UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

A PROPOSAL FOR MODEL-BASED AUTOMATION OF ENTERPRISE SERVICE CHANGE MANAGEMENT PROCESSES

TESIS DOCTORAL

FÉLIX CUADRADO LATASA
Ingeniero de Telecomunicación
2009

DEPARTAMENTO DE INGENIERÍA
DE SISTEMAS TELEMÁTICOS

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE MADRID

A PROPOSAL FOR MODEL-BASED AUTOMATION OF ENTERPRISE SERVICE CHANGE MANAGEMENT PROCESSES

Autor: FÉLIX CUADRADO LATASA
Ingeniero de Telecomunicación

Director: JUAN CARLOS DUEÑAS LÓPEZ
Doctor Ingeniero de Telecomunicación

2009

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día 4 de diciembre de 2009.

Presidente: _____

Vocal: _____

Vocal: _____

Vocal: _____

Secretario: _____

Suplente: _____

Suplente: _____

Realizado el acto de defensa y lectura de la Tesis el día 22 de diciembre de 2009 en la E.T.S.I.T. habiendo obtenido la calificación de _____

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

A mis padres y
a Nacho

Agradecimientos

En primer lugar tengo que expresar mi sincera y total gratitud a la persona que ha hecho posible esto: mi director de tesis, Juan Carlos Dueñas. Muchas gracias por tu confianza y por tu apoyo, por tu disponibilidad para aconsejarme y guiarme en este proceso, encontrando siempre una solución a los problemas que han aparecido.

También quiero agradecer a todos los compañeros del C-215 (y Siberia), que han contribuido para que éste sea un entorno de trabajo motivador y acogedor. Gracias Jose, por ayudarme desde el primer día, por tus consejos y por tu amistad. A Rodrigo, por ser un gran compañero y amigo, que me ha apoyado todo este tiempo. A Don Hugo A., siempre juicioso y cercano, Álvaro, Boni, Marta, Freakant, Samuel, Chema, Rubén, Bea, Antonio, y Sandra, y los integrantes del equipo ITECBAN despliegue y testing. A July, que hace que todo esto funcione. Por último, gracias al departamento y la universidad, por el apoyo prestado estos años.

I would also like to mention the European partners of this journey. Thanks to Hans, Patricia, Rahul, Qing, Maryan, Viktor, Adam, Rik, Remco, Daniela and Elly for three great months at Vrije Universiteit. Also, thanks to the external reviewers of this thesis, Didier and Patricia, for their valuable advice.

Gracias a mis padres, a Nacho, y al resto de mi familia, por darme su cariño y apoyarme en todo momento, aunque haya escogido este camino tan complicado.

Y por último, no me olvido de todos mis amigos, que han sabido comprenderme y animarme en los momentos menos buenos. Gracias Perico, Marisol, Bea, Carlos, Irene, Ignacio, Miguel, Evaristo, y muchos otros que se merecen estar en esta lista.

Abstract

In a globalized world enterprises have to face greatly increased competition, demanding agility to release new products and update to customer demands. From a technological perspective, these factors have lead to the adoption of the service oriented paradigm, which must be supported by a robust IT infrastructure. One of the main competitive factors is the quality of service provided, ensuring the elements of the services portfolio have high availability, and unnoticeable response times. These non functional requirements are partially supported by the execution infrastructure, composed by multiple, heterogeneous servers with specialized roles, distributed over a network. However, the combination of these factors greatly complicate technical management processes of the infrastructure such as diagnosing the environment status, planning the required changes or applying corrections to improve its performance. Frequently those tasks are manually executed by an IT administrator, but this approach is very costly and hampers the desired agility. An increased degree of automation in service change management operations is a must for obtaining the potential advantages of the service oriented approach.

This dissertation proposes an enterprise service management architecture with automated operation capabilities. One of the cornerstones of this proposal is an information model of all the relevant management information. The proposed model builds upon the common ground of the main information model standards to characterize both the logical artifacts, originated from the service development process, and the managed runtime elements, ranging from hardware nodes to the provisioned services. The model not only allows to represent different environment configurations but also provides well defined expressions for validating the correctness of any system state, and automatically obtain the required configuration values for some of the managed resources. In addition to the information model, the business objectives, desired functionality, and changes to the domain have been defined using the same concepts. This way, the effect of external changes to the environment configuration, as well as its impact on the stability and functionality of the environment can be automatically analyzed.

After defining all the relevant management information through a cohesive model that covers both technical and business aspects, this dissertation proposes an algorithm for automating the execution of service configuration change management activities, based on pseudo-boolean SAT techniques. The proposed algorithm analyzes the current state of a managed domain and, in case the situation is not stable or desirable obtains the set of required changes to restore the system to its intended functionality. Instead of defining separate processes for installation, reconfiguration, or removal of selected elements, the same reasoning steps produce a change plan with the necessary operations.

Finally, after taking into account the requirements of enterprise applications, an architecture for a service change management system has been proposed, based on the described models and reasoning techniques. A prototype of the proposed architecture and algorithms has been developed and validated through a set of case studies taken from the context of a real banking organization. The results of the validation show how different situations such as initial provisioning or reaction to hardware malfunctions are correctly addressed by the architecture, as well as how the proposal scales with increasingly larger environments and defined services.

Resumen

La globalización ha incrementado el nivel de competencia entre las empresas, obligando a una mejor adaptación a las necesidades de los clientes, y a recortar los ciclos de desarrollo de nuevos productos. Estos factores pueden ser soportados a nivel técnico por una infraestructura orientada a servicios, que tenga suficiente robustez para apoyar las necesidades del negocio. En este contexto, mejorar la calidad de servicio es un posible factor para diferenciarse de la competencia, ofreciendo servicios con alta disponibilidad y un tiempo de respuesta imperceptible. Con el objetivo de poder soportar estos requisitos no funcionales la infraestructura base de ejecución está formada por un conjunto de servidores heterogéneos, distribuidos sobre la red de la compañía. La combinación de estos factores complica enormemente las actividades gestión de los servicios, como el diagnóstico de la situación del entorno, o la identificación de los cambios necesarios para corregir una incidencia o mejorar el rendimiento de los servicios en ejecución. Estas actividades son frecuentemente realizadas manualmente por un administrador de sistemas, aunque el esfuerzo que conlleva este tipo de cambios imposibilita aplicarlos con la agilidad necesaria. Para aprovecharse de las ventajas de la orientación a servicios resulta necesario incrementar el nivel de automatización de estos procesos.

Esta tesis propone un conjunto de modelos y técnicas para automatizar las operaciones de cambios de configuración a los servicios empresariales. Como base de la propuesta se propone un modelo genérico que captura toda la información del entorno relevante para su gestión, con el objetivo de ser automáticamente interpretable por los sistemas de control de los cambios. El modelo se basa en las principales abstracciones definidas en los estándares de gestión, y sobre ellas modela tanto los elementos lógicos, que provienen directamente del proceso de desarrollo, como los elementos del entorno de ejecución, caracterizando desde los nodos hardware hasta los servicios en operación. El modelo no sólo permite representar la configuración del entorno, ya que también define cómo validar la estabilidad del mismo, así como obtener el valor correcto de configuración de algunos elementos. Sobre estos mismos conceptos también se ha formalizado la definición de los objetivos de negocio que debe cumplir el sistema, o los cambios que puede experimentar. Esto permite un análisis automático del efecto de un cambio externo en la configuración actual, así como estimar el impacto del cambio en la estabilidad o funcionalidad del sistema.

Tras capturar toda la información relevante de gestión con los modelos propuestos, esta tesis propone un algoritmo para gestionar las actividades de gestión de cambios, basada en un sistema resolutor SAT pseudo booleano. El algoritmo analiza el estado actual del dominio gestionado y, en caso de que la situación actual no sea estable o deseable, obtiene un conjunto de cambios que restaurarán la funcionalidad deseada del sistema. En lugar de definir procesos independientes para instalar, reconfigurar, o eliminar componentes del sistema, la solución propuesta es capaz de generar un plan de cambios con las operaciones necesarias mediante el mismo procedimiento.

Por último, teniendo en cuenta los requisitos propios de las aplicaciones empresariales, se ha propuesto una arquitectura de un sistema de gestión de cambios de servicios empresariales, basada en los modelos y técnicas de razonamiento descritas anteriormente. También se ha desarrollado un prototipo de esta arquitectura, que se ha validado mediante un conjunto de

casos de estudio extraídos del contexto de una organización bancaria. Los resultados de este trabajo de validación muestran cómo la arquitectura propuesta es capaz de tratar correctamente distintas situaciones, desde el aprovisionamiento inicial de un nuevo servicio hasta el diagnóstico y reparación de una avería en uno de los dispositivos hardware del entorno. Finalmente, la escalabilidad de la propuesta se ha evaluado mediante una serie de experimentos con modelos del entorno gestionado y la lista de servicios disponibles progresivamente de mayor tamaño.

Table of contents

List of Figures

Glossary

ACEL: Autonomic Computing Expression Language.

ADL: Architecture Description Language.

AM: Autonomic Manager.

API: Application Programming Interface.

B2B: Business-to-Business

B2C: Business-to-Consumer

BPEL: Business Process Execution Language.

BPMN: Business Process Modeling Notation.

BRM: Business Rules Management.

CDN: Content Delivery Network.

CENIT: Consorcios Estratégicos Nacionales en Investigación Técnica.

CI: Configuration Item.

CMDB: Configuration Management DataBase.

CORBA: Common Object Request Broker Architecture.

CPU: Central Processing Unit.

CRM: Customer Relationship Management.

D&C: Deployment and Configuration

DBMS: Database Management System

DIT: Departamento de Ingeniería de servicios Telemáticos.

DMTF: Distributed Management Task Force.

DMT: Device Management Tree.

DPWS: Device Profile for Web Services.

DS: Discovery Service.

DSL: Domain-Specific Language.

DU: Deployment Unit.

EAI: Enterprise Application Integration.

EAR: Enterprise Archive file.

ECA: Event Condition Action.

EJB: Enterprise JavaBeans.

EMF: Eclipse Modeling Framework.

EPL: Eclipse Public License.

ERP: Enterprise Resource Planning.

ESB: Enterprise Service Bus.

eTOM: enhanced Telecom Operations Map.

ETSI: European Telecommunications Standards Institute.

EU: European Union.

FCAPS: Fault, Configuration, Accounting, Performance, and Security

HTTP: Hypertext Transfer Protocol.

IBM: International Business Machines Corporation.

ICT: Information and Communications Technologies.

IEEE: Institute of Electrical and Electronics Engineers.

IETF: Internet Engineering Task Force.

IFIP: International Federation for Information Processing.

ITECBAN: Infraestructura TECnológica y metodológica de soporte para un core BANcario.

IP: Internet Protocol.

ISO: International Standards Organization.

IT: Information Technology.

ITIL: Information Technology Infrastructure Library.

ITSM: Information Technology Service Management.

ITU: International Telecommunication Union.

ITU-T: International Telecommunication Union - Telecommunication Standardization Sector.

JEE: Java platform, Enterprise Edition.

JAR: Java Archive file.

JCA: Java EE Connector Architecture.

JDBC: Java Database Connectivity.

JEE: Java Enterprise Edition.

JMS: Java Message Service.

JMX: Java Management Extensions

JRE: Java Runtime Environment.

JVM: Java Virtual Machine.

LAN: Local Area Network.

LAS: Logical Application Structure.

LDAP: Lightweight Directory Access Protocol.

MAN: Metropolitan Area Network.

MDA: Model-Driven Architecture.

MIB: Management Information Base.

MQ: Message Queue.

MOF (from CIM): Managed Object Facility

MOF (from OMG): Meta Object Facility.

MOWS: Management Of Web Services.

MUWS: Management Using Web Services.

NGOSS: Next Generation Operational Support System.

OASIS: Organization for the Advancement of Structured Information Standards.

OCL: Object Constraint Language.

OMG: Object Management Group.

OSGi: Open Services Gateway Initiative.

OSI: Open Systems Interconnection.

OWL: Web Ontology Language.

PB-SAT: Pseudo Boolean SATisfiability problem.

PDDL: Planning Domain Definition Language.

PDP: Policy Decision Point.

PEP: Policy Enforcement Point.

PIM: Platform-Independent Model.

PR: Policy Repository.

PSM: Platform-Specific Model.

QoS: Quality of Service.

RAM: Random Access Memory.

RFC: Request For Change.

REST: Representational State Transfer.

RMO: Resource Management and Operations.

SAT: boolean SATisfiability problem

SCA: Service Component Architecture

SCM: Source Code Management.

SDK: Software Development Kit.

SISL: Service Information Specification Language.

SIP: Strategy, Infrastructure, Product.

SLA: Service-Level Agreement.

SLM: Service-Level Management.

SMO: Service Management and Operations.

SNMP: Simple Network Management Protocol.

SOA: Service-Oriented Architecture.

SDD: Service Deployment Descriptor.

SID: Shared Information/Data Model.

SP: Service Provider.

SQL: Structured Query Language.

SML: Service Modeling Language.

SPDF: Simple Prerequisite Description Format.

TCP: Transmission Control Protocol.

TINA-C: Telecommunication Information Networking Architecture Consortium.

TMF: TeleManagement Forum.

TMN: Telecommunications Management Network

UA: User Agent.

UML: Unified Modeling Language.

UPM: Universidad Politécnica de Madrid.

URI: Uniform Resource Identifier.

URL: Uniform Resource Locator.

W3C: Word-Wide Web Consortium.

WAR: Web Archive file.

WAS: WebSphere Application Server.

WLAN: Wireless Local Area Network.

WS: Web Service.

WSDM: Web Services Distributed Management.

WSDL: Web Service Definition Language.

WS-BPEL: Web Services Business Process Execution Language.

XML: Extensible Markup Language.

XSD: XML Schema Definition.

XSLT: eXtensible Stylesheet Language Transformation.

1. Motivation

The IT infrastructure has increasingly taken a central role in modern enterprises, which can't be nowadays conceived without it for working efficiently and competitively. The use of IT allows adopting faster and more efficient business processes, becoming a key factor in competitive advantage in the last years. Currently we can even talk about Service-Oriented-Organizations [78], whose business model is providing IT computing or business-service functionalities to other organizations and companies.

The increased importance of IT infrastructure has led to important investments in infrastructure, which must be amortized over long periods of time. However, the IT systems evolve rapidly, rendering purchased units as legacy technology before their lifetime has been completed. On top of that, it is necessary to upgrade applications and enterprise services, in order to continuously improve process efficiency to gain a competitive advantage. This force lends to adopt new technologies for Business to Business (B2B) and presentation services. Thus, systems are composed by not only legacy systems, mainframes, databases, but also Java Enterprise Edition (JEE) application servers, or Business Rule Manager (BRM) systems. The resulting enterprise infrastructure is a highly complex distributed system, composed by dozens of different servers and application containers, deployed over hardware machines interconnected through complex network distributions, containing firewalls, virtual private network and other access restriction and security mechanisms. To further complicate the situation, server vendors try to differentiate from each other by providing additional features over the common standards, which effectively breaks the interoperability between artifacts from different providers.

Interoperability between all the components of the IT infrastructure is usually achieved by adopting a higher-level integration layer, which is based on Service Oriented Architecture and Business Process Management (SOA/BPM). This way, each artifact of the system is presented as a service, hiding its implementation details and providing a uniform high-level view. Services are published in directories and connected through an Enterprise Service Bus (ESB), where additional non-functional capabilities can be added to the communications, such as logging, or data transformation. On top of that, BPM technologies, such as BPEL (Business Process Execution Language), orchestrate the activities, bridging the gap between the IT infrastructure and the business processes.

The SOA / BPM approach has contributed to maximizing the use of the existing IT infrastructure, but managing the system is still a pending issue. Although they are now treated uniformly with the abstraction of services, managers still need to cope with the enormous heterogeneity and complexity of the infrastructure.

Traditional management processes are identified with human operation over a management administration console. Monitoring information and events are collected and aggregated into the console, and the human administrator invokes specific operations on the environment based on the identified objectives and the collected information.

Operations are executed in scripts, containing the exact set of machine-specific instructions for achieving a concrete task. Because of that, scripts lack reusability and suffer tremendously from the increased complexity, distribution and heterogeneity of current IT systems.

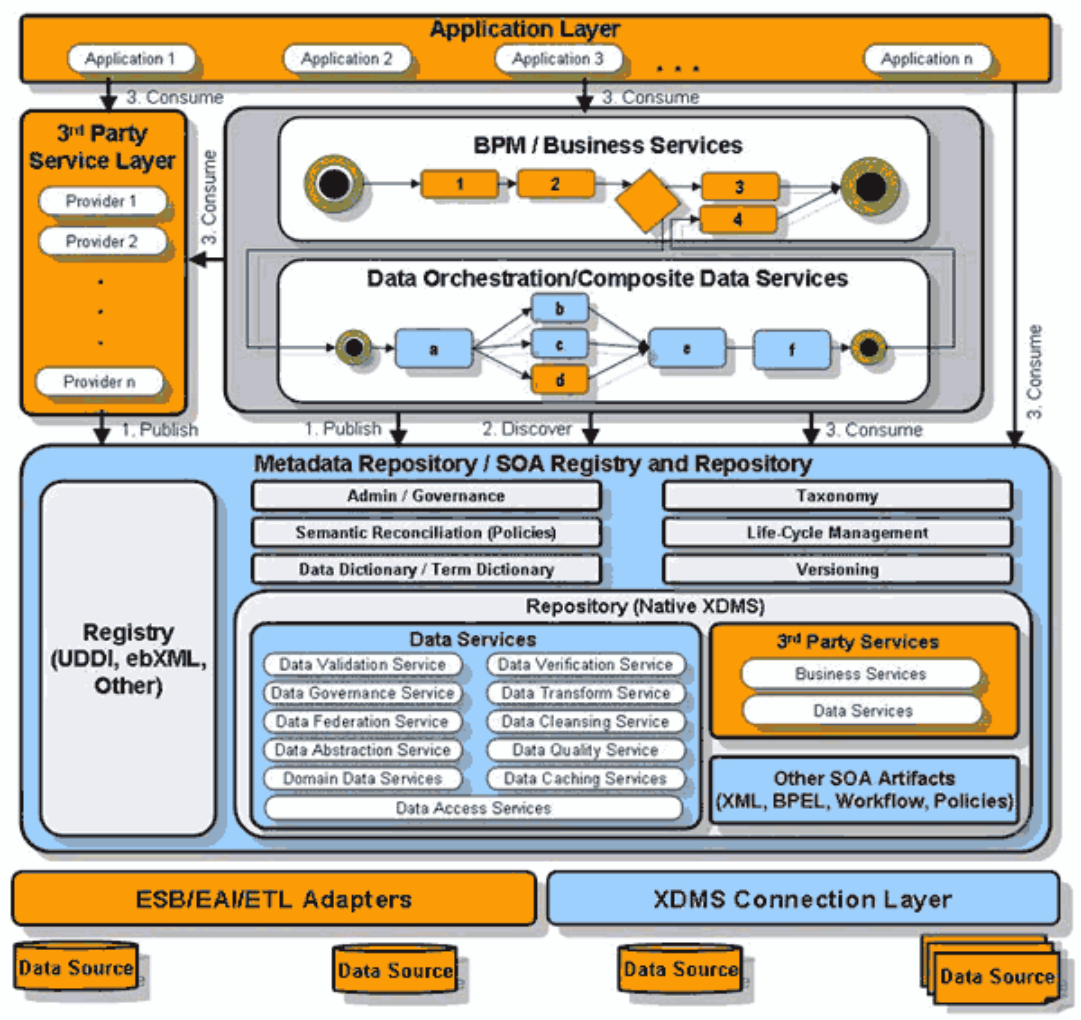

Figure 1 Functional view of an Enterprise SOA Infrastructure

This approach has severe limitations, which become more evident as the complexity and heterogeneity of the managed systems keep growing. Changes to the environment impact the complete management process, as the configuration and workflows must be manually adapted to the specifics of the environment. Management operations are manually created and composed, requiring lots of knowledge from the administration experts, and can hardly be reused. A runtime system configuration has component and configuration dependencies between heterogeneous artifacts, propagating the impact of any change or error throughout the whole system.

The importance of addressing these limitations is widely acknowledged not only by the academia but also by the main industrial companies. As a reference, the third edition of the ITEA Roadmap for Software Intensive Systems and Services [51], defined by the ITEA2 Consortium industrial members (such as Nokia, Phillips, Alcatel, Siemens or Telefonica), selects it as one of the four main challenge areas which should be addressed by the European research community. The roadmap highlights the criticality of improving the deployment and

reconfiguration processes of large, heterogeneous distributed systems, while at the same time supporting the non functional capabilities such as efficiency, or reliability.

There is clearly a necessity of finding models, paradigms and mechanisms to address all the different factors, lessen human intervention and automate parts of the enterprise service management processes.

1.1. Research Methodology

The SOA paradigm presents new opportunities for providing more scalable, flexible and agile systems. However, at the same time it poses new and updated challenges to every software engineering discipline, which must be addressed while respecting the already existing concerns and requirements from the domain stakeholders.

Some of these challenges have been identified over the development of the CENIT ITECBAN project (2006-2009). ITECBAN is a Spanish Research Project from the Ingenio 2010 CENIT program which aims at defining a complete core banking solution, based on the SOA-BPM approach. From this general objective, my participation in the project was centered in developing a deployment and configuration system for the banking services. The work performed in the context of the project revealed the main challenges for implementing this management system: service distribution must be controlled over time by the architecture, a range of different runtime artifacts (database information, business rules, web applications, business logic services) from multiple vendors must be supported, and a large set of use cases (service installation, container configuration, service update, and so on) should be implemented by the proposed system, while at the same time providing the scalability and efficiency required for enterprise systems.

After identifying the main challenges for an automated management of enterprise services, the current state of the art was analyzed in order to see whether those concerns were completely supported. In this process two main aspects were analyzed: the existing standards on information models, and the different paradigms for service management. The study revealed numerous initiatives which addressed some of the concerns but which were generally focused on the network aspects, or constrained to a specific technology. There was not a single solution that completely addressed the different changes in a generic way. The work by Dr. Ruiz [93] was the most interesting approach, as it supported deployment operations reasoning over a generic model, analyzing logical dependencies and supporting artifact heterogeneity. However, the proposed solution was focused on the initial provisioning aspects, lacking both model expressivity and functionality to support other scenarios such as update or reconfiguration. Nonetheless, it showed how basing the architecture on a generic model for representing all the relevant information could allow supporting the requirements of this scenario.

After that conclusion, the modeling abstractions from the analyzed standards and initiatives were distilled in order to propose an abstract model built on the common ground of the standards which could represent all the information and dependencies which must be addressed over a heterogeneous distributed environment. Based on those generic abstractions, a specific model was defined which could represent the relevant elements of an enterprise service management infrastructure. On top of that, referring to those abstractions,

the changes which could impact the environment were identified and classified. With these concepts properly defined, the base function of the management system was defined as obtaining and applying the required changes to the managed domain in order to restore its correctness.

Once the problem was completely established and modeled, alternatives were looked for defining an algorithm with can reason over all the information from the domain, and automatically suggest a set of changes that can return the system to a desirable and stable state, regardless of the type of change. After evaluating the available approaches a pseudo-boolean satisfiability (PBSAT) solver was selected as the base technique. The proposed solution converts the domain information into boolean variables, which represent the potentially reachable states by applying a set of operations, and functions, which restrict the valid combinations of these variables, in order to ensure that the obtained result represents a correct and stable configuration of the environment. Once the resolution engine has been completely configured and invoked, variable assignations are interpreted, obtaining from them the set of required changes which need to be applied to the domain in order to achieve the desired configuration.

On top of that, in order to enable the adoption of the proposed solution in an enterprise context, a reference architecture was defined which would support the complete functionality of a service change management system. The architecture components leveraged the proposed algorithm as the core reasoning module of the enterprise management architecture, and described how to integrate it with the rest of the enterprise infrastructure.

Finally, a prototype implementation was developed, and a number of validation tests were defined an executed. The results from those experiments verified that the solution described at this dissertation supports the identified management functions, and can successfully operate over a set of different scenarios with the described abstractions and algorithms.

1.2. Structure of the document

After this chapter, which provides the motivation for the work and the research approach followed, the next chapter details the results of the analysis of the state of the art, comparing the main standardization initiatives in the information modeling, and the main approaches for systems management.

Chapter three defines the objectives of the work under this thesis, after determining that the current solutions did not address all the concerns about the automation of service configuration activities. The next two chapters detail the main contributions of this work. Chapter 4 describes the proposed model abstractions for representing heterogeneous, distributed management, as well as a specific model for service-based applications management. Chapter 5 defines the objectives of a service management system and characterizes the set of changes that can occur at an environment. For the specific case, internal operations allowed by the management system are defined. After those concepts have been clearly established, an algorithm based on pseudo-boolean SAT solvers is described for solving the management functions. This contribution takes as only input the previously defined models, and automatically obtains a stable and desirable configuration for the system.

Chapter 6 describes a reference architecture for the service management system that uses the proposed models and algorithm to automate the main tasks. The results from a set of validation experiments are discussed in chapter 8.

Finally, the last chapter details the main conclusions of this work, as well as a description of the most interesting future research activities which have been identified during the development of this work.

2. State of the art on service configuration management

Service configuration management of distributed heterogeneous systems is an active research field, for both academia and industry. Over the years, several standards, models and research paradigms have been proposed to address the challenges of service management. For the purposes of this document, I will define service management as the set of methods, processes and techniques used to efficiently manage application services. Although these entities are often viewed only as logical software artifacts, the scope of the management system must also extend to the running services, the supporting platforms, as well as the physical infrastructure and the network topology of the system.

This section is structured into four parts. First, I will introduce some basic definitions and describe the different functions, areas, and dimensions of a management system, as well as the business processes where the management activities are involved. Once these terms have been properly defined, I will describe the most relevant standards for enterprise service environment modeling. Modeling standards capture the expertise of management professionals, offering a common representation model for the relevant information, and an associated set of management model and operations. Finally, I will describe the most promising service management paradigms, describing their main characteristics and which problems they are trying to solve. Each paradigm is represented with concrete research initiatives, which show how their principles are taken to practice.

2.1. Basic management definitions

First of all, I will try to define what we understand for management, using Hegering's [42] definition of networked system management: "the management of networked systems comprises all the measures necessary to ensure the effective and efficient operation of a system and its resources pursuant to an organization's goals". As this is a very broad topic, contributions usually focus on managing specific parts of the system. If the fundamental elements are the network elements we will speak about *network management*. If the main entities are the end systems we will use the term *system management*. As the software layer relevance keeps growing, the management of existing applications has acquired a greater relevance, to the point of opening the field of *applications management*. The SOA (Service-Oriented Architecture) paradigm has spread over distributed architectures, shifting the focus from the software components to the runtime functionality exposed to the complete system. *Service management* works at that abstraction level. Finally, business processes, strategies, and policies are the focus of *business management*. This classification of management levels was established by The TMN (Telecommunications Management Network) [102], a standard defined by the telecommunications operators derived from OSI. Next figure shows the TMN management pyramid.

Figure 2 The TMN management pyramid

An *integrated management* approach spans over several management layers in order to provide complete traceability among the different layers, aggregates agents and tools from different levels and reasons over the complete information. As these definitions are too general I will describe the more widely accepted classifications of management systems in order to have a clearer picture.

2.1.1. Management functional areas

In addition to the abstraction level, another widespread criterion for classifying management solutions is the type of supported functionality. As the management definition is very broad, it is often segregated into different functional areas. In this context the universally accepted classification is the FCAPS model proposed by the OSI network management architecture [49]. According to the specification, the functionality of any management system can be broken down into the following five functional areas:

Fault Management: Includes the processes related to detecting failures in the managed system, identifying the source of the errors and returning the system to a correct state. Additional tasks include network and systems monitoring, alert propagation and processing, or user assistance interfaces.

Configuration Management: Covers all the operations which manipulate the structure of the distributed system, its state, and parameters of managed objects in order to ensure a normal functioning of the system. The concept is very broad, including specific configurations such as software installation and update, network element parameters (router tables, IP interface attributes), or server and services configuration.

Accounting Management: This category covers fundamentally user-related topics. It is in charge of user management, including registration of usage data, account policies, and billing. Its place as one of the five main categories is due to its vital importance in the business model of telecommunications operators.

Performance Management: includes all operations which improve the way the system performs its normal operations. Some examples of specific functions are QoS (quality-of-

service) monitoring and assurance, performance bottleneck detection, capacity planning, threshold analysis and failure predictions.

Security Management: covers all actions related to managing the security of the distributed system. Specific examples include intruder detection, protection from passive and active attacks, threat analysis, as well as confidentiality (encryption) and data integrity assurance.

2.1.2. Dimensions of technical management.

In order to provide a useful framework for classifying management systems, Hegering proposes a Kiwiatt-like, multi dimensional taxonomy. This way, a management system can be classified with respect to five different dimensions. In addition to the previously described **functional areas** and **management levels**, the following categories are proposed:

Development stages: Every managed element which is part of the runtime system has its own lifecycle, starting from the initial planning stages where its requirements are defined, going over its implantation, maintenance, and final retirement. A management system can support all of these stages or focus on a specific part of them.

Network types: The network topology of the managed system heavily influences its design, functionality and limitations. Depending on the context, factors such as scalability, infrastructure coordination, or network capacity limitations come into play. The considered alternatives range from local networks to increasingly larger topologies including Internet-wide management architectures.

Information types: Similarly to the previous point, the nature of the managed information types not only impacts the managed distributed system, but also the management architecture.

Next figure shows the graphical representation of the five dimensions of management.

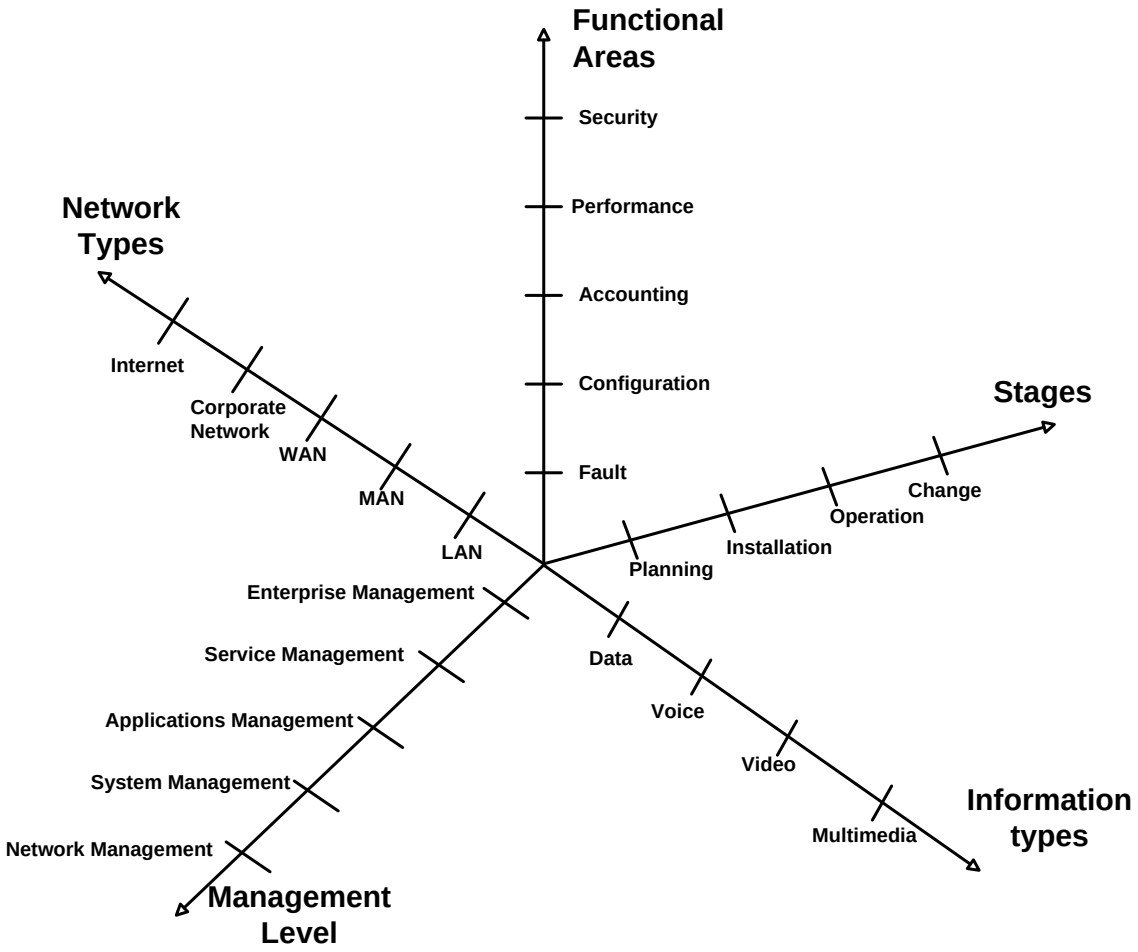

Figure 3 Dimensions of management

2.1.3. Business-Level Management

In order to complement these base definitions, and before analyzing technical solutions to the problems of management, the most important business process frameworks related to enterprise management activities will be described.

In the following pages I will briefly analyze the two main business process frameworks where service management activities are involved: ITIL and eTOM. Although these specifications do not go into low level detail of the technical challenges and solutions of the service management problem, they must be considered for the proposed solution as they provide the organizational context where the service, configuration, and deployment activities must be executed, often providing additional requirements.

2.1.3.1. IT Service Management standards

The Information Technology Infrastructure Library (ITIL) [84] [85] is a set of process-centric best practices for the management of IT services. ITIL was developed in the United Kingdom Office of Government Commerce although its popularity has extended over the whole world. Because of that, the ISO has formalized ITIL as the ISO/IEC 20000-1 [50], establishing certification requirements. ITIL processes allow IT organizations to manage their services efficiently and reliably.

The most extended version of ITIL is v2, which focuses on two main process areas: Service Delivery and Service Support. On May 2007 the ITIL Refresh Project released an updated version, frequently referred as ITIL v3, with a new focus on the business value provided by the different IT processes [54].

ITIL is a very large specification, from which I will focus on the related concepts for service management, defined in the Service Support process. This part of the specification contains the best practices obtained by professionals over the service maintenance process, including activities ranging from incident management to the solution of the problems. The next picture shows the main elements of the model. In the central part it can be seen one of the central pieces of ITIL: The Service Desk, which is a unified point of interaction with all service users (user level). It handles all incidences, queries and requests, acting as the only interface for all the service support processes.

The other central element of ITIL is the Configuration Management DataBase (CMDB), seen in the lower part of the picture. This component centralizes all the management-relevant information of the service lifecycle, including the Configuration Items and its relationships, as well as incidences, errors, or releases.

The service support process starts when service users report an incident to the service desk. After analyzing it, it is handed to the incident management process. In this step the incident is recorded, and compared against the existing database of known system incidences. In case there are no previous records of the notified issue, it is escalated to the problem management activity, where a root cause analysis is performed in order to diagnose the technical cause of the problem. The found solution can either be directly applied , notifying the user and closing the process, or may need to perform changes on the software services or the base infrastructure, which will be applied in the change management process through a Request for Change (RFC) notification.

The change management process interprets the request and implements the necessary changes to the services and / or the infrastructure to ensure the initial SLA levels can be restored. This process is executed in a controlled environment, with approval of the proposed changes to the infrastructure. These changes are executed through the release management activity, where the impact of the modifications is evaluated and the system configuration is adjusted to them. Finally, the configuration management process keeps the CMDB synchronized with the updates to the Configuration Items (CI), as well as its relationships after the system configuration has been updated to deal with the raised incident.

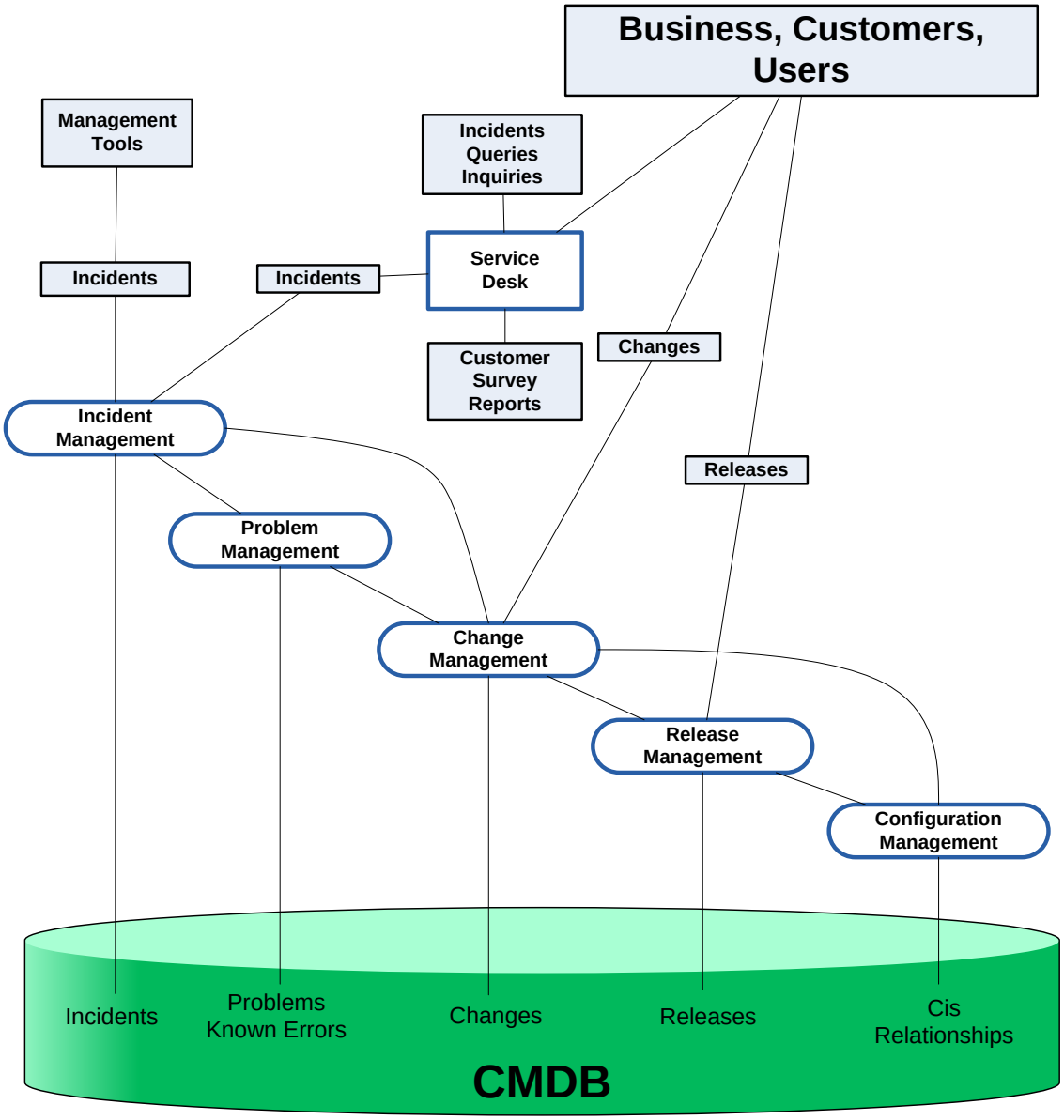

Figure 4 The ITIL Service Support Process

ITIL processes are modeled at an organizational level, with both the company departments and external clients and providers as the process actors. Because of that, it is more focused on high-level processes and does not provide direct links to specific technologies, processes and architectures for realizing the best practices described in the specification.

However, ITSM frameworks should be the basic frame for management systems, as they provide the organizational context where the business concepts are integrated with the technical process. In [97] an example process s described for the definition of a Service Level Management architecture. It starts from the relevant Service Support processes, and through a set of scenarios described a management architecture supporting them.

2.1.3.2. eTOM

The enhanced Telecom Operations Map (eTOM) [35] is a business process framework for Internet and Communications Service Providers. It is published and maintained by the TeleManagement Forum (TMF). Its predecessor, the Telecom Operations Map (TOM) was first

published by the TMF in 1998 and was superseded by the eTOM in 2001. Since 2004, eTOM is also an ITU-T Recommendation (M.3050). The standard is one of the main components of the New Generation Operation System and Software (NGOSS) project, whose objective is to define a vendor-independent architecture for management systems.

eTOM defines five view levels of processes, from level 0 to level 4. The views provide increasingly more detailed views of the processes. Level 0 defines only three fundamental processes [7](also known as 'process areas'): SIP (Strategy, Infrastructure and Product), Operations and Enterprise Management. Service configuration processes would fall under the operations area, which is also the most developed part of eTOM, as it was the scope of the original TOM. The standard refines operations processes up to level 3.

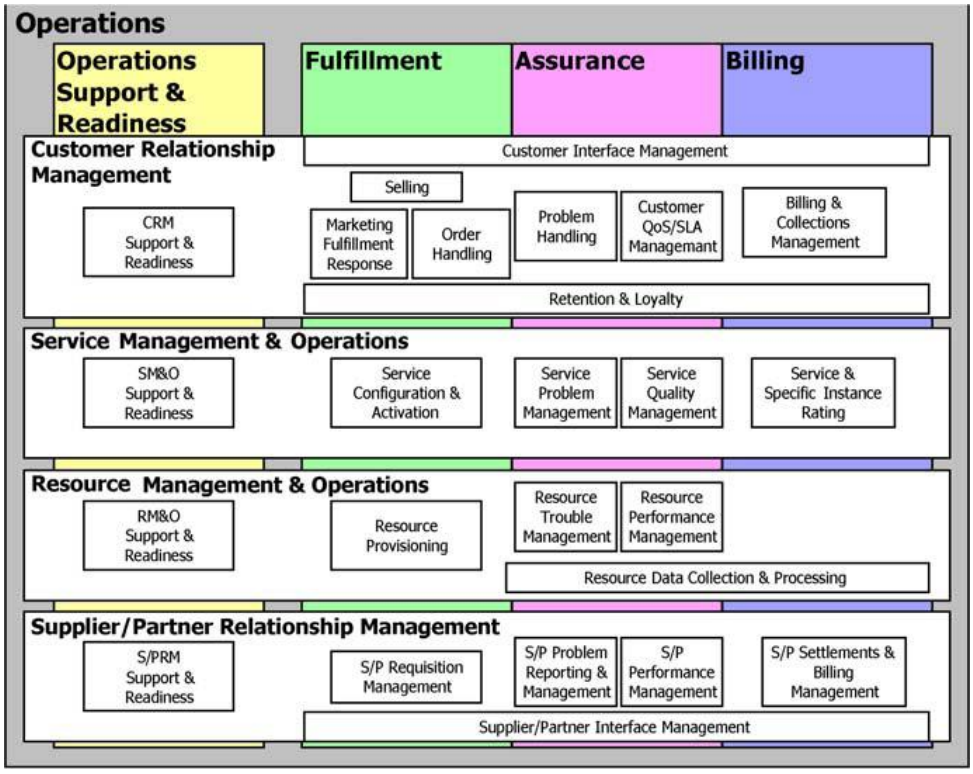

Figure 5 eTOM operations level 2 process matrix

We can see in Figure 5 the process matrix of the level 2 view of the *Operations* processes. The processes are aligned to four business-level functional areas, resembling organization departments: Customer Relationship and Management (CRM), Service Management and Operations (SMO), Resource Management and Operations (RMO) and Supplier / Partner Relationship Management (SPRM). The matrix is completed by four vertical business goal areas: OSR (Operations Support and Readiness), Fulfillment, Assurance and Billing. The technical processes related to service configuration clearly belongs to the level 2 SCA (Service Configuration & Activation) process, although in a broader context a service configuration workflow will possibly involve additional processes of the fulfillment category. Figure 6 shows the level 3 process decomposition of SCA defined by eTOM.

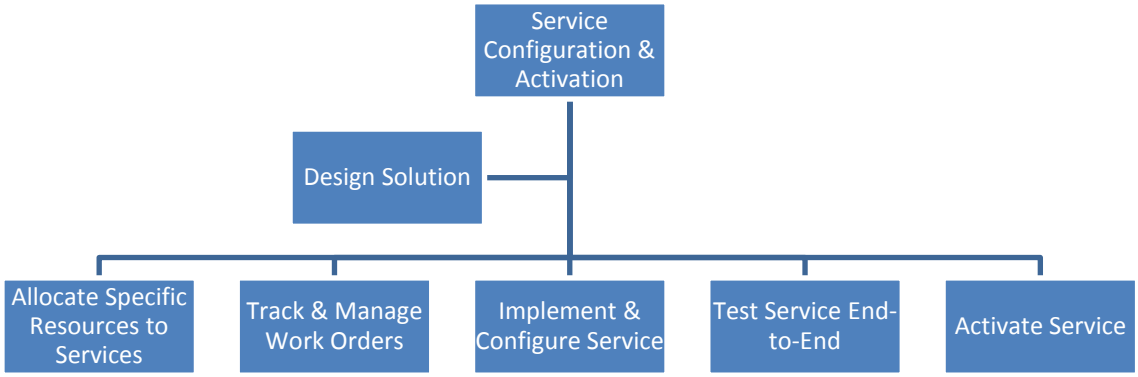

Figure 6 eTOM Service configuration & Activation Level 3 decomposition

eTOM provides a fine detailed process framework for service provisioning and configuration. However, it does not provide proposed implementations of the activities or even guidelines, as they are out of the scope of the specification. Nonetheless, eTOM is a fundamental reference for the organizational aspects of service management.

2.1.3.3. ITIL and eTOM comparison

Although the two analyzed standards cover the service management processes and workflows their approach and structure is quite different. A complete analysis of the similarities and differences can be found in [98]. First, the original background of each initiative differs. eTOM is defined in the telecom industry, and is part of the NGOSS initiative, whereas ITIL originates in the IT industry. In spite of their separate origins, the convergence phenomenon has caused some degree of overlapping between both specifications.

If their scopes are compared ITIL defines the set of business processes that are required to support each service in the SLA, and focuses on the expertise acquired through the years in the form of best practices. On the other hand, eTOM defines a high-level process framework, and defines mechanisms for managing the relationships between the different actors, services providers, consumers, and partners. Therefore, the scope of eTOM is much broader, although in the current version it is much more detailed in the operations processes. Approximately ITIL would fall under the operations assurance eTOM process, also impacting the fulfillment and billing sub-support processes.

Nonetheless, the TMF has identified the need of a mapping between the two frameworks, and has published in [110] a mapping of ITIL processes to eTOM level 2 processes.

2.2. Enterprise service environment modeling

The mandatory element for service and systems management is a practical representation of the managed environment. This model must contain all the relevant information regarding management about the target environment. It must allow defining the characteristics of each element of the environment as well as the topology, including the existing relationships between these elements. Because of the importance of this topic there are several standards devoted to defining the reference information model for managing a distributed system. In this section, I will discuss the main characteristics of the most relevant specifications. As it can

be hard to separate the information model from the management operations (which are tied to its), this study will cover both aspects.

2.2.1. SNMP/MIB

The Internet Engineering Task Force (IETF) defines a set of standards which aim to provide a complete management solution for networked systems. The result of this work is the Simple Network Management Protocol (SNMP) [9], which, in combination with the Structure of Management Information (SMI) v2[74], and the Management Information Base (MIB) [75] define together a simple management model, a way of describing the management objects and a mechanism for packaging and storing the information.

The model is oriented to variables management. Information is represented by collections of variables, either as simple primitive types or organized as conceptual tables. As this set can grow to huge figures, SNMP provides a four-level naming system which ensures a unique identifier for each managed variable. In this model, the two upper-level layers define the context engine and the context name where the variable is stored. Next two levels identify variable type and instance. This structure also eases the management functions, as the information can be explicitly aggregated in contexts, which are frequently self-discovered, and context variables can be easily iterated.

The SMI data model is based on conceptual tables, which have a primitive data type for each column. Columns can be marked as index, similarly to a relational database, although the model is much more limited (for example, only one index can be used for fast lookups). As a conceptual table is a property, it has a unique identifier. Managed objects defined by SMI are usually called MIB objects. A set of interrelated MIB objects is a MIB module.

Thanks to the simplicity of the data model, SNMP protocol operations are very simple. There are read operations (get, getnext, getbulk), write operations (set) and notify operations (trap, inform).

SNMP / SMI define low-level management operations, and a restricted data model. These characteristics complicate building advanced functionality or automate managing processes through these standards. Thus, it is a very successful standard, but its scope is centered on event notification, and monitoring a large amount of simple devices. In an enterprise environment, services are the base granularity level, and complex dependencies must be managed. Because of that, SNMP is not well suited to automating high-level enterprise management operations, although it is the reference standard for network management operations.

However, there are some initiatives such as the work by Danciu et al[17], which propose an approach to aggregate the low level properties usually managed by the defined MIBs into coarser grained Service MIBs, with all the relevant information for the adequate management of the instrumented service. In their work they also propose a methodology for identifying the relevant management properties of each service, a language, the SISL (Service Information Specification Language) for expressing the relationships between the low level resource data and the service attributes, and an architecture for supporting those concepts. However, this approach is focused on the service abstractions from lower level elements, but still does not

consider the environment-wide dependencies of the elements running the services, ending somewhat limited in its expressivity.

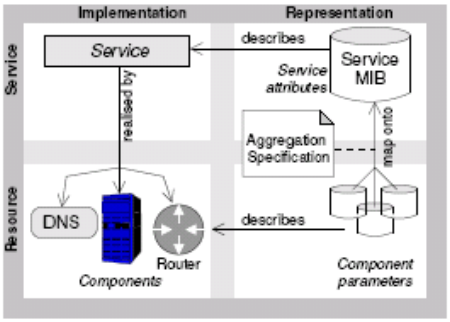

Figure 7 Representation of services as an aggregation of MIB described resources
Extracted from [17]

2.2.2. CIM (Common Information Model)

The Distributed Management Task Force (DMTF) is an industrial standardization organization, whose main work areas are management and integration technologies for enterprise and Internet environments. The Common Information Model (CIM) [23] is one of its most relevant contributions. CIM is an object-oriented model for describing overall management information in a networked enterprise environment. The model is specified as a set of UML models and complementary MOF (Managed Object Format) files, textual files expanding the semantics of the defined elements. The latest version of CIM is 2.22.0, which was released in June 2009.

CIM is a modular, extensible standard; it models a very broad set of elements including databases, networks, user preferences, and applications among others. Internally, CIM is divided into a core model, defining the basic elements, and additional models extending the base elements with additional details on one specific area. Some examples of extensions profiles are application, network, or computer.

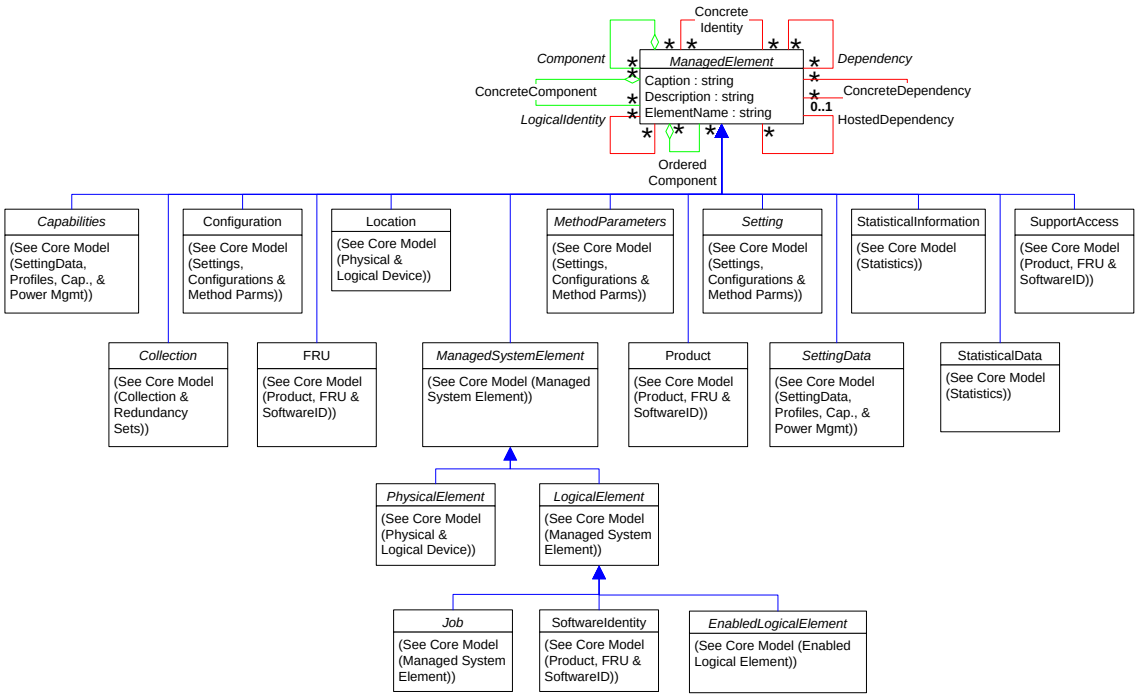

Figure 8 Main Elements of the CIM Core Model

Every CIM element is an extension of the base type *ManagedElement*, defined in the core. From this common base the core model defines an initial set of subclasses, such as *PhysicalElement*, *Configuration*, or *Product*, which serve as abstract definitions for the CIM extensions. Those concepts are refined in the extensions provided by the profiles, which provide a management characterization of elements ranging from J2EE application servers to file system directories and files).

The CIM Application Model extends the base elements with the concepts of the installed software artifacts. It provides a model similar to the concepts handled by software vendors. A Software Product is a unit of software acquisition, with all the provider and product information. Products aggregate a set of Software Features, which are considered the base units perceived by the users of the software. Each feature is provided by a collection of Software Elements, which are the base elements actually installed in the system. The model allows specifying constraints on these artifacts to work properly in a target environment. This extension process is not only applied to the core model, but also profiles can be further refined by additional profile extensions. As an example of that, the application model is further extended by the Application_J2EE model, providing a more concrete example of servers and applications.

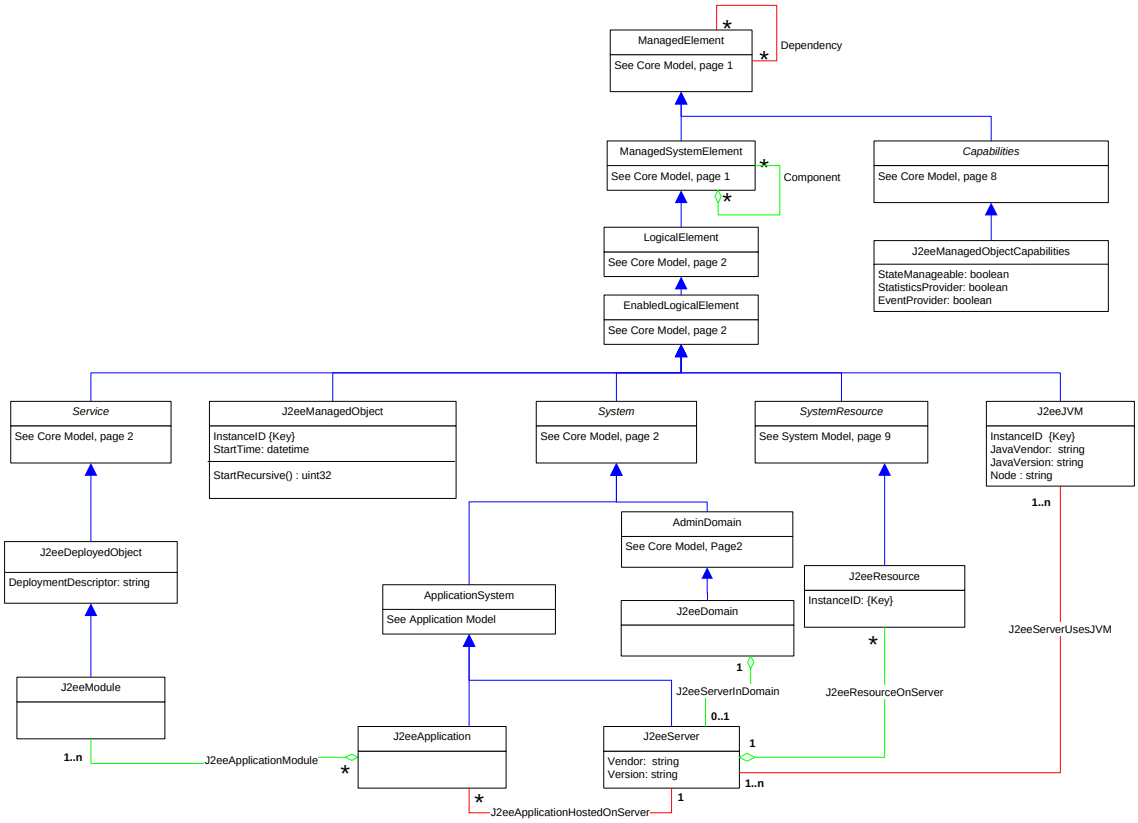

Figure 9 CIM J2EE Application Model

In addition to the environment model defined by the schemas, CIM provides a management interface for the defined elements. CIM Management operations can either be explicitly defined, as methods in the modeled objects, or be implicit operations, covering model instance creation, modification and reading, in a similar way to SNMP operations.

The main strength of CIM is the depth and scope of the specification, as it covers an enormous range of system management elements, at different degrees of details. However, because of the enormous complexity of its scope, it should not be treated as a monolithic standard which has to be completely supported. Instead, depending on the specific requirements of the supporting management systems, selected parts of CIM (including the core and base dependencies) will be implemented, and additional extensions to the model may be defined over those common concepts.

However, the main strength of CIM can also be seen as its main limitation in some fields, as the modeling effort for CIM-based models carries a tremendous overweight, and must be continuously updated with the newly appearing requirements of an enterprise runtime environment.

2.2.2.1. WS-MANAGEMENT

WS-Management is another DMTF standard [24] which was defined in April 2006. It specifies a Web Services based protocol between managed resources and the management infrastructure. Managed resources have an associated resource class, dictating its information model. Although it could be theoretically decoupled, the information model for WS—Management is the CIM object model. The standard also defines the allowed operations over the resources, composed by the following set: get, put, create and delete resource instances, iterate over managed collections, subscribe to notifications from the resources, and execute specific management methods with strongly typed parameters. These operations are similar in nature and degree of abstraction to the SNMP equivalents. In [90] the performance of WS fine grained protocols is compared against traditional SNMP protocol, with the results as it was expected clearly favoring the latter one for fine grained scenarios. Clearly, Web Services are more suited to a SOA model, implying higher-lever, coarser-grained way of management, but they are not the best option for supporting every operation of a management architecture.

2.2.3. Solution Deployment Descriptor (SDD)

The Solution Deployment Descriptor (SDD) [79] is an OASIS Standard for providing a generic information model for software and services deployment activities. The current version of the specification is 1.0, released in August 2008. SDD provides a model for defining the required deployment operations for installing, uninstalling or configuring a set of software elements over a distributed environment. Models are defined in XMLSchema, and stored in two deployment descriptors which must be included in every installation package.

The package descriptor contains information about the identity and contents of the *software package*. A software package is a collection of elements for installing, modifying, or uninstalling a software component. I.e., an “install program” (or uninstall). Among the contents of a package descriptor it is mandatory to include exactly one *DeploymentDescriptor*. *PackageIdentity* informs about the package type - defining the type of activity, from the following: *baseInstall*, *baseuninstall*, *configuration*, *maintenance*, *modification*, *replacement*, *localization*). It also provides general information, such as name, version, and *contentType*. However, unlike other standards such as CIM, no complementary typology is provided.

The Deployment Descriptor defines the topology of the resources related to the solution, the requirements and conditions associated to the operation and the actual deployment activities.

It is always associated to a parent package descriptor. Figure 10 shows the main attributes of this descriptor.

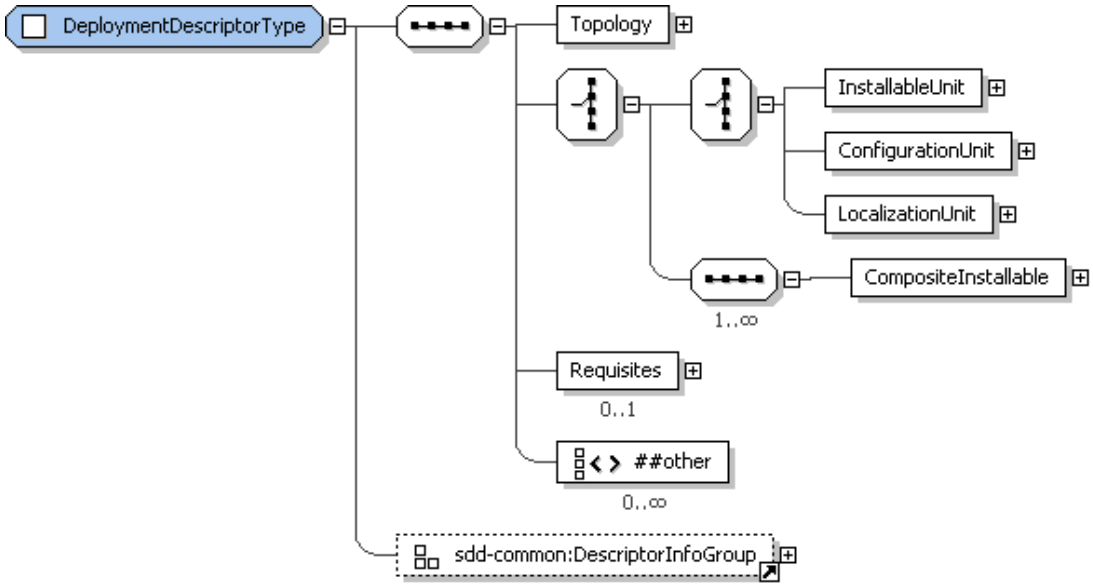

Figure 10 SDD Deployment Descriptor Schema

The Deployment Descriptor contains a set of Atomic Content Elements, which can be of three different kinds: *InstallableUnit*, *ConfigurationUnit*, and *LocalizationUnit*. Each one of them defines a set of artifacts. An artifact defines inputs, outputs, variables and types associated with the artifact files (installation scripts, .sql files, .jar files, zip files…). Each one is processed by a *targetResource*, and will generate a set of *ResultingResource* and *ResultingChange* elements. A descriptor can either contain multiple atomic elements or one *CompositeInstallable,* which aggregates a set of atomic elements into one logical entity.

Deployment operations are executed over an environment model, which is based on the concept of resources (*ResourceType*). Resources have a name, a set of properties and a type (without an attached topology). Additionally, resources can be nested through a host relationship. Some of these elements are also *targetResources*, which process the deployment artifacts. Those elements are frequently containers which host the processed resources. The topology includes descriptions of every resource involved in the deployment. A resource participates if it is required for, created by or modified by the deployment activities. It must be noted, that those resources are “logical”, as they are not actual resources existing in the deployment domain. The mapping from logical to real resources is out of the scope of this specification.

The base specification defines a very generic model, without modeling specific types of resources which can appear at the environment. However, profiles can extend the base standard to provide those specific domain models. As an example, the SDD Starter Profile, although not mandatory, strongly suggests combining SDD with CIM for the domain modeling. SDD complements CIM providing a logical model of software components and a set of defined operations which allow executing deployment operations, whereas CIM application model provides a vendor-centric view of the managed software artifacts.

2.2.4. OMG Deployment & Configuration of Distributed Systems

The OMG Deployment and Configuration Model [83] defines a model for representing deployment and configuration operations over a distributed target. The target environment is called a domain in its terminology, and is described by an object-oriented information model. This model is simple and flexible. The base elements of the domain model are resources, which are named entities classified into one or more types. Resource instances model physical artifacts, mainly: nodes, bridges and links.

On top of that, resources can be parameterized with a collection of properties. Properties have name, value and a kind. The property kind determines the consumption nature of the resource. The combination of types and properties enables resource managers to operate on different environments working with the same base concepts.

This model is very simple and flexible, although it is quite biased to model hardware-related resources such as disk space, physical memory or network interfaces. On top of the environment model, D&C is aligned with the OMG approach, modeling a Platform Independent Model (PIM) of component-based applications, and providing an example Platform Specific Model (PSM) mapping the concepts to a CORBA architecture [45].

The specification also defines the supported management operations, consisting on the installation and configuration of the selected software components. Operations are encapsulated in a deployment plan. It is a deployment-centric specification, so the role of configuration related activities is not covered with the same degree of detail.

2.2.4.1. A model-based services deployment architecture

The work performed by Dr. Ruiz in his PhD Thesis [93] presents a very interesting refinement of those concepts, defining a deployment model which combines the best parts of OMG D&C with the CIM application model. In a similar approach to SDD, logical deployment information, environment modeling and vendor terminology are unified in a single set of models. The logical model proposes additional refinements over D&C, with each deployment unit definition describing which resources will be offered to the environment and which resources will be required from it in order for the unit to correctly work. Also, resources are versioned and dependencies can express a range of compatible versions, enabling advanced modeling of version compatibility. Those refinements over the standard allow developing a deployment system where deployment unit dependency resolution and runtime resource constraints can automatically be analyzed and satisfied. However, part of that advantage is lost for runtime reasoning because of the D&C target model. This model represents every managed element as a resource from the node. However, part of the logical expressiveness is lost as there are no explicit links between the runtime instances of deployment units and their exported resources, not between the resources which satisfy dependencies)

Similarly to D&C this proposal is only focused on deployment, as it becomes evident by analyzing the set of defined management operations, which are described in the deployment plan model. The types of supported changes consist of install, uninstall, stop, start and update of deployment units. Consequently, neither unit nor target configuration are not considered in the proposal, which are required capabilities for a complete management of enterprise services.

2.2.5. WSDM (Web Services Distributed Management)

The Web Services Distributed Management is a set of standards from OASIS devoted to the management of IT distributed systems with the use of Web Services technology. The specification is composed of two parts: MUWS (Management Using Web Services) [115], focused on how to describe and manage resources through Web Services and MOWS (Management Of Web Services) [118], focused on how to represent Web Services as manageable Resources for interoperability. The latest released version is 1.1, made available on August 2006.

MUWS defines a management model supported by the Web Services stack. In this overview we will focus on the domain model instead of the management model. MUWS handles every manageable element of a distributed system as a resource. Manageable resources have a well-defined set of operations, known as capabilities. The specification defines a set of basic capabilities, such as description (which allows to obtain resource's name and version), state, metrics or configuration (through properties configuration). This mechanism is extensible, allowing some resources to expose specific management interfaces in addition to the base mechanisms. In this vein, there is a proposal from IBM [63] for extending the set of basic capabilities provided by base MUWS resources. The proposal defines an additional mandatory capability, called *ResourceType*. This extension forces every Resource to state its type or types, greatly increasing the expressivity of Resources with respect to the management systems. The resource modeling is completely defined by WSRF (Web Services Resource Framework) [39], another OASIS standard for defining resources which take part in WS operations. Resources are identified by a set of properties [40]. Resources can also be aggregated in *ServiceGroups*

The WSDM specifications provide a management architecture for environments modeled with resource-centric standards such as CIM or D&C. MUWS overlaps with the WS-management standard previously described, and shares the same limitations on the scope of applicability because of the heavy nature of Web Services-based management protocols.

2.2.6. Service Modeling Language (SML)

The final selected standard related to information modeling is also the most recent addition to the state of the art. The Service Modeling Language 1.1 [87] is a W3C standard which was released on May 2009. Its aim is to provide a set of constructs which allow the definition of models of complex systems and services. This way, it complements the previously described initiatives, because instead of proposing another information model it ensures that the definition language is expressive enough to capture the actual relationships and constraints that occur at a service based environment. SML can be seen as an extension to XML Schema [30]. However, it is relevant to the context of this work because it has been defined with the context of supporting the requirements of existing information models. For the authors of this standard, the following characteristics describe the importance of information models:

1. Models focus on capturing all invariant aspects of a service/system that must be maintained for the service/system to function properly.
2. Models represent a powerful mechanism for validating changes before applying the changes to a service/system. Also, when changes happen in a running service/system, they

can be validated against the intended state described in the model. The actual service/system and its model together enable a self-healing service/system — the ultimate objective. Models of a service/system must necessarily stay decoupled from the live service/system to create the control loop.

3. Models are units of communication and collaboration between designers, implementers, operators, and users; and can easily be shared, tracked, and revision controlled. This is important because complex services are often built and maintained by a variety of people playing different roles.
4. Models drive modularity, re-use, and standardization. Most real-world complex services and systems are composed of sufficiently complex parts. Re-use and standardization of services/systems and their parts is a key factor in reducing overall production and operation cost and in increasing reliability.
5. Models enable increased automation of management tasks. Automation facilities exposed by the majority of services/systems today could be driven by software — not people — both for reliable initial realization of a service/system as well as for ongoing lifecycle management.

The latest characteristic is especially interesting as it points to a key requirement of service-centric information models: they must contain sufficient relevant information about the domain to enable a more automated approach to the management functions. This way, not only the elements must be characterized by the model, but also the relationships and constraints must be properly specified, in order to enable an automatic validation.

With these objectives in mind, SML contributes two main extensions to the XML Schema. First, it allows defining references between model instances which span over multiple documents, extending the single document support of base schema. This way, the structure of the complete domain can be validated among multiple, complementary model sources. In order to further enrich the expressivity of the models, additional constraints can be defined for the models as rules, which are Boolean expressions that constrain the structure and content of documents in a model. Rules are defined in Schematron[52] and XPath[10].

Because of its recent public release the impact of this specification is still low, so it has not been taken into account for the context of this dissertation. However, by taking into account the relevance of the standards organization and the companies involved in its definition (such as BEA, Microsoft, HP or IBM) it will probably be a very important solution in the future.

2.3. Management Paradigms

Heterogeneous distributed service management brings a new set of complex factors which must be addressed by new initiatives, originating numerous contributions from the scientific community. In [112] a classification of the different approaches to software and services deployment is provided, which can be used as a reference. This taxonomy distinguishes between manual, script-based, language-based, and model-based paradigms. I will structure this section following that classification, and extending it with behavior-based approaches: policy and ontology-based management and the autonomic computing paradigm. However,

these categories are not perfect partitions of a set; there is overlapping to a certain extent among the selected categories.

2.3.1. Model-based Management, Language-based management

The most common approach for dealing with the heterogeneity of enterprise IT systems is the adoption a model-based management approach. Model-based service management consists of defining and externalizing software, services and environment information in a separate model. The model provides an abstraction layer between the managed infrastructure and the management architecture. This way, similar concepts are grouped and handled the same way, and relationships between configuration elements can be tracked and automatically processed.

Depending on the specific proposal, the extent of the modeling scope usually differs. We can consider modeling the environment information, the logical, developer-centric view of the software applications and services being managed, as well as management operations. In most cases model-based management is based on one of the presented standards for environment modeling.

A similar approach is language-based management [112], where the models' expressivity is enhanced with set of defined operations, obtaining Domain Specific Languages (DSL) [33] for management, sort of executable models.

The following sub sections will detail some of the most relevant initiatives in this field. However, the variability between different approaches is significant, as this category covers the ground between traditional scripts, models, and DSL management languages.

2.3.1.1. Model-Driven transformation based configuration and deployment

The authors at [29] propose a complete model-driven, transformation-based architecture for service deployment and configuration. Service configuration processes are very complex because in order to prepare a change plan it is necessary to analyze the targeted environment, the characteristics of the specific services, the current system state, the requirements and SLA defined for the services, and so on. The proposed architecture maximizes the idea of separation of concerns, defining different models which will be handled by different stakeholders of the discipline of service development, provisioning and management. A set of transformation rules enable the aggregation of the dispersed knowledge into a concrete change plan.

We can see in the next figure a logical view of the proposed architecture. The configuration and deployment process is a workflow, where different actors participate and the architecture provides a framework for automatic transformation between the different models.

First, the developers define the logical application structure (LAS). This model defines components, modules and services dependencies. The deployer enriches this model with packaging and software container information, as well as strategies for distributing the deployment (such as spreading the services over the containers, or concentrating them in the same node) obtaining a logical deployment model. Business managers also take part in the process by contributing a model with the high-level business requirements. The deployment topology generator takes these abstract models, consults with a repository of defined transformations, containing management knowledge and best practices, and the information

from the target environment, stored in the data center model. The result of this search is a physical deployment topology, where the logical elements have been assigned to the physical resources existing in the environment. The final product is an executable deployment and configuration plan, which is handled to the provisioning system.

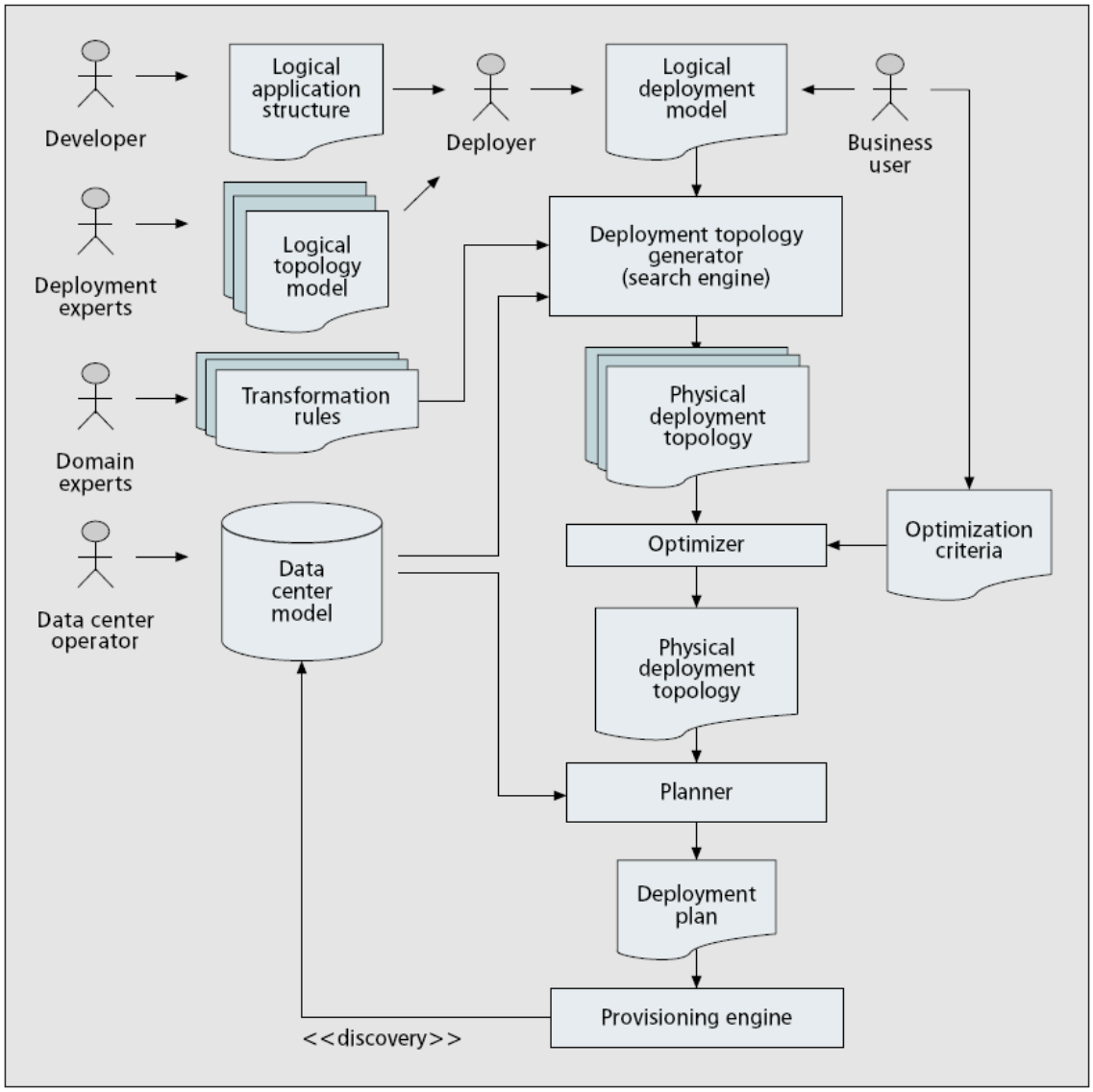

Figure 11 Model transformation deployment architecture

The proposed deployment topology generator is the center piece of the architecture, which can be considered an inference engine, similar to the ones presented in the sections about Policy-Based Management and Ontology-Based Management, described in the following sections.

The most interesting contribution of this approach is the management of complexity not only by modeling abstraction but also through problem partitioning into different model artifacts. This approach also has the added benefit of converting human knowledge into models which can be reused in different contexts, improving the process automation. However, it is not clear how scalable this approach is, neither how do the different models seamlessly convert into one or another. Also, the central element automatically applies multiple model transformations potentially coming from different sources, which could be not consistent

between themselves. Finally, the proposed architecture does not build on any well-established information model, greatly complicating its adaptation to different scenarios.

2.3.1.2. Model Driven Provisioning

Usually in an enterprise environment deployment and configuration plans are defined and stored as workflows, which are manually defined. For improved reutilization, commercial management product suites such as Tivoli Provisioning Manager help in its definition, storage and maintenance. However, in this scenario each workflow is equivalent to an executable script; specific to a set of applications and services as well as its environment, and costly to maintain, with an unclear level of reusability. With that motivation, in [72] the authors propose a provisioning architecture for generating on-the-fly configuration plans.

The proposed architecture is model-based, using the Planning Domain Definition Language (PDDL) [36] as the language for defining the provisioning operations, as well as the object model (which is converted to a collection of logic statements, detailing type, inheritance, and property information).

Dynamic plan generation consists of the following three steps. First, the current state of the environment and the desired deployment are modeled. Then, the configuration operations are defined, stating their preconditions and effects. With all that information, a partial order planner (POP) is invoked, obtaining the sequence of operations required to reach the desired state. This planner algorithm has been optimized for the characteristics of distributed application deployment, with modifications such as prioritizing the resolution of open dependencies, or deferred variable instantiation.

This approach can automatically reason and obtain a plan for the provided problem. However, it is not clear how applicable would be to complex, heterogeneous enterprise environment. Knowing with a reasonable degree of confidence the required time for each activity can be an unfeasible requirement, which can render the planning solution proposed here unfeasible for the targeted environments.

2.3.1.3. Smartfrog

The SmartFrog language [37] is a prototype-based language for distributed systems configuration. Components are the base language elements, and are described by a set of properties. Components can have two types of relationships: containment and inheritance. Inheritance provides a way to address the lack of typing in the components and properties. The most powerful feature of the model is the capacity to define values by reference, which may be resolved at definition time or lazily, in the runtime environment, enabling dynamic configuration of some attributes. Finally, the model also defines some basic operations (i.e. concat for creating URLs).

A system is defined as a collection of applications running over a distributed collection of computing resources. Applications are a collection of components, defined statically with a descriptor or generated dynamically at run-time. Finally, components are Java objects that implement a specific API, which binds them to the SmartFrog component model lifecycle. Components may create and manage other objects, including processes and programs written in other languages. This characteristic of Smartfrog makes the framework tightly coupled with the desired solutions, as it is necessary to define a description model, as well as a Smartfrog

Java component for each managed element. Smartfrog defines a distributed deployment infrastructure, where the infrastructure elements read component descriptions, instantiate them, and manage the lifecycle of the created applications and components.

In [81] two extensions to Smartfrog are described, which extend the runtime management functionality of the deployment infrastructure. Anubis extends the automatic discovery mechanisms of the deployment infrastructure (adapting accurately the runtime information), and continuously monitors the system for failures in the agents. Woodfrog is a runtime configuration snapshot extension. Woodfrog records several states of the system and allows reverting back to stable configurations. The combination of these extensions with the base functionality enables infusing autonomic behavior to Smartfrog-managed systems.

Smartfrog presents some very interesting ideas, specially related to the expressiveness of configuration properties and dependencies, which are also built from a simple base. However, its high level of intrusiveness, as well as the considerable effort for applying it to a very heterogeneous environment hampers its applicability.

2.3.1.4. Mulini, XACCT

The Mulini transformation tool chain [111] tries to automatically deploy components and services, as well as application-specific instrumentation, taking as only parameters design models. This way, the only input parameters for the tool chain are design-level component specifications, generated by Cauldron (in CIM MOF format, with Gantt-style workflow activity dependencies), as well as test service parameters and SLA. The generator produces Smartfrog descriptors and Java code for the core components and the required instrumentation agents.

The code generator component of Mulini is the ACCT (Automated Code Composable Translator) [96]. It applies an XSLT transformation to component dependencies and additional metadata in order to obtain an internal XML workflow model, XACCT. In this model the activity dependencies are expressed, with event sending between the different activity deployers. In the final stage, Java scripts and XSLT transforms convert this independent model into executable Smartfrog deployment scripts and component definitions.

Mulini shows an interesting approach to addressing the high entrance of barrier of SmartFrog, the integration of Model-Driven Engineering approaches, for automatically generating SmartFrog specific configuration and code. However, for highly heterogeneous systems, this approach can even be more costly than the original, as the actual cost of defining all those transformations can be greater than the initial effort of those constructs.

2.3.1.5. CHAMPS

The CHAMPS System [56] is a prototype under development at IBM Research for CHAnge Management with Planning and Scheduling, providing an execution platform for IT Change Management activities (one of the disciplines of the ITIL best practices). CHAMPS creates automatically executable plan workflows applying the modifications specified in a Request For Change (RFC) Document.

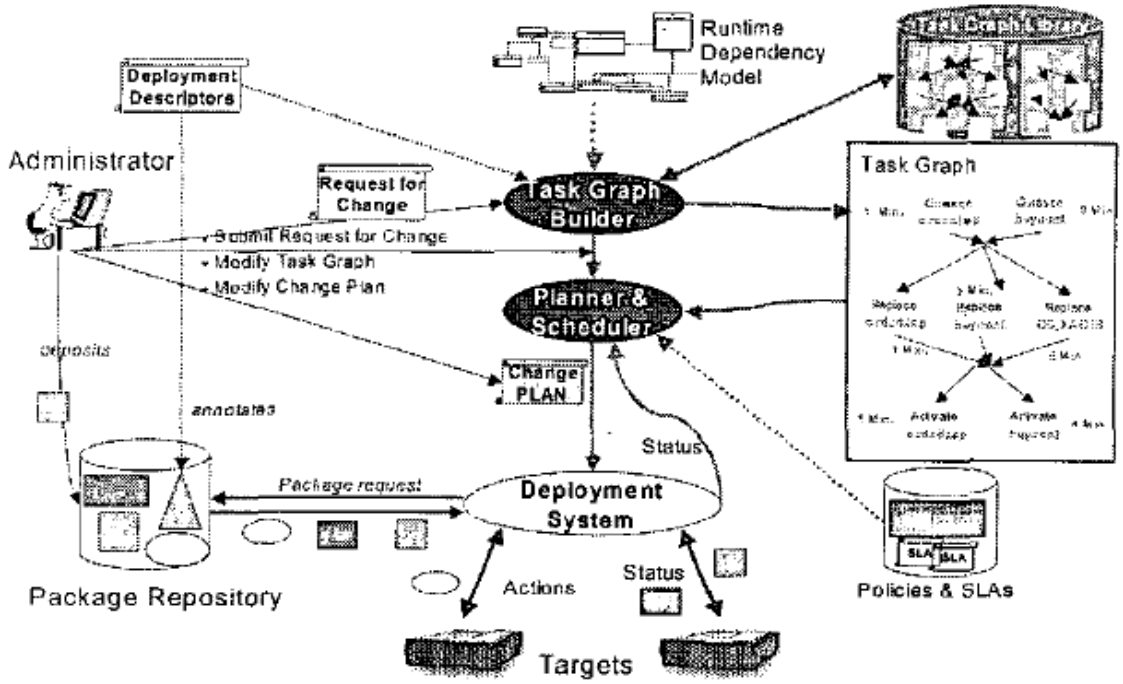

Figure 12 CHAMPS architecture (Extracted from [56])

CHAMPS consists of two main elements: The Task Graph Builder (TGB) and the Planner & Scheduler (P&S). TGB evaluates temporal and location dependencies between the elements involved in the change operation, using as inputs the specified RFC document, the deployment descriptors, expressed as SDD elements, and a runtime dependency model of the system which contains remote dependencies among elements. With all that information, the system produces a Task Graph, which is an abstract workflow defining tasks, their dependencies, and time estimations for each one of them. This model does not contain specific information about the runtime.

The P&S converts the Task Graph into a Change Plan, taking into account the defined policies & SLAs. The Change Plan is another workflow, defined in WS-BPEL, with tasks assigned to specific runtime elements, and deadline estimations. This executable tries to maximize parallel execution of activities [57], thanks to the use of Constraint Satisfaction Problem techniques in the logic of the P&S. As a consequence of that, workflows have enhanced temporal constraints expressivity, adopting the available temporal constraints of GANTT diagrams. This way, tasks can depend with the four following expressions (Finish-To-Start, FS: task A must finish before task B begins, Start-to-Start, SS: task B cannot start until A does. Finish-to-Finish, FF: task B cannot finish before task A does. Start-to-Finish: task B cannot finish until task A starts).

The researchers have compared the benefits of the automation provided by CHAMPS with a manual approach [55], taking as reference the installation and configuration of the SPECjAppServer, a multi-tier J2EE application which serves as benchmark for the different application server vendors. The metrics obtained show a significant reduction in the human intervention, and about 33 percent decrease in required installation time.

2.3.1.6. RollbackITCM

One of the most frequent assumptions for configuration plans is the assured correct execution of each activity. However, this assumption makes failures in the configuration operations leave

the system in an unstable state. The paper [71] proposes a general architecture for dealing with failures over the execution of ITIL change management process.

Conceptually, rollback execution is obtained by applying the concepts of atomicity and transactions to the execution of configuration change plans. In order to do so, it builds on the BPEL process definition engine base mechanisms, marking initially some activities as atomic, and at a later stage, processing the initial plan in order to generate a rollback one.

This proposal clearly describes how to cope with failures in the execution of activities, although it still imposes a very strong requirement on the actual operations: each atomic operation must have a reverse one.

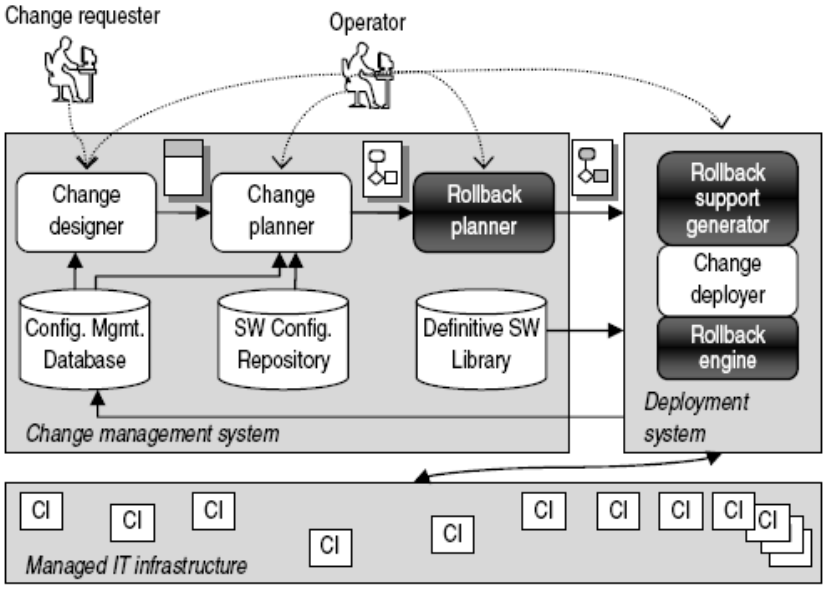

Figure 13 Transactional change management architecture

2.3.2. Policy-based Management

In the field of network management, policies are one of the most used techniques to cope with the complexity of managing a distributed system [5]. As the policy concept is somewhat ambiguous, we will use for this chapter the definition found at [104]: “policies are rules governing the choices in behavior of a system”. According to this definition, there are several types of policies used in network management: Security authorization policies, routing policies, behavioral policies and management policies. In service configuration the latest group is the most relevant. Management policies modify the performance of the system. Some examples of that are strategies for resource allocation, – ensuring a Service Level Agreement, as well as defining reactions over unexpected events in the environment – configuring a load balancer to cope with a change in the received server traffic. Defined policies have different levels of granularity, ranging to specific network configuration parameters to business-level objectives [116].

Before understanding policy-based management systems, first how policies are defined must be explained. There are multiple ways to express policies, although typically management policies are defined in the form of event-triggered condition–action (ECA) rules. Each rule is executed when the trigger occurs, it checks whether the condition is valid, and if true invokes

the defined action. There is a generalization of this model, called Event Trigger Response [5] which separates the three base elements, allowing rule definition by composition of the separate parts.

The IETF and DMTF have jointly defined a policy model and architecture [80]. This model defines policies are simple condition rules, in the form of if <condition(s)> then <action(s)>. The trigger part of the ECA paradigm is left open for the policy managers. The model is object-oriented, extending CIM core classes, and can be serialized to both XML format and LDAP-like tabular format.

We can see in the next figure a typical architecture of a policy-based system. A policy management tool allows creating, modifying and deploying policies, storing them in a policy repository (PR). Policy Decision Points (PDP) communicate with the repository, interpret the policies and transmit the results to the Policy Enforcement Points (PEP). Finally, PEPs apply the policies to the system. The roles of PEP and PDP are often taken by the same device. An event monitoring system sends monitored events from the environment to the PDPs to trigger the execution of relevant policies.

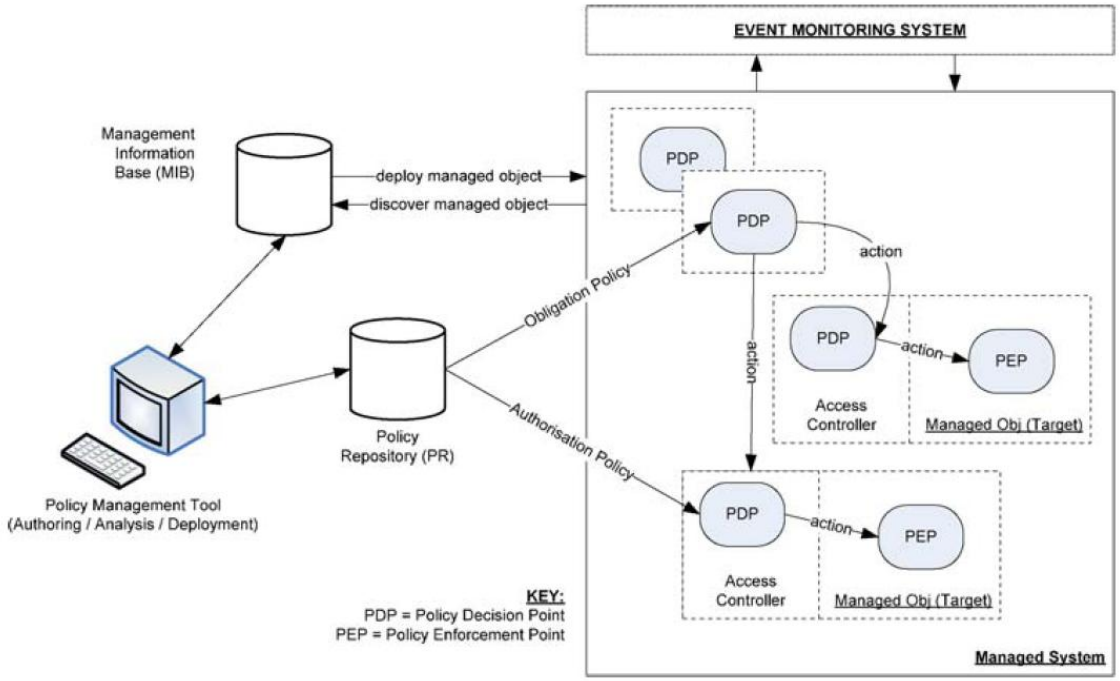

Figure 14 Policy framework architecture. Extracted from [5]

PBM can be applied to the configuration of distributed systems for contributing high-level requirements to the management loop, such as SLA, or load balancing policies. The behavior of a system can be partially determined by the specific policies, hiding the managers from the complexity of the system.

2.3.2.1. PlanIT

The PlanIT system [4] is an automatic configuration change planner for distributed systems. It supports both initial provisioning planning as well as dynamic reconfiguration planning. PlanIT uses an environment model derived from ADL. The model defines components, connectors and machines (places where components are deployed). Every model component has a state. PlanIT uses PDDL for defining these environments as well as the plan model.

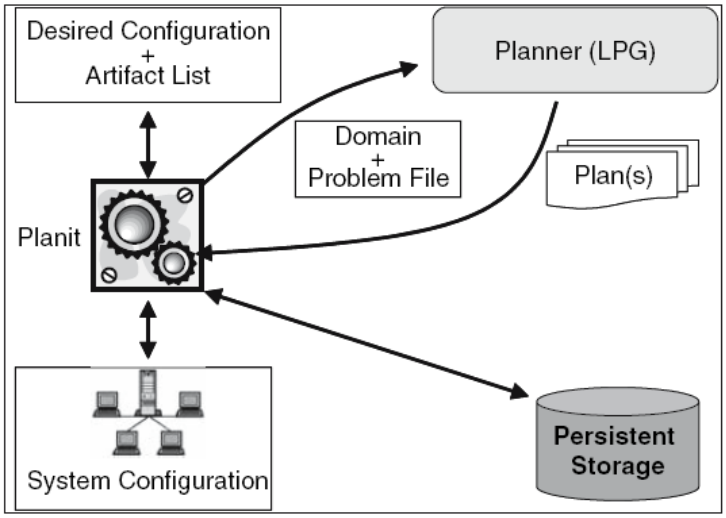

Figure 15 Planit Architecture. Extracted from [4]

In order to construct a configuration plan the following input is required: the domain elements, the initial state, the goal state, and basic utility functions. Those elements contain not only the environment topology information but also additional component properties such as operation metrics or SLA requirements. When the collected information is sufficient to generate a plan PlanIT calls LPG (a PDDL planner) and iteratively obtains valid deployment plans. When the timeout finishes, PlanIT chooses the best plan among the obtained ones. The generated plan is a sequence of steps, scheduled in specific time intervals, and with an estimated duration for each step. There will be a different sequence of operations for each environment machine.

The combination of PDDL and a planner builds on the same foundation of policy-based management, applying well-known techniques from the intelligent systems domain into automatic operations inference from the stated problem. The main limitation of this approach is the use of a proprietary environment model described in PDDL The required effort for describing with sufficient detail a heterogeneous enterprise system severely limits its applicability, as well as the already mentioned difficulties of establishing exact timing for the execution of the operations. Additionally, the authors provide an analysis of PlanIT-LPG performance for generating plans for increasingly large environments, and it shows the scalability limitations of an automatic planner solution.

2.3.3. Ontology-based management

Previous sections have described the advantages models and policies bring to enterprise service management. Recently, some of these initiatives have evolved to the use of ontologies, in an attempt to better characterize the managed environment and provide a natural way of automating system management. The concept of ontologies originated from philosophy, and it was adopted later by Artificial Intelligence practitioners as a solution for the lack of interoperability between facts from different sources. Recently the advent of the Semantic Web has renewed the interest of applying ontologies for solving problems. Strassner provides in [107] the following description for ontologies:

An ontology is a formal, explicit specification of a shared, machine-readable vocabulary and meanings, in the form of various entities and relationships between them, to describe

knowledge about the contents of one or more related subject domains throughout the life cycle of its existence. These entities and relationships are used to represent knowledge in the set of related subject domains. Formal refers to the fact that the ontology should be representable in a formal grammar. Explicit means that the entities and relationships used, and the constraints on their use, are precisely and unambiguously defined in a declarative language suitable for knowledge representation. Shared means that all users of an ontology will represent a concept using the same or equivalent set of entities and relationships. Subject domain refers to the content of the universe of discourse being represented by the ontology.

An ontology commitment represents a selection of the best mapping between the terms in an ontology and their meanings. Hence, ontologies can be combined and/or related to each other by defining a set of mappings that define precisely and unambiguously how one node in one ontology is related to another node in another ontology.

As ontologies are bound to a subject domain, if the definition is narrowed to the domain of network and systems management it becomes the following:

An ontology for network and system administration is a particular type of ontology whose subject domain is constrained to the administration of networks and systems. Administration is defined as the set of management functions required to create, set up, monitor, adjust, tear down, and keep the network or system operational. One or more ontologies must be defined for each device in the network or system that has a different programming model.

The definition clearly shows how ontologies can help in the management of enterprise systems. Ontologies can in principle provide a better modeling of the managed environment, as well as the management operations. Ontologies can also enable interoperability and a reference frame for management devices from different vendors and protocols.

Many of the advantages ontologies provide have already been provided by environment and management operations models. However, ontologies' expressivity goes beyond standard object-oriented models, resulting in these additional capabilities:

- Ontologies allow greater expressiveness in object relationships compared to standard modeling such as UML (dependencies, associations, aggregations, and inheritance). With ontologies it is possible to express relationships such as time (past, future), synonyms and antonyms.
- As it is the case with expert systems, in an ontology-based system the initial knowledge can be automatically extended through reasoning with the available facts.
- Ontologies provide the necessary means to identify the same concepts expressed in different formats (e.g, two equivalent management interfaces from servers from different vendors).

There are several specific languages for defining ontologies, such as Semantic Web's OWL [19]. However, models and ontologies are not completely separated domains. It is possible to enrich some well-established models so that they comply with the requirements to become ontologies. In [67] the CIM metamodel is extended using the Object Constraint Language (OCL) [117] to contribute the strictness required for an ontology definition.

However, the adoption of ontologies to heterogeneous domains has been slower than expected, as they demand a considerable greater modeling effort, requiring behavioral

characterization. On top of that, the automatic knowledge inference capabilities of ontologies share the scalability limitations traditionally presented by expert systems. Another alternative is to embed the additional knowledge in the supporting architecture, instead of inside the ontology. Because of these factors the validity of ontologies to the enterprise service domain must be further tested.

2.3.3.1. Semantic Management Information

The work from Dr. López de Vergara [69] shows how the different management information models and operations being adopted in network management build upon the same foundations, but can't efficiently interoperate. Even products from different vendors complying with the same standard do not provide the same set of instructions.

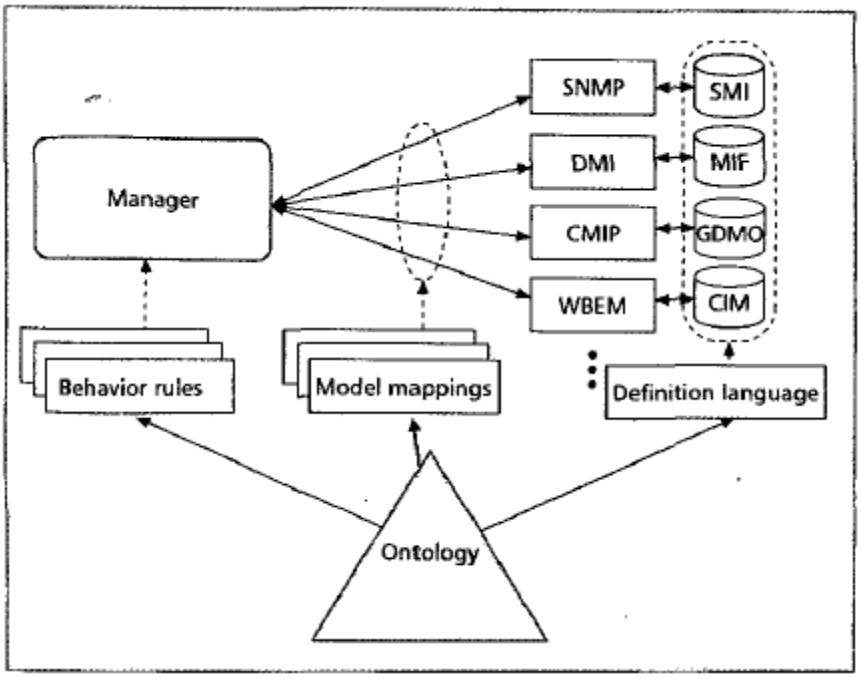

Figure 16 Semantic operations network management

In order to alleviate this interoperability problem, this work proposes the adoption of ontologies. This way, each different management protocols is defined as an ontology. On top of that, mappings between different ontologies are defined, so that all of them form part of the same knowledge base. This way, an ontology-based manager can work with the aggregated set of knowledge from all the systems managed by heterogeneous protocols. As it can be seen in the picture, the proposed ontologies allow a mapping between the main network management information models.

2.3.3.2. Semantic mapping for service-level deployment

The work performed by Frenken et al. [34] also shares a similar approach. This proposal describes an architecture for service deployment over distributed embedded systems, with pluggable strategies for defining the runtime mapping of the services. Although the characteristics of the target hardware elements (embedded devices) differs from a typical enterprise infrastructure (servers with large computing power), the proposal is based on the assumption that the Web Services model will extend to those environments, making them

comparable to our target environment. This assumption seems reasonable as specifications such as UPnP [114] or the DPWS [76] are experiencing a growing popularity in this domain.

The proposed management architecture is built over a simple control loop. A monitoring infrastructure obtains a snapshot of the system state. The user wants to apply an operation (service install or update), and selects a mapping strategy for deciding the distribution of the components over the environment. Finally, the obtained mapping is applied to the environment by the injector component. The main contribution of the paper is the description of several strategies for obtaining this mapping. For identifying the candidate nodes for a deployment, two strategies are proposed: 'Property Expressions', consists on resource matching and consumption from the target service to the available resources at the runtime, with the information expressed in simple properties format. A refined strategy is called 'Semantic Linking', where the device and service information are expressed as OWL ontologies. After the candidate nodes have been identified, a second type of strategy is applied for selecting the nodes for each service. In this category the described methods are a Best Fit Decreasing variation (taken from the bin packing problem), Monte-Carlo based approaches, and complete iterations of the search space. This way, a plan can be found using different information models and reasoning mechanisms, depending on the problem complexity and the specific environment. On the modeling approach, the authors highlight that ontologies clearly have greater expressivity than simple properties, although they require a much greater effort for modeling every device and service. This way, depending on the specific problem, one or other approach might be the better option. However, up to this point they are not following any of the previously mentioned standards for defining neither the models nor the ontologies, so no conclusions can be extracted in this topic from the paper.

2.3.4. Autonomic Management

The objective of the autonomic management paradigm is to lessen the burden on system administrators by using infrastructure to manage infrastructure [58]. A self-managing system can greatly reduce its operation cost and perform automatically some well-known management processes. The potential advantages which can be obtained by this paradigm are usually expressed in the form of four distinct self-management capabilities:

- **Self-Configuration**: The components of an autonomic system must install and configure automatically, without needing human intervention.
- **Self-Healing**: The system diagnoses itself continuously, detecting functional failures. After detecting a fault, the system reconfigures itself in order to correct it.
- **Self-Optimization**: The system monitors its resources with respect to defined requirements and policies. System parameters are continuously configured in order to improve its service-level quality.
- **Self-Protection**: The system can identify proactively intruders and defect from their attacks.

Next figure shows a multi-level view of a complete autonomic system. The low-level elements are the managed resources, instrumented through a manageability endpoint, or touchpoint in this terminology. This interface is the entry point to the resource, exposing a sensor channel and an actuator channel. On top of that infrastructure, autonomic managers implement an intelligent control loop for automating some of the management aspects of the resource. For

environment-wide operations managers can be orchestrated, and even managed manually in some critical cases [11].

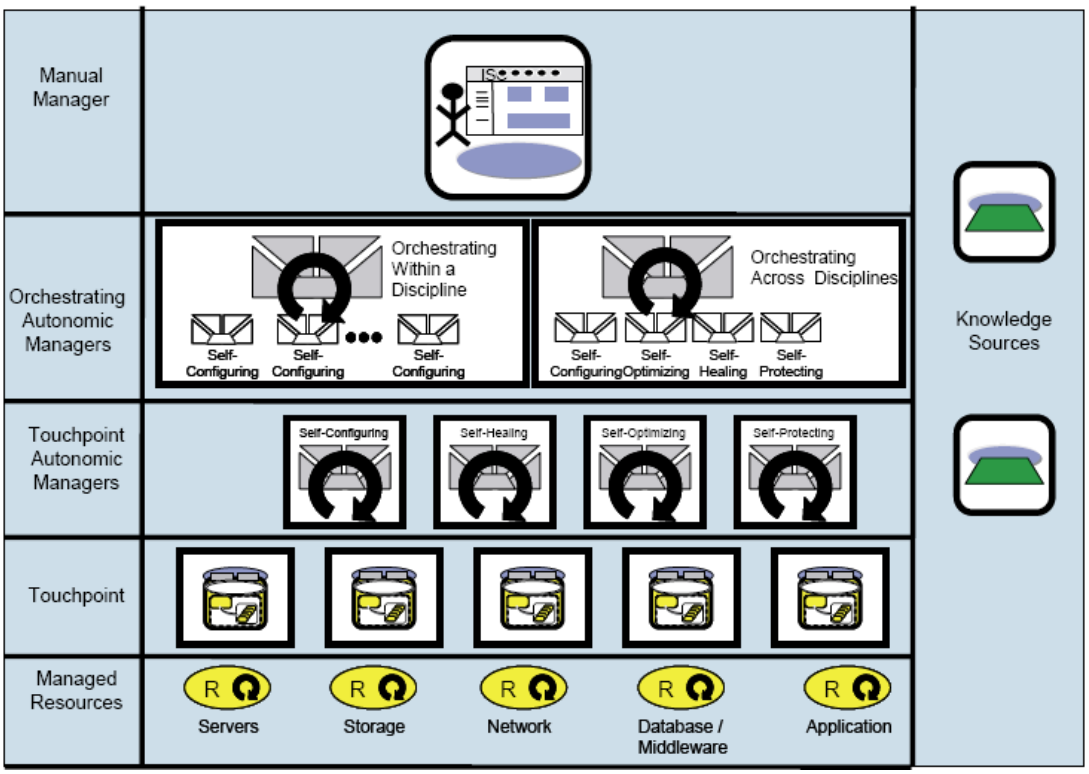

Figure 17 Architecture of an autonomic system.
Extracted from [11]

The behavior of an autonomic manager is determined by a set of high-level requirements. One of the most common approaches for implementing an autonomic manager is the use of policies. Rules enable reasoning and deciding the required corrections to be applied through the actuator channel, and provide a simple mechanism for modifying the behavior of the autonomic manager.

We can see that the autonomic management paradigm provides a complementary approach to the previously defined initiatives. The problem addressed by autonomic management is the uncontrolled complexity of traditional management approaches, as the number of management applications and services keeps growing. Because of that, in the following examples we will see autonomic management initiatives are combined with the other presented paradigms for a more complete solution.

2.3.4.1. PMAC

Researchers from the IBM Watson Research Center have developed PMAC (Policy Management for Autonomic Computing) [2], an autonomic manager (AM) whose behavior is controlled by policies. PMAC architecture consists of a policy definition tool (PDT) for creating policies, the policy editor storage (PES), for providing policy deployment and persistence, the AM and the managed resource sensors and actuators (MR libraries) for policy enforcement. The AM manages one or more MR through the deployed sensors and actuators.

PMAC Policies are defined in the ACPL (Autonomic Computing Policy Language), an XML-based rule definition language. ACPL rules are composed by four components: condition, action, priority, and role. The addition of a role allows scoping the policy to a subset of the resources controlled by the autonomic manager, improving the scalability of the autonomic manager. ACPL expressivity if enhanced by ACEL (Autonomic Computing Expression Language)[1], an XML-based language for expressing conditions. PMAC also accepts externally defined policies in CIM-SPL [3].

The AM provided by PMAC builds upon a policy ratification engine. This component uses specialized strategies for improving the efficiency of policy resolution and improving its automation capabilities. These techniques are: dominance check – allowing to detect whether a new policy doesn't affect at all the behavior of the system; conflict check – detecting conflicting conditions in policies; coverage check – measuring the variety of events addressed by the defined policies; conflict resolution – assigning priorities to different policies.

PMAC is a prime example of the synergies between policy-based management and autonomic computing. It also proposes some valuable approaches to improve the performance of automatic policy triggering and resolution, which are frequently limited to its application in complex system because of its scalability limitations.

2.3.4.2. FOCALE

A similar approach is adopted by FOCALE (Foundation Observation Comparison Action Learn eReason) [53], an autonomic network manager implemented with ontology-based policies. FOCALE is designed to deal with the main current challenges in network management: managing heterogeneous functionality, adapting to changes in user requirements and the environment, and integrating learning and reasoning techniques to network management.

FOCALE reasons over an information model based in DEN-ng (Directory Enhanced Networks - Next Generation) [108], an object-oriented information and policy model for telecommunications. This model is complemented with Finite State Machines for modeling behavior and ontologies for embodying semantic information.

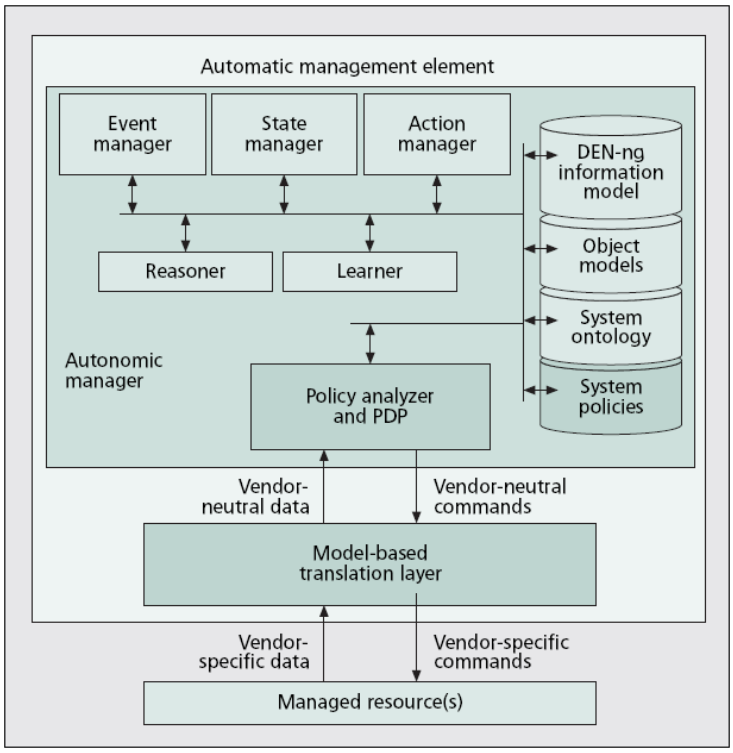

Figure 18 FOCALE Architecture

FOCALE architecture adds a Model-Based Translation Layer (MBTF) between the AM and the MRs. This layer is in charge of translating vendor-neutral commands to resource-specific operations as well as converting resource-specific data to the internal model handled by FOCALE, isolating the AM from all the vendor-specific details. The implemented prototype defines a configuration DSL using XText from the OpenArchitectureWare Eclipse project [28]. Models are later converted to OWL with EODM from EMF.

The AM also implements two control loops: a maintenance control loop, used when the system works normally, and an adjustment control loop, used when there are changes in the defined policies. Control loops are driven by ECA policies, which are executed with Drools inference engine.

FOCALE architecture is a good example on how to implement an autonomic manager over a potentially heterogeneous environment, as well as another example of the use of policies to implement an autonomic manager.

2.3.4.3. Autonomic J2EE Server

The edge computing paradigm aims at distributing the computing resources away from centralized servers, to physical nodes closer to the service clients. This approach is currently being applied to Content Delivery Networks (CDN), but some authors are applying it for managing autonomically an edge-server like infrastructure of J2EE servers [20][21].

Their implementation uses an Architecture Description Language (ADL) to model the software products and services available at a J2EE server (including web applications, enterprise java beans, servlets, and databases). The model allows managing dependencies and composition. The system follows a standard AM architecture. The sensor channel receives application and service-level metrics, which feed the AM. Internally, the data is processed by a Drools reasoning engine. ECA policies can trigger with the collected data, potentially invoking

actuations on the MR such as deploying a new instance of the application in a node nearby to absorb an increase in the number of requests.

This research line highlights the convenience of evaluating policy-based autonomic management for the domain of enterprise systems and service management.

2.3.4.4. DACAR

Another interesting initiative in the field of autonomic management is DACAR (Distributed Autonomous Component-based Architecture) [25]. The main objective of this autonomic manager is dealing with the heterogeneity and dynamics of managed environments.

DACAR models managed resources with the OMG D&C model. On top of that, policies defined in ECA (Event-Condition-Action) style are deployed. DACAR implements two control loops: one for endogenous events (coming from the knowledge base, such as a new policy defined) and another one for exogenous events (coming from the environment, such as a new node appearing in the system). Because of the selected domain model, DACAR has been successfully applied to CORBA systems, which are the de-facto PSM defined for D&C. However, that fact complicates the adoption of this approach into environments with a more heterogeneous infrastructure.

2.3.4.5. Autonomic OSGI Gateways

Another example of the combination between ontologies and autonomic computing is the research work of J.L de Vergara et al. [69][38]. The objective of this work is to develop a semantic, autonomic manager for an OSGi home gateway, instead of the traditional management model based on the centralized role of the Control Center [26]. In this context, the home gateway acts as the bridge between the operator infrastructure and the residential environment. Control of home network devices and services is carried out through that interface.

In this context, the contribution of the authors is to install an autonomic manager in the home gateway. The manager will autonomically find and configure the services for each user, instead of depending from the centralized management infrastructure. The autonomic manager is based on a closed control loop, which is provisioned with OWL ontologies defining the service model and dependencies, the component model, and the user preferences. A set of defined rules in SWRL [47] are applied by the autonomic manager for finding suitable services and configuring them based on the user profile.

This line of work further reinforces the combination of autonomic management and ontologies. However, in this case the domain presents some different characteristics; there is a specific component and service model, inside a controlled domain (the digital home) with a limited set of well-defined services. However, taking into account the important effort required to define all the participating ontologies, the feasibility of this approach for supporting highly heterogeneous and complex enterprise domain would be very costly.

2.3.4.6. Automatic configuration service of distributed component systems

The paper by Kon et al. [62] provides an overview of the main conclusions of a six year work in the context of automatic configuration. Although the specific word ‘autonomic’ is not used for describing the work, the contributions of the work can be classified under its paradigm, as the

intent is the complete automation of the configuration operations, by taking into account the characteristics of the managed components and the available runtime resources.

In order to support these objectives, the work proposes an architecture framework built over CORBA services, which is formed by three main functions: a mechanism for dependency management and representation, a hardware resource management service, and an automatic configuration service dynamically instantiates components using the other two. A general view of the main architecture components is depicted in Figure 19.

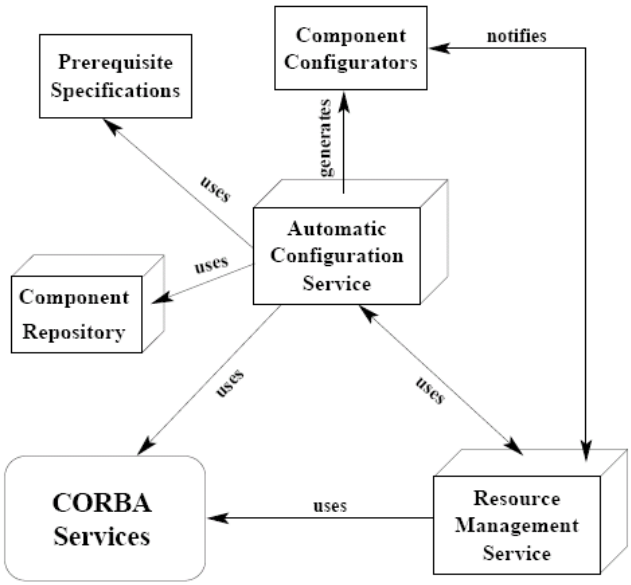

Figure 19 Architectural Framework for Dynamic Automatic Configuration
Extracted from [62]

The architecture reasons over components, ensuring that the logical and runtime dependencies are satisfied whenever they are instantiated. These requirements are defined in component descriptors with the XML based Simple Prerequisite Description Format (SPDF). In order for the components to be supported by this architecture, developers have to contribute these descriptors. However, it is unclear how hardware requirements can be properly defined without a common ontology for characterizing the required hardware elements, or at least a shared taxonomy for referring to them with the same names.

The resource management service instruments the monitored elements, providing an updated model of the available resources, and reasons about the available resources in order to allocate the desired component instantiations. Hierarchical structure with local agents (Local Resource Managers) and a control point, the Global Resource Manager. The Automatic Configuration Service involves both elements in order to deploy new components to the system or perform dynamic reconfiguration for the components in case some changes occur to the environment.

Unfortunately, the scope of the work makes it unfeasible to be directly translated to the enterprise heterogeneous environments, as it lacks a supporting modeling base, and the prototype tests are built over a CORBA / 2K operating system infrastructure. However, the reasoning approach of this architecture, can serve as reference for addressing some of the requirements of the domain of work.

2.4. Conclusion

In this section I have analyzed the most relevant initiatives in the field of network and services management. The proposals have been evaluated for their suitability to the domain of enterprise services. In order to provide the initial context, some basic service management concepts have been defined, and complemented with an overview of the most relevant business-level service management processes, ITIL and eTOM. Those specifications define a conceptual framework of the organizational for the technical problem discussed in the rest of the analysis.

The analyzed standards aggregate the experience gathered from different sectors over the years (mainly network management and IT administration). However, none of them provides a complete model for enterprise service management, including the characterization of the environment resources and the possible operations. Standards are created and evolve for different domains, but they still haven't catch up to the requirements of a service-based environment. In order to efficiently manage complex heterogeneous systems, one mandatory requirement is the availability of a uniform representation of every service-relevant configuration element. This model should allow describing distributed systems, runtime services, their configuration and the available operations over the environment.

This chapter also presents an analysis of numerous research initiatives for distributed service management. By looking at the general picture the first identified commonality is that every proposal is built over a representation model, which can be either the already mentioned standards or self-developed models. Usually these models are designed to cover the new requirements of services. Also, they usually have greater expressivity and can solve the identified problems. However, without the modeling work provided by the existing standards, many of these proposals can't be efficiently applied to heterogeneous systems. Moreover, complex models imply a considerable modeling effort which impacts the advantages provided by these solutions. The most promising initiatives provide transformation mechanisms to combine the knowledge provided by existing standards with additional expressivity and adapted scope to solve their needs.

By looking at the functional proposes, it is clear that the main concern is the automation of the management operations. Multiple examples have been presented which show how approaches such as autonomic computing or policy-based management can partially automate some of the service management processes. However, no analyzed proposal provides a complete automated solution to the problem. Some of them address only part of the services lifecycle (e.g, software deployment [34][93], or network and device configuration management [2][53]). On the other hand, several contributions can support the complete problem, but use ad-hoc information models, rather than following a standards-based solution [4][29][37][69]. Because of that, their applicability is restricted to their specific domains, they cannot be efficiently translated to support complex, heterogeneous enterprise domains. Therefore, none of the analyzed proposal can handle the diversity of changes that can happen to an enterprise infrastructure and automatically react with the required changes.

3. Objectives

The topic of this PhD thesis is the automated management of distributed enterprise services. For this to be achieved, the complete life cycle of the management resources (including initial provisioning, initial configuration, reconfiguration and removal of no longer needed components) must be supported. After the analysis of the state of the art in these topics I concluded that there was not an existing solution which addressed at once all those challenges, while at the same time supporting different heterogeneous environments.

The objective of this work is to propose a set of information model abstractions and techniques which allow automating the deployment and configuration activities of enterprise services distributed over heterogeneous environments. In addition to the metamodel definitions for abstracting all the relevant management information and the set of functions for supporting the management operations, the proposal must also provide guidelines on how to develop management systems which can make use of the base contributions.

In order to better establish the context where this work will be executed I will show the graphical representation proposed by Hegering for reflecting the dimensions of technical management systems. The shaded area delimitates the scope where this work will be focused on.

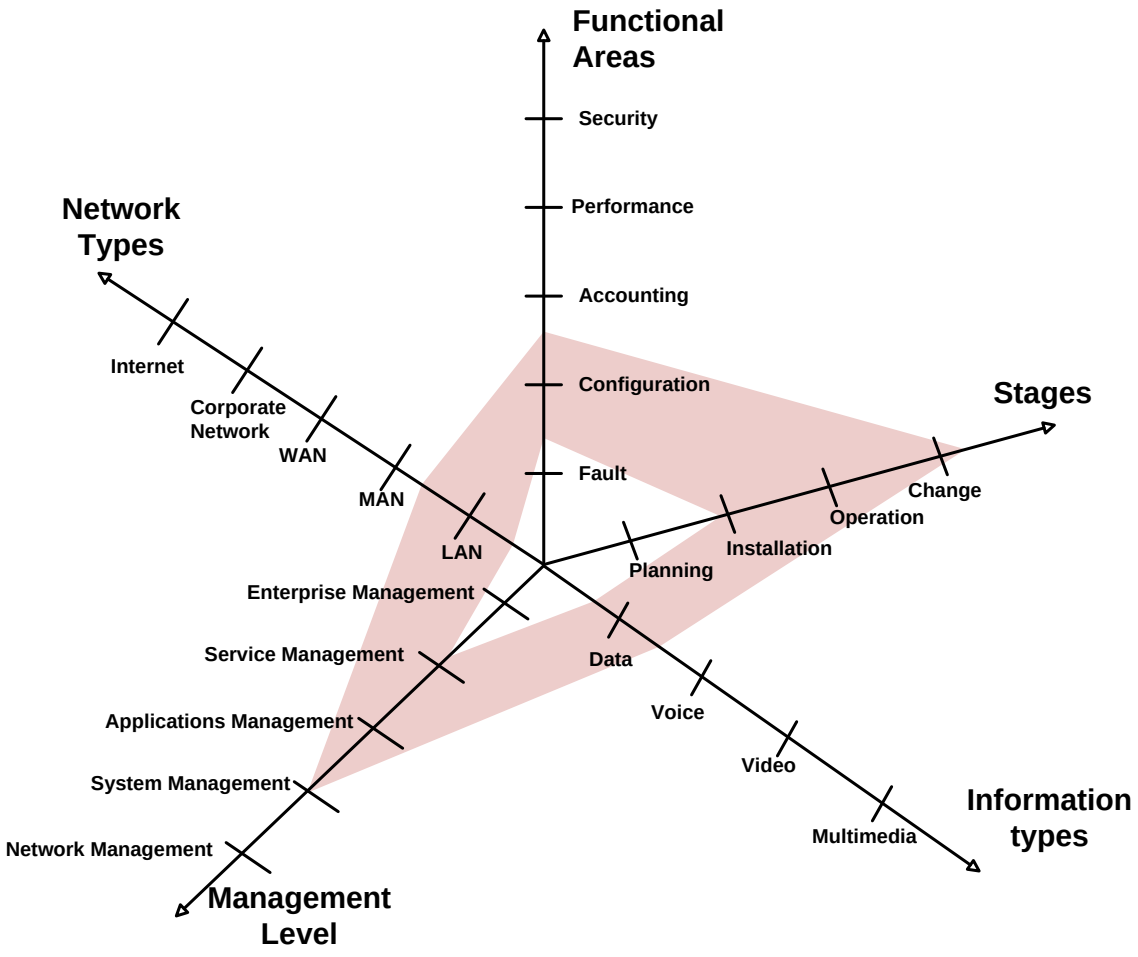

Figure 20 PhD Thesis Scope Definition

By looking at these dimensions of management the first categories which can easily be determined by analyzing the nature of the targeted domain are the type of networks involved and the kind of information exchanged. Enterprise environments are managed at LAN level, because the security restrictions greatly restrict the external communications of the environment. The type of exchanged information is purely data. About the level of management, the work on this PhD dissertation is focused on applications and service management, whereas on a functional classification the main functional area is the configuration. Fault management must also be covered, as the architecture must be able to diagnose the correct functioning of the system and react accordingly if an incidence is detected. Finally, regarding the management stages, depending on the type of situation the proposed solution should support initial installation of applications and services, as well as further operation and changes to the environment.

By breaking down the general objective into specific tasks the following specific objectives of the PhD work have been identified:

- *To define a set of models which characterize the problem of managing enterprise distributed services*

The proposed models must contain all the required information for an automated management of the environment. This way, they must characterize the managed infrastructure, the containers, services, applications and its configuration, including all the existing dependencies and relationships among those elements. On top of that, the models must also define the changes which can occur to those elements and the functionality that the managed system must provide.

The models supporting the proposed system architecture must be aligned with the modeling standards of the domain, building upon the commonality between management standards and trying to extend their expressivity whenever it is required.

- *To support the complete lifecycle of enterprise services*

Most of the analyzed research work focuses on specific service management processes, such as provisioning and deployment, reconfiguration, network management or inter dependency management. The proposed techniques and modeling concepts must support the complete services life cycle, ranging from the initial provisioning of the services to its runtime reconfiguration and final removal from the managed environment.

- *To support the complete automation of the service management*

Traditional enterprise management requires human intervention and manually defined workflows. The architecture must support the partial automation of these processes, reducing the effort required to manage enterprise services, taking as reference the autonomic computing paradigm. This way, the proposed architecture must detect incidences occurring at the managed domain, diagnose the current state and if necessary, detect the required changes for restoring the system to a correct situation, constituting a management close control loop.

- *To propose a reference architecture and validate the proposed models and algorithms*

The described models and algorithms must be supported by a proposed reference architecture, which can support the management activities over multiple, different

heterogeneous distributed environments. It must interact with the rest of the enterprise infrastructure, instrumenting the physical elements of the environment and also communicating with the rest of information management systems from the enterprise.

Also, a working prototype of these elements will be developed and validated against a set of representative cases to verify that it addresses the objectives presented in this chapter.

4. Information Model Definition

In order to provide techniques for an automated execution of service management operations, it is necessary to define the underlying information model, which will characterize the elements from the managed environment in a format suitable for programmatic processes. This chapter presents the proposed information model for characterizing enterprise heterogeneous distributed systems and the process followed towards its definition.

First, the main characteristics that must be supported by the model, as well as the selected language for its representation will be described. Those fundamental concepts will be analyzed and defined, building upon the conceptual abstractions provided by the main standards. This way, a generic model will be defined, and a set of functions will be provided to diagnose the stability of model instances. Once the main concepts are completely described, they will be specialized into a specific information model for enterprise service management.

4.1. Modeling requirements

The first step which must be followed is to clearly establish 'what must be modeled'. In this process I will follow the criteria proposed by SDD for the environment: the model must include every relevant element, which can participate in some way over the management operations. In the model definition, I will skip non technical concepts such as inventory codes and vendor information, which are actually included in some of the analyzed standards such as CIM. Nonetheless, if those factors were determined to be mandatory for the model, they can be integrated along the base definitions.

The model must characterize the services which compose SOA enterprise applications, and every participating element. This includes the logical definition of the service, the providing components if applicable, the enclosing deployment artifacts, the required configuration attributes or the requirements and dependencies for it to work. In addition to that, the available runtime infrastructure must also be modeled, describing the running services, the available resources, the possible containers for the applications, or the network connectivity capabilities. Traditionally logical and runtime information has been managed separately, but approaches such as [62][93] have showed that the right approach is to provide a unified view of both concepts in order to increase the automation capabilities. This consideration can also be extended to the remaining aspects which will be characterized besides the information model. All the management related knowledge must be consistent in order to enable automated reasoning.

4.2. Modeling language

As the final prerequisite for starting with the base modeling definitions, it is necessary to establish the language which will be adopted to define the modeling concepts. The analyzed approaches have selected a large variety of specification languages, ranging from ad-hoc, unstructured data, to XML based structured specifications, object oriented models and ontology-based models.

The main requirement for the selected language is for it to be expressive to model the required information as well as the relationships between the elements, which has lead to discard the lesser structured, most basic options.

If the existing standards are looked as reference, it is clear that current trends point to object oriented models, such as the ones defined by CIM and OMG D&C. The constructs provided by UML class diagrams constitute powerful modeling concepts not only for the information but also to characterize the relationships through the well known abstractions of composition, association or inheritance.

These factors have motivated the selection of an object oriented metamodel, as it combines powerful expressivity and easy access to already available modeling knowledge (e.g. CIM modeling abstractions). From the available object oriented languages I have selected Ecore [106]. This language is a simplified set of MOF (Meta Object Facility), the main metametamodeling language promoted by the OMG as part of the MDA model [59]. Ecore is designed to define metamodels (in this case, belonging to the structure of the information models). Ecore contains the same constructs and expressions seen from class diagrams (such as object, attribute, method, composition, or inheritance), while at the same time does not bring the additional often overlooked complexity of complete UML models (mainly derived from the dynamic aspects of the specification, unsuited for defining an information model). Ecore is a mature language, considered the de-facto standard of MDA based tools, such as the numerous projects available at the Eclipse Modeling Project.

Ontologies were the other main alternative as the base modeling language, but were discarded for several reasons. The required effort to model with ontologies the characteristics of heterogeneous, complex systems, was a deterrent factor. This was aggravated by the lack of available information model standards defined with ontologies, so an additional effort would be required if those concepts were to be incorporated into the proposed model. Nonetheless, there are several research works which show how object-oriented models can be transformed from / to ontologies with additional formalization of the base models [68], which does not rule out this possibility. Regardless of the format, clearly the additional knowledge contained by the management specific ontologies described in the previous chapter must be present in one way or another in the complete management architecture.

4.3. Model foundations

Before providing a detailed information model I will define the management modeling abstractions that will enable the specific definitions. These abstractions will be applicable to any management information model, not being specific to model enterprise concepts. The definitions include a characterization of the basic managed elements, the resources, and its most important characteristics and relationships.

4.3.1. Resource

As every analyzed management information model builds on the concept of resource, it will be the first definition of the proposed model: resources are the base unit of abstraction for a management view. This way, a managed environment is composed by a set of resources, which comprise all the relevant information for an effective operation of the environment. In a

service management architecture, the most representative elements modeled as resources are services. However, every other relevant software and hardware artifact, such as operating systems, containers, dll or jar libraries, TCP ports, processor speed, ram memory, or peripherals, will also be modeled as resources in the management view of the system.

An efficient operation of resources requires the availability of sufficient information about each resource state and configuration. In order to model this knowledge I will add a set of properties, composed by pairs name-value, to characterize the configuration of every resource. This solution is the most extended in the current management specifications. If for specific subtypes of resources, one property turned to be mandatory for all the similar instances, it could be promoted to a named attribute of the class. Some examples of candidate properties for specification are the status information of runtime resources with a defined lifecycle, or the version of software resources. Basically, if every similar resource should have this characteristic the state information should be moved to the class definition. However, I will try to reduce to a minimum the amount of defined types in order to keep the model simple and flexible.

A management environment is composed by a very large number of resources, from a potentially infinite number of them. However, many of those possibilities actually represent different states of the same resource. For instance, in two different time intervals, a resource representing the RAM memory is completely identical, except a variation in the property 'freeMemory'. These two resources represent the same manageable element, a hardware piece, in spite of having a different value for a property. Clearly, a mechanism for establishing resource identity, and differentiate unique entities from the infinitely large set of possible number of modeled Resources is required. The concept of resource identity is managed in the proposed model with a meta-attribute of every resource, called UID (Universal Identifier), which will always be different for two distinct resources. This allows determining if two resources are the same just by comparing their UIDs. I have called it a meta-attribute, as its value is not independently defined, but is derived from the evaluation of several resource properties. This way, a resource is completely characterized by the collection of containing properties, and a specific subset of them also defines the resource identity. The set of properties which are relevant for the identity will differ from some resources to other. In the following sections we will provide additional restrictions to the identity concept. Figure 21 provides a graphical representation of the abstractions of the base resource model.

Figure 21 Resource model

4.3.2. Instantiation (logical and physical resources)

The list of elements which should be modeled includes elements from two different categories: logical resource definitions, such as the available services from an artifact repository, and runtime instances, representing the software and hardware elements available at the runtime environment. Logical resources can also appear as runtime instances of the environment, and a relationship should be established among them (e.g. the one between a

service definition and a running instance of the service over a container). This relationship is analog to the object orientation difference between logical classes and runtime instances. Logical and runtime resources establish a partition from the complete resource set. However, there is no intrinsic information which is common to all logical resources neither to all physical resources, although there will be subsets of each of them and relationships between resources which can only be applied to logical or runtime elements. Because of that, from now on, I will specify whether each definition applies to one or both groups.

In addition to that, a base relationship exists between a runtime element and its logical definition. A runtime element is an instance of its logical definition, and these definitions are instantiated at concrete places in the runtime structure. However, this does not necessarily imply that runtime elements contain all the information defined at the logical level. As resources are abstractions defined for an effective management, there might be invariant information, that, although relevant for its logical definition, would be redundant as part of the runtime view of these instances (for instance, information about licenses, and vendors of an installable library). However, there is a minimum set of common information between a logical resource and their instances, composed by all the identifying attributes of the logical definition. These shared attributes allow to univocally identify the corresponding logical definition of an instance. On the other hand, runtime resources' UID will be constituted by additional properties, which allow distinguishing different instances.

The logical-runtime duality will not occur for every managed resource. A dual definition (logical and physical) of resources only makes sense for elements whose complete lifecycle (including instantiation to and removal from the environment) is controlled by the management architecture. There will be runtime resources without a logical definition, as it becomes clear that non-instantiateable resources (which can't be added to or removed from the runtime environment) are only relevant to the management architecture as runtime instances. However, the opposite is not true. It makes no sense for a management model to include logical resources without a corresponding runtime realization.

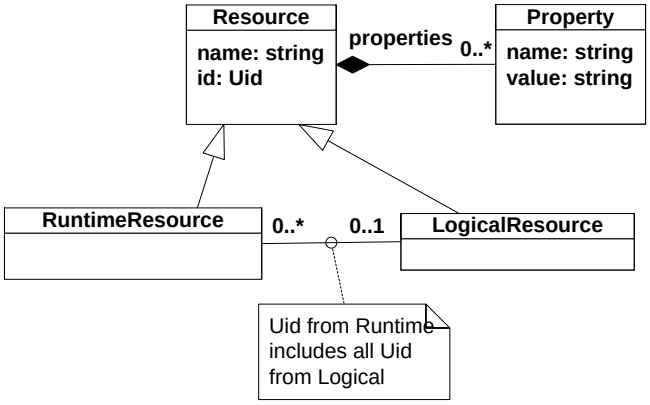

Figure 22 Logical and Runtime Resources

Although it is not reflected explicitly in the model, there is another key difference between these two kinds of managed resources: Logical resources can exist by themselves whereas runtime resources must be contained by other element of the runtime environment. For one logical resource defined (e.g., a characterization of an xml parsing library), multiple runtime instances can exist in the environment. This attribute is the latest factor for defining the

identity of runtime resources: Two runtime resources are actually the same resource if all their identifying fields share the same values, and their placement in the environment is the same.

Instantiation is an operation that takes as parameters a logical resource, and a host element from the environment, and creates a runtime resource at that point which is an instance of the logical definition. I will provide a more formal definition and description of resource hosts in the following section.

4.3.3. Composition

I have already defined the basic characteristics of the resources, and how the management information can be adapted to a homogeneous model. However, in many cases, the expressivity allowed by a set of properties is not enough to model adequately all the information related to a managed element. Because of that, I will introduce another extended type of resource, with enhanced expressiveness.

A *CompositeResource* is a resource which presents its internal management information in a more structured view, through the use of additional composed resources. The individual resources are indivisible from the main one. *CompositeResources* can both be logical or runtime. As regards instantiation, sub-resources cannot be instantiated individually, but all of them will be present whenever the main resource is instantiated. It would be theoretically possible to define *CompositeResources* whose internal resources were also composite (and so on), but those cases don't have a clear application, so they will be substituted by additional resource relationships. Next figure shows the *CompositeResource* model.

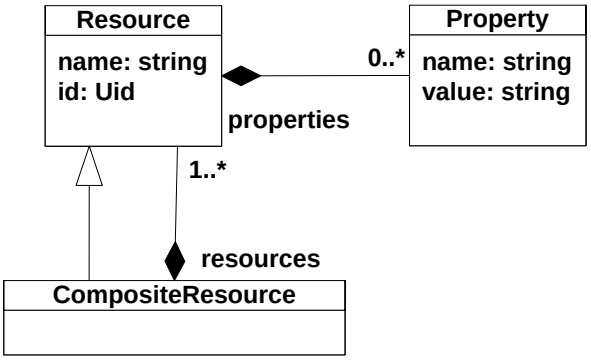

Figure 23 Composite Resources model

Composition is the most basic structural relationship, as the resulting element is a single resource. However, in a runtime environment there are additional relationships which must be traced between distinct resources from the environment. The next two sections describe those two relationships: the previously mentioned containment relationship, and the more general dependency relationship.

4.3.4. Runtime Containment

It has already been mentioned that the management view of a runtime environment cannot be defined as a simple set of independent resources. Therefore, the proposed model must effectively capture the existing inter-dependencies between the managed elements. I will start defining the containment relationship, which captures the hierarchic structure of the managed resources. That relationship occurs at many levels in a runtime system: General programs and services are hosted by the operating system, enterprise services are deployed over an

application server, and even virtual node instances are managed by a virtualization manager or hypervisor. In the remaining of this work I will use the terms 'container' and 'host' to refer to the containing elements.

At runtime, a resource host provides an execution context, providing several capabilities and configuration to the existing units. From a management perspective, the containment relationship also indirectly enables the operations consisting on adding or removing resources to the runtime environment (or, to be more precise, to a host of the runtime environment).

After that introduction I will provide a simple definition of the hosting relationship: "A (runtime) resource *A* is hosted by other (runtime) resource *B* if *A* exists only inside the execution context of *B"*. This way, if B is removed A must also be removed from the environment. The hosting relationship cannot be symmetrical. The containment relationship can only occur between two runtime resources, as it makes no sense in the logical domain.

After describing those concepts I will translate them to the model. First, it is necessary to define a resource subtype for the runtime instances, the *RuntimeResource*. As we have already mentioned in the previous sections, runtime identity is obtained from the combination of logical UID, and its containing resource at the environment. On the other hand, a *HostResource* is a *RuntimeResource*, which contains (aggregates in OO terminology) a collection of hosted runtime resources. This general definition will be specialized in the specific models, where specific kinds of hosts will only aggregate specific subclasses of resources, as I will detail in the further sections.

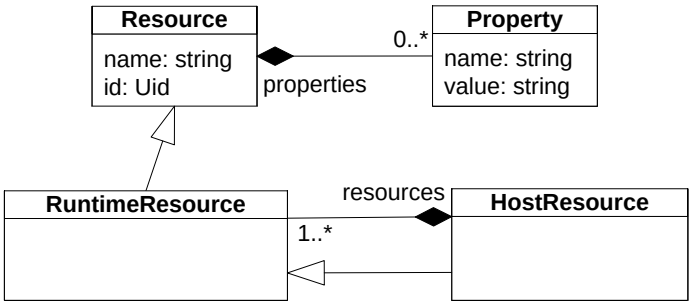

Figure 24 Containment Relationship model

The containment relationship defines exactly the identity of runtime resources. I have already mentioned that two different runtime resources can have the identical logical image and still be unique. We can state that two runtime resources contained by different hosts are intuitively different, as each runtime resource can only have one host. In the same host, two resources are unique if their logical identity is the same.

It must be noted that, although very similar, there is no possible confusion between *HostResources* and *CompositeResources*. Composite resources are indivisible, and they are treated as a single entity. *HostResources* can exist at the runtime environment without any hosted element, and contain a variable number of hosted elements. On the other hand, *CompositeResources* always appear with the same set of internal resources.

At this point, I will define an additional runtime element, which has already been informally mentioned. The *Environment* is a special kind of *RuntimeResource*. There is exactly one instance of this element, and is not contained by any other resource. The Environment has no

logical definition, as it represents the runtime management domain. Every other information element will be part of a root hierarchy initiated by it. As the hosting relationship is mandatory for every runtime resource, it changes the topology of the managed environment from a flat set to a directed tree (a single-origin directed acyclic graph). The root of the tree is the managed environment, and every runtime resource is part of it under a hierarchical structure. This concept has already been applied by standards such as SDD (without any specification) or OMG D&C (Although it is actually implemented by the Target, Node and Resource classes, it is not defined as a general relationship).

Management environments will have runtime resources which are simultaneously hosts of other resources, and are hosted by another element from the environment. Figure 25 shows a sample management view of an enterprise node, modeled as a set of interrelated resources. Every element is a resource. Darker tone rectangles represent host resources, and the elements on top of them are the contained elements. We can see in this example several cases of host-hosted resources, building the intermediate layers of the management tree. Partially overlapping resources represent parts of the *CompositeResources* existing at the moment that snapshot was taken.

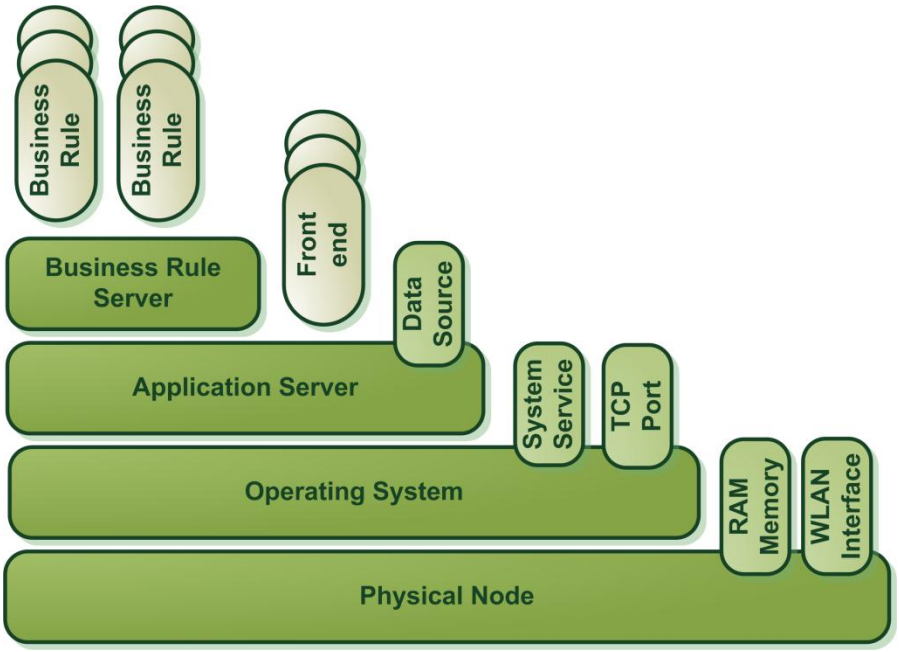

Figure 25 Sample model of a managed node with Hosted and Composite Resources

After all those concepts have been defined I can finally provide a definition for Configuration. Let's define R as the complete set of *RuntimeResources*. If R^* is defined as all the possible combinations of *RuntimeResources* then $C = R^* \cup Env$ is the complete set of possible configurations of the environment.

An environment configuration $c \in C$ is a representation of all the management-relevant information from an administration domain. It is composed by the collection of *RuntimeResources*, classified in a containment hierarchy tree, whose root is the Environment.

4.3.5. Dependency relationships

The previous definitions allow defining the structure of a management environment, through the runtime elements definition and the containment relationship. If there were no additional

relationships between the management environments the tasks of managing a distributed environment would be very simple, as the impact of a change in the configuration would affect only the sub leaves of the tree. However, the experience indicates that this assumption is not realistic, as it is confirmed by existing concepts such as network services, software component dependencies, or client-server architectures. Because of that, a new type of resource relationship must be defined: dependency relationships. Dependencies can be assimilated to uses relationships, as in object-oriented design. Dependencies define an overlay on top of the hierarchical tree relationships structure of the environment, as runtime resources can depend from any other from the runtime.

After describing this new relationship I will define it in the information model. It becomes clear that, as it was the case with containment, dependencies can only be defined among two *RuntimeResources*. Logical resources can't have relationships between them. For runtime instances, a dependency relationship is expressed with a binding to the depending resource. Bindings are properties of the dependant resource which point (by identification) to the runtime resource it depends on. Bindings can point to any resource except their own. Bindings must be defined to explicitly represent the dependency relationships. A resource can have any number of bindings defined, with potentially multiple references to the same resource. In case a resource is bound to a subresource of a *CompositeResource*, the binding will be expressed to the main resource, as it identifies the composition.

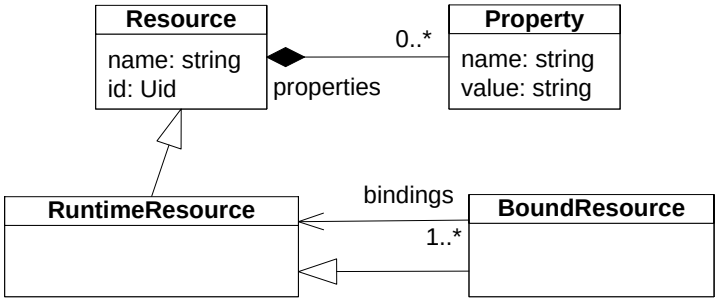

Figure 26 Binding Model

Dependencies add an extra layer of complexity to the processes of change management, and impact assessment. However, they are very frequent in runtime environments so they must be able to be properly modeled and treated by the management architecture.

4.3.6. Configuration Stability

I have already defined the basic concepts for modeling the complete structure of the environment management information. However, just providing the structure is not enough for an effective management. From the possible set of existing configurations, C, only a small subset will cause the system to work properly. Because of that, it is necessary to define along the structural information of the model the rules and validations that will allow to programmatically determining whether one specific configuration is valid or no one. In order to do so, I will introduce the concept of resource stability.

A *RuntimeResource* r is stable whenever its stability condition is evaluated to true, when tested against the environment configuration where the resource belongs. Only *RuntimeResources* can be stable (or even tested for stability). However, both logical and runtime resources can define stability conditions. In case both elements exist, the conditions

can be absent from the runtime instances, this way being evaluated against its logical definition. For runtime-only resources, that cannot be the case.

The stability condition is composed by a set of validations against the environment configuration. It can check for existence / non-existence of another *RuntimeResource* from the environment, or check that the value of a specific property of a *RuntimeResource* is within a defined range of accepted values. The stability condition can be decomposed into a set of individual checks, each one defining a valid set of values for a property or attribute of any resource. This way, the stability of a single *RuntimeResource* from the environment can be expressed as a single logical formula, checking each stability condition is true for the current environment configuration.

Each individual stability check is a function which will be evaluated against several variables from the complete environment configuration. Depending on the scope of the analyzed environment, the following categories can be identified:

- **Local conditions** only evaluate the properties of the *RuntimeResource* where the condition is defined. They are the simplest ones, but allow to express concepts such as checking that initial settings for an application have been defined.
- **Hosting conditions** are evaluated against any property or attribute of the contained *RuntimeResources*. They propagate to additional elements other than the one defining the check, but they are constrained to the hierarchical runtime resource structure.
- **Constraint conditions** can refer to any characteristic of the containing execution context, which is composed by the host resource, as well as its own hosts, ending at the environment. The context provides to the contained resources a set of properties and resources, where specific elements will be required to appear in order to ensure the stability of the hosted elements. The sub resources aggregated by composite resources are also considered in the search scope of a constraint condition, as they are part of the host resource. However, the same is not the case for other contained elements of the hosts, which are outside the scope of this type of conditions.
- **Dependencies** are the remaining type of conditions. They can be evaluated against a characteristic of any resource from the environment configuration, regardless of its place in the containment hierarchy. It is clear that from the dependency and binding definitions, any resource can be bound to any other one, which in most cases would not produce stable configurations, hence the need to define constraints to these relationships. These restrictions constitute the most important subset, because they greatly impact configuration management processes.

The next picture shows a generic runtime configuration where several examples of the different types of conditions are shown. Each shape represents a stability condition, and the stars indicate the configuration element it is being validated against. Different shapes (square, triangle, diamond and circle) identify the different types of relationships. The shaded background boxes represent the environment configuration subset where each specific condition can look for values. Dependency checks have a shared search space, which comprises the complete environment configuration.

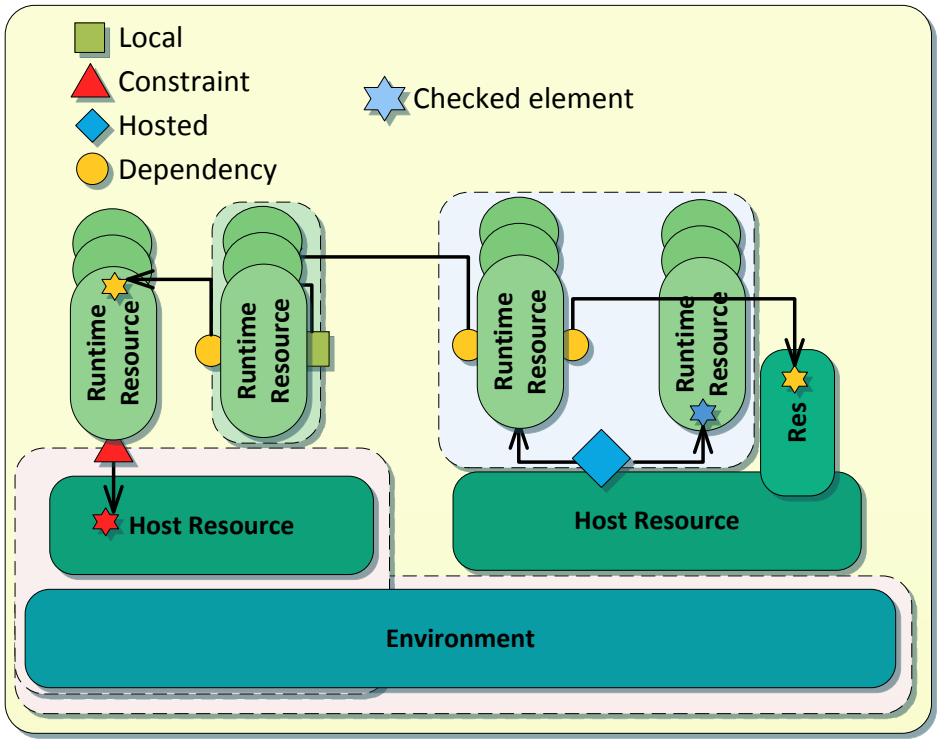

Figure 27 Stability check types and search scopes

In the case of *CompositeResources* and *HostResources*, the stability condition must be true for every participating element, including the main resources and all the containing / composed resources. If we apply recursively that operation to an environment configuration we obtain the definition of the stability of the complete environment. An *Environment* is stable if and only if all the runtime resources which are part of its current configuration are stable. This way, a single formula can be defined for validating the stability of an environment configuration, composed by the validation of each condition defined in its resources.

$$c = \{r_1, r_2, \dots, r_n\}$$

$$Stability(c) = \bigwedge_{i=1}^{n} Stability(r_i)$$

It must also be mentioned that *RuntimeResources* are stable by default. Therefore, resources without stability conditions will be stable. This can be expressed assigning them the following stability function: $Stability(r) = true$. Because of that, these resources can be ignored at the complete stability function of the environment, although they may still satisfy the conditions by the rest of runtime resources.

4.4. Heuristics definition

The basic concepts which have been defined in the previous section allow characterizing the information model for managed environments. However, the model must be evaluated not only for their expressivity but also for their suitability to be automatically processed by management architectures. In this second category, the defined concepts leave a great degree of freedom which can complicate its application. With these factors in mind I will further

evaluate the complexity of the configuration management problem, in order to better estimate the role the models must play in this regard.

In the literature there are some studies which analyze the algorithmic complexity of the problem of obtaining the correct configuration for a distributed system [8][109]. The two contributions analyze the problem from different approaches, looking at finding a correct system configuration, and looking at the achievable transitions by composing operations, but their conclusions are the same: The problem is at least of NP-HARD complexity, making it in principle unfeasible to be handled in a polynomial time by an algorithm.

However, those intrinsic characteristics of the problem have not impeded systems to have been successfully managed over the years, mostly by human personnel. The reasons for this are mostly two, as are discussed in the latest reference. First, not all the configuration problems are of the maximum complexity, most cases being more tractable that the worst one. On top of that, management experience and documentation provide a set of well-accepted guidelines, abstractions and processes which can be repeated and allow solving the same problems. The available management standards convey a considerable amount of that knowledge, further reinforcing the aim of the models to be consistent with their definitions.

In addition to those considerations, the presented work mentions a set of additional measurements which can be taken to improve the tractability problem. The most relevant for this work is the suggestion to reduce the state space, by applying a set of heuristics that limit the potential configurations and consequently reduce the complexity of the problem. In this category two additional recommendations are provided: to define a simple set of operations and to structure the elements with hierarchies and relationships (already addressed by the proposed model).

After those considerations, it is clear that the specific model for enterprise service modeling will be more tractable that the initial one, as the concrete classes and operations tackle the general complexity. However, before defining those elements I will also refine the generic model with a set of heuristics that apply an initial filtering to the enormous amount of possible combinations obtained from the initial definitions.

4.4.1. Scope of reasoning for management processes

The management functions reason over the modeled resources, which have been classified into two partitions: logical resources and runtime resources. In principle, the size of these sets is arbitrarily large, but in practice that is not the case for real management processes; the relevant resources are a finite subset of each of them. In this section I will provide definitions for these two subsets, ensuring that the search space for management operations is finite and well-defined.

Before providing the definitions of these sets the concept of domain state must be defined. The management view of the domain is composed by a set of resources, which will change over time. Only the defined resources (both logical and runtime) at the current state affect management operations. This way, in order to determine the concrete subsets, it will be necessary to define which resources are available at the current state.

The concept of configuration has already been defined for the space of runtime resources. At any state, the set of relevant runtime resources is completely represented by its configuration.

This way, for management purposes, the only relevant runtime resources are the elements from the current configuration.

For the space of logical resource definitions it is necessary to define the finite subset of relevant elements. A LRB (Logical Resource Base) is a subset of the potentially infinite space of logical resources, which contains only the available resource definitions of a management context. As it is the case with the environment configuration, management processes can only reason with the logical elements defined at the LRB. Similarly to configurations, the contents of the LRB can also change over time, by factors external to the management system.

The definition of configuration and LRB as the only relevant resources constitutes the first heuristic for addressing the inherent complexity of the problem. The outcome of every management operation will be only based on the LRB and the current configuration. These concepts are assumed in every management system, but have also been explicitly defined along the proposed information model.

4.4.2. Resource typology

One of the reasons for the complexity of managing distributed environments is that the configuration is composed by a very large number of resources, and each instance is a completely different entity, although the hosting, composition, and dependency relationships partially address this problem by defining an environment structure. However, it is clear that there are sets of resources which share many common characteristics and can be managed similarly (e.g. the different web applications installed at a server container, or the listening TCP ports reserved at operating system). One initial solution to characterize those groups is to define subclasses for each category, starting from the base *RuntimeResource* concept, obtaining a different class for each kind of managed resource. This is the followed approach of the CIM Resource model specification. However, requiring to define a specific subtype for each different element presents scalability concerns, as large models are considerably harder to define and maintain, and require a much greater effort from the supporting management architectures. However, the expressivity of the specialization mechanism is also very valuable for identifying similar concepts. With these concerns in mind, I have followed an approach based on the work by other standards such as OMG D&C, defining a hierarchic resource typing mechanism. This retains the advantages of specialization while being more flexible than CIM's mandatory explicit inheritance. As an additional advantage of this option, it does not rule out the possibility of also specifying some subclasses for key modeling concepts, as it will be explained later.

It has already been established that resources are identified by a selected set of properties. If those identity properties are analyzed, it can be seen that some of them do not provide an individual identification of the element (as, for instance, happens with the name attribute), but constitute a common ground shared with other similar resources. I will define the resource type as the combination of all the group identity properties. As those concepts have been identified for every resource, they will be promoted to attributes of the base class. Therefore, from this point on, resources have a name, belong to a type and contain a list of additional properties. The addition of types completely defines the "commonality" part of each resource identity, allowing to group similar resources. Because of the similarities of members from the

same type, they will share a set of properties, although it is not enforced in the way sub classing would do.

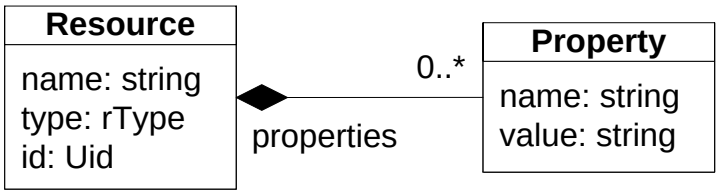

Figure 28 Updated resource model with types

The type definition is applicable to both logical and runtime resources. Moreover, as resource type is an identity property, the same type must be shared by logical resources and runtime instances. This is a key concept that enables a simultaneous reasoning over logical and runtime resources.

After providing the base definition of the concept of types, I will detail how that information will be modeled. Resource types are of the kind *rType,* which allows to support the concept of hierarchical resource classifications. Resources can only have one value for its type. However, depending on the degree of detail applied in its identification, a resource could have many different types (e.g., a windows service and a database are clearly both software resources, but their actual characteristics are not quite the same, implying they belong to different types). In order to enable a greater expressivity, types aggregate hierarchically, some of them being further restrictions of already existing ones, which allows resources to belong simultaneously to multiple subsets. This way, types achieve a complete hierarchical clusterization of the complete space of resources. For the remainder of this document, I will represent types as Strings, with a format similar to Java packages. E.g., the type *software.database.jdbc.oracle* is a specialization of the type *software.database.jdbc*.

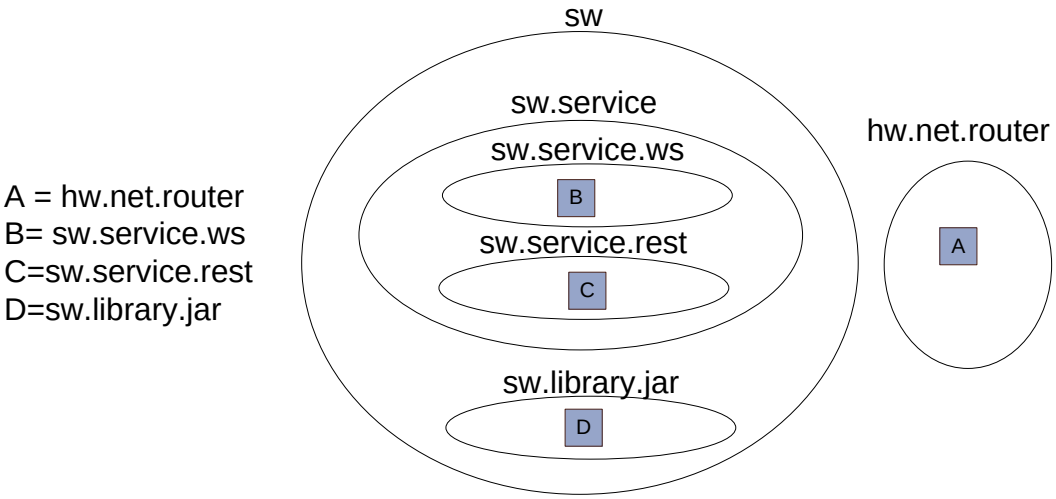

Figure 29 Set Representation of the Type Containment

Types also allow restricting the set of potential configurations by defining a type taxonomy which declares the supported types by the architecture. It is clear that the process of defining that taxonomy cannot be automatically extracted with any algorithm, being defined instead by an analysis of the domain by experts, performed at the adequate abstraction level of the

management architecture. Those initiatives can be performed coordinately, so that different management models develop with a finer level of detail different sectors of the type hierarchy.

The addition of types to the base resource definition greatly reduces management complexity. The obtained hierarchical clustering greatly reduces the scope of most operations and relationships. Over the following points the type concept will be used to provide a concrete definition of several stability checks, which will express the conditions for a correct behavior of the system. Also, this is achieved without the need to define a very complex subclass hierarchy, which would be necessary to represent all the information from complex, enterprise heterogeneous environments.

4.4.2.1. Resource inheritance

After stating the concept of resource types, it is possible to extend the expressivity of the classification mechanism by allowing resource inheritance for some specific cases, in concordance with the typing concept. In specific domains, a reduced number of resource types play a prominent role in the successful management of the environment. These elements are present at any domain configuration, and are identified by a set of well-known configuration properties, playing a fundamental role in every scenario. For those cases, it is possible to take the notion of typing a step further by the definition of resource subclasses.

A subclass identifies a specific subset of resources, sharing a base type and a common set of properties. The subclass base type must be shared for all the instances, but the specific types can be different to characterize different subsets. Regarding the logical/runtime relationship, if there is a specific class for a logical resource, its realization in the environment will necessarily be represented by another specific subclass, both of which will share the same base type.

4.4.3. Stability checks definition

The initial model defined the stability checks as functions whose input was the environment configuration, or a specific subset for some of the types. In order to simplify the check of a configuration stability I will instead define a concrete set of functions which can be easily evaluated for each type (local, host, constraint, dependency). The defined functions will be simple but at the same time must allow representing the different relationships which appear in a runtime management environment.

The conditions will be defined at two levels. For each category identified previously (local, host, constraint, dependency) I will define a stability function. That formula will be a composition of base checks, which will be the concrete validations. Each base check will be defined on their first appearance, although they won't necessarily be exclusive to one specific type of stability checks, potentially appearing in several categories.

4.4.3.1. Local restrictions

Local stability checks are the simplest validations to be made, as they only refer to the configuration of the defining resource. Moreover, the identifying part of a resource is invariant so it cannot be evaluated on the stability checks of the resource. They don't verify the structural stability of the resource but can easily express concepts such as Service-Level Agreements of a managed service.

In order to define the concrete checks to be made I will take as reference the OMG Deployment and Configuration Standard [83]. The specification defines a model element for

expressing conditions to the value of a property: the *PropertySatisfierKind*. This enumeration described the types of value checks, which are the following: MINIMUM (at least the specified value), MAXIMUM (at most the specified value), SELECTION (one from the specified options), ATTRIBUTE (exactly the specified option), QUANTITY(at most the number of restrictions over it), CAPACITY(at most the a total consumption of the specified amount). I will adopt them as the base checks for local conditions, except Quantity and Capacity. The nature of those two checks is different from the other ones, as the first elements are simple functions with one input argument, which can be evaluated independently, whereas quantity and capacity checks must be evaluated between multiple competing resources. I will provide additional details on them in the definition of the Constraint Checks.

4.4.3.2. Numeric value discretization

As some of the defined local constraints perform numeric comparisons between expected and actual values I will at this point introduce how numeric value discretization allows simplifying the configuration states by focusing on relevant magnitudes which enable a more effective management.

In order to express the values from numeric properties the model will not use the sets of mathematical types (i.e. $\mathbb{Z}, \mathbb{R}$), opting for the sets of programmatic types instead (int, float, long). Moreover, for an efficient management purposes, the precision of the accepted values will also be limited (i.e. 2 or 3 digits of precision for numeric values, thus storing values such as 2.4Ghz, instead of 2369,24 Mhz). For the forthcoming definitions, I will name the set of discrete values $NM \subset \mathbb{Z}$.

With those considerations in mind, the following four base checks have been defined:

MIN(p, val)

***Description*:**	Checks that the value of a resource property is over a minimum accepted value
***Checked element*:**	$p \in r.properties, r \in Resource$
***Additional arguments*:**	$value \in NM$
***Formula*:**	$MIN(p, value) = \begin{cases} true, & p.val \in NM \wedge p.val \geq value \\ false, & p.val \notin NM \vee p.val < value \end{cases}$

MAX(p, val)

***Description*:**	Checks that the value of a resource property is below a minimum accepted value
***Checked element*:**	$p \in r.properties, r \in Resource$
***Additional arguments*:**	$value \in NM$
***Formula*:**	$MAX(p, value) = \begin{cases} true, & p.val \in NM \wedge p.val \leq value \\ false, & p.val \notin NM \vee p.val > value \end{cases}$

ATT(p, val)

***Description*:**	Checks that the value of a property is equal to a expected value
***Checked element*:**	$p \in r.properties, r \in Resource$
***Additional arguments*:**	$value \in \{NM, String\}$
***Formula*:**	$ATT(p, value) = \begin{cases} true, & p.val = value \\ false, & p.val \neq value \end{cases}$

SEL(p, range)

***Description*:**	Checks that the value of a property belongs to a set of accepted values
***Checked element*:**	$p \in r.properties, r \in Resource$
***Additional arguments*:**	$range = \{val_1, val_2, \dots, val_n\}, val_i \in NM \cup STR$
***Formula*:**	$SEL(p, range) = \begin{cases} true, & p.val \in range \\ false, & p.val \notin range \end{cases}$

In order to collectively refer to the four defined base checks I will define the local check $LOC(p)$ which can be either of them.

$$LOC(p) \in \{MIN(p), MAX(p), ATT(p), SEL(p)\}$$

The local checks allow restricting the values of the resource's properties. In order for a resource r to be locally stable, every local check $LOC_i(p_i), p_i \subset r$ must be met at the same time. This way, the local stability function has the following shape:

$$LOCAL(r) = \bigwedge_{i=1}^{n} LOC_i(p_i)$$

4.4.3.3. Constraint restrictions

After providing the local definitions I will focus on defining the restrictions and validations which are evaluated against the execution context of a resource. Constraint restrictions define conditions which must be met by the resource containment hierarchy in order for the resource to be stable.

At this point, it is necessary to provide a concrete definition of the execution context, as it defines the search space where resource constraints will be validated.

$$r \in RuntimeResource, ExCtx(r) = \{r.host, r.host.host, \dots Env\}$$

I will also define the complementary set, which is the set of runtime resources directly or indirectly contained by a *HostResource*. Starting from the designated host, it can be seen as a sub tree of the complete configuration hierarchy. That set of resources will be known as descendants from this point onwards, taking the name from tree graph theory.

$$h \in HostResource, Desc(h) = \{RuntimeResource\, r \in C \,/\, h \in ExCtx(r)\}$$

Constraint restrictions defined by a resource can be satisfied by any resource from its execution context, without the need to explicitly reflect in the model which one will be

satisfying the constraints. However, before providing additional restrictions on the identified resources, it is necessary to provide the means for defining these resource identification functions.

In order to do so, I will introduce base checks for expressing constraints over a resource identity. I will first define the general check, which potentially analyses every identifying property of the resource. More specific functions can be defined at a later stage which only use some concrete identifying fields to make the matching.

RESID(r, idCheck)

Description:	Checks that the resource identity belongs to the identified subset
Checked element:	$r \in RuntimeResource, r.uid = \{name, type, \dots, ui_n\}$
Additional arguments:	$idCheck = \{nameCheck, typeCheck, \dots, idCheck_n\}$
Formula:	$RESID(r, idCheck) = \begin{cases} true, & r.uid = idCheck: \\ & r.name = nameCheck \wedge \\ & r.type \supset typeCheck \wedge \\ & \dots \\ & r.ui_n = idCheck_n \\ false, & r.uid \neq idCheck \end{cases}$

The local checks previously defined can also be used to further restrict a resource filter defined through a *RESID* function, by imposing additional requirements over the resource properties. However, there are many unstable situations that cannot be expressed with the current functions. The local base functions define checks that are evaluated independently. They don't allow representing constraints over the access to shared resources, where there is competition for consuming a limited resource (possibly partitioning them, such as RAM memory, or completely locking them, as a TCP port from the machine). Those situations match the Quantity (of value 1) and Capacity D&C satisfier kinds, which were previously mentioned.

Both identified cases will be supported by additional base checks. First, I will define a new element for allowing the expression of D&C Capacity restrictions. The special characteristics of this function imply that it can't be individually evaluated at the resource constraint analysis, but must be collectively evaluated at the hosting level, as each new definition can alter the verdict on the previously processed restrictions on the same property.

CAP(p)

Description:	Checks that the value of a property is large enough to be used by all consuming resources, each of them requiring a quantity *cons(p)*
Checked element:	$p \in r.properties, r \in RuntimeResource$
Additional arguments:	$\{s_i \in Desc(r) \mid RESID(s_i, rid) = true \wedge s_i.cons(p) > 0\}$ $s_i.cons(p) \in NM$
Formula:	$CAP(p) = \begin{cases} true, p.val \in NM \wedge & p.val \geq \sum_n s_i.cons(p) \\ false, & otherwise \end{cases}$

As the second competitive check, instead of adopting the D&C Quantity function I will provide a check for demanding exclusive resource reservation (e.g. TCP port process bindings). This alternative check has several advantages over the D&C concept. The restriction is expressed over the complete resource, instead of just on the value of a specific resource property of the resource. This simplifies the modeling of these real concerns as it is no longer necessary to define a virtual property named quantity with an integer value of 1, which in the proposed approach is simply substituted by the existence of the *RuntimeResource*.

EXCL(r,rid)

***Description*:**	Checks that the identified resource is demanded exactly once
***Checked element*:**	$r \in RuntimeResource$
***Additional arguments*:**	$rid \in idCheck = \{nameCheck, typeCheck, \dots, idCheck_n\}$
***Formula*:**	$EXCL(r, rid) = \begin{cases} true, & \lvert\{s \in Desc(r), RESID(s, rid) = true\}\rvert = 1 \\ false, & \lvert\{s \in Desc(r), RESID(s, rid) = true\}\rvert \neq 1 \end{cases}$

The available functions allow detailing what must be present at the execution context for a runtime resource to be stable. However, those constraints cannot express incompatibilities between two resources, in other words, what must NOT appear at the context if the demanding resource is instantiated. This concept is present in some of the analyzed standards – they are named ex-requisites in the SDD standard. Because of its relevance I will define it as another base function, defined on top of RESID.

NOT(r, rid)

***Description*:**	Checks that the identified resource is not present at the execution context of the mentioned resource
***Checked element*:**	$r \in RuntimeResource$
***Additional arguments*:**	$rid \in idCheck$
***Formula*:**	$NOT(r, rid) = \begin{cases} true, if\ \nexists s\ \in Exctx(r) \mid RESID(s, rid) \\ false, \quad otherwise \end{cases}$

After the base concepts have been established, it is now possible to define the constraint stability function of a runtime resource. However, before that it is necessary to define how that information can be added in the model, as the conditions cannot be implicitly derived from the current information. I will use the term *ConstrainedResource* for referring to the *RuntimeResources* which either define constraints or have a corresponding logical element which defines them. Both logical and runtime resources can define constraints, but only runtime resources can be analyzed for stability. Resources can declare any number of constraints.

The following picture represents how these constraints can be expressed. *ConstrainedResources* contain a number of *Constraints* which provide the arguments for the already defined base constraint checks.

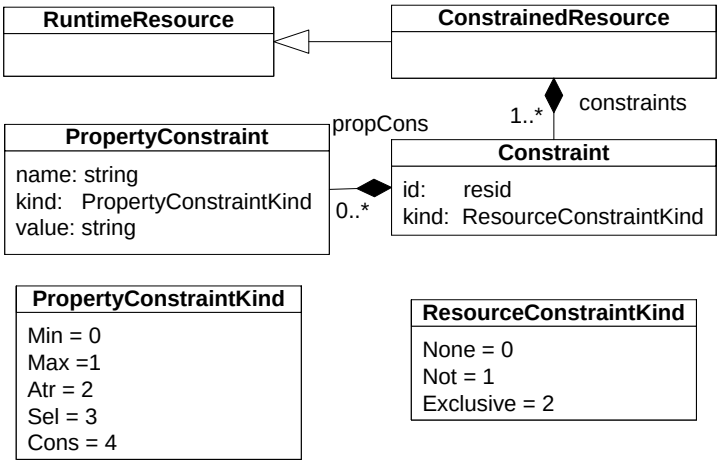

Figure 30 Constraint definition model

A constraint definition is composed by two parts. One mandatory part identifies the resource to be evaluated, which must belong to the execution context. Optionally one or more constraint checks can be defined over the identified resource, allowing to express property restrictions *pCons* (including both local checks and *CAP* checks), exclusive consumption of the resource or incompatibility with it. With that information into account, we can define the stability function corresponding to each check with the following function:

$$CPROP(p) \in \{MIN(p), MAX(p), ATT(p), SEL(p), CAP(p)\}$$

$$r \in ConstrainedResource, r.constraints = \{c_i\}, i = 1..n$$

$$c_i.propCons = \{pCons_j(p_j)\}, pCons_j(p_j) \in CPROP(p), j = 1..m$$

$$c_i.kind = \{None, Not, Excl\}$$

$$CONSTR(c) = \begin{cases} true, & \begin{array}{l}(c.kind = Not) \wedge NOT(c.parent, c.id) \\ (c.kind = None) \wedge (\exists s \in Exctx(r) | RESID(s, c.id) \wedge \\ \qquad \wedge pCons_j(s.p_j)) \\ (c.kind = Excl) \wedge (\exists s \in Exctx(r) | RESID(s, c.id) \wedge \\ \qquad \wedge pCons_j(s.p_j) \wedge EXCL(c.parent, c.id)) \end{array} \\ false, & otherwise \end{cases}$$

$$CONSTR(r) = \bigwedge_{i=1}^{n} CONSTR(c_i)$$

4.4.3.4. Hosting restrictions

After defining the validations which will be applied to the execution context elements I will define how to express and validate the complementary validations. Hosting restrictions evaluate the correctness of the contained resources, checking the containment relationship.

It has been described how the resource type completely defines the group identity information of a resource. Therefore, checking the type information will be enough to determine compatibility of hosted resources. The need to restrict the potential guest resources can be clearly seen: with no limitations, any *RuntimeResource* can appear over any *HostResource*. However, it would make no sense for an operating system (a *HostResource*) to host a physical node. With this in mind, the hosting stability condition will be based on checking the types of the resources to be contained.

Figure 31 shows how that restriction has been captured in the model. *HostResources* will specify an additional property, defining the *supportedTypes* of the contained resources. The property includes a list of compatible resource types. This way, the contained resources do not define the condition, it is their types which will be validated. As the proposed definition of inheritance for specific models builds over the base resource type hierarchy, the same restriction can be applied to the specific subclasses, which will share the same base type.

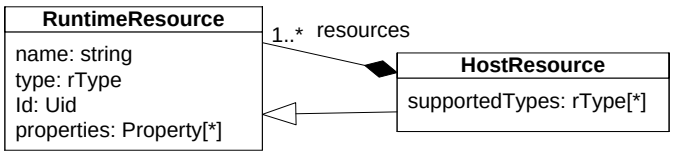

Figure 31 Supported types by host definition

In order to support automatic validation of the hosting constraints, I will first define a specific resource identity check for verifying that a resource belongs to a set of accepted types. As types define a simple hierarchy, the check will be true whenever the type of the evaluated resource is contained into the one provided as a parameter:

RESTYPE(r, typeCheck)

***Description*:**	Checks that the resource belongs to a type / subtype
***Checked element*:**	$r \in Resource$
***Additional arguments*:**	$typeCheck \in rType$
***Formula*:**	$RESID(r, idCheck) = \begin{cases} true, & r.type \supset typeCheck \\ false, & r.type \not\supset typeCheck \end{cases}$

The complete definition of the stability function is as follows. For a Let it $h \in HostResource$, with the following characteristics: $h.supportedTypes = \{type_1, \ldots, type_n\}$. Let it be $h.res = \{r_1, r_2, \ldots, r_n\}, r_i \in RuntimeResource$. The stability function *HOSTING (h)* can be defined as:

$$HOSTING(h,r) = \bigvee_{j=1}^{m} RESTYPE\left(r, type_j\right)$$

$$HOSTING(h) = \bigwedge_{i=1}^{n} HOSTING(h, r_i)$$

The provided definition of the supported types host stability check can be easily validated, and can be applied at different management levels and heterogeneous resources, thus helping to

reduce the management complexity while capturing at the same time the underlying conditions on the configuration stability. It has been defined for every host resource with the property *supportedTypes*, thus avoiding the need to define it explicitly for each element.

4.4.3.5. Dependency checks

In the previous section I have introduced the concept of dependency. Dependencies define an additional layer of relationships over the runtime resources, represented by the binding attributes. However, clearly not every possible binding and resource configuration leads to a stable state, which points at the need of defining dependency stability checks.

As it happened with constraint checks, they must be defined explicitly. However, there is a fundamental difference. Constraints are evaluated globally over the complete set of resources from the execution context. On the other hand, dependency checks have a much more limited scope; they are applied one by one to the defined bindings. This way, there will be one dependency check definition for each binding. As it happened with constraints, both logical and runtime resources can contain dependency checks, but only the runtime elements will be evaluated.

The dependency stability check is a function that identifies the set of resources which can be referred in a binding, with the depending resource remaining stable. Similarly to constraints declaration, the set of valid resources will be identified by a RESID function. In order to keep model simplicity, no local property checks will be allowed for bindings.

Before defining the structure of dependency stability checks (of a binding) there is a remaining factor that affects binding stability checks which did not apply to the previous definitions. Unlike constraints, dependencies can be referred to any other part of the environment. However, it cannot be assumed that a resource can be accessed by any other element from the configuration. For instance, it is evident that in a runtime environment with two networked PCs, the operating system services of one node cannot be accessed from the other one. This aspect is covered by the D&C standard, through the definition of locality constraints. In this information model, dependencies can be restricted, to be matched only by resources available at the same node or the same process of the demanding element.

I have contributed that information to the model but choosing a different approach. Instead of defining locality constraints I have added to the model information about the visibility of resources. Visibility is part of the resource identity, whereas dependent resources should not need to know about that characteristic. There are four levels of resource visibility, defined with the concepts of the information model, instead of through domain specific concepts such as processes. This way, the visibility of runtime resources can be classified as any of these categories:

- *Locally visible* resources can be accessed by other resources which share the exact host as the providing one.
- *Host visibility* resources can be accessed only by the directly contained resources of the host, but not by their own descendants.
- *Context visible* resources are accessible to any resource belonging to the execution context of the providing resource.

- Finally, *environment-wide visible* resources can be accessed by another resource from the environment.

This information must be transferred to the base resource model, by adding an additional attribute to the base resource definition. In case it is not specified, the default visibility of any resource will be local. The following picture details the updated resource model with visibility information:

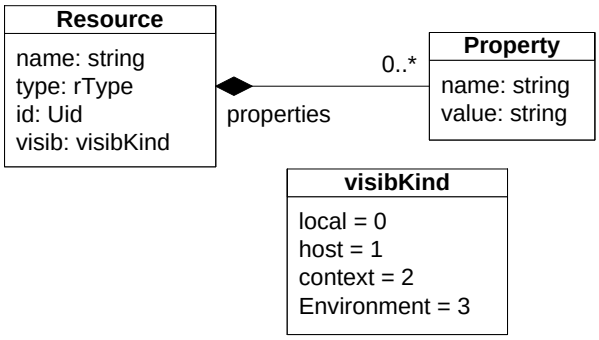

Figure 32 Resource with visibility information

Regarding composite resources, the visibility of compound resources is evaluated starting from the main resource. The following picture shows examples of these four visibility types in a sample environment. The small figures depict runtime resources, with the type of shape showing the visibility of the resource. The subset of the environment where that resource can be accessed is showed by a square frame of the corresponding tone. Environment-wide resources have a shared scope equal to the complete configuration.

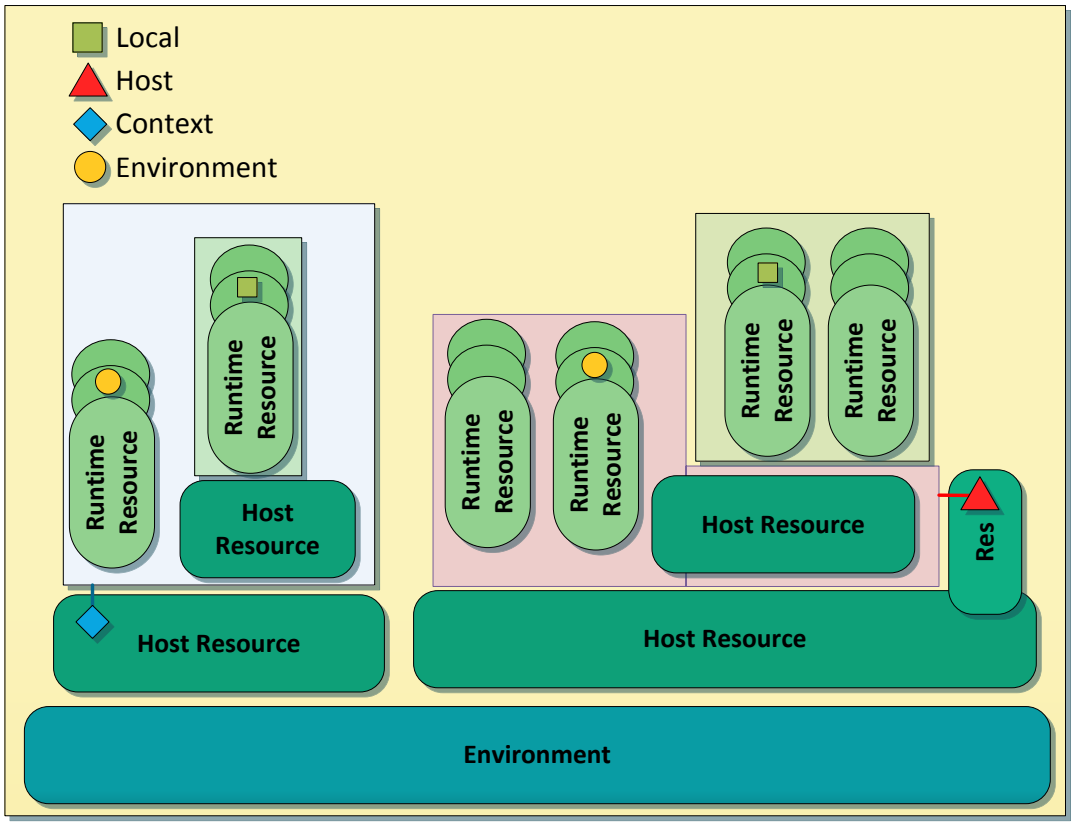

Figure 33 Resource Visibility Scopes

After introducing the concept of visibility to the resource model, it is very simple to define the visibility base check:

VISIBLE(r, s)

***Description*:**	Checks that one runtime resource can access another resource
***Checked element*:**	$r \in RuntimeResource$
***Additional arguments*:**	$s \in RuntimeResource$
***Formula*:**	$VISIBLE(r,s) = \begin{cases} true, & \begin{array}{c} s.visib = ENV \\ s.visib = CTX \ \wedge \ s \epsilon ExCtx(r) \\ s.visib = HOST \ \wedge \ r.host = s \\ s.visib = LOCAL \ \wedge \ s.host = r.host \end{array} \\ false, & otherwise \end{cases}$

Dependencies represent uses relationships between the environment runtime resources. However, if no additional mechanism is provided, the actual relationship between the two elements is kept hidden. Nonetheless, for an automatic management of complex systems, it would be very desirable that the underlying configuration dependencies were explicitly reflected by the model. This way, it would be possible for a configuration value (expressed in the model as a property) to depend on the configuration of the bound runtime resource. As an example of that two-level view on dependency, I will use an example of remote services communication. Defining the service binding between the provider and the consumer allows to estimate the impact of changes to the configuration (e.g, the providing service disappears). However, it is possible that some manual configuration operations are required to allow the consumer access the bound provider. If that configuration property was explicitly related to the binding, it would be possible to automatically obtain the correct value, and apply it after the binding has been established, removing the need for manual operation. I will call the properties dependent of the binding *BoundProperties*. Next figure shows an example of this concept. The resource R has a dependency which is satisfied by the resource S. This relationship is materialized with the binding b. On top of that, R has a *BoundProperty* p, whose correct value must be the one of the property q of whatever resource is bound to the binding. This way, after the binding to S is established, p value is updated to match q value automatically.

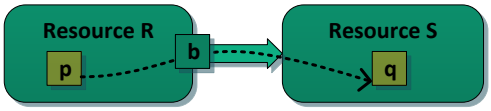

Figure 34 Bound Property and Binding

The concept of bound configuration can be defined by using the existing base checks. If the value of property p of the dependant runtime resource r is bound to the configuration value of the property q from the bound runtime resource s through the binding b, this check can expressed as *ATT(r.p,b.q.value).*

In order to explicitly represent those restrictions in the model I will extend the concept of *DependentResources*. These elements define a variable number of dependencies, each of them composed by a binding to another *RuntimeResource*, and a resource identification which must be met by the bound runtime resource. In addition to that, these elements can define

BoundProperties, whose value will match the value of a property of the bound resource, whose name is specified by the *BoundProperty*. These concepts are illustrated at the following picture.

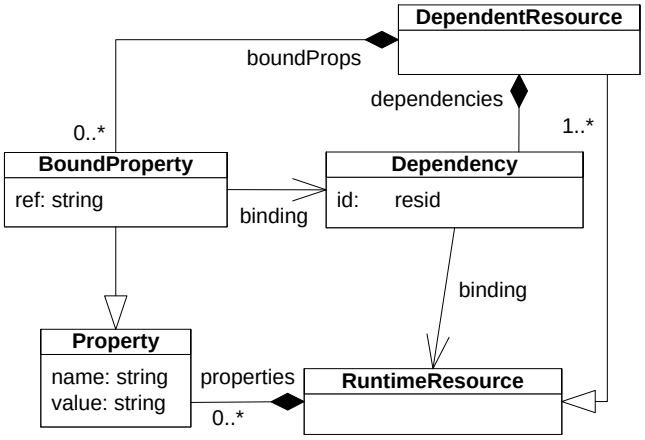

Figure 35 Dependent Resource Model

After defining the required base functions to completely evaluate dependency stability, I will provide the general function. As it was the case for constraints, first each dependency definition must validate the identity of the bound resources. Visibility will also be evaluated. Finally, the configuration of the bound properties will also be verified. Taking into account those factors, the dependency stability function of a resource can be obtained as follows:

Let it be r a $RuntimeResource$ with defined $Bindings\ b_i,\ i = 1..n$, and each binding with defined $BoundProperties\ p_j, j = 1..m$,which are bound to the properties $qj.$ of the bound resources.

$$DEP(r) = \bigwedge_{i=1}^{n} RESID(b_i) \wedge VISIB(r, b_i) \bigwedge_{j=1}^{m} ATT\left(r.p_j, b_i.q_j.value\right)$$

4.4.3.6. Overview on the stability checks

After these heuristics have been established the definition of stability has been refined down to four basic functions (Local, Hosting, Constraint and Stability) which are evaluated for each *RuntimeResource* from the configuration. Local functions evaluate the correctness of individual resources configuration (by restricting the allowed values of its properties) whereas the remaining three functions evaluate the structure defined by the *RuntimeResources.* Hosting and Constraint functions validate the hierarchic structure, defined through the hosting relationships. Finally, Dependency functions validate that the overlay structure defined by the established bindings between *RuntimeResources* is also correctly configured.

The four functions are defined by the combination of ten basic stability checks which can be easily evaluated over single or reduced properties of the configuration. Next table shows the base constraints which are involved in the definition of each stability function. It can be seen how some checks are used only for one category whereas in some cases they appear over several definitions. The latter is the case with RESID, taking part in the three structure checks. This shows how restrictions on resource identity are fundamental to reduce the possible configurations. In addition to that, the MIN, MAX, ATTR, and SEL have been used at the local level as well as for further identifying the constrained elements, but they could also have been

integrated into dependency expressions in order to further restrict the resources verifying resource identity.

Table 1 Participating Primitives of Stability Functions

	LOCAL	HOSTING	CONSTRAINT	DEPENDENCY
MIN	X		X	
MAX	X		X	
ATTR	X		X	X
SEL	X		X	
RESID		X	X	X
CAP			X	
EXCL			X	
NOT			X	
DEP				X
VISIB				X

4.4.4. Resource identity heuristics

The previous section has defined a function to evaluate the stability of a system configuration, which greatly simplifies the complexity of diagnosing the configuration status. The function can be decomposed into a set of individual, atomic checks, all of which must be met for the result to be stable. However, one of the ten base checks, the RESID stability check, cannot be easily evaluated, as the set of identifying resource properties is not completely defined (name and type have been defined, but there are potentially additional properties which also constitute the resource identity).

Therefore, in order to enable an efficient evaluation of the stability function it is necessary to define completely the set of identifying properties of the resources. This way, RESID check will be specified as a fixed set of checks against the identifying properties. After these changes, the final model will be simple enough that an efficient evaluation of its stability will be feasible.

The previous analysis in resource identity defined two properties which were common to all the resources and influenced the resource identity. The resource *type* completely covers the resource group identity. On the other hand, the resource *name* will always be part of the individual identity, but depending on the characteristics of that type of resources, it was not determined whether additional properties were necessary to distinguish between different resources with the same name. With the aim to close this gap in the model, I have analyzed the existing information models, looking for characteristics of the modeled resources which separate the identity of specific resource types.

After this review there is one additional property, identified in the CIM Application model and WSDM resource model specifications which further restricts the identity of resources: *version* information. *Version* differentiates resources with the same name and type, complementing the existing identifying properties. The concept of versions is well-known in the software engineering discipline, and its insertion into the management model contributes one fundamental concept missing until now: the managed system and its managed resources are elements of a continuously evolving ecosystem. Over time, internal changes in the elements can originate new elements, whose identity differs from the initial. These two elements are

different (at least the *version* will change), but they have a strong shared identity, further than the likeness reflected by sharing a *type*. This way, starting from the set defined by all the resources of a common *type*, additional subsets can be identified by grouping resources with the same *name*. Each resource from that subset will be at least different from the others by its *version*. Unfortunately, more than *type* and *name*, it is not possible to establish exactly what set of properties is shared between resources of two different versions. Moreover, the commonality will possibly change between two different pairs of resources taken by the same subset. It must be noted that, as an identity property, *version* is a common aspect of logical and runtime resources, and will be shared between logical definitions and runtime instances.

The combination of *name*, *version*, and *type* provides a framework strong enough for establishing a managed resource identity, built upon the abstractions from the main standards. No additional properties have been detected which influence the identity, so these three will be the only ones evaluated. However, it must also be considered that the concept of *version*, although prevalent, is not defined in every resource from the management domain. In those cases the identity definition of those resources will be determined just by its *type* and *name*; the latter one enclosing the individual information of the resource.

Figure 36 shows the updated resource model and the extended type for representing resources with version information. *VersionedResources* have a *version* attribute, of type *rVersion*, for identifying members of a family of resources with similar characteristics. For *VersionedResources*, version must be evaluated as part of its identity, as two *VersionedResources* with the same name and type, but different versions are granted to be separate entities.

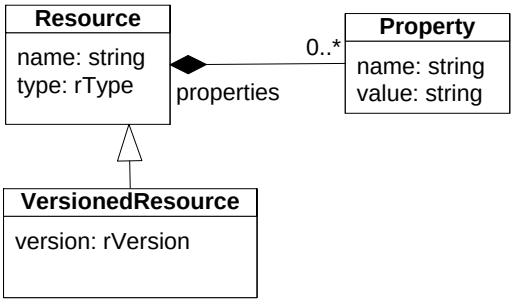

Figure 36 Versioned Resources model

The exact format of the *rVersion* type can't be comprehensively defined at this abstraction level, because the actual format of versions differs depending on the specific domain of the work, the characteristics of the resources (its type), or the company policy, each concrete format shares some key characteristics. A thorough analysis of the main versioning strategies and its most important differences is presented at [12]. From that comparison it can be extracted that any version space can be represented as an acyclic graph. As an additional restriction, I will consider the domain of possible *rVersion* values to be a partially ordered set. This way, version ranges can be defined, which are intervals representing subsets of the complete *rVersion*, defined by lower and upper limits. When referring to intervals, I will use traditional mathematical notation to represent version ranges, with the form *[(lower_bound,upper_bound)].*In the following definitions, I will use the following notation to specify that a *version* v is included in a *VersionRange* range $:v \in range$

In order to close this heuristic I will provide an updated definition of the RESID base check, detailing all the possible checks on the three definitions.

RESID(r, idCheck)

***Description*:**	Checks that the resource identity belongs to the identified subset
***Checked element*:**	$r \in RuntimeResource, r.uid = \{name, type, version$
***Additional arguments*:**	$idCheck = \{nameCheck, typeCheck, vRange\}$
***Formula*:**	$RESID(r, idCheck) = \begin{cases} true, & r.uid = idCheck: \\ & r.name = nameCheck \ \wedge \\ & r.type \supset typeCheck \ \wedge \\ & r.version \in vRange \\ false, & r.uid \neq idCheck \end{cases}$

4.4.5. Conclusions on the heuristics simplification

Along this section the initially proposed information models have been streamlined by the defining of three fundamental simplifications. After those changes have been applied the resulting information model can be automatically analyzed and the contained information can be more easily interpreted.

First, the set of potential elements which needed to be analyzed has been greatly reduced to two well-defined finite subsets, the configuration and the LRB. Only those defined resources are relevant for the management architectures. After that, the initial resource model has been refined, by providing a complete definition of what constitutes the identity of a resource, represented by three attributes (*name*, *type*, and *version*), common to the main information model standards. These concepts also simplify the management operations, as similar resources can be detected and grouped, reducing the complexity of the operations. Finally, after those changes have been proposed, a complete function for evaluating the stability of an environment configuration has been defined. The configuration stability function is based on simple base checks which can be easily evaluated just from the actual state of the environment.

4.5. Model definition

The previous sections have defined a solid foundation for defining information models, complemented with several heuristics which address the potential complexity of the management activities. The proposed abstractions allow characterizing the runtime information and its structure, as well as the logical assets related to the system. On top of the elements definition, a series of checks have been defined which can be evaluated to determine the stability of a configuration. The concepts, abstractions and heuristics proposed are not specific to enterprise environments, being applicable to any management information model.

Over the remainder of this chapter, I will propose a specific information model that builds upon these established concepts and applies them to the domain of enterprise services configuration and deployment. As it is the case with the base abstractions, the specific model will also be aligned to the base structural aspects of the D&C, CIM, SDD and WSDM standards which were described at the State of the Art analysis.

Building on those base concepts, I will detail the fundamental elements of the management information model. Starting from the concept of resources previously described, I will identify special classes of resources which constitute the main domain concepts for the management of distributed enterprise services. The model will cover both runtime elements, describing the state of the management environment, and logical elements, representing the configuration resources and characteristics which can be instantiated by the management architecture. Both definitions will build upon the final iteration of the resource concept as it was depicted in Figure 32.

I will start by defining how to characterize the central elements of the management architecture: the services. The logical information model defined in the next section defines how they are represented according to the base modeling abstractions, so that both their main characteristics and their requirements for working correctly at the runtime environment are sufficiently represented. Once these concepts have been properly established, I will propose the specific information model for the elements of the runtime environment. The runtime model must support the definition of distributed systems, containing several types of servers (e.g. application servers, business rule managers, database management systems and enterprise orchestration services), as well as the structure of the environment, and the available characteristics provided as the execution environment for the services. The specific model will build upon concepts such as resource identity and types to be applicable to characterize the huge variety of environments present in this domain, without the need to define a specific subclass for each new supported resource.

4.5.1. Logical Service Management Model

The logical service management model will characterize every logical resource which must be considered over the deployment and configuration activities. Services are the central elements of this model, which will also characterize the artifacts providing them to the execution environment, which will be called from this point onwards deployment units. This model provides a deployment and configuration perspective of the logical artifacts obtained from the service development process. These elements constitute the cornerstone of the logical model. As they are logical entities, the defined elements won't appear directly as part of the runtime environment, but can be instantiated on it. Figure 37 shows the main elements of the model.

The model defines a specific subclass of *Resource*, the *DeploymentUnit*, for representing the artifacts which can be physically deployed and configured for the environment. A unit's *name* indicates the symbolic name of the artifact, but does not correspond with the specific file name. Nonetheless, that information can easily be specified as a property "*packageName*" of the *DeploymentUnit*. The *type* of a *DeploymentUnit* indicates the kind of packaging of the artifact, which allows identifying compatible execution environments for the units (typical packaging types at an enterprise environment comprise WAR and EAR JEE deployable files, OSGi JAR Bundles). Finally, units have a *version* attribute. It has already been discussed how this mechanism is integral to the software elements, so it is clear that information must be present at every deployment artifact. This way, the identifying elements of a deployment unit are the trio of *name*, *version* and *type*. Finally, by default DeploymentUnits are only visible locally. The exported resources from the unit will be the ones which can be accessed from the rest of the environment.

Version has been defined at *DeploymentUnit* level instead of at the base *Resource* level because, as it was mentioned during the definition of the version identity, it cannot be applied to every resource (e.g. a TCP port, or a graphical card). However, it must be supported for the base Resources which require version information. For those elements, instead of defining a specific subclass which would complicate the model, it will be enough to define a resource property named *version*. This way, restrictions over the *Resource* version will be evaluated with this property.

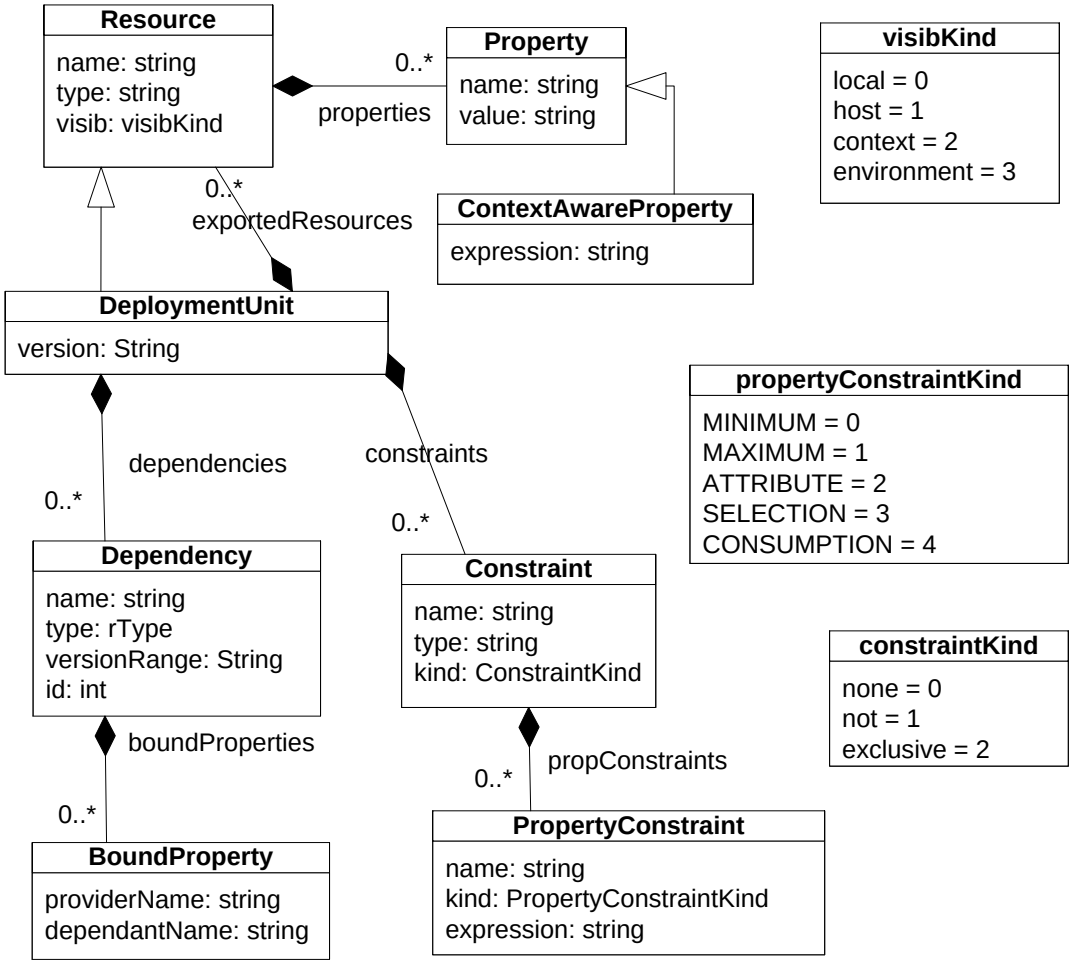

Figure 37 Deployment Unit Model

A *DeploymentUnit* aggregates several exported resources, such as services, libraries, web interfaces, and business logic components, which will be made available to the rest of the environment if the unit is instantiated. These elements are modeled as resources. The flexibility of the *Resource* model and the typing classification allows a rich characterization of different elements through the use of the same common concepts. Each exported resource will be accessible to different parts of the environment depending on its visibility. A *DeploymentUnit* can only aggregate base resources, no subtypes are allowed (e.g. a *DeploymentUnit* cannot have another *DeploymentUnit* as an exported resource). Finally, the unit contains a set of properties that will be used to include configuration details of the unit for its correct functioning, which would not fit as exported resources. These properties reflect the

internal configuration of the component (e.g., the parameters of an authentication LDAP server).

4.5.1.1. Deployment Unit Requirements

Most *DeploymentUnits* aren't independent entities; they collaborate instead to provide the final services. In addition to inter-unit collaboration, some *DeploymentUnits* also require some capabilities from the execution environment for a correct runtime operation. This can be expressed by requiring the availability of specific resources as part of its execution context. These two relationships of a *DeploymentUnit* with the rest of the system correspond to the previously described stability concerns of dependency and constraint. In order to take into account these specific requirements of the units, they must be explicitly reflected in the model. It is interesting to mention that *Dependency* and *Constraint* concerns are evaluated over different partitions of the environment. Dependencies are satisfied by the instances of other logical *DeploymentUnits*. On the other hand, constraints are imposed and evaluated against the execution environment, which, as it will be described in the following section, is composed by physical and logical resources at the lower layers that support the execution of services. As DeploymentUnits are logical resources, they can only express the requirements for both types of conditions being correctly met, although the actual evaluation will only be possible over their runtime instances.

The format for expressing unit dependencies is very straightforward. For each requirement, the *DeploymentUnit* will add a *Dependency* declaration, which contains sufficient information to identify the required resource. The *Dependency* identifies the valid candidates by declaring a matching filter composed by the *name* of the required resource, its *type* and optionally a single *version* or a range of valid values. If no version restrictions are specified version information will not be evaluated. For the representation of version ranges I will adopt the notation for specifying version ranges proposed in the OSGi dependency model [86] - e.g. [1.0.0, 2.0.0) - which mimics the traditional mathematical notation for intervals [92]. However, if it was deemed necessary other formats could be defined instead. In addition to the filter attributes, each *Dependency* from the *DeploymentUnit* will have a unique *id*, which can be automatically be assigned as a self incrementing value starting from 1.

The *DeploymentUnit* dependency model follows a SOA approach, decoupling the actual dependency matching (required - providing resource) from the enclosing units. This approach provides additional flexibility to the packaging of units and services, focusing instead on the actual services provided to the system. Direct dependencies between units are also supported as a *Dependency* declaration can also be matched by *DeploymentUnits*, which are also resources. An important quality of *Dependency* evaluation is that both the dependency definition and the potential search space for its satisfaction are contained in the logical set of resources. This fact enables to reason about dependency requirements without considering the runtime configuration. For instance, it is possible to analyze whether a *DeploymentUnit* from the LRB can see all of its dependencies satisfied by other logical units, thus evaluating the completeness of the LRB with respect to satisfying unit dependencies. A unit can declare any number of dependencies, so in order to differentiate among the multiple *Dependencies* of a *DeploymentUnit*, a different *id* will be assigned to each declaration.

Constraints are declared in the *DeploymentUnit* definition with a similar mechanism. A unit can define one or more *Constraints*, each one of them requiring a specific resource to be present (or not) at its runtime execution environment. Each *Constraint* identifies the valid resources, by defining a filter consisting of *name* and *type* comparison. *Version* restrictions are not included as in the general case environment resources do not contain version information. The main difference between *Dependency* and *Constraint* identification lies in the search space. *Constraints* can only be satisfied by the resources from the execution context of the instantiated *DeploymentUnit* (that is, the container, the node, or the environment).

The *Constraint* model allows a further refinement of the resource identification by also expressing restrictions over the properties of the identified resource. These additional requirements are represented by *PropertyConstraint* elements, which follow the base checks defined in the generic model, based on the defined restrictions of OMG D&C model [83]. *PropertyConstraints* simply declare the name of the property, the kind of evaluation to be done (minimum, maximum, attribute, selection, consumption), and finally the expression value to be compared with (e.g. a typical *Constraint* would identify a resource of type "*hardware*.*processor*" with an additional *PropertyConstraint* over the property "*speed*" of kind "*minimum*" and expression value "*2000*"). In case it was considered necessary, this mechanism can also be used to add *version* identification to a *Constraint* specification.

A *Constraint* belongs to one of three different types, which define the relationship between the requiring unit and the environment resource, determined by the *ConstraintKind* attribute. *DEFAULT Constraints* require that the identified resource be present at its execution environment. On the other hand, *EXCLUSIVE Constraints* further restrict this relationship by demanding that no other unit can access this resource. Finally, *NOT Constraints* define incompatibilities, meaning that the identified resource must not be present at the execution environment for the *DeploymentUnit* to be stable.

4.5.1.2. Automated unit configuration support

Up to this point, the described characteristics of the *DeploymentUnit* metamodel allow to express exactly the resources provided by the *DeploymentUnit* to the environment if it is instantiated, and the stability requirements for its instantiations to work correctly. The final concepts introduced in the model go a step further in the automation of the basic unit configuration. This is achieved by linking configuration values to the actual resources which satisfy the *Dependencies* and *Constraints* of the unit. This enables the automatic configuration of an instance of a *DeploymentUnit* (by means of modifying the value of its properties) adapted to the characteristics of the rest of runtime elements.

As regards dependency-based configuration, *DeploymentUnit* definitions can express *BoundProperty* values, which were also introduced in the general resource model (the relationship between dependency and bound property is shown in Figure 34). They allow expressing how the configuration of the dependant unit is obtained from the satisfier. This way, the *value* of a *Property* is substituted by the *value* of another *Property* from the bound resource. A typical example of this type of automatic configuration is providing the URL of a remote service to the unit which has been bound to it so it can access it correctly. In order to reflect them in the model, each unit *Dependency* can declare any number of *BoundProperties*, which will be evaluated against the unit which satisfies the dependency. The *dependant* field

marks the *name* of the local *Property* whose *value* will be automatically configured and the *provider* field indicates the *name* of the source property for configuration. The next picture shows an example of this automated configuration. If unit A contains a dependency D that is satisfied by the unit B, and D contains a *BoundProperty* with "dependant" p, and source "q", the resulting configuration operation would be: *A.p.value = B.q.value*. As dependencies can be transitive among several units it is possible that several *BoundProperties* are linked, with the value being propagated over the dependency chain. In those cases, it is important to apply them in the correct order, in order to transmit the original value over the dependant elements.

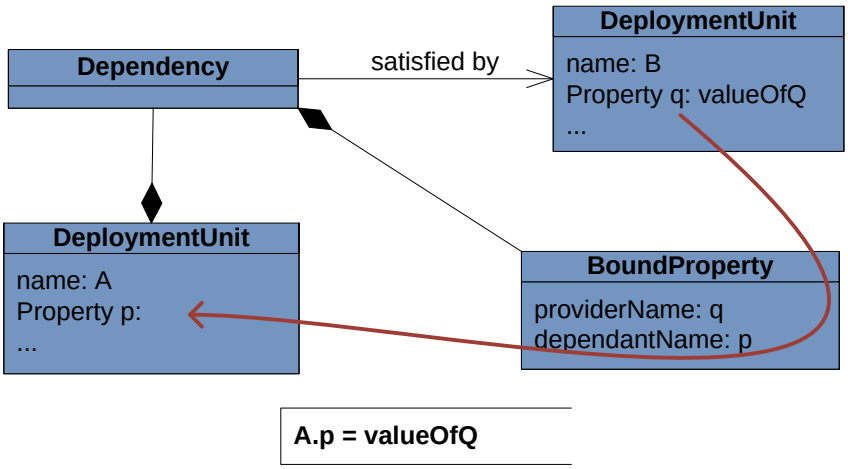

Figure 38 Bound Property Configuration Example

In addition to *BoundProperties*, there are also several configuration aspects which can be automatically extracted from the execution context where the unit has been instantiated. These configuration aspects are supported in the model with the *ContextAwareProperty* concept. Although the specific elements of the runtime environment model will be defined in the next section, it is clear that the environment model will contain runtime resources which represent the nodes and containers where *DeploymentUnits* will be instantiated. Those elements contain in their description the relevant context information that will influence many aspects of the unit and the provided services. As an example, any remote service running at the environment, will have a connection URL which depends on aspects such as the IP of the physical node, or the service port which is reserved by its container. By providing a mechanism to identify and obtain those context-dependant configuration parameters, the part of the unit configuration which depends on the context can also be automated. Before simply explaining how this kind of automated configuration has been integrated in the model, I will briefly discuss the most relevant concerns which must be taken into account.

Up to this point, resources have been considered first on an individual basis, and later on linked through several relationships (composition, hosting, and dependency). However, In the case of enterprise environments, there is a certain type of configuration which should be shared by every resource of the environment, or at least a hierarchic slice of it. I will refer to this type of configuration as Global configuration. This concept will be supported in the model by means of value inheritance from the properties defined in the execution context resources. This way, *Property* values from parent resources of the hierarchy can be accessed by runtime unit instances and become part of its configuration. This concept allows representing both

global configuration parameters, which are applied over the whole environment, or specific context information, such as the base network address of all the node resources.

The inheritance of property values over the containment hierarchy can potentially lead to conflicts, in cases where several resources from the hierarchy define a *Property* with the same *name* (although in real scenarios this situation would rarely occur). For avoiding model uncertainty, in those situations the more specific *value* of the *Property* will always take precedence over the ones appearing farther in the resource hierarchy.

In addition to that, simply substituting the *value* of a local *Property* with another one from the context can prove to be too limited to solve the needs of context-based automatic configuration. As an example, in order to construct the *serviceURL* of a deployed Web Service, it is necessary to retrieve both the IP and the listening port from the execution context of the deployed service. Both values must be combined in order to obtain the required configuration value. These requirements have been supported by the model by defining a specific element which can be automatically configured based on its execution context. This approach was not extended to *BoundProperties* as no scenarios were identified where the property combination aspect would be necessary.

A *ContextAwareProperty* is an extension of the base *Property* class which can only appear in *DeploymentUnit* definitions (either in the base unit resources or in the properties of the composed resources). On top of the inherited name and value fields, these special properties contain an expression that mandates what the Property *value* should be depending on all the context information. For establishing the syntax of these expressions and the computing approach, the string variable substitution [31] model supported by Ant [46] has been selected. Ant defines a mechanism for property automatic derivation, including inheritance from multiple files, and the same conflict resolution mechanism described earlier. The similarities between the two concepts clearly indicate the feasibility of adopting this approach for context aware expressions; Ant properties are syntactically identical to the properties of our model. On top of that, in many cases they reflect the established global configuration, the current context information for a specific operation, which is similar to the execution context, but without the multiple nested scopes present in the model case. Finally, the domain of application of Ant scripts covers deployment and configuration operations, thus these abstractions have already been successfully applied to enterprise service management.

An *expression* is a string template, which represents the final value of the property. Inside the expression one or more variables will be defined, identified by the character sequence *${variable}*. There is a minimum of one variable for each expression (else it should just be a regular property, as there would be no context dependence), and no established maximum. The *name* of each variable identifies the *name* of a *Property* which will be present at the execution context of the declaring unit. As an example, the context-based configuration of a "*serviceUrl"* previously described, could be automatically supported by defining the following expression: "*http://${ip}:${port}/ContextPath/Service*". If instead of a composite configuration a unit simply requires a global configuration attribute it can be retrieved by defining the same property with the value*: commonLogChannel = ${commonLogChannel}*.

The following picture shows an example of the complete process for context-based automatic configuration. The topmost resource represents an instance of a *DeploymentUnit*, which

contains in its logical definition the shown expression for automatically configuring the correct value of its *serviceUrl* property. In order to resolve the two variables (*ip* and *servicePort*), the *Property names* of the unit execution context (in this case, composed by the remaining two resources) are examined. As there are no name conflicts in this simple scenario, the value from *servicePort* is obtained from the *container* resource, and the value for *ip* is obtained for the *node* resource. After the process has been completed the value for the automatic configuration is obtained, as it is shown in the picture.

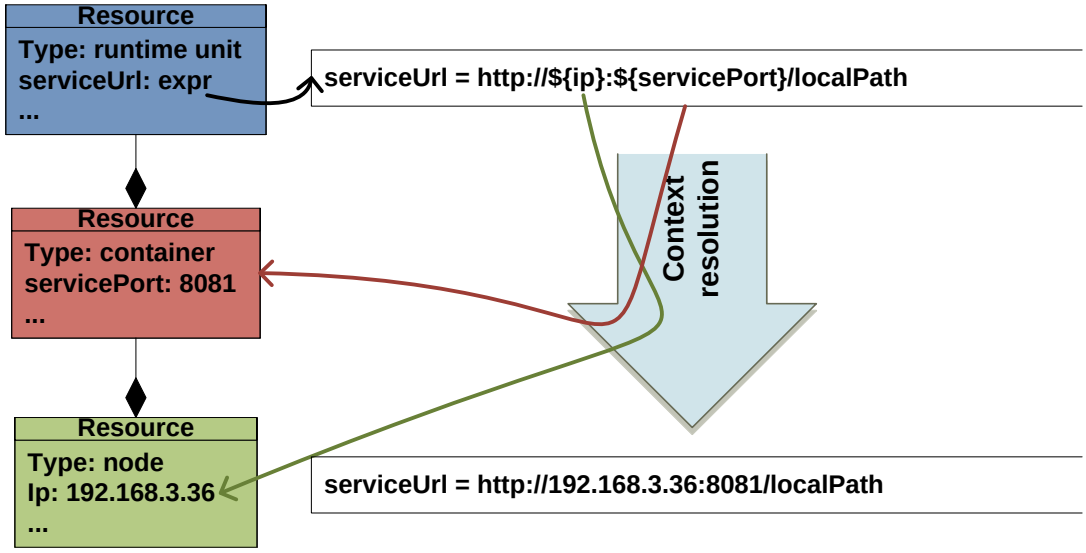

Figure 39 Context Aware Property Configuration Example

The defined *DeploymentUnit* model allows a rich characterization of both the deployment-relevant elements (a deployment unit, its configuration, its requirements to run correctly), as well as the design-time artifacts provided (services, libraries, components and so on). Because of that, it acts as a bridge between the software architect view, of services, and the runtime stability constraints which must be manually managed by IT administrators.

The following section complements this logical view with the runtime environment model, which defines the runtime resources constituting the environment, ranging all the way from the general environment to the instantiated deployment units.

4.5.2. Runtime Unit Lifecycle Definition

Before providing definitions for the runtime resources which will appear at the environment, I will model the lifecycle of the instances created from the main logical definitions: the *RuntimeUnits*. Those elements represent a realization of a *DeploymentUnit*, over an element of the runtime environment. In addition to the logical information, *RuntimeUnits* also have state, showing whether they are running correctly or are not started. However, from a generic management view, it is difficult to define the set of states and the lifecycle of *RuntimeResources*, as each type of container has their own interpretation of these elements. The states defined by this model must capture the essential, shared concepts among them. As the starting point I have selected the deployment lifecycle of one of the most relevant standards in the enterprise server domain: the OSGi specification [86]. Currently every major Java EE implementation is built over this component model, with several solutions proposing it as the new packaging standard for applications. On top of that, the lifecycle for OSGi

components is one of the most complete ones, detailing the possible transitions and states of their management elements. The left-hand side of the next picture shows the complete lifecycle adopted by this specification:

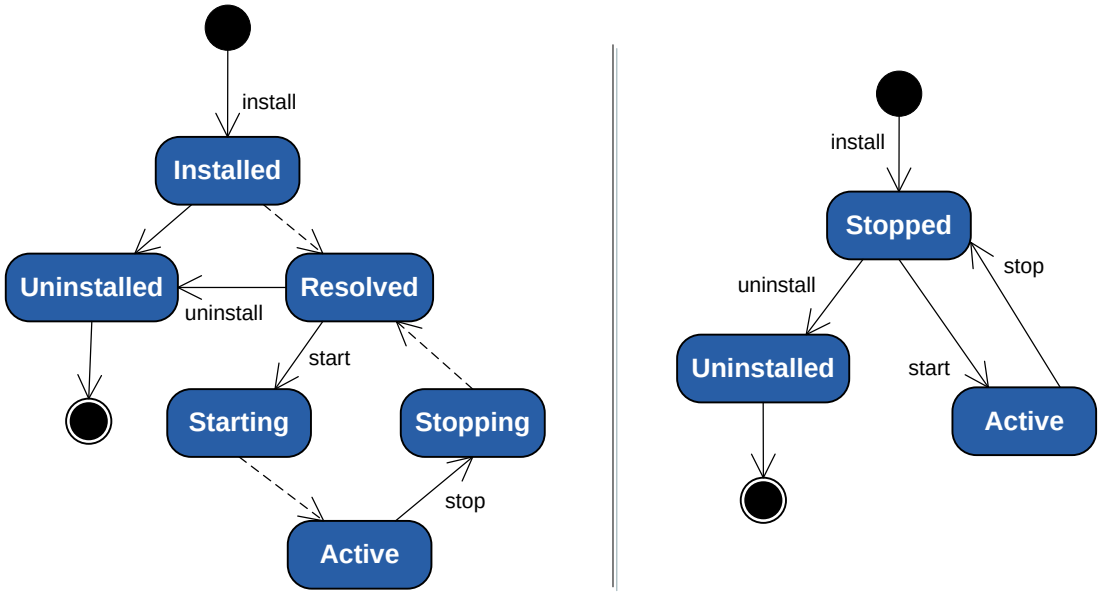

Figure 40 OSGi component lifecycle and proposed shared lifecycle

In an OSGi container each unit beings its existence with the installation operation and can potentially go through six different states. Transitions with actions on them and solid lines are initiated by the manager, whereas transitions with dashed line arrows are automatically processed by the framework when certain internal triggers occur (e.g., once all internal dependencies of a newly installed unit have been satisfied, the framework changes its state from installed to resolved).

From a management perspective, transition states such as starting or stopping are only relevant over a short period of time, after which a stable state will be reached. Moreover, the manager does not control these transitions, only being able to operate on the final states. These fine-grained details are not present at the lifecycles of other platforms. However, the fundamental notions of installation, activation, active and stopped states, are a well-accepted base for the great majority of enterprise containers. Because of that, the general lifecycle for the *RuntimeUnits* of the defined model will be the one defined at the right-hand side of Figure 40. This state diagram is in fact identical to the OSGi model, after the removal of the transitory states. *RuntimeUnits* are stopped when initially installed, and must be activated for them to perform their functions at the environment. In addition to that, every management operation which modifies the unit configuration must be applied with the unit in the stopped state, as 'hot configuration' is not supported by every type of services container. This way, a reconfiguration activity over an active unit will be preceded by a stop command, and followed by a start command.

4.5.3. Runtime Environment Model

The logical deployment units are deployed at an execution environment, comprising from the application servers to the hardware elements. It is clear that in order to enable automated

service management operations a model of the runtime environment is necessary. The proposed information model addresses all those concerns while focusing on the key aspects for enabling an automated service configuration. The main reference for this runtime model is the D&C target model and its refined version proposed in [93], although it has been modified to be completely integrated with the *DeploymentUnit* and *Resource* definitions provided earlier. In addition to that, in the same vein as it was mentioned in the logical model, I have focused only in the fundamental concepts for service management (namely the services, its relationships in the environment, and its execution context). In case additional information must be added to the environment model, it can easily be extended to integrate those concepts. The next picture shows a high-level view of the elements which constitute the runtime model. It can be seen that, as it was the case with the logical model, the definitions are built over the base concept of *Resource*, with a set of subclasses constituting the runtime hierarchy. Over the next paragraphs I will explain the concepts behind each model element, starting from the higher-level element of the hierarchy (the environment)

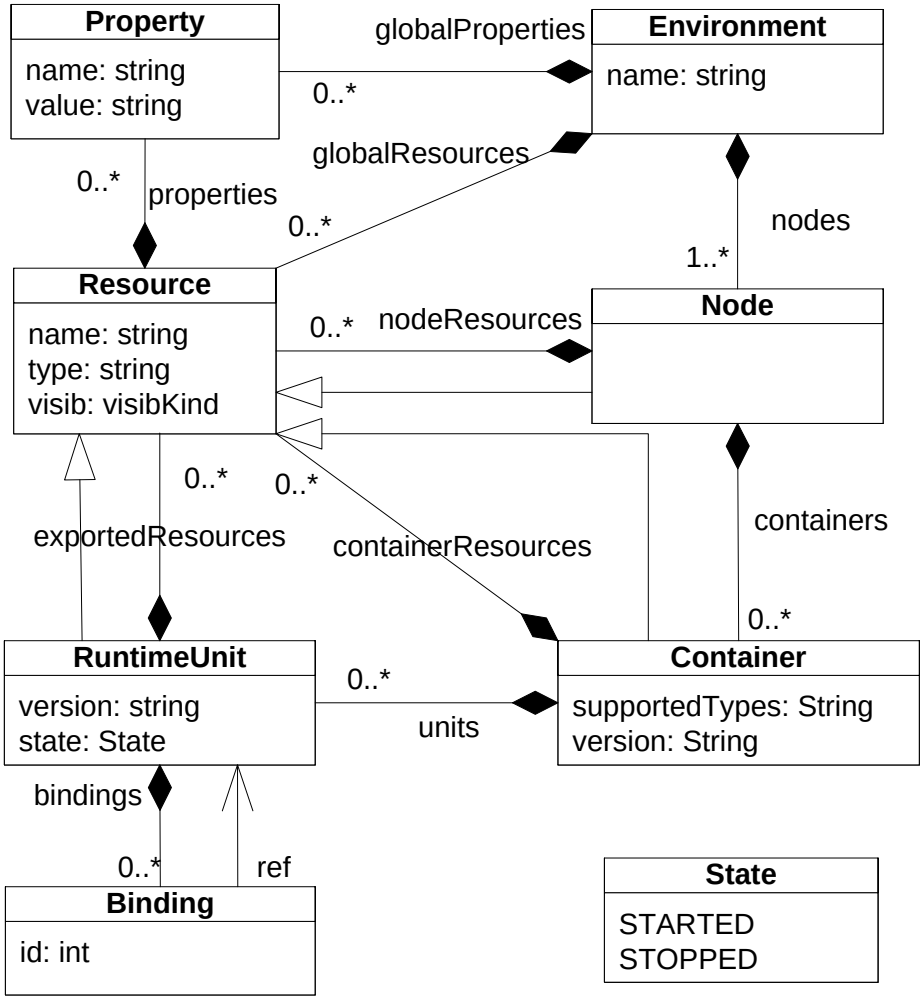

Figure 41 Runtime model

The root element of the runtime model is the *Environment*. An environment must have a uniquely identifier for its base name, as one management system can manage several environments, which are completely independent among them. The environment is composed by at least one resource with computing power: a *Node*. In addition to that, it can also provide global resources which are available to every application in the domain (such as an LDAP

authentication provider). These resources are not directly controlled by the management architecture, but they appear in the management view as they can satisfy unit *Constraints*.

In addition to those global resources, the *Environment* object defines the current global configuration through a series of properties. These elements, which were briefly mentioned when detailing the context-based configuration capabilities of the logical model, are mandatory for the internal management of enterprise infrastructures, as they simplify the configuration of several aspects by making then common for the complete environment. Deployment units which require those values to work correctly simply declare them as *ContextAwareProperties*.

Nodes are the basic elements of the environment model, representing resources with computing capabilities. They are specializations of resources that in the general hierarchy are directly hosted by the environment. *Nodes* are directly or indirectly connected to the rest of the network, having access in principle to any other resource from it (unless the visibility does not allow it). A *Node* resource comprises all the hardware, firmware and low-level software layers of the device (such as the operating system), abstracting the specific components, libraries, communication channels and devices as *nodeResources*. On top of that substrate, a node hosts any number of *Containers*. The name of the node must be unique over the environment, allowing an univocal identification of the contained elements. For the purposes of service management, there is no difference between real physical nodes and virtualization nodes.

A *Container* is the base execution platform for *DeploymentUnits*; the specialized resources where units are instantiated. *Containers* have a *name*, which must be unique over the environment, and a container-specific *type* (examples include "*container.database.jdbc.oracle*" or "*container.jee.websphere*"). In addition to that, containers must be versioned, as they are software elements. The same reasons for versioning *DeploymentUnits* also apply to these elements. *Containers* provide a set of platform services to the host units, which are expressed as a set of container resources (in a typical application server these resources would be Datasource connections, JMS queues, or external system connectors). Additionally, container properties contain additional configuration details, such as the service port of an http server.

The main function of *Containers* is to host the runtime instances of the *DeploymentUnits.* These instances are explicitly represented in the model as *RuntimeUnits*. However, clearly not every unit can be instantiated on any *Container* from the environment. It is necessary to define a mechanism for matching compatible units and containers. This is managed at *Container* level through the *supportedTypes* attribute, which specifies which types can have the deployed *RuntimeUnits*. This information is equivalent to the hosting stability conditions which were previously mentioned at the general model. In addition to that, more specific requirements for *Container* compatibility can be declared by the *DeploymentUnits,* through the definition of *Constraints* for the container resources.

The defined runtime model restricts the possible resource host relationships by specifying resource subtypes as the only valid nested elements. This way, *Nodes* can only be hosted by the *Environment*, *Containers* are always hosted by *Nodes* and *RuntimeUnits* are provisioned over the *Containers*. However, there are some enterprise environments were containers host additional containers (e.g. a domain where the system manages both operating system-level

packages, and web applications deployed over an application server, or another one where an OSGi container is deployed as an application on top of the regular application server). Although initially it may not seem the case, those scenarios can be supported by this model, by means of defining both Containers to be hosted by the Node containing the first. Although this management view does not exactly represent the physical topology, at service-based management it would be equivalent. Avoiding the support to recursive hosting of the same type greatly reduces the space of potential configuration, consequently simplifying management operations.

As it has been mentioned, the instantiation of a *DeploymentUnit* in a *Container* from the environment creates a *RuntimeUnit*. The identity information of the logical definition (*name*, *version* and *type*) is also shared with all the *RuntimeUnit* instances. As it is the case with logical definitions, the *RuntimeUnit* internal information is described through properties and exported resources. *RuntimeUnits* also have a state, which follows the previously defined lifecycle.

In addition to the information provided by the *DeploymentUnit*, these elements also included attributes for specifying the *Constraint* and *Dependency* stability constraints of the instantiated *RuntimeUnits*. That information is not replicated at the *RuntimeUnits* as it is already present at the logical repository, and thus it can be retrieved without unnecessarily increasing the footprint of the complete environment information. *Constraints* must be supported by the execution context of the *RuntimeUnit*, but no additional information must be included in them. On the other hand, as regards *Dependencies*, it has been already mentioned that they can potentially be satisfied by multiple units. As each logical definition can appear at the environment multiple times, it is clear that the number of possible satisfiers for each dependency increases. However, at the runtime level, only one *RuntimeUnit* can satisfy each *Dependency*. This way, if no additional information is defined, it is not possible to know exactly how the *Dependency* is satisfied at the environment. As this knowledge is necessary in order to correctly evaluate the impact of changes to the environment, it will be explicitly reflected in the model.

The *RuntimeUnit* model encapsulates how dependencies are satisfied as a set of *Bindings* to other *RuntimeUnits*, which are the realization of the logical requirements. Each logical *Dependency* must result in a runtime *Binding* to another unit, both of them sharing the same *id* in order to identify the corresponding dependency definition of each one. Each *Binding* contains a reference to the satisfying unit. Multiple *Bindings* can be defined to the same unit, each one because of a different *Dependency*. Finally, it must be clarified that *Binding* relationships can only occur (both source and destination units) whenever the unit is at the active state. *Bindings* improve the obtained information from the environment, defining an overlay structure over the runtime containment hierarchy. This is vital to correctly apply any changes over already existing *RuntimeUnits*.

As it was the case in the general resource model, the structure of a runtime environment configuration is a hierarchical resource structure. This specific model streamlines the hierarchy through the use of resource subclasses. This presents a clear management view of the environment, with specific elements representing the key concepts of the system (nodes, containers, units). The structure does not allow the recursive hosting of any of the elements, which further simplifies the model. The next figure depicts a tree representation of a

configuration, where the different resource subclasses are allocated in the respective layers. This view of the managed elements is very simple, although it must be mentioned again that there is an additional dependency overlay defined by the *RuntimeUnit Bindings*. The combination of both types of runtime resource relationships enables to estimate the real impact of changes occurring to an element of a distributed service architecture.

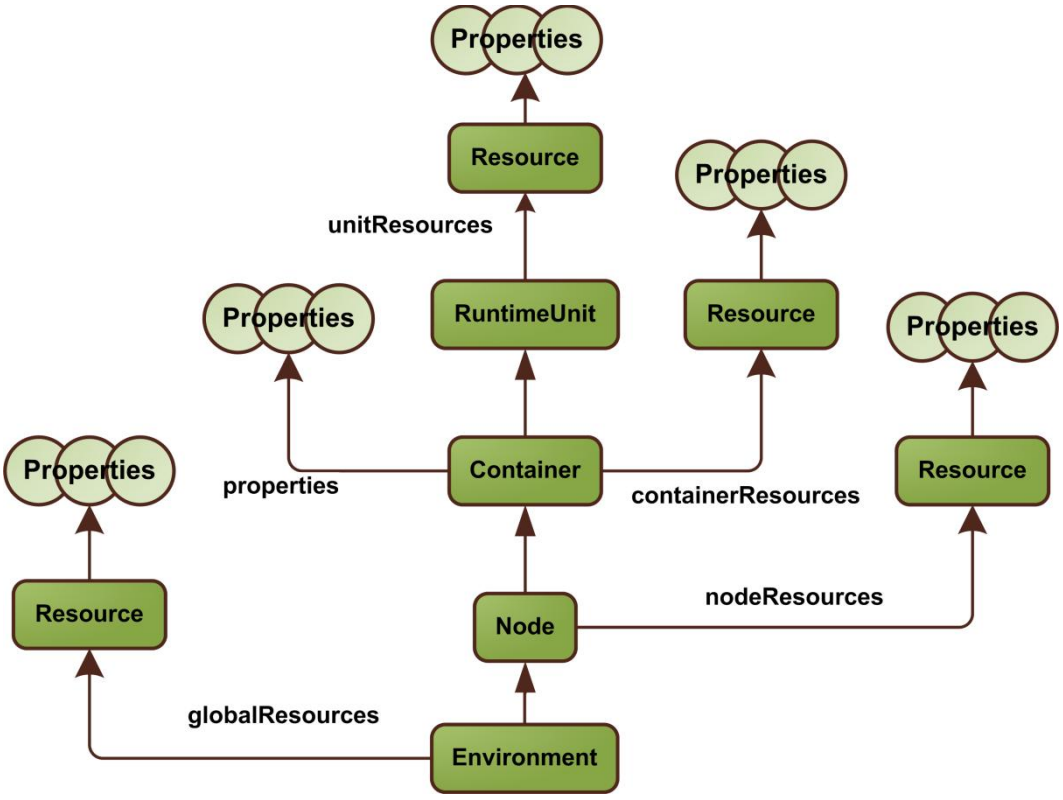

Figure 42 Tree view of the runtime model

4.5.3.1. Configurable Container Resources Definition

A service management system must control not only the *RuntimeUnits*, but also their execution platform. Because of that, some management operations will not be ordered on *RuntimeUnits* but on *Containers*, altering their current configuration. *Container* identity is not configurable – composed by *type*, *name*, and *version*, neither is the list of *supportedTypes*. On the other hand, *Container properties* can be modified. Moreover, the container *resources* must also be controlled, potentially adding or removing them with the management operations. However, in order to create new resources at the *Container*, similarly to the *RuntimeUnits*, these elements have to be defined previously at the logical space. Because of that, I will define a new logical element, the *ContainerResourceConfiguration* (CRC), which contains all the required information for adding a new runtime resource to an existing Container.

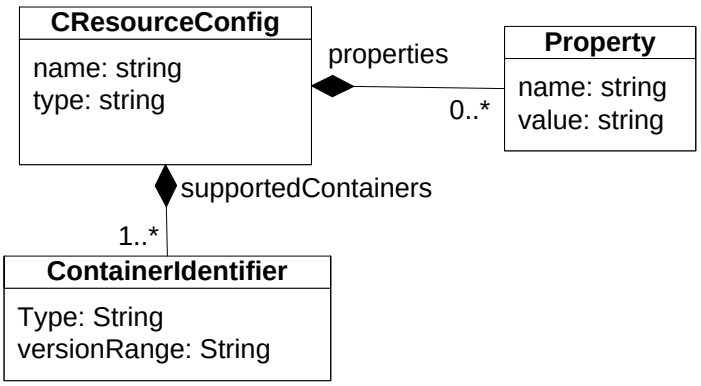

Figure 43 Container Resource Configuration Model

The Figure 43 shows the main characteristics of this new element. A CRC is a logical template that can be instantiated to add a resource to an existing runtime container. The CRC contains several attributes which will be shared by the resources created based on it: the type of the created resource, the initial set of properties, and optionally the resource name. The name is optional as there are numerous cases where more than one resource of the same *type* can be present at the same container (such as Datasources). In those cases the *name* differentiates one resource from another, and will be provided as an additional argument of the operation. Finally, the CRC must identify the set of compatible containers, where the specific instances can be created. This is declared by elements of the *ContainerIdentifier* type, which similarly to *Dependency* definitions, filters compatible containers by matching their identifying properties, in this case the name and the version (with the possibility of defining a range of compatible instances).

After the models of both the logical definition and runtime instances of deployment units have been defined it is important to clarify exactly the relationships between them. These connections have been mentioned over the previous paragraphs but they will be clarified at this point because of its importance for management activities. The next picture shows a side by side comparison of the two models, detailing the structure of each element, and the correspondence between the fields. In order to explain these relationships I will describe them from both perspectives (first from the logical side and then from the runtime side).

4.5.3.2. DeploymentUnit to RuntimeUnit Relationship

After all the runtime elements have been properly defined I will provide a greater explanation A logical definition of a *DeploymentUnit* can be instantiated at a runtime *Container*. The resulting element is a *RuntimeUnit*, which inherits several attributes from the logical definition. First, the identity is transferred to the runtime instance; this way, *name*, *version* and *type* will be the same for both entities. In addition to that, the internal information of the unit (the list of properties and resources) will also be initially shared by the instance. Finally, for each *Dependency* of the logical definition, there will be one *Binding*, whose id will be equal to the one of the *Dependency* (allowing to retrieve the requirements of the bound resource). The remaining information of the *Dependency* describes several stability conditions that the bound resource must satisfy in order for the *Binding* to be stable. Finally, defined logical *Constraints* must be satisfied by the selected execution context in order for the runtime element to be created.

From the perspective of an existing *RuntimeUnit*, it is also fundamental to be able to trace back to the original definition, in order better diagnose existing runtime instances. In order to do so, the trio of identifying properties allows retrieving from the logical definitions space the complete information about the unit. The *Constraints* and *Dependencies* from the logical definition will be taken into account in order to ensure the stability of current and future configurations.

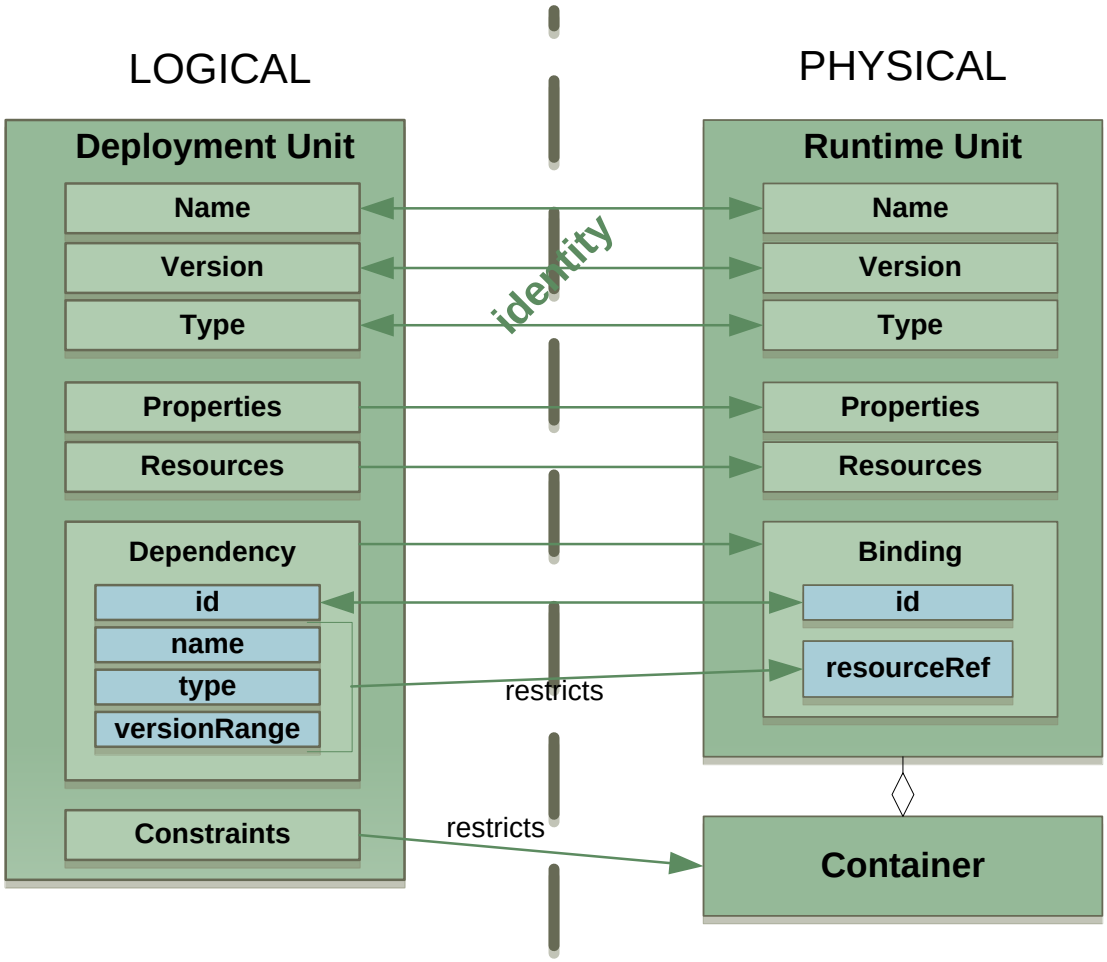

Figure 44 Relationships between Deployment Unit and Runtime Unit

4.6. Enterprise Service and Environment Model conclusions

The presented runtime information model allows a rich characterization of heterogeneous environments without the need of an additional effort to model each particular case. This has been made possible thanks to the abstractions that were identified at the general resource model. These characteristics were also shared by the model proposed in [93], which has been one of the main references for these definitions. On top of that, the runtime definitions are linked to the logical concepts, covering both the deployment units and the services. Relationships between related elements from both sets, (i.e. *DeploymentUnit* and *RuntimeUnit*) have been explicitly identified and explained, allowing to simultaneously reason about all the related information.

The model not only allows representing the information of the domain, but also defines the concept of configuration stability, which can be automatically evaluated from the instances of

the logical and runtime models. The definition of two models also enables a separation of concerns with the defined information: the stability concerns are expressed as logical definitions, and the actual configuration values of the environment are contained at the runtime model.

Finally, the defined model also enables a partial automation of the configuration of the managed units and services. On one hand, global configuration values can be established and retrieved from the environment information, reducing the repetitive tasks of applying them specifically to each element of the runtime configuration. On the other hand, the concepts of context aware properties and bound properties allow further automate the detailed configuration of the runtime units, deriving the actual configuration values from the dependant and required elements which are part of the environment.

5. Service Change Management Foundations

The objective of this chapter is to completely define the two main entities for management purposes: the managed environment and the management system, enabling an automated reasoning over the status of the domain and the required operations to correct detected problems. In order to achieve that, the existing information model abstractions will be used as the base to describe both concepts. This way, I will start defining a set of abstractions to explicitly model what functions must be supported by the managed environment, with the corresponding means of an automated verification. In addition to that, I will also propose a definition for the role of the service change management system, based on the concept of domain changes. Finally, I will propose an algorithm, which can automatically support the main functions of the management system based upon the previous modeling abstractions: identifying the required changes to ensure that the domain is kept at a correct state.

5.1. Managed Environment Definition

A formal definition of environment stability allows automatically evaluating that environment resources operate correctly. However, that validation is not enough for ensuring the correct management of the environment. The managed environment is a business domain entity which exists to fulfill a purpose for the owning organization, by providing a desired functionality. In order to ensure a correct configuration of the environment, the environment must also be validated with respect to offering the required functions and objectives.

Similarly to the stability function, this new set of requirements will be represented by the desirability function, which is evaluated against an environment configuration. A desirable configuration is a configuration from the space of possible configurations which, when applied the desirability function evaluates to true. The configurations satisfying that property constitute the set of desirable configurations.

5.1.1. Managed Environment Objectives definition

The defined desirability formula must check that all defined objectives for the managed environment are supported with the provided configuration. In order to define the general formula I will define specific functions for each specific objective which must be supported by the environment. As the information model is based over the concept of resources, the objective checks will apply to the same elements.

In the previous chapter I introduced a set of base checks which allow expressing rich conditions over the resources from the system. Some of them, specially the local base checks, will be reused in this other context. However, the existing checks are not enough for expressing the management objectives of the environment. Some fundamental concepts are missing, as they only appear when reasoning about restrictions to the environment as a whole, instead of analyzing the stability of individual elements. These concepts are the simplest base checks defined until now, and allow demanding the presence or absence of a specific *Resource* in the environment configuration. As *Resources* are the management abstraction for services, applications, software and hardware elements, they are the actual elements which perform

the desired objectives. This way, declaring the functions which must be provided by the system can be expressed by requiring that certain *Resources* appear at the runtime environment.

The formal definition of these additional checks is provided at the following paragraphs. Because of the duality between logical and runtime resources, separate checks have been defined to allow referencing either type.

EXISTS(rr), EXISTS(lr)

Description:	Checks that the identified resource is present at the runtime configuration
Checked element:	$rr \in RuntimeResource$ $lr \in LogicalResource$
Additional arguments:	$-$
Formula:	$EXISTS(rr) = \begin{cases} true, & rr \in C \\ false, & rr \notin C \end{cases}$ $EXISTS(lr) = \begin{cases} true, & \exists rr \in C, rr\ instanceof\ lr \\ false, & \nexists rr \in C, rr\ instanceof\ lr \end{cases}$

NOTEXISTS(rr), NOTEXISTS(lr)

Description:	Checks that the identified resource does not appear at the runtime configuration
Checked element:	$rr \in RuntimeResource$ $lr \in LogicalResource$
Additional arguments:	$-$
Formula:	$NOTEXISTS(rr) = \overline{EXISTS(rr)}$ $NOTEXISTS(lr) = \overline{EXISTS(lr)}$

5.1.2. Desirable Configurations

In this work I will only focus on functional objectives for the management environment, which determine what must do or not do. Aspects such as how it should do those functions (i.e. SLA levels) won't be considered. This way, two types of checks can be used to specify what objectives must be supported by the managed system. I will define the concept of an *Objective o* (r), as a desired characteristic which must be supported by the environment configuration. Each objective encapsulates one specific check for the environment, which can be either $EXISTS\ (r), or\ NOTEXISTS(r)$.

Once the individual objectives have been identified, it is possible to define the set of objectives *O*, which is composed by all the possible objectives which can be defined over the resources from R.

$$O = \{o\ (r) | \ r \in R\}$$

Similarly to the search space reduction applied to the logical resources set, after defining the general set I will try to characterize a more manageable subset of *O*. Although *O* contains all

the possible objectives, clearly only a fraction of them will be relevant for the managed environment. I will call this relevant subset *Defined Objectives*, $DO \subseteq O$. *DO* contains all the high-level policies and objectives which must be satisfied by the environment configuration. The composition of *DO* can change over time, but as the mechanisms and strategies to modify the contents of this set are the subject of the higher level management domains, I will consider this set not modifiable for the management architecture, analogous to the *LRB*.

However, unlike *LRB*, not any subset of objectives constitutes a valid *DO*. It is possible that some objectives from *DO* are inconsistent among themselves, or with respect to the environment, invalidating the applicability of the correctness function. At the very least, the two following conditions must be met by the *DO* members:

For each objective $o_i(r_i) \in DO$, the identified resource $r_i$must either belong to the runtime configuration or be defined at the *LRB*. Otherwise $o_i(r_i)$ can never be evaluated against the current configuration (because the resource is not even defined), rendering the objective meaningless.

The *DO* cannot contain two objectives with contradictory conditions, which cannot be possibly true simultaneously. This can occur when two conflicting conditions are defined as objectives against the same resource. E.g. *EXISTS(x)* and *NOTEXISTS(x).* Clearly, if that were the case, the correctness function would be wrongly defined.

Once the base checks and functions have been defined, it is possible to define the *Desirability* formula of a managed environment, which will simply consist of evaluating every objective defined at the DO:

$$Desirability(C) = \bigwedge_{i=1}^{|DO|} o_i \, , o_i \in DO$$

5.1.3. Correct Configurations

As it was the case with the stability formula, the correctness formula defines a new subset of the complete space of possible configurations. I will name the set *DC*, *Desirable Configurations*, which is composed by every possible configuration that complies with the high-level management objectives.

However, not every element from this set is a desired state of the system. A desirable configuration containing unstable resources clearly would not be correct. Correct configurations must be stable and desirable at the same time. Correct configurations are the most important ones, as they provide correctly the required functionality. Because of that, I will define *CC*, the set of correct configurations, which can be mathematically defined as $CC = SC \cap DC$. Next picture shows a graphical definition of the *CC* set, as the intersection of the set of stable and the set of desirable configurations.

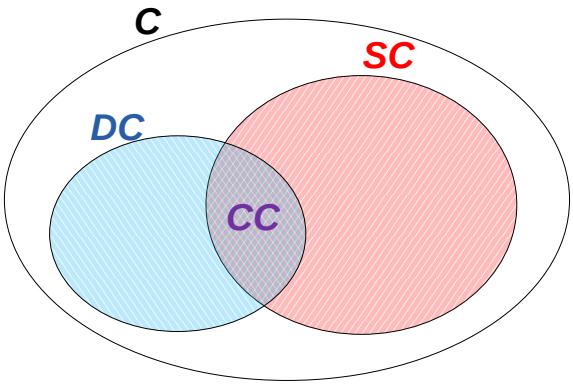

Figure 45 Graphical Definition of the Correct Configurations

Because of the relevance of the set of correct configurations, I will provide a way to directly identify its members. In order to do so I will define the correctness function, which when applied to an environment configuration and an associated *DO*, returns true if the configuration is at the same time stable and desirable. As those conditions have already been defined as functions, the correctness formula will simply be a conjunction between those two.

$$Correctness(c) = Stability(c) \wedge Desirability(c)$$

The correctness function can be easily evaluated, as it is composed by stability and desirability functions which can be in turn decomposed into atomic base checks, which can be evaluated individually. This way, the complete correctness formula will be evaluated to true if and only if each and every base check is evaluated to true.

$$Correctness(c) = \bigwedge_{i=1}^{|c|} Stability(r_i) \bigwedge_{i=1}^{|DO|} o_i, r_i \in c, o_i \in DO$$

5.1.4. Managed Domain definition

In Section 4.4.1 I delimited the scope of reasoning for management processes, which should consider only the relevant information for correctly applying management operations. With respect to the information models, I reduced the potentially participating resources to two specific sets, the current configuration, and logical definitions contained at the *LRB*. Over the last section I have defined a third set of elements, the objectives defined at the *DO*, which also belong to the scope of management activities, as those defined objectives are the base arguments to evaluate the desirability of an environment configuration.

As those three sets contain all the management-relevant information, the managed environment can be effectively represented for management purposes by the combination of these three sets. Therefore, I will define a management domain *D*, as a triplet composed by the current configuration C_0, a *LRB* and the currently defined objectives *DO*.

$$D = (C_0, LRB, DO)$$

From this point on, the management *Domain* will be the element of analysis for the management system. A Domain can be evaluated for correctness, as *DO* defines the desirable objectives, the LRB contains the stability requirements of the participating resources and the elements of C_0 will be the subject of the evaluation.

5.2. Domain Changes

Once the role of the managed environment has been defined, and expressed as an evaluable function, it is possible to define the role of the management system. The objective of the management system is keeping the Domain configuration at a correct state (meaning both desirable and stable).

This definition brings a key concept about the managed Domain: it is a continuously evolving element. As time passes, the runtime configuration can experience changes, leading to new current configurations, and the two logical sets (*LRB* and *DO*) can also experience variations in its composition. In this section I will develop the concept of Domain changes, as it is fundamental to understand the functions of the management system. This way, although the initially observed configuration can be correct, the effect of changes originating outside the control of the management system can potentially alter that. In those cases, the role of the management system will be reacting to these events in order to restore the domain to a correct state. This will be achieved by in turn invoking a set of changes over the domain.

From those concepts, an initial classification of the changes can be established, depending on the originating element. This way, changes can be either *external changes* (initiated by external entities) or *internal changes* (applied by the management system). In addition to the initiating agent of the changes, the scope of both types of changes is different: external changes can modify any of the three elements of the domain, whereas internal changes can only modify the current configuration. Those concepts are also coherent with the autonomic control loop concept: an external change occurs to the domain, the system diagnoses the change, evaluating if it breaks domain correctness. If that is the case, a set of internal changes are applied, until the domain is restored to a correct state.

The following sections will further detail each category of changes, providing a better characterization of the management operations. External changes will be evaluated on the potential impact over the overall correctness, while the possible range of internal changes will be defined in order to determine the possible scope of actuation of the management system.

5.2.1. External Domain Changes

As the scope of external changes is very broad I will provide a further classification based on the one proposed at DACAR [25]. In this proposal, the external events which can occur to the system are classified into two types depending on the affected part of the domain. Changes coming to runtime environment (C_0 in the proposed terminology) are called *exogenous events*. On the other hand, changes to the established knowledge base (equivalent to the combination of the *LRB* and the *DO*), are named *endogenous events*. This initial classification is useful as depending on the affected part of the domain, the nature of the change and its potential impact will be different. This way, I will describe each type of external change, depending on the part of Domain which has been affected.

- ***Configuration* changes**: Those changes are exogenous events occurring at the physical managed system, and later on reflected through the abstractions of the information model. Those changes can represent hardware failures, network connection problems, traffic fluctuations or even performed changes by other competing management systems. They are perceived by the management system as unexpected changes to one or more

runtime resources from the configuration. They will generate a new current configuration, C'. Clearly, there is no guarantee that the new instance is either stable or desirable, thus it will possibly force the management system to react to them.

- ***LRB* changes**: They are endogenous changes consisting on the addition, removal or modification of a logical resource from the *LRB*. As those elements are not evaluated for stability or desirability, they won't affect domain correctness. However, they will be relevant for future change operations, as the logical definitions greatly restrict the scope of action of the management system.
- ***DO* Changes:** The other kind of endogenous changes affect to the defined objectives for the domain. They consist on objectives being added, removed or modified from the *DO* set. As objectives determine the conditions for domain correctness, the desirability of the current configuration will have to be reevaluated, potentially triggering internal management actions in case the updated objectives are no longer supported by the current configuration.

The following picture provides a summary on the possible domain changes.

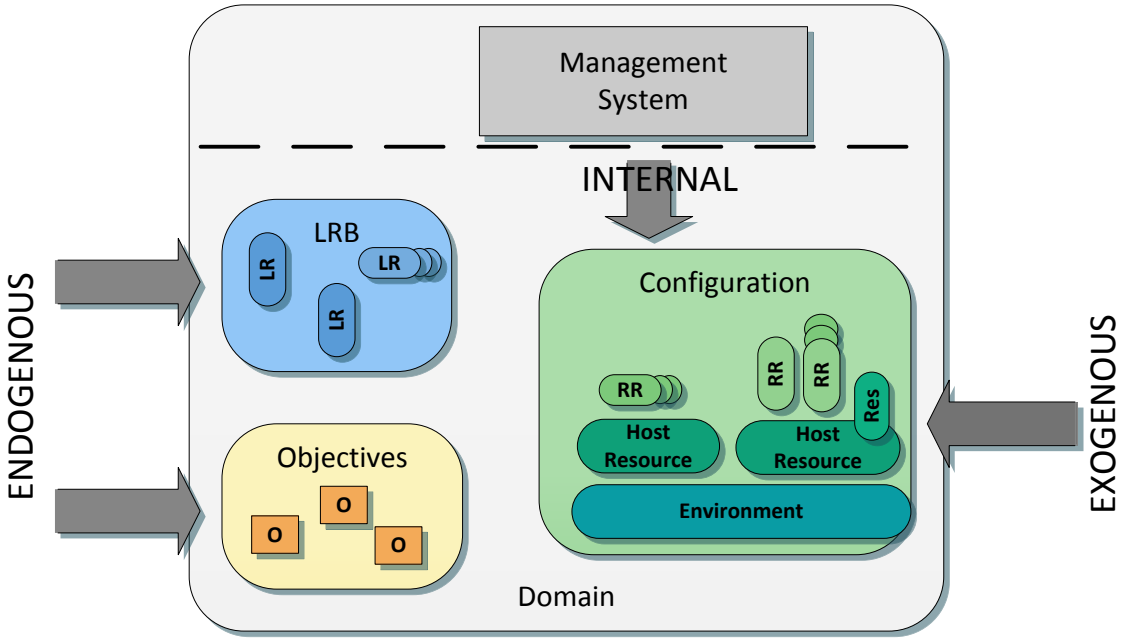

Figure 46 Nature of Domain Changes

5.2.2. Internal Domain Changes

Internal changes can be invoked by the management system in order to correct the actual state of the domain. Those changes can only affect the current configuration. Neither the *LRB* nor the *DO* can be changed by internal actions. However, those parts of the domain must also be considered for the internal changes, as they will determine both the correctness of the configuration and restrict the potentially applied changes. At this point I will only identify the general types of changes which can be applied to *RuntimeResources*, without distinguishing among specific resource subtypes. The changes belonging to the service management information model defined at the end of the previous chapter will be identified at a later stage.

As C_o is a set of *RuntimeResources*, and internal changes can only affect the configuration, every possible internal change will affect in one way or another to a *RuntimeResource*.

Because of that, in order to identify the different types of internal changes, I will analyze the kinds of modifications which can be applied to the set of *RuntimeResources*. This way, three main categories of changes can be identified: adding a new *RuntimeResource* to C_o (additive change), removing an element from C_o (substractive change), or altering in some way one of the existing elements (substitutive change). I will analyze each type of changes with the objective of defining the primitive operations. I won't define more complex changes as they can always be expressed as a combination of the base modifications.

Additive changes consist of adding a new *RuntimeResource* to the configuration. As every operation is defined only from the Domain information, in order to add a *RuntimeResource* its logical definition must be defined in the *LRB*. By following the terminology provided earlier, I will name this type of change *instantiation*. As the configuration is a hierarchical structure, the operation will be executed over a runtime host from the current configuration, which will become the root of the execution context of the new *RuntimeResource*. The corresponding operation is defined as follows:

INSTANTIATE(lr, rhost)

Operation arguments	$lr \in LRB, \quad rhost \in C_0$
Applied Change	$C' = C_0 \cup \{rr \mid rr\ instanceof\ lr \wedge rr.host = rhost\}$

The opposite type of changes consists of removing *RuntimeResources* from C_o. As I intend to define only the base primitives, I will only allow removing resources without any hosted element in the current configuration. The rest of elements cannot be individually removed without breaking configuration structure (as they would be left without a containing element). Those cases will be supported with a combination of remove changes, targeting initially the set of the hosted elements, and finally the hosting resource. The primitive remove operation can be defined as follows:

REMOVE(rr)

Operation arguments	$rr \in C_0, rr.hosts = \emptyset$
***Applied Change*:**	$C' = C_0 - \{rr\}$

Finally I will detail the modifications which can be applied to an existing *RuntimeResource*. The most common types of changes alter the resource configuration, while obviously respecting the resource identity. This way, they will consist of a set of modifications to the resource property, defining, eliminating or altering their values.

MODIFY(rr,conf)

Operation arguments	$rr \in C_0, conf \in \{addProp, removeProp, confProp\}$
***Applied Change*:**	$C' = C - \{rr\} + \{rr' = conf(rr)\}$

Another potential change to *RuntimeResources* is the modification of its hosted resource, consequently modifying the structure of the domain configuration. I will define it as a base operation, although in some scenarios it could also be viewed as the combination of an

instantiate and remove operations. However, if the target resource does not have a matching logical definition, this operation needs to be independently defined.

MOVE(rr, rhost)

Operation arguments	$rr, rhost \in C_0, r.host \neq rhost$
***Applied Change*:**	$C' = C_0 - \{rr\} + \{rr' = rr \wedge rr.host = rhost\}$

Without using a specific information model, the four defined types of basic changes represent all the actions which can be initiated by the management architecture. Specific management systems will define their own set of internal changes, by applying two iterations to these generic definitions. First, in case the information model defines specific resource subclasses, the changes will have to specify the affected types, potentially increasing the number of operations. After that, it is also possible to define compound changes, such as a service update (consisting internally of a resource removal followed by an instantiation of a different version of the same resource over the same host resource).

5.2.3. Reachable Configurations

Once the internal changes which can be ordered by the management system have been detailed, I will try to explain how they allow controlling the evolution of the Domain towards a correct state. The execution of a internal change modifies the current environment configuration. This way, the complete set of potential internal changes represents all the possible transitions that can be triggered by the architecture from one starting Domain Configuration. I will define *Change(D)* as the total set of potentially applicable internal changes starting from a Domain $D = (C_0, LRB, DO)$. Only C_0 and *LRB* affect the members of *Change(D)*, as the objectives do not open up additional possibilities for changes. The members of this set are different instances of the four change types, with specific parameters for its input elements (e.g., for each logical resource *lr* defined at the *LRB*, one potential instantiation will be defined for each valid host resource from the configuration). The next figure shows a graphical overview of the tree generated by the possible transitions from one initial domain state.

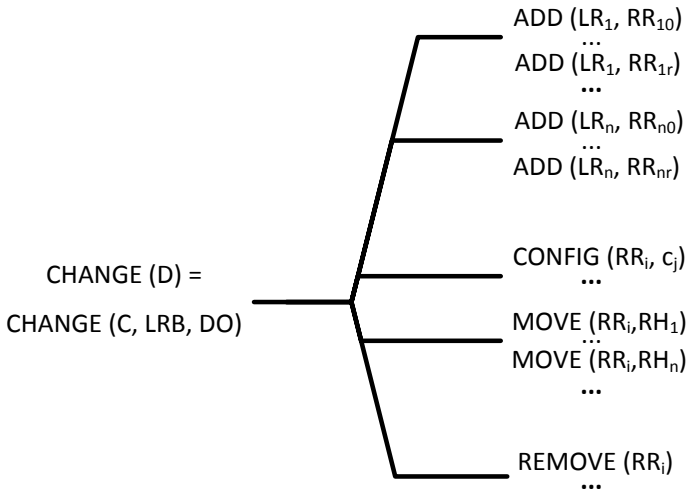

Figure 47 Change Tree of a Domain

Each of the potential changes described in the change tree will modify the targeted domain. This way, starting from the initial state, a total of |*Change(D)*| different states can be reached by applying one of the potential internal changes to the environment. The application of the change modifies the domain configuration, obtaining a new set of potential changes, *Change(D')*. This way, as long as there is no external change to the domain, it is possible to iteratively explore the possible actions from each additional end state, until obtaining the complete set of potential obtainable configurations. In the following paragraphs I will try to explain how to build this set. Starting from a domain d_0, and exploring the potentially executable internal the following set can be obtained:

$$D_1 = \{d_i \mid d_i = d_0 + chg_i, chg_i \in Change(d_0), i = 1..|Change(d_0)|\}$$

$$d_i = \{C_i, LRB, DO\}$$

In order to obtain the second order set of potentially obtainable *Domains*, it will be necessary to repeat the same process, starting from each element of the set d_i, and aggregate the results. When the complete results of this set have been gathered, the aggregated set will have some repeated Domain states (as a pair of commutative operations has been applied in different order), as well as the initial state D_0 (obtained by applying a pair of inverse operations). These repeated elements should be removed from D_2 as all the potentially obtainable branches from them would also be repeated elements. The second order of reachable domains can be defined as follows:

$$D_2 = \{d_{i,j} \mid d_{i,j} = d_i + chg_j, chg_j \in Change(d_i)\}$$

$$d_{i,j} = \{C_{i,j}, LRB, DO\}, i = 1..|Change(d_0)|, j = 1..|Change(d_j)$$

The same process defined up to this point, applied recursively to every unique Domain state obtained, will produce the complete set of reachable environments. This way, the nth iteration can be defined as:

$$D'_{n-1} = \{d_i \mid d_i \in D_{n-1}, d_i \notin D_j, j = 0..(n-2), n > 1\}$$

$$D_n = \{d_i \mid dn - 1 = D + c \in Change(d_i), i = 1..|Change(D)\}$$

By expanding over the change tree concept introduced earlier, it is possible to visually represent how the different Domains are reached. The proposed method for searching the potentially obtainable configurations generates a spanning tree, which can be seen in Figure 48 after applying z iterations.

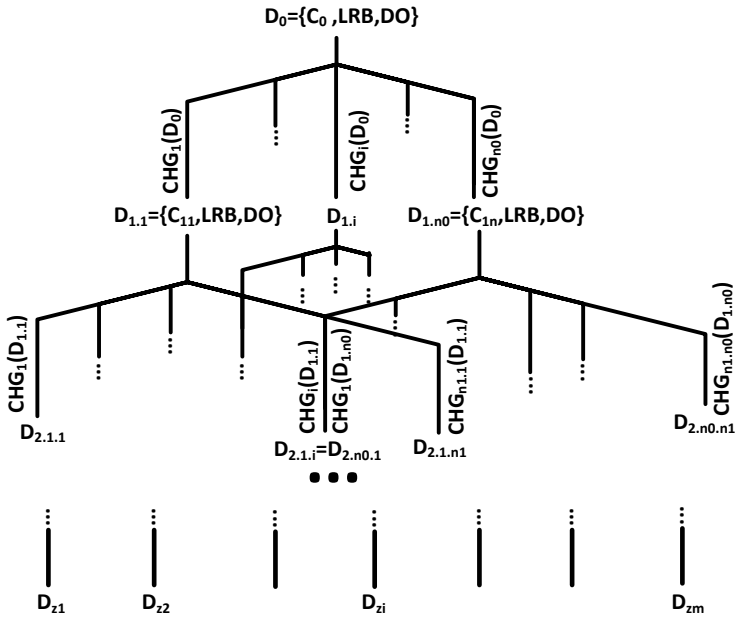

Figure 48 Reachable Configurations Definition through Change Exploring

The followed process has brought the concept of *Reachable Configurations* (*RC*), which are all the potential configurations, which can be obtained by applying a set of internal changes to the initial configuration state $C_0 \in C$. RC can be easily defined by aggregating the contents of the nth grade reachable *Domain* states:

$$RC = \bigcup_{i=0..n} C_i\,, D_i = \{C_i\,, LRB, DO\}$$

The definition of the reachable configurations set supersedes the previous superset of possible configurations as the root of the configurations space. For management purposes, only the configurations contained in RC need to be considered, as they include the current state C_0, and every reachable configuration through the intervention of the management system. This way, the size of the set of Configurations which must be considered by management operations is greatly delimited, further reducing the complexity of the base problem. As it can be seen in the next picture, from this point onwards RC will be the superset for all the management operations. Also, the same set of definitions for the sets of stable configurations, and desirable configurations still apply, but they will be evaluated now against this base superset.

The RC is only valid as long as there are not external changes to the Domain. Any external modification to the elements of the domain will need RC to be reevaluated. However, that does not invalidate this set for management purposes, as it will be relevant as long as the control of the modifications is held by the management system.

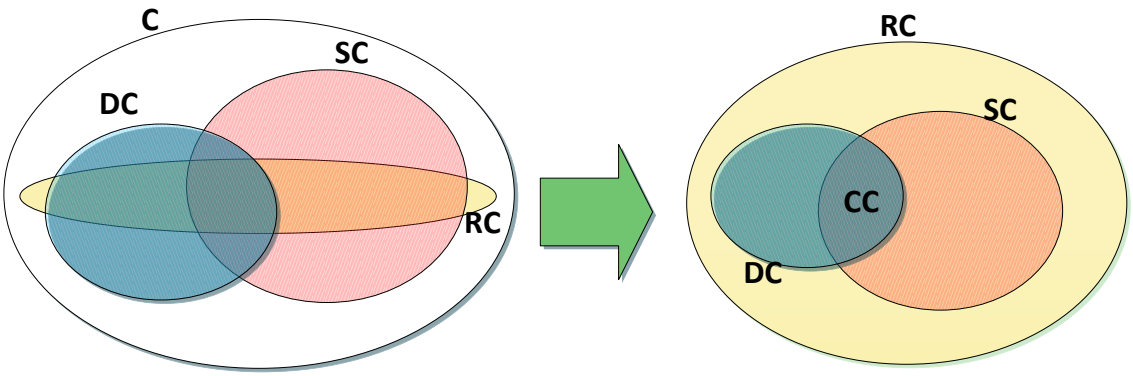

Figure 49 Reduction of the Configuration Space to Reachable Configurations

The definition of the *RC* set also allows a better definition of the role of the management system: If after suffering an external change, the managed Domain ends in a non correct state, the management system must explore the *RC* from that initial state in order to detect a correct configuration from this set and apply the set of required internal changes to reach it.

5.3. Analysis of an Enterprise Service Management System

The previous sections have expanded the concepts defined with the information model, until providing a detailed definition of the two basic entities of the context of this work: the managed domain and the management system, including the potential actions which can be applied to perform its role. These concepts have been described at a generic level, but need to be specified for the type of management system which is the main subject of this dissertation: an Enterprise Service Change Management System.

At this point I will review the autonomic computing principles described in the state of the art, to see whether they can be applied in this context. This way, I will try to define the autonomic computing principles using the definitions of the domain and management system provided earlier. Self Protection does not have a translation to these concepts, but the remaining three can be expressed as follows:

- **Self-Configuration**: Whenever an external change modifies the elements of *DO*, the management system will find a reachable configuration that is adapted to new objectives while keeping at the same time system stability.
- **Self-Healing**: Whenever the Domain is affected with external changes to its configuration which leave the system at a non correct state, the management system will obtain (and if possible invoke) a chain of changes that will restore the system to a desirable state. Depending on the severity of the change there might be some aspects which cannot be restored by the management system, as they are not part of the modeled Domain information.
- **Self-Optimization**: Optimization consists on improving the 'quality' of the environment state by applying a set of internal changes. Those preferences can be simply expressed as objectives evaluated against the runtime properties (e.g. minimize the value of the response time property from one resource which represents a running service). However, those values cannot be directly modified by the management operations, their evolution depends on external factors whose relationship with the measured element cannot be

represented by the current models (e.g in the previous case, the response time could be improved by migrating the service to a more powerful node, or creating a new instance of the service and balancing the traffic among the two of them). They can be expressed with the current abstractions but cannot be automatically addressed, so they have not been considered.

This way, in order to support an automated behavior of the management architecture, the previous scenarios must be included into the set of defined uses cases for the system. With these requirements in mind, I will first identify the most relevant use cases which must be fulfilled by the proposed management architecture, and finally define the set of internal changes which can be ordered by it to support them.

5.3.1. Use Case Analysis

Over this section I will describe the most relevant use cases which have been identified in the context of Enterprise Service Change Management Systems. For each identified element I will provide both a textual, context-relevant description of the scenario and an interpretation based on the Domain changes concept.

The criteria for the completion of all the use cases will be identical: Each use case will be completed when the management system obtains a domain $D' = \{C', DO', LRB'\},\ C' \in RC$, being *C'* a correct configuration. The system will apply the required changes to reach the correct state, fulfilling the use case.

5.3.1.1. Service initial deployment and configuration

After the development process of a new service is complete, a company decides to provide the newly defined service into the production environment. The company knowledge base is updated with the *DeploymentUnit* definition of the new element providing the service, and a defined a new objective for the environment stating the required existence of the new service. After these changes are detected the management system will apply automatically the required changes for the new service to be correctly running at the environment.

This use case can be easily defined with the Domain and changes concept by first defining which have been the external changes applied to the *Domain*'s *DO* and *LRB*:

$$DO' = DO \cup \{EXISTS(s')\}, LRB' = LRB \cup \{du'\}$$

$$s' \subset du', du' \in LRB$$

5.3.1.2. Service update

After the users of a remote service report multiple occurrences of an error, a new service version is developed with the required fixes and updated to the knowledge base. After it has passed all the regression tests, a new objective is defined to replace the newer version instead of the faulty one. After these changes are detected the management system will apply automatically the required changes for the updated version of the service to be correctly running at the environment.

$$DO' = DO \cup \{EXISTS(sn)\} - \{EXISTS(so)\}, LRB' = LRB \cup \{dun\}$$

$$sn \subset dun, so \subset duo\ , (dun, duo) \in LRB$$

5.3.1.3. Service uninstallation

Once the maintenance period of a company service has concluded, a management decision is taken to no longer extend it. In order to do so, the runtime objectives are updated to reflect that the service must no longer be part of the environment. After these changes are detected the management system will apply automatically the required changes to remove all the instances of the service.

$$DO' = DO \cup \{NOTEXISTS(du)\} - \{EXISTS(du)\}$$

$$s \subset du, du \in LRB$$

5.3.1.4. Runtime diagnosis

A hardware malfunction causes a node of the managed environment to stop running. When the incidence is detected and notified to the management system it will be reflected with the disappearance from the updated configuration of the resource representing the node as well as all the resources from its execution context. The management system will analyze the correctness of the updated *Configuration*, and in case it is no longer correct, the required changes to restore correctness are identified and applied to the domain.

Nonetheless, it is possible that the changes occurring to the real elements of the environment cannot be expressed by the provided management modeling abstractions. A hardware malfunction can cause the loss of data, as well as abruptly breaking all the established sessions from the affected applications and services. Because of that, it cannot be guaranteed that the service management system will be able to completely restore the correctness of the management environment, although it can operate based on the known information (the Logical Units, Configuration and Objectives)

$$C' = C - \{n, r_i\}, n \in Node, n \in Exctx(r_i)$$

5.3.2. Internal changes definition

The definition of some representative use cases allows delimiting the exact set of internal changes which can be invoked by the proposed service management system. Configuration change operations consist of changing the state of the domain configuration, either adding, removing or modifying the characteristics of a *RuntimeResource*. The service-management centric information model identifies four different subtypes of *RuntimeResources*, each of them potentially being controllable by the management system. However, only the set of operations which are considered necessary and at the same time coherent with the abstraction level of the management system will be defined. Consequently, I will analyze each specific type of resource to determine to what extent it should be modifiable by internal changes.

Environment-level operations: The possible changes to the *Environment* object are *Node* creation, *Node* elimination, and configuration of *environmentResources*. Clearly, those changes neither are at the abstraction level of a service management system, nor would be required for the previously described use cases. Consequently, no environment-level operations will be defined. On the other hand, these operations would be mandatory in a cloud computing environment. In that scenario, management policies are applied transparently to the underlying runtime platform, adapting elastically to performance and traffic demands.

Node-level operations. The possible configuration operations to *Node* elements are *Container* instantiation and elimination, and configuration of the *nodeResources* and *nodeProperties*. The same reasoning applied for environment-level operations can be applied to this category, so no defined internal changes will target the runtime *Nodes*.

Container-level operations: At this level most of the required internal changes will be defined. The potential list of container-level changes includes *RuntimeUnit* instantiation / elimination, and *Container Properties* and *Resources* configuration. *RuntimeUnits* operations must be clearly supported, covering the complete lifecycle of hosted units, from the instantiation to its removal or update. *containerResources* configuration must also be supported for its potential role in the satisfaction of *RuntimeUnit Constraints*. On the other hand, *containerProperties* modification will be left outside from the management scope, as they only reflect internal details of the Container whose modification could not reach a correct state from an incorrect one.

Runtime Unit-level operations: Finally, I will analyze the potential configuration changes to existing *RuntimeUnits*. As unit lifecycle must be controllable by the management system, unit state modification (*STARTED*<->*STOPPED*) must be included among the internal changes. Regarding additional unit configuration, *exportedResources* configuration will not be supported. These elements are considered and integral part of the *RuntimeUnit* and there are no identified use cases where those operations would be necessary. *Properties* configuration will also be supported, as it is clearly a required functionality, as both *ContextAwareProperties* and *BoundProperties* need to be controllable by the management operations. For analogous reasons, *Binding* configuration must also be supported.

After delimiting the scope of the allowed operations I will completely define the set of potential internal changes, which is composed by 10 elements. Each configuration primitive will be completely defined by the following information, the *target* (*RuntimeResource* where it will be applied), the *arguments* (required additional information for the operation), the *pre-conditions* (mandatory initial state for the change to be applied) and the *post-conditions* (mandatory runtime state after applying the change). In case some of these elements include optional sections they will be marked with a question mark *(optional_term)?*, similarly to the EBNF (Extended Backus–Naur Form) notation.

INSTALLDU(cont,du)

Name	*Install Deployment Unit*
Target	$Container\ cont, cont \in C$
Arguments	$DeploymentUnit\ du, du \in LRB$
Pre-conditions	$\nexists ru, ru \subset cont \wedge ru\ instanceof\ du$
Post-conditions	$\exists ru, ru \subset cont \wedge ru\ instanceof\ du$

UNINST(cont,ru)

Name	*Uninstall Deployment Unit*
Target	$Container\ cont, cont \in C$
Arguments	$RuntimeUnit\ ru \in C, ru \subset cont$
Pre-conditions	$\exists ru\ \wedge\ ru \subset cont$
Post-conditions	$\nexists ru, ru \subset cont$

UPDATEDU(cont,ruo,dun)

Name	*Update Deployment Unit*
Target	$Container\ cont, cont \in C$
Arguments	$RuntimeUnit\ ruo \in C, DeploymentUnit\ dun \in LRB$ $ruo.name = dun.name, ruo.type = dun.type$ $ruo.version \neq dun.version$
Pre-conditions	$\exists ru\ \wedge\ ru \subset cont$
Post-conditions	$\exists run, ru \subset cont \wedge run\ instanceof\ dun$ $\nexists ruo, ruo \subset cont$

ADDCRES(cont,crc,name)

Name	*Add Container Resource*
Target	$Container\ cont, cont \in C$
Arguments	$ContResConf\ crc, (name \in String)?$
Pre-conditions	$\nexists cr, cr \subset cont, cr \in Resource,$ $cr\ instanceof\ crc, (cr.name = name)?$
Post-conditions	$\exists cr, cr \subset cont, cr \in Resource,$ $cr\ instanceof\ crc, (cr.name = name)?$

RMVCRES(cont,rc)

Name	*Remove Container Resource*
Target	$Container\ cont, cont \in C$
Arguments	$Resource\ rc, rc \in C$
Pre-conditions	$\exists rc, rc \subset cont$
Post-conditions	$\nexists rc, rc \subset cont$ s

CNFCRES(cont,rc,props)

Name	*Configure Container Resource*
Target	$Container\ cont, cont \in C$
Arguments	$Properties\ props, Resource\ rc$
Pre-conditions	$\exists rc, rc \subset cont$
Post-conditions	$\exists rc, rc \subset cont, props \subset rc$

STARTDU(ru)

Name	*Start Deployment Unit*
Target	$RuntimeUnit\ ru, ru \in C$
Arguments	-
Pre-conditions	$\exists ru \wedge ru.state =$ STOPPED
Post-conditions	$\exists ru \wedge ru.state =$ ACTIVE

STOPDU(ru)

Name	*Stop Deployment Unit*
Target	$RuntimeUnit\ ru, ru \in C$
Arguments	-
Pre-conditions	$\exists ru \wedge ru.state =$ ACTIVE
Post-conditions	$\exists ru \wedge ru.state = STOPPED$

CFDUPROP(ru, props)

Name	*Configure Unit Properties*
Target	$RuntimeUnit\ ru, ru \in C$
Arguments	$Properties\ props$
Pre-conditions	$\exists ru, ru \subset Container\ cont$
Post-conditions	$\exists ru, ru \subset cont, props \subset ru$

CNFBIND(ru,bindId,rb)

Name	*Configure Unit Binding*
Target	$RuntimeUnit\ ru, ru \in C$
Arguments	$RuntimeUnit\ rb, int\ bindId\ ru \in C$
Pre-conditions	$\exists ru, \exists rb \in C, ru \neq rb, ru.bind(bindId) \neq rb$
Post-conditions	$\exists ru, \exists rb \in C, ru \neq rb, ru.bind(bindId) = rb$

5.4. A Service Change Identification Algorithm

Up to this point, I have completely characterized the required information for managing a domain, including its information model and the objectives it must comply with. Side by side with the models functions have been defined to validate only from that information whether the current state is correct. Finally, the role of the service configuration system has also been defined, based on the concept of domain changes. Over this last section of the chapter, I will address the initially established objective of automating the management functions, more specifically, the process of finding a correct configuration among the reachable configurations, and identifying the set of required internal changes to reach that state.

In order to find a solution to this problem there are two main approaches for finding a correct configuration among the RC space [13]: imperative and declarative solutions. Imperative solutions consist of exploring the reachable configurations by progressing over the search space by applying changes sequentially. They belong to the mathematical field of search algorithms [95]. On the other hand, declarative approaches try to find directly a correct, reachable configuration and from that point compare it with the current state, obtaining the required internal changes which must be applied.

In order to define an imperative algorithm, it would be necessary to establish a search strategy, which at each iteration selected a element from *CHANGE(D)* which, when applied obtained a new state D' closer to the required correctness. However, this approach can be difficult to implement correctly. Without having the complete scope, local strategies can lead to broken paths, and a high degree of back steps which can render the algorithm invalid for some contexts.

On the other hand, a declarative approach can be followed by defining a function with the set of *Reachable Configurations* represented by the variables, and the conditions for correctness (both stability and desirability) also integrated in the function. Starting from the established models and conditions, defining this formula would be feasible. This way, the limitation for this approach consists of efficiently finding a solution to the described formula, taking into account the amount of relevant information Nonetheless, over the recent years in the mathematical field important performance breakthroughs have been achieved in the solvers for one of the

most well-known NP Complete problems: the boolean satisfiability problem (SAT). The SAT problem consists of finding a solution to a function composed only by boolean variables and logical operands [103]. In addition to the base SAT problems, there are several variants as MaxSAT, consisting of finding the maximum number of positive literals of the formula, or Pseudo Boolean SAT, which expands the number of allowed functions to include boolean linear expressions, as well as an optimization function.

In the early sixties the DPLL (Davis-Putman-Logemann-Loveland) algorithm was presented [18], providing a sound and complete way of obtaining the solution to the problem of satisfiability.. This algorithm first tried to discover if the problem had a solution, and in the affirmative case started to explore the variable space by assigning them potential values through a technique called Boolean Constraint Propagation (BCP). In case at one step of this process a conflict was found (assigning both true and false to a variable), the algorithm would backtrack to the point where that decision had been made and try the other possibility. This base algorithm has been heavily optimized in the recent years, by adopting strategies such as conflict analysis techniques (which try to obtain the reason for an assignment conflict), conflict-driven learning (adding clauses to avoid the same conflict in the future) or the incorporation of heuristics to detect conflicts before they occur or optimizing the way the search space is explored [91]

The relevance of SAT goes beyond its mathematical interest, as the first discovered NP complete problem. Thanks to the described advances in SAT resolution algorithms, current solvers can obtain a solution for SAT problems of a considerable size in a short time, which has lead to their adoption as the base algorithm for solving numerous problems of the electronics and computer science fields [65]. The problems expressed in SAT include automatic test pattern generation, redundancy identification and elimination, FPGA (Field Programmable Gate Array*)* routing, or model correctness checking [73].

There is one specific field of application of SAT which has lead to its consideration for the proposed algorithm: the support for dependency resolution in installation processes. This application was initially proposed by the OPIUM prototype [113]. This work describes how to automatically obtain dependency closures of Linux packages, respecting both expressed dependencies and incompatibility constraints. The practical applicability of this approach has been demonstrated since 2008, when a SAT-based algorithm was incorporated to the Zypp dependency manager of the openSuse 11.0 Linux distribution. In addition to this example, there is another successful use of SAT for dependency management among logical component and service definitions: the Eclipse p2 provisioning engine which, since the 3.4 version of Eclipse, released in June 2008, completely manages the dependencies of the platform plug-ins for install, update and uninstall operations. The p2 use of SAT [66] refines the notions presented in OPIUM, in order to support a more expressive dependency model, with version information and decoupling among the packages and the providing /dependant elements. Both of those refinements are also considered by the model proposed in this dissertation, which further reinforces the notion that SAT may enable a declarative resolution of the change identification problem. In addition to the increased expressivity, the performance results obtained by the p2 implementation are very positive, as it has been successfully tested with problems of more than 10000 literals, and a solution involving more than 3000. Finally, p2 uses a PBSAT solver instead of a regular one, which not only provides a simpler definition of the SAT clauses, thanks to the use of linear functions, but also allows to specify an optimization

function, enabling to find not only a correct solution but also improve the quality of it through optional restrictions.

Although there are similarities between these proposals and the problem of change identification there are also several differences. The main difference is the distributed nature of the problem of finding a domain configuration, which adds an additional layer of complexity to the problem, and forces to partially alter the strategy of literals definition. On top of that, several additional requirements need to be expressed as SAT clauses, such as *Constraints* over the execution context, or visibility restrictions. Finally, the solution must not only decide what *RuntimeUnits* will be available, but must also provide a correct configuration for aspects such as *Bindings* or *containerResources*. Nonetheless, after an analysis it has been determined that these factors can also be expressed into an SAT problem.

With these factors in mind, I have opted for an imperative approach, consisting of the characterization of the change identification problem as a Pseudo Boolean SAT. In this approach, first a set of literals must be defined, representing the *Reachable Configurations*. On top of them, constraint functions will be identified ensuring that the solution obtained by the solver respects the structure of the information model, and is stable and desirable. Once that information has been expressed in the format required by the SAT it will be necessary on a final step to process the results of the engine invocation, in order to identify the required internal changes to the domain.

5.4.1. SAT-Based Service Change Identification Process

In order to model the problem as a SAT, the first step must be the definition of boolean variables (literals) which will be evaluated by the solver. Each defined literal must represent a decision over the final state of the environment configuration, after applying the required changes. Not every *RuntimeResource* will have an associated literal, as only the aspects potentially controllable by the management architecture will be considered (e.g. the instantiation of a *RuntimeUnit* in a *Container*, but not the removal of the Container, as that change is not among the allowed properties for the management architecture). The literals will represent the decisions over the final state of the runtime environment.

The literals will be defined from the elements of the management domain. These elements will be referred to with the following notation: The domain *D* is composed by the *Logical Resource Base*, the environment *Configuration* and the *Defined Objectives*:

$$D = LRB \cup Cf \cup DO,\ LRB = \{LRi, CRC_j\},\ Cf = \{RR_i\},\ DO = \{O_i\}.$$

From the set of *RuntimeResources* of the *Configuration*, a subset of them, referred as $C_i \in RR_i$, constitute the runtime *Containers*. Finally, the potential results of instantiating a $CRC_j \in ContainerResourceConfiguration$ on over a $C_i \in Container$ will be referred as $RC_{ij} \in RuntimeResource$.

In addition to the literals, it is necessary to define a set of clauses which will ensure that the proposed solution is at the same time stable and desirable. For this goal the stability and objective functions described in the previous chapters will be translated to boolean functions, referencing the defined literals. In order to improve the quality of the proposed solutions, an optimization function can be defined to maximize some desired characteristics of the final environment state.

Finally, after invoking the resolution engine the proposed values for the defined literals will be interpreted and compared with the current configuration, in order to obtain the list of required changes.

The following picture represents a high-level representation of the complete change identification process. In the following sections I will describe the required algorithms for, starting from the Domain information, generate the required input for the SAT Solver (Literals, Clauses and Optimization Function), invoke the engine, and interpret the results as the set of required changes.

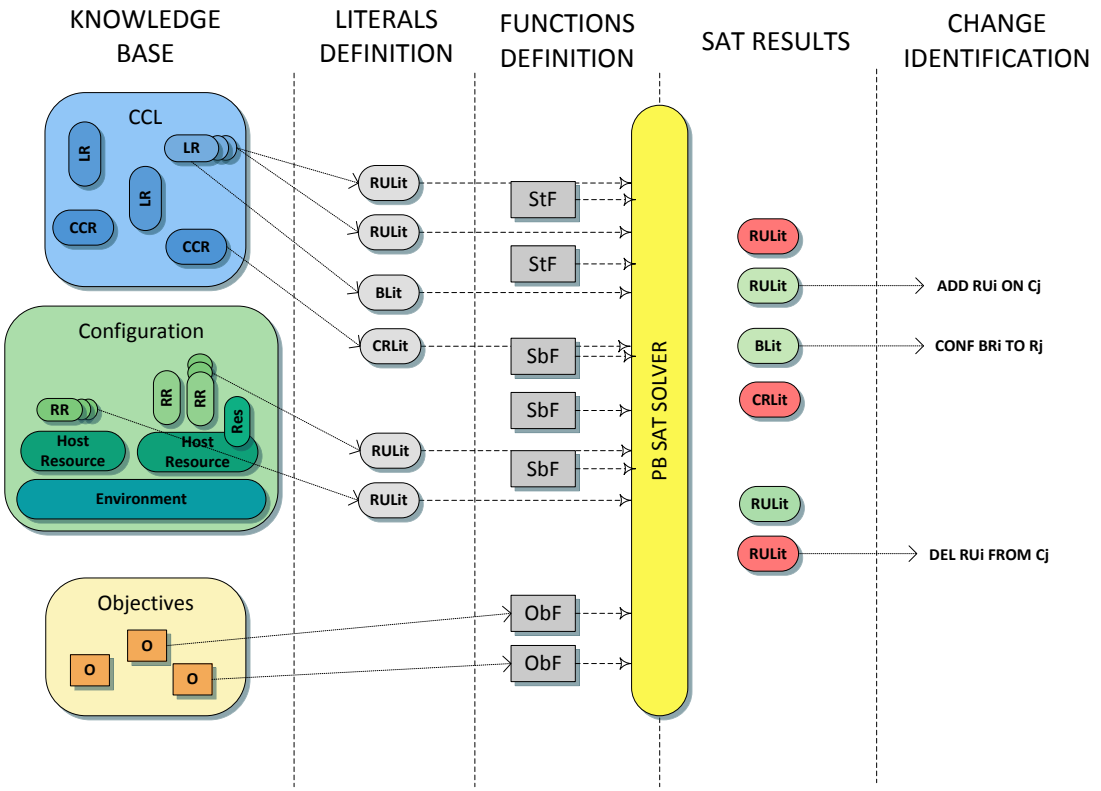

Figure 50 Change Identification Via PB SAT Solving

5.4.2. Literals Definition

Over this section I will identify the complete set of literals which will be added to the SAT solver.

The first defined group of literals represents the potentially existing *RuntimeUnits* in the future configuration. In order to obtain all the possibilities the logical definitions from the LRB will be checked against the runtime *Containers*. In order to reflect all these possibilities, there will be one literal for each *DeploymentUnit* for each compatible *Container*. The set of *RuntimeUnit*-based literals and the specific literals will be referred to with the following notation: $RLit = \{rul_{i,j}, C_j \in Cf \wedge DU_i \in LRB\}$. The worst-case number of defined literals of this group would be $|RLit| = |\{C_j\}| * |\{DU_i\}|$, although type compatibility and additional restrictions will further reduce this number.

The proposed method considers the *RuntimeUnits* obtainable from the logical definition, without requiring to separately evaluate the resources already present at the environment. Most of the existing *RuntimeUnits* from the current configuration will have a matching *DeploymentUnit* in the LRB, meaning they are already included in the literals definition. The only missing elements are the existing *RuntimeUnits* without a correspondent logical definition at the LRB. Although they can potentially be removed by the operation $REMOVE(RU)$, it has been decided to leave them out of the scope of this algorithm, as in case it was decided to apply that change, there would be no way to reverse the operation. Nonetheless, these units still play a role in the process, as they might be required by other *RuntimeUnits* for satisfying their dependencies.

In order to completely capture the possible structure of the final configuration the proposed *RU* literals are not enough. The decision on how to configure the *Bindings* of dependent *RuntimeResources* must also be provided by the change management process. In order to support that, additional literals will be defined to represent the possible *Binding* values of the existing *RuntimeResources*. For each *RuntimeUnit Binding*, there will be one literal for each other *RuntimeResource* which can potentially satisfy the relationship. The set of *BindingLiterals* will be referred to with the following notation: $BLit = \{bl_{i,j}, rul_i \wedge rul_j \in RLit\}$ In the worst case, the number of defined *Binding* literals would be: $|BLit| = |\{B_i\}| * (|RLit| - 1)$, with $\{B_i\}$ being the set of *Bindings* among the potential *RuntimeResources*.

Finally, in addition to the decisions about the *RuntimeUnits*, the change management system can also modify the configuration of the existing *Containers*, modifying their *Resources* through the use of the *Configurable Container Resource* templates. With an analogous reasoning about *RuntimeUnit* literals, I will define the Container Resource Literals, $CLit = \{crl_{i,j}, C_j \in Cf \wedge CRC_i \in LRB\}$, which represent the decision to create a *RuntimeResource* in a *Container*, based on a CRC template. As potentially any number of *RuntimeResources* can be created on the same *Container* from only one CRC template, it is not possible to provide a worst case estimation of the cardinality of the $CLit$ set.

The three sets of literals are enough to capture all the decisions about the future configuration of the domain. There are additional aspects of the configuration which haven't been defined as literals, such as context aware or bound properties. Those elements do not belong as literals because their value is a result of the decisions about the *RuntimeUnit* structure, which is already supported by the existing set.

The following table summarizes the defined literals and the worst case estimation of the number of required literals for analyzing a base domain. It is evident that the number of literals grows exponentially as the size of the LRB and the number of logical elements increases. Fortunately, the actual complexity of the problem is much simpler than that, as the already defined stability constraints greatly reduce the number of potential configurations. Although some of them will be processed as SAT clauses later on, there are also at this point some simplifications which can directly reduce the cardinality of the three sets of literals. Over the following paragraphs I will propose an algorithm for defining the set of literals which only contributes to the SAT the uncertain elements, filtering out variables whose value would always be false or true.

Table 2 Defined Literals for the Change Identification Process

Name	Definition	Number of literals
$\mathbf{DU_iC_j}$	Existing and possible *RuntimeUnits* from the LRB	$\lvert\{C_j\}\rvert * \lvert\{DU_i\}\rvert$
$\mathbf{RU_iB_jR_k}$	Potential bindings between the RUs	$\lvert\{B_i\}\rvert * (\lvert RLit\rvert - 1)$
$\mathbf{CR_i}$	Potential Configurable Resources from Container	$Unbound$

The *CLit* set elements are stable by themselves, but their only role is to support the stability of *RuntimeUnits* with *Constraint* definitions. Because of that, instead of an undetermined amount of *crl* literals, only the ones required for the stability of the constrained *RLit* elements will be defined. This way, *CLit* literals will be defined on the fly over the processing of Resource Constraints.

I will start the simplifications by proposing an algorithm for the definition of the *RLit* elements. Over the iteration of the *DeploymentUnit* elements of the LRB against the configuration *Containers*, some stability checks will be applied to verify whether the *rul* literal could potentially be stable, ignoring it in case it is not possible. This way, the following checks will be applied:

- If the type of the *DeploymentUnit* is not among the *supportedTypes* from the host *Container* the literal will not be defined.
- For *DeploymentUnits* with *Constraint* definitions, the algorithm will check whether the Execution Environment of the *RuntimeUnit* contains or can potentially contain (by means of container resource creation from a CRC template) the identified resource. This way, for each defined *Constraint* definition there will be three possible results: The resource is NEVER available at the environment, the resource is ALWAYS available at the environment, or the resource is not available but could be configured. RLit literals won't be defined whenever the resource is NEVER at the environment, or otherwise, whenever the *Constraint* definition is of NOT kind (incompatibility) and the resource is always available at the environment. In the remaining cases the *RLit* element will be defined, with also a *CLit* being added for the undetermined cases, representing the decision of creating or not the container resource from the CRC template.

By applying these restrictions, the following algorithm is proposed to only define the *RLit* and *CLit* which can potentially be evaluated as true or false before defining the rest of structural and management constraints.

```
filterUnits {
   define RLit as list of RuntimeUnit literals
   define CLit as list of ContainerResource literals
   for each DeploymentUnit du ∈ LRB{
     for each Container c ∈ Cf{
     ru = instance(du,c)
```

```
            if(ru.type ∈ c.supportedTypes){
                 if(ru is constrainedResource){
                      for each Constraint co ∈ ru{
                           if( !(co.kind==not) AND r=co.resid(ru.exctx))
                                if(co.MAX(r) AND co.MIN(r))
                                     RLit.add(ru)
                                else if(!(co.kind==not)){
                                     //try to create a conf cont resource
                                     cr = instanceof(crc_i,c)
                                     if(co.resid(cr)){
                                          RLit.add (ru)
                                          CLit.add(cr)
                                     }//end if
                           }//end if
                      }//end for each
                 }//end if
            else
                 RLit.add(ru)
            }//end if
       }//end foreach
       }//end foreach
       return RLit, CLit
}
```

After reducing the size of the *RLit*, *BLit* is also simplified, as the total number of potential *Binding* configurations depends on *RLit* cardinality. In addition to that, as *Bindings* originate from a logical *Dependency* definition, I will evaluate some of its stability checks in order to avoid defining literals for impossible bindings. This way, the following simplifications will be applied:

- In order for each *Binding* to be valid, the dependant resource must be able to access the bound resource. Resource accessibility is evaluated through the combination of visibility of the bound resource and the placement of both resources in the overall hierarchy. Unreachable *Bindings* won´t be defined as literals.
- The *Dependency* correspondent to each *Binding* includes a resource identification function which must be met by the identity of the bound *Resource*. In case this check is not true the potential literal will not be defined.
- As a consequence of both simplifications, it is also possible that some *RuntimeUnit* literals can never be evaluated to true, as some of their *Bindings* could never be satisfied. If that

was the case, it will be known over this step of the algorithm, so the invalid *RULit* literals will be assigned the *false* value at this point of the process.

The following algorithm obtains the reduced set of *BLit* literals after evaluating those *Constraints*:

```
filterBindings {

     define BLit as list of Binding literals
          for each DependantResource ru ∈ RLit{
               for each Binding b ∈ ru{
                    for each br ∈ RLit –{ru}

                         if(visible(rr,br) AND b.resid(br))
                              BLit.add(ru.b.br)
                    }
               }

               if(no possible binding was found)

                     set ru to false
          }

     }

     return B

}
```

After these procedures have been executed the complete set of literals which will be evaluated at the SAT solver has been defined. $SATLit = RLit \cup BLit \cup CLit$. The decided value of these boolean variables will be enough to determine the final state of the managed system.

5.4.3. SAT Functions Definition

After the algorithm for defining the SAT literals has been explained I will detail how to express the remaining stability and desirability functions from the domain as Boolean functions whose variables are the identified literals. In order to structure this analysis I will separate the description in three groups, depending on the type of restrictions defined: expressions that assure the identified solutions have a valid structure, are stable and provided the desired management functionality.

5.4.3.1. Structural functions

The objective of the structural functions is to ensure that the results from the SAT execution are coherent with the intended meaning of each type of literal. Whereas any combination of values from the *RLit* or *CLit* elements could be interpreted as a desired state of the environment (another topic would be its stability or desirability), the same is not the case for the *BLit* literals. If no structure clauses were defined, possible SAT solutions would imply a *Binding* pointing to multiple units at the same time, which clearly does not make sense. This is the case because the *BLit* elements are not actually independent among them, but constitute groups related to the same decision. A group of *BLit* literals is a set of elements whose originating *Binding* and *RuntimeResource* are the same. Because of that, it is necessary to define a set of structural functions which guarantee that for any SAT solution the *RLit* and *BLit* literals will be coherent with the relationship between a *RuntimeUnit* and its *Bindings*.

These requirements will be defined the following way. Let it be $A \in RLit$ with n defined *Bindings*. The group of binding literals corresponding to the *Binding i* is the following set: $Group = \{Ab_i B_j\}, B_j \in RLit, j = 1..m$ For each group of literals the following two conditions must be enforced: if the *RuntimeUnit* is present at a *Container*, exactly one literal from the group must be true. If the *RuntimeUnit* is not present, all the group literals must be false. Those concepts can be expressed with the following two boolean functions:

$$A \rightarrow ExactlyOneIsTrue_j(Ab_i B_j)$$
$$\bar{A} \rightarrow \bigwedge_j \overline{Ab_i B_j} = \bigvee_j Ab_i B_j \rightarrow \mathrm{A}$$

The first function cannot be directly inserted in the pseudo-boolean SAT engine, which only supports logic functions and linear expressions over the boolean variables. However, that function can easily be decomposed into two factors, a disjunction among all the binding literals and a linear function restricting the maximum amount of true literals from the group to one. Moreover, the disjunction is the reverse implication of the other identified restriction, so both can be combined as a double implication. This way, for each group of binding literals, the following boolean functions will be added to the SAT engine:

$$A \leftrightarrow \bigvee_{j=1}^{\mathrm{m}} Ab_i B_j$$
$$\sum_{j=1}^{m} Ab_i B_j \leq 1$$

The definition of structure functions can be easily added to the previous algorithm where the set of *BLit* elements is defined. After completely processing the potential candidates for a Binding of a *RuntimeUnit*, the group of literals has just been defined so it is immediate to process them in order to generate the required structure functions.

5.4.3.2. Stability functions

Up this point, the result from evaluating the defined literals with the provided structure functions will always be a correct model structure. This section defines an additional set of constraints which will further restrict the potential results from the SAT solver in order to ensure that the proposed configuration is stable. Over this section I will review the four categories of stability functions and for each of them will define the required SAT boolean functions which transform the requirements on the model to the defined literals.

Local functions. The restrictions about local functions evaluate the actual value of a resource property. That kind of restrictions cannot be efficiently represented as SAT literals, as they would need at least integer variables for a proper characterization. Fortunately, this limitation is not problematic, as they can both be evaluated beforehand on the current configuration, and configured with the desired values after the proposed changes to the environment have been processed.

Hosting functions. Hosting functions restrict which *RuntimeUnits* can be deployed on each environment container. As they can be evaluated independently, they have already been

applied at the filtering stage for reducing the number of defined *RuntimeUnit* literals. This way, no additional functions need to be defined because of those conditions.

Dependency functions. Dependency functions ensure that the *Bindings* between *RuntimeUnits* are correctly defined and that the internal unit configuration is correct with respect to the bound resource (the values of the *BoundProperties*). The solution proposed by the SAT solver must respect these requirements. Parts of them are already enforced by the algorithm for identifying the binding literals, which applied the visibility and resource identification restrictions in order to filter out unstable possibilities to the *Bindings*. Thanks to that, the *Binding* literals already represent stable bindings. Moreover, the previously defined structure functions ensure that the solution includes exactly one binding decision whenever the *RuntimeUnit* is present at the proposed configuration.

Nonetheless, the previous operations are not sufficient to ensure a correct solution with respect to *Dependencies*. It is also necessary to add functions to the SAT engine which ensure that, whenever a *RuntimeUnit* with defined *Bindings* is present at the configuration solution, it will have a correct configuration for each one of its *Bindings.* Structure functions ensure that one of the binding options will appear at the solution. However, it still must be enforced that the *RuntimeUnit* referenced by the *Binding* also appears in the proposed solution. This concept can be expressed very similarly to the way OPIUM defined the component dependencies as SAT functions. The main difference is that, as this algorithm reasons about distributed instances instead of logical elements, the dependency restriction must be defined for each binding literal, instead of only one among the provider and consumer elements. This way, for every defined binding literal, the following function will be added to the SAT.

$$A_i b_j B_k \rightarrow B_k$$

Finally, the only aspect of dependency expressions that has not been addressed is the handling of *BoundProperties*. Analogous to local restrictions, property values cannot be efficiently evaluated with a SAT, as it would be necessary to use integer variables. However, in this case those values do not need to be expressed in the SAT, as they are obtained after the decision on the Binding, which has been already defined through literals and functions. The SAT solver decides to which unit the *Binding* must point, and the required value of the *BoundProperty* is automatically obtained from that decision, without needing to apply any complex reasoning process. This way, the configuration of *BoundProperties* will be directly obtained from processing the results from the SAT solver.

Constraint Functions. Finally, the constraint functions will be translated and added to the solver. Similarly to *Dependencies*, several aspects from the *Constraints* have already been taken into account at the filtering stage. First, the resource identification of the required resources to the constraint was already evaluated, combined with incompatibility check, to filter out *RuntimeResource* literals which could never be true.

However, additional restrictions must be defined to ensure that the *CLit* elements appear at the solution when required to satisfy the *Constraints* of *RuntimeResorce* literals. This relationship is similar to the one expressed in the dependencies evaluation. If the constrained *RuntimeUnit* appears at the solution, the satisfying *CRC* must also be present. Although they were initially considered, incompatibility must also be evaluated at this point as it obviously alters the defined formula.

This way, for each *CLit* literal, there will be at least one defined function (one for each *RLit* which depends on the Configurable Resource existence for its stability). The following algorithm can be followed for iterating these elements and defining the required functions:

```
for each ContainerResource cr ∈ CLit{
   for each ConstrainedUnit cu ∈RLit, cu constrained by cr
            if(cu.cons.not) //constraint is incompatible
                 addFunction(cu → c̄r)
            else      //constraint is additive
                 addFunction(cu → cr)
    }
}
```

The definition of these additional functions ensures that for any case of the three possible ways of satisfying the constraint resource identification, the solution will be stable with respect to that. However, there are additional kinds of Constraint definitions which required additional restrictions to ensure they are correctly respected by the proposed solution.

First, additional checks must be defined for EXCLUSIVE constraint declarations. For those cases, it must be enforced that at most one of the demanding resources can be instantiated at the *Container* for the configuration to be stable. This was not possible to evaluate during the filtering stage, as it must be evaluated collectively for all the conflicting units which require the same resource.

This restriction can be converted to SAT terms with one additional function for each group of RLits with a *Constraint* of EXCLUSIVE access to the same resource. I will refer to each group of literals with the same exclusive constraint as: $GrpExcl = \{A_i\}, A_i \in RLit, i = 1..n$ The SAT function to respect the special kind of Constraint of these elements can be expressed as:

$$\sum_k A_k \leq 1$$

After defining the functions for exclusive relationships, there is only one kind of constraint base check which hasn't been enforced by the SAT literals and functions: consumption constraints. These constraints can be evaluated similarly to exclusive declarations, as in practice this is a more general case of it. Fortunately although initially it seems that integer variables would be required to evaluate this condition, this is not the case. The variables only have to represent the decision on the existence or not of the consuming *RuntimeUnits*, with each one of them consuming an already established weight. As the PBSAT solver accepts linear Boolean functions, this condition can directly be expressed into the solver.

This way, one additional function will be added for each group of *RuntimeUnits* consuming the same property of the same resource. The resource which is required by the units is called *r*, with an initial capacity of the consumed property *rcap*. The group of RuntimeUnits consuming the resource is $GrpCons = \{A_i\}, A_i \in RLit, i = 1..n$, and the amount consumed by each *RuntimeResource* is $cons(A_i, r) = A_i c_r$. The consumption function of the group of units can be expressed as:

$$\sum_{i=1}^{n} A_i * A_i c_r \leq rcap$$

5.4.3.3. Objective functions

The previous sections have defined a set of literals and functions which, when evaluated with the PBSAT engine, will yield a stable solution, respecting the structure of the affected resources. Finally, the management objectives which must be supported by the environment Configuration will be translated to SAT functions, which will ensure that the proposed solution is also desirable.

The *DeploymentUnit* EXISTS objective functions mandate the presence of specific resources in the runtime environment. These resources are provided by the *DeploymentUnits* from the LRB, so the management system can provide them to the environment configuration. In order to support that type of objective functions it is necessary to identify the *DeploymentUnits* (and by extension *RuntimeUnit* literals) which can provide the demanded resource. The set of RLits providing the identified resource is called $Providers(r) = \{Ar_i\}, i = 1..n, Ar_i \in RLit$. In order for satisfying the objective, at least one of these elements must be present at the proposed solution, which is equivalent to the following boolean function:

$$\bigvee_{i=1}^{n} Ar_i$$

A similar process applies for the NOTEXISTS objectives, which mandate the non existence of the selected resource in the runtime configuration. In order to express this objective the same set of $Providers(r) = \{Ar_i\}, i = 1..n, Ar_i \in RLit$ will be involved. However, in this case the interpretation of the objective won't result in another formula, being directly a false assignation for the affected literals.

$$\forall\, Ar_k \in Providers(r), Ar_k = false$$

The *RuntimeUnit* versions of EXISTS and NOTEXISTS objectives, are much simpler to represent as SAT clauses. The *RLIt* element representing the target of the objective will be evaluated beforehand as *true*, in the case of a mandatory existence, or *false*, in case it should not appear over the environment.

5.4.3.4. Proposed Solution Optimization

The structure, stability and objective-based functions which have been defined over the previous sections ensure that the result of the solver execution will obtain, if possible, a correct configuration of the environment. However, as it was previously discussed, one of the reasons of selecting a PBSAT engine was that, as predictably a solution could be obtained for most problems, it would be valuable to have additional expressivity for declaring additional restrictions to the satisfiability problem which could improve the quality of the proposed solution. This way, I will describe in this section two possible optimizations which can be defined to further improve the control of the desired solution. I have selected two requirements which, although might not appear in every scenario, will be common to many enterprise environments: avoid duplicate instances of the *DeploymentUnits*, and minimize if possible the number of changes in the proposed solution.

The first restriction which will be modeled is the control of the maximum number of allowed instances of each *DeploymentUnit* over the environment. Depending on the characteristics of each unit, and the environment defined policies, the specific requirements might change. For instance, a policy might mandate that each relevant resource should be replicated if possible, enabling a simple configuration for reacting to individual failures. On the other hand, in many scenarios the preferred state is to avoid redundancy and allow only one instance of each element. Those restrictions can be easily enforced by defining additional clauses for each group of *RLit* literals. As an example, I will define the case where only one instance of each unit is allowed in the solution. Let it be $du_a \in DeploymentUnit$, and $\{rua_i\} \in RLit, i = 1..n$, the set of defined literals representing the possible physical instantiations of the unit. In order to ensure that only one is selected, the following boolean linear function must be added to the SAT engine:

$$\sum_{i=1}^{n} rua_i \leq 1$$

The second example optimization addresses an important aspect: the problem of minimizing the amount of changes. Whenever possible, the proposed solution should respect the current state of the configuration while addressing at the same time the desirability and stability concerns. However, the set of previously defined functions does not provide any indication that this would be preferred. The general objective of minimizing the set of changes, although apparently clear, is actually difficult to represent correctly as part of the satisfiability problem, because the actual impact of applying each type of changes usually differs. For instance, the impact of removing a *RuntimeUnit* from the environment is usually much greater that the one of reconfiguring a binding. As the specific details will depend on each domain, I have selected a simplification of that general objective: to give preference to valid solutions that include the *RuntimeUnits* existing at the current environment configuration. This constraint will partially reduce the number of required changes, as whenever a resource provided by a *DeploymentUnit* is required for the solution can be accessed from an existing *RuntimeUnit*, the existing one will be used.

This objective will be translated to the PBSAT engine as the optimization function, as it represents a general goal instead of a strict requirement. The complete optimization formula will be built by the combination of a set of terms derived from each group of literals $\{rua_i\} \in RLit, i = 1..n$, being the potential instantiations of $du_j \in DeploymentUnit.$, with $LRB = \{du_j\}, j = 1..m$. The evaluation of each group will provide the function GRP_j , with the defined optimization function being:

$$MINIMIZE(\sum_{j=1}^{m} GRP_j)$$

The definition of GRP_j functions is obtained as follows. For each group $\{rua_i\}$ of cardinality *n*, if *m* is greater than or equal to two, and one of the literals rua_x represents a *RuntimeUnit* already present at the configuration, a set of summation terms with the literals as variables will be added to the function. The one representing the existing resource will have a low weight (1 in this example), whereas the rest of options will have a weight W, of value greater than 1 (or the weight for the already existing options). The exact weights can be adjusted

based on the specific domain heuristics. This way, when minimizing the function, if possible the existing value will be selected as they have a lesser weight:

$$GRP_j = \begin{cases} 0, & \nexists rua_x \mid rua_x.ru \in Env \ \vee |\{rua_i\}| < 2 \\ rua_x + \sum_{i=1}^{n-1} W * rua_i), & \exists rua_x \mid rua_x.ru \in Env, |\{rua_i\}| \geq 2 \end{cases}$$

The specific value for the weights on the existing and not existing literals cannot be established at this point, because it depends on whether there are additional factors that must also be expressed through the optimization function, in which case a relative ordering among them should be applied, restricting the values of the weights in the optimization formula.

5.4.4. Results interpretation

The final step of this algorithm consists of collecting and interpreting the results from the SAT invocation, in case a solution could be found for the presented domain. Otherwise, the current state will need diagnosis from a manual agent. In those situations, SAT solvers provide debugging mechanisms that expose the chain of reasoning followed and pinpoint the problematic elements.

In this section I will describe the steps which will be taken after obtaining a solution from the SAT. The result from the solver execution is an assignment of true or false to each of the literals which were previously defined at the literals identification stage. The value of each literal represents a decision about the final configuration. The analysis will compare the assigned value to each literal with the initial configuration state, in order to determine the required configuration changes.

However, analyzing the value of all the literals would impose a heavy cost in this final process, which is not actually necessary. The objective of the literals identification was to cover every possible combination. However, as only the changes surrounding the current and future configuration are relevant, most of these elements won't have to be processed, because they represent discarded options. Because of that, strategies to select the relevant literals will be described as part of the activity.

The analysis of the literals will be described in the following order: first *CRLits* will be analyzed *to* determine what *containerResources* must be configured in the *Containers* to satisfy *RuntimeUnit* constraints. After that, the *RLit* results will be evaluated, determining the *RuntimeUnits* which must be present at the final configuration. Once that has been determined, *BLit* results will provide the Bindings configuration among the elements. For each category I will provide a description of the meaning of literal values and an strategy for selecting the relevant literals and the derived changes for each one of the results.

5.4.4.1. Container Resource Literals

CLit literals contain the decisions about the creation of additional *containerResources*, based on a *ContainerResourceConfiguration* definition. They will only be created to satisfying constraint-related restrictions of the *RuntimeUnits* which will be part of the updated environment.

As initially none of the resources identified by the literals exist at the environment, only the literals evaluated to true need to be processed. False literals represent potential configurations which have not been selected, and consequently won't require additional operations. As regards positive literals, for each positive *RLit* a new *RuntimeResource* will be created over an environment *Container*, through a $ADDCRES(cont, crc)$ operation.

5.4.4.2. Runtime Unit Literals

Each one of those literals represents the existence or not of a *DeploymentUnit* at a specific runtime *Container*. Those *RuntimeUnits* can either be initially present at the environment Configuration or be defined as a logical entity, and be compatible with the *Container*. True literals represent the complete set of *RuntimeUnits* which must appear at the environment.

This way, if the literal is true, there are two possible scenarios. If the *RuntimeUnit* appeared at the initial environment configuration, no changes are necessary. Otherwise, it will have to be instantiated from the *LRB* if it did not appear beforehand. This will be achieved by a pair of change operations: first an $INSTALLDU(c,du)$, followed by an $STARTDU(c,ru)$.The latter change will only be applied after all the $CNFBIND$ and $CFDUPROP$ changes have been applied to the *RuntimeUnit*, ensuring that the unit is correctly configured before being activated.

For all those cases it will also be necessary to evaluate the *DeploymentUnit* definition of the affected unit, in case it declares some *ContextAwareProperties*. The value of each configurable *Property* will be obtained by substituting the variables with the actual values from the elements of the updated execution context. The correct values will be configured to the unit by applying $CFDUPROP(c,ru,props)$ changes.

In case the literal is evaluated as false there will be no change required if it was not present in the initial configuration. On the other hand, if the runtime unit was present at the initial configuration, it will have to be safely removed from the environment, by applying $STOPDU(c,ru)$ followed by $UNINSTDU(c,ru)$.

The set of *RLit* elements, because of the nature of the process (considering every possibility), contains many discarded options for the final decision, which in turn are not related to the obtained changes. Because of that, I will propose a searching strategy which will skip the non-relevant false literals. The method for processing the *RLit* values will be the following:

First, every positive literal will be analyzed, as they are the ones representing which elements must appear at the solution. After that, all the *RuntimeUnits* present at the initial configuration will be retrieved, and their corresponding literals will be analyzed. As these elements were already present at the environment, their literals being false would imply subtractive changes to the environment, and thus they must be analyzed. That is not the case for false literals of *Runtime* Units not present at the beginning, as they just represent non taken options which can be safely ignored.

5.4.4.3. Binding Literals

A *BLit* literal represents a possible configuration of a *RuntimeUnit,* which will satisfy one of its dependencies through a *Binding* to the pointed unit. Both source and destination units are guaranteed to exist at the solution thanks to the previously defined rules. For each binding of a *Runtimeunit*, a *BLIt* was defined for every valid binding configuration. From all of them, only one will be set to true because of the structure functions defined. Because of that format of

BLit literals, false ones don't need to be analyzed for derived operations; they either belong to a *RuntimeUnit* which will not exist in the final state, or represent an alternative option for a *Binding* which was not selected.

A *BLit* literal evaluated to true implies that for the final configuration the real binding will be configured to the designated *RuntimeUnit*. If the dependant *RuntimeUnit* did not exist at the initial configuration, or it did exist but the *Binding* it was pointing to a different unit, it will need to be configured by a change operation, with a $CNFBIND(ru,bindId,rub)$ change. As regards execution order of the changes, *Binding*-related operations will be applied after all the $INSTALLDU$ changes have been executed, in order to avoid configuring Bindings to nonexistent *Resources*.

In addition to the base *Binding* configuration, additional changes will be required if the *DeploymentUnit* definition contains one or more *BoundProperties* dependent of the *Binding*. In those cases, the required value for the properties will be obtained and configured through a *CFDUPROP(cont,ru,props)* change. Those changes will be applied after the automated context-based *Property* configuration changes have been applied. An additional factor must be taken into account when executing those changes: the possibility that the value of a *BoundProperty* could be transitively dependant, so that the base configuration value does not originate from the bound *Resource* but from a third one instead. This is the case with resources acting either as proxy (apparently exact copy) or façade (adapted image) of other resources. These intermediation elements are bound to the original *Resource*, so that other *Resources* bound to the proxy or façade will have their configuration equal to the one of the original resource. Because of that, it is necessary to apply the configuration changes for *BoundProperties* transitively, ensuring that the original value is transferred over the whole *Binding* chain.

If Binding-related changes need to be applied to an initially existing *RuntimeUnit*, a $STOPDU(ru)$ operation will be executed before them, in order to avoid hot configuration of the unit. Finally, after the configuration is complete, the *RuntimeUnit* will be activated again through a $STARTDU(ru)$ change.

5.4.5. Wrap-up of the Service Change Identification Algorithm

Over the previous pages I have described the proposed algorithm for identifying the required changes to restore any managed Domain to a correct state. The proposed solution transforms the general problem to a Pseudo Boolean SAT function, with the literals representing the potentially reachable configurations, and the correctness (stability and desirability) restrictions over it expressed as a set of boolean function which restrict the potential solutions. This way, if the solver can find a solution, it is ensured that it will respect all the requirements which are expressed in the information models. Moreover, additional refinement can be supported by defining an optimization function which prioritizes the reachable correct configurations.

The first stage of the algorithm consists of defining the SAT literals and functions, by analyzing the elements from the Domain. A detailed explanation has been provided of not only what elements need to be analyzed but also what strategies can be followed for efficiently processing them into SAT elements. After those activities, the resolution engine is invoked, and a solution consisting of a true-false assignment for the introduced literals is obtained.

The final part of the algorithm consists of obtaining the required internal changes to reach the proposed configuration by the SAT result. In this process the translation of boolean values to the intended state of the Configuration have been provided, as well as a simple strategy to obtain the set of required changes from the assigned values to each type of literal.

5.4.6. SAT-based Change Impact Analysis

The previous section has described how to implement the change identification functionality as a PBSAT problem. However, that is not the only function from a management system which can be solved with this technique. It was described in the motivation for selecting a PBSAT-based solution, how purely logical, dependency resolution processes can be solved with this approach.

In this section I will provide another example of SAT application to automate a management-related process: Estimation of the impact of a single change to the environment (Change Impact Analysis). A change which modifies or removes a *RuntimeResource* from the environment usually affects not only the targeted element but has an environment-wide scope. Changes can propagate over the two structural relationship layers between *RuntimeResources:* the containment hierarchy and the overlay network defined by the *RuntimeUnit Bindings*. Estimating the impact of the change over the containment hierarchy is a simple task, as the potentially affected elements are only the branches and leaves from the tree whose root is the affected resource. On the other hand, the ramifications of dependency relationships are harder to analyze, as they depend on the specific configuration of each one of the *Bindings*, and can span over the whole environment.

In the current information model, the first type of propagating impact applies only when the affected element is a host *Resource*: either a *Node* or a *Container*, whereas the second type can only occur when the change affects a *RuntimeUnit*. Also, it must be considered that changes of the first type can propagate to the second one, but the opposite is not true.

In the remaining of this section I will describe a PBSAT-based process for automatically obtaining the impact of a change to either a single runtime unit or a group of affected units. In order to illustrate the proposed solution, I will use a fictional example, whose details are described in the next picture. The environment hosts three *Nodes*, five *Containers* and seven *RuntimeUnits* (named alphabetically A to F). The units have in total five *Bindings*, configured as shown by the arrows from the picture.

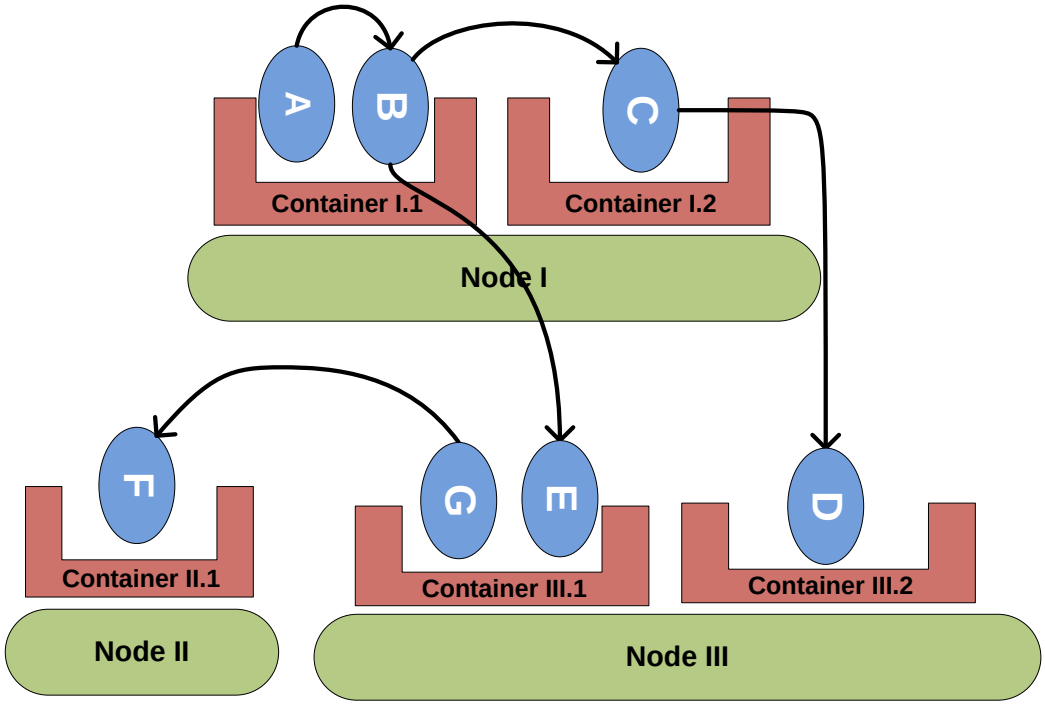

Figure 51 Sample Abstract Runtime Environment

The structural impact of changes affecting the host resources will extend to those in the hierarchical structure. For instance, if Node I disappears, this would imply that Containers I.1 and I.2, as well as units A, B, and C, will be affected.

As regards *Binding*-derived, there are three possible sets that can be identified starting from a unit. The most important set is the one composed by the transitively dependant units (those bound directly or indirectly to the affected unit). Clearly, the stability of all those elements can potentially be affected by any change to the initial unit. In the provided example, if the unit C was affected, the complete set of dependant units (including the source one) would be:

$$Dependant(C) = \{C, A, B\}$$

The second set is composed by the satisfying units, which are transitively bound to the unit Bindings. Although initially less critical, this set can also be relevant to analyze as those elements were providing stability to a *RuntimeUnit* which might not be present after the change. This may imply that these elements are no longer fulfilling a function in the system and thus could possibly be removed. In the previous example, the satisfiers of C would be the following set:

$$Satisfiers(C) = \{C, D\}$$

Finally, the complete graph of Binding related units (obtained by transitively applying both relationships) allows partitioning the *RuntimeUnits* into closed sets that collaborate for providing its functionality, thus providing a useful global view for the manager. Starting from C, the related units would be:

$$Related(C) = \{C, A, B, D, E\}$$

This example shows that the set of related elements cannot be obtained just by the union of the previous two ones, as the transitive nature of the relationships makes its calculation more

complex. Only units F and G are in no way related to C with the information handled by the management architecture.

For any of the three categories, the aggregated impact of changes to several *RuntimeUnit* equals the impact of the union of their sets. This way, in the most general case, the total impact of a change to a *Node* or *Container* can be calculated by first obtaining the set of affected hierarchical resources. After that, for each runtime unit affected, obtain the binding-related changes, and apply a union to all those sets.

Once the characteristics of the problem have been described I will provide the required definitions to handle it correctly with a PBSAT solver. I will only cover the units *Binding* reasoning as the hierarchical impact is immediately obtained. The following terminology will be used: the runtime environment contains the set of runtime units $Units = \{RU_i\}$. The units are related through the set of $Bindings = \{B_i, S = B_i.src, D = B_i.dest, D, S \in Units\}$. Only those sets need to be analyzed to define the SAT problem.

In order to convert the problem to a SAT function I will define one literal representing each element from *Units.* The logical evaluation of these variables will dictate whether that *RuntimeUnit* has been impacted by the source one (true value) or not (false value). Clearly, the same set of variables will be used for obtaining any of the three impact graphs described previously.

After defining the literals and the interpretation of their values, I will define the objective function that will be optimized by the pseudo Boolean solver. As only the affected units must be evaluated to true, the defined function will try to minimize the number of positive runtime units. Again, this is also the case for the three different graphs to be calculated. This way, the following objective function is obtained:

$$Objective = \sum RU_i$$

After those initial definitions have been established, different clauses will be defined depending on the type of impact graph to be obtained. In the case of the dependant graph, for each defined *Binding*, an implication will be added from the destination to the source of the *Binding*.

$$\forall Binding\ Bi\ define: B_i.D \Rightarrow B_i.S,\ \ B_i.D, B_i.S \in Units$$

For the satisfier units it is necessary to traverse the *Bindings* the reverse way, so the opposite functions will be defined:

$$\forall Binding\ Bi\ define: B_i.S \Rightarrow B_iD,\ \ B_i.D, B_i.S \in Units$$

Finally, for obtaining the complete closure of binding-related elements, both relationships must be transitively applied. This way, the defined functions must be:

$$\forall Binding\ Bi\ define: B_i.D \Leftrightarrow B_i.S,\ \ B_i.D, B_i.S \in RuntimeUnits$$

Up to this point, all the necessary restrictions have been established. However, one last statement must be defined to the solver in order to obtain a solution, as the current set of definitions will always evaluate every literal as false. This is the case because none of the aforementioned concepts specify which unit or units are the source of the impact analysis. This can be expressed simply by evaluating to true the literals representing the directly affected

elements. One of the main advantages of adopting a SAT engine for addressing these calculations is that, because of its declarative nature, the impact from multiple originating elements can be evaluated simultaneously just by setting each corresponding literal to true. This way, in order to select the units to be evaluated, the final assignments have to be made:

$$\forall\, RU_i \in AffectedUnits, \qquad RU_i = true$$

5.5. Conclusions

Over this chapter I have provided a complete characterization for service management purposes of the two main affected entities. First, the management Domain has been defined, consisting of the combination of a set of logical resource definitions, contained in the LRB, the currently available runtime resources, which constitute the runtime Configuration, and the established functional objectives for the Domain. All the relevant information is defined over the base concepts introduced at the previous chapter, which enables an automatic reasoning over the managed environment. This way, a function has been defined which determines whether the state of the domain is correct or not, taking into account both the functional objectives and the stability conditions.

The role of the management system has also been characterized as a corrective agent who can invoke internal changes to restore the correctness of the domain state, after it has experienced external modifications. in this model the set of allowed operations for the service management system have been defined, and a complete algorithm has been proposed which automatically analyzes the domain state, finds a reachable correct configuration and proposes the set of required changes to alter the domain and obtain the selected state. The proposed solution is based on a problem resolution technique of proven maturity, which is adopted in areas such as logic circuit design or dependency management of complex software distributions.

6. Reference Architecture

The previous chapters have detailed an analysis of the complexity of automating the change and configuration operations of enterprise services. In order to cope with those problems, initially the characteristics of general management systems and the enterprise services were analyzed, in order to obtain a model abstraction of the relevant information. This way, the terms of the problem are clearly described, using a common set of definitions for the managed assets and the business objectives. On top of them, the function of the management system was described and an algorithm to automatically solve its main activity, the service change identification, was described. However, no guidelines have been provided to implement those concepts in a specific scenario.

The objective of this section is to propose a reference architecture for an enterprise service change management system. The architecture will be based on the models and reasoning processes detailed in the previous sections. The description will only focus on supporting the functional characteristics of the system, abstracting over aspects such as security, reliability or auditability which are common non functional requirements to enterprise systems. This has been decided in order to provide a more focused view of the architecture.

6.1. Architecture Description Model

Before starting the architecture description, I will select a view model from the literature which enables a clear description of the architecture of the service change management system. This way, I will first analyze the best known model in the literature: the 4+1 view model established by Kruchten [64].

This model consists of four base views which focus on different aspects of the architecture: The logical view describes the functionality provided by the system, through the use of UML class and /or sequence diagrams. The Development view provides a logical representation of the architecture, detailing its internal structure into development packages. The Process view focuses on the runtime behavior of the system, reflecting how the different elements interact and communicate. Finally the Physical view covers the physical deployment aspects of the proposed architecture. These four aspects of the system are complemented by the scenario view, which illustrate the architecture description through a set of use cases or scenarios, putting into context the other four views. The model allows a certain degree of flexibility, conceding that for specific cases, some of the views might not be necessary as the remaining set comprise all the important details.

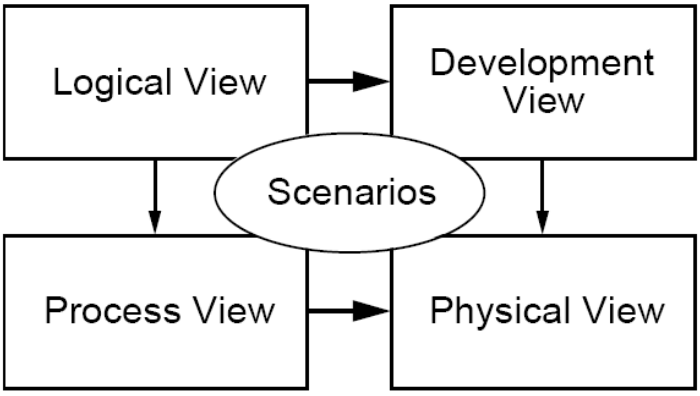

Figure 52 The 4+1 Architecture View Model
Extracted from [64]

The 4+1 view model has the greatest acceptation among the proposals of the literature. However, it is intentionally a generic model for describing the architecture of any system, abstracting the specific characteristics of each domain. Because of that, there is room for additional view models that provide a more specialized focus. In the management domain, the most accepted model for describing management systems is the OSI management framework for integrated management of networked systems [60]. Although the OSI management architecture has not become the dominant solution, the high level concepts created for its description have proven to be flexible enough to be adopted to describe any management architecture [42]. This description model documents a management architecture by detailing the following four basic sub models:

- Information model: Describes all the relevant management information. Management models define the characteristics of the managed objects. The model must include all the relevant information for the management operations, the elements, its characteristics and relationships.
- Organization model: Defines the relationships between the managed entities, collectively referred as environment, and the management infrastructure. Depending on the domain approaches vary from central organization, to hierarchical entities to completely autonomic management.
- Communication model**:** Defines the exchange of management information. This concern includes both the communication patterns (notifications, polling, synchronous operations), and the exchanged information between the elements.
- Functional model**:** Defines the management function areas supported by the architecture. This includes the detail of the functions supported by the architecture, as well as how they are achieved through the collaboration of the different components.

As it was described in the state of the art introduction, management can be understood at different levels of abstraction, ranging from business processes to technical configuration activities. Because of that, in addition to the OSI partition, the abstraction level of the views should also be specified. In [97] three reference perspectives are proposed, process view, system specification and technology model, going from the most abstract to the most concrete one. The process view focuses on the high-level processes (similar in abstraction to the ITIL Service Management and Service Delivery process views). The system specification perspective

details any required information to design a management system, except for the aspects which are platform specific, covered by the technology perspective. This presents some similarities with the PIM (Platform-Independent Model) - PSM (Platform-Specific Model) characteristic of MDA processes. Clearly, in the context of this dissertation, the selected perspective will focus on the system specification. The combination of OSI sub-models and description perspectives is presented at Figure 53.

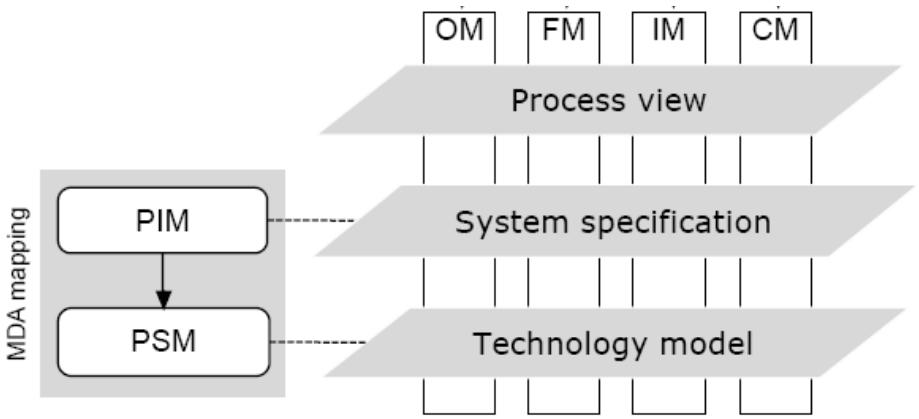

Figure 53 View points of a OSI Management Architecture

Although initially 4+1 and OSI might seem to be quite different, after analyzing the information which is provided by each view /sub-model there are clear mappings which can be established between both models, considering that one is generic and the other focused on the specifics of management. The logical view completely characterizes the functionality of the system. This information must be contained in any architecture description, as is the case of the OSI model. Two sub-models of OSI are roughly equivalent to the logical description: the information model describes the data which is being handled by the architecture, and the functional model details the possible operations. The process view is represented by the communication model, which in this case is specialized to focus on the main communication mechanisms between the main components of the architecture and the agent infrastructure. Similarly, the physical view is similar to the organization model information, which details the multiplicity and distribution of the instrumentation infrastructure and the main management functions. The development view is not present at OSI model, although that information can be covered in the functional model. Finally, the scenarios are missing from the OSI model, which does not define any mechanism for linking the different information views.

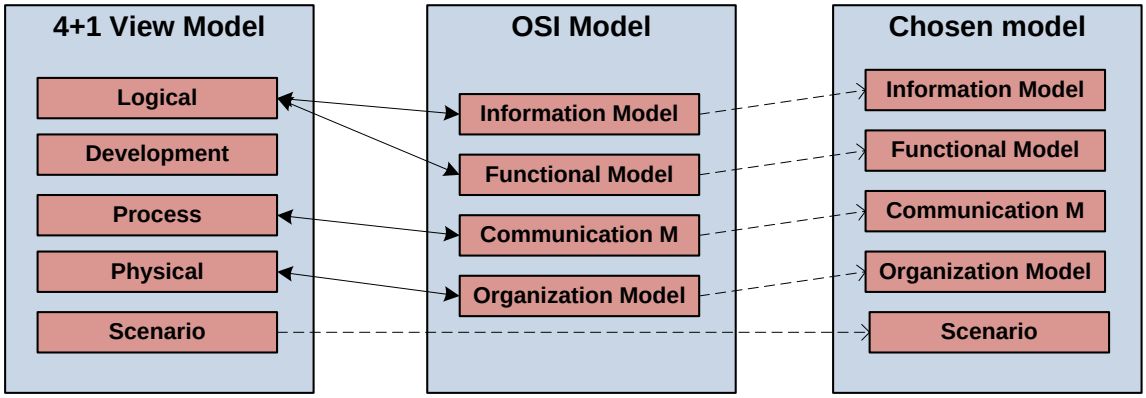

Figure 54 Coverage Comparison between the Architecture Description Models

After analyzing and comparing both architecture description models I have selected a mixed approach for detailing the management architecture. As a base reference I will adopt the OSI model, with the addition of the +1 scenario view to better link those concepts. OSI has been selected as the base model because it adapts better to the management concerns, and at the same time reflects the key aspects covered in the 4 views. Development view information will not be contributed, as it is sufficiently covered with the information provided in the other models. However, the scenario view will be also adopted in this description to clearly reflect how those factors come into play.

Once the description model has been selected, the following sections will describe each sub models, with the scenarios providing the final link.

6.2. Information Model

The information model of the reference architecture characterizes every aspect of the domain which is relevant for the supported management operations. The metamodels must be generic enough to capture the variability present in different domains, while at the same time retaining a level of expressivity that allows capturing every relevant detail. As those metamodels have been defined over the previous chapters I will provide here just a recap of each one and a reference to the full description. The reference information model is composed by the software model, the runtime model and the objective model.

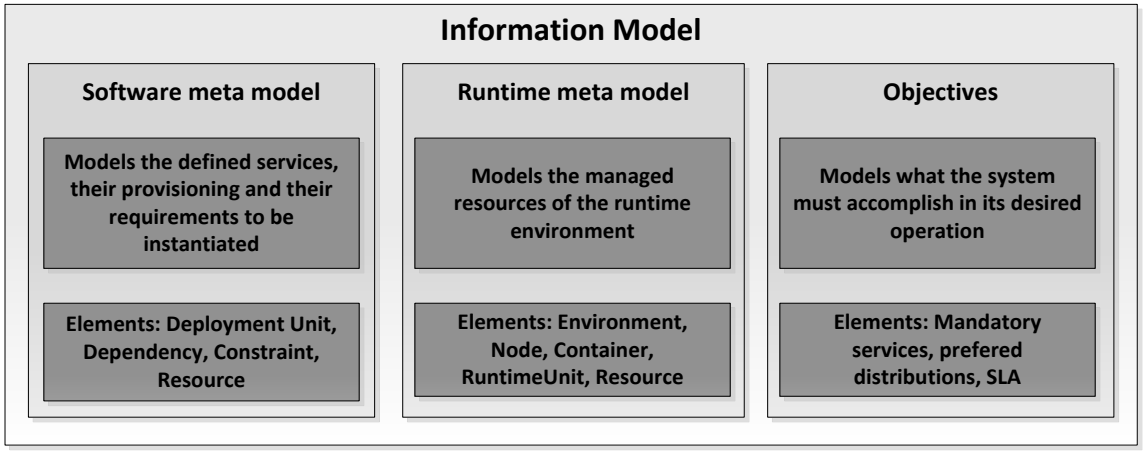

Figure 55 Reference Architecture Information Model

The software model provides information about the logical elements which can appear instantiated at the managed environment. Model details are provided in Section 4.5.1, Service Deployment and Configuration Model. The basic elements of the metamodel are resources, which represent software services, libraries and additional assets. Deployment units aggregate those logical resources and represent the actual elements which are provisioned to the environment, thus becoming the focus of the deployment operations. The model allows expressing both the provided resources for each Deployment Unit and the restrictions on the rest of the environment or them to be instantiated correctly. The elements of the software model are represented at Figure 37.

The environment model allows characterizing the runtime execution environment where the services are deployed. It is based on the common abstractions defined by the most relevant standards in enterprise modeling (such as WSDM, CIM and D&C), while at the same time extending the resource semantics so that there can be links between the runtime elements and the logical definitions. This way, it is flexible enough to support the modeling of environments with very different topologies and characteristics. It must be noted that, although there is no explicit dependency between two metamodels, the elements from the logical model appear instantiated at runtime, and at the same time the stability of the structure of the environment model is restricted by the constraints and dependencies defined in the software model. The elements of the runtime model are described with details in Section 4.5.3, and its main elements are depicted at Figure 41.

The Objectives model captures the established definitions of what the environment as a whole must do, in order for it to fulfill its intended purpose. As it this was the case with the other ones, they are also expressed over the base concept of resources. The different types of objectives are described in the section 5.1.1.

Those three models enclose the main management concepts that must be taken into account for managing the service configuration changes over a distributed environment. Although additional, specialized models might be used by some of the architecture subsystems, they will be derived from the information contained in those three.

6.3. Organizational Model

The organizational model identifies the main entities of the management domain, as well as their base relationships and the cardinality orders between them. The selected model will be influenced by the main characteristics of enterprise environments, which naturally lead to identify these main elements and their relationships.

The management architecture controls the managed system, also referred as the runtime environment. Security and reliability requirements in the enterprise infrastructure cause a multiplication of managed environments; systems must be isolated for a greater resilience against external attacks, and production environments are complemented with pre-production and integration replicas, in order to ease the applications functional testing without affecting the availability of the already provisioned services.

It has already been described how the complexity and heterogeneity found in the environments is managed by the abstraction into the runtime information models previously

described. However, there is a gap between the core management systems, which process those generic elements and the low-level, specific information of the environment. This mismatch will be addressed by a third element which acts as a bridge between them: the agents constituting the instrumentation infrastructure. Its role is twofold; on one hand, it must capture all the relevant information from the managed runtime elements, and convert it to the runtime model, enabling its processing by the core management systems. On the other hand, it must be able to receive the changes identified by the main systems and apply them by invoking the specific management interfaces of the concrete runtime elements.

The described relationships link the whole environment with the management functions. However, in principle that should not be necessarily the case, as an environment is not a monolithic element but a set of nodes with computing power. Each node provides multiple resources, and containers, which on top of that also contain runtime units and services. In principle, it seems there could be specific managers for each node of the environment, each of them reasoning only over a limited set of the information. However, this is not possible as distributed dependencies span over the whole environment, ignoring the containment hierarchy, and objectives are defined and must be maintained over the complete set of resources that constitute the environment. Fortunately, the existence of multiple, specialized environments instead of a single one per organization allows this centralized model to be adopted without scalability concerns about the size of the organization. The following picture shows how the management architecture will be organized with respect to the managed environment.

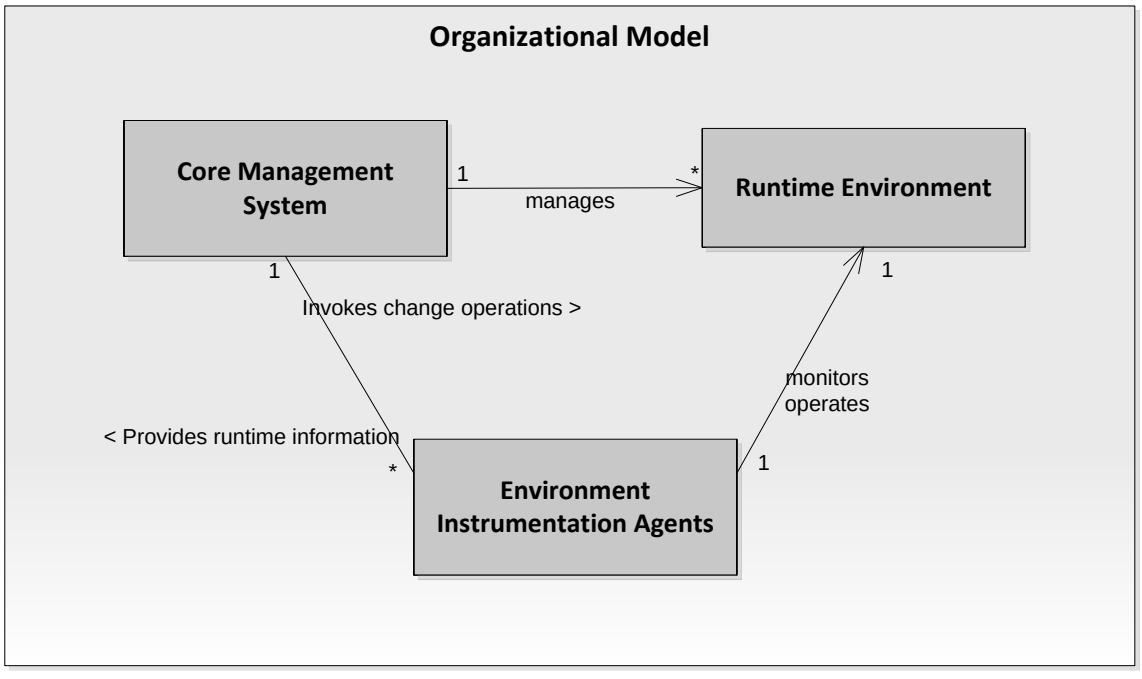

Figure 56 Reference Architecture Organizational Model

6.4. Communication Model

The communication model describes the management information which is exchanged between the different entities, as well as the type of messages and the communication mechanisms.

The organizational model has identified three main elements (core, agents, environment), and subsequently three potential communication channels, as shown in Figure 56. This way, the first step to describe the communication model will consist of analyzing the nature of these three connections. First, from the previous description it becomes clear that the core does not directly communicate with the environment, as every operation is executed through the instrumentation agents, which act as an adaptation layer between the model-based core and the real runtime elements. On the other hand, the core-to-agents communication channel plays a fundamental role, and must be thoroughly covered in this view. Finally, although undeniably there is a great deal of communication between the agents and the environment it cannot be detailed in this model, as the communication details will be different depending on the technology of the specific managed elements. Moreover, as the architecture must be able to adapt to multiple, different heterogeneous systems, it is unfeasible to know and describe them beforehand as part of the architecture description.

In addition to the inter subsystem communications just discussed, these entities also present complex internal communications requiring specific mention and detail in this view. This is the case with the instrumentation agents, required to work under a high degree of uncertainty about the actual managed environment. In order to cope with the fact that the topology and characteristics of the environment are unknown at design time, the agents instrumentation infrastructure has been broken down into a hierarchy of specialized agents that collaborate and communicate to mediate between the abstract concepts and the low-level information and operations. Therefore, the internal organization and communication of these agents will also be covered as part of the communications model description.

The following structure will be followed to describe the different facets of the communication model: First, the communication between the core and the instrumentation will be covered, detailing how both monitoring and operation are supported. Once these aspects have been sufficiently covered, the internal details of the inter agent communication will be explained.

6.4.1. Core to Instrumentation communication model

The organization view has identified the two basic types of communications that occur between the core system and the instrumentation agents. Monitoring collects information about the runtime environment, providing to the core system a model of the current runtime state. Change operations consist of the core ordering the execution of a set of changes to the environment through the agent infrastructure.

As both communication channels are part of the management architecture, there are no third party concerns, thus providing a high degree of flexibility in the selection of the underlying management protocol for these operations. The Web Services stack would be the most general option for providing the remote communications between those two entities. Over the State of the Art analysis standards such as WSDM were described, which could be adopted by this architecture. However, it should be considered that these XML-based solutions impose an important performance penalty when compared to other alternatives, which sacrifice interoperability for efficiency. In this second category, depending on the selected technology environment for the implementation of the architecture, there are multiple standards in the field of enterprise applications, such as the JMX (Java Management eXtensions) stack [32], which provides a simple information and management model. Because of that, a specific

protocol will not be selected in this architecture description. However, this view will address the characterization of the exchanged information and the communication patterns between these entities.

6.4.1.1. Monitoring Communications

Monitoring operations allow the core architecture to keep synchronized the internally managed model of the environment with the current status of the physical environment. Depending on the specific scenario, there are two communication patterns for monitoring information: push and pull mechanisms.

Pull communications are initiated by the core system, requesting from the instrumentation infrastructure an updated snapshot of the environment information. The push model works the other way around, with the agents actively updating the information of the environment as the changes are detected.

Whereas the initiation of pull requests depends on the internal functional details of the core architecture, additional information can be provided about the triggering of push notifications. There are two, compatible approaches that can be applied depending on the characteristics of environment, and the severity of the changes. The simpler alternative consists of automatically scheduled notifications, which periodically inform the core about the status updates of the system. This kind of notifications must be implemented with an adequate balance between the frequency of the changes (a too low frequency lowers the usefulness of the status updates) and the traffic load imposed to the network (too high frequency can cause network congestion when combined with a large environment). The second approach to the pull model consists of asynchronous alert notifications, sent whenever some specific changes occur at the environment. Because of the generality of the proposed architecture it is not possible to detail at this point what specific triggering rules should be enforced, as that depends on multiple factors about the domain, but some examples can be provided. For instance, any structural change, consisting of a variation in the number of resources available at the environment, should be immediately notified, as it usually has a large impact over the environment state correctness. Alterations to property values cannot be globally decided on, as the severity of the change depends on both the information reflected by them and the specific concerns of the management system. In an actual implementation, a subset of them, identified and linked to several SLAs, should be immediately propagated, while changes to the rest would be notified at the regularly scheduled updates.

As regards the exchange of information related to monitoring, only the messages emitted by the agents contain attached information, composed by instances of the environment model, either representing the complete current state, or the runtime changes since the latest communication. Pull requests message size can be ignored when compared to the incoming downstream information.

6.4.1.2. Internal Changes Communication

When the scope of the management system was defined, the internal changes that could be initiated by it were identified and detailed. However, it is clear that for reacting to any external change, it won't be sufficient to order one of those operations, but a combination of them. In addition to that, over the change identification algorithm several restrictions in the execution order or timing of these primitives were described. For instance, if the objective is to provide

an active service, it will be at least composed by an install activity, followed by a start activity. All those factors have been captured in the definition of the change plan model. This model allows defining these complex changes and provides a simple mechanism for expressing the set of required activities, with all the inter-dependency information that ensures they are executed correctly. The next figure shows the elements of the change plan model.

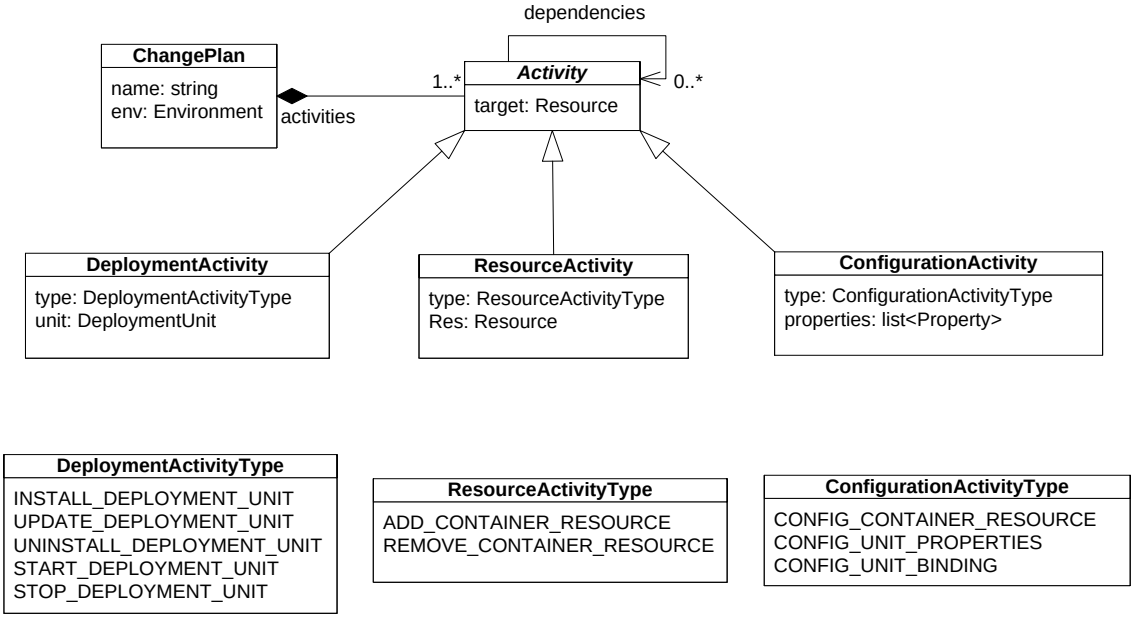

Figure 57 Change Plan Model

A *ChangePlan* is uniquely identified by a name, and is executed over a single Environment, which must be explicitly defined. This way, both for scheduled executions of a plan, or post execution storage, the base target of the plan will be registered. The application of the plan to only one environment is consequent with the monitoring and reasoning scope.

A plan is composed by a set of activities, which correspond to the atomic change operations that can be initiated by the management architecture. They are represented in the model by the *Activity* abstract class, which identifies the target *Resource* from the runtime environment where it will be applied. As the environment model structure is composed by several subclasses of resource, all of them can be identified by this target. The specific set of operations will be defined through *Activity* subclasses, which identify the type of primitive, provide additional information about the required parameters for executing the activity and restrict which resources can be targeted for executing it. The set of subclasses must support the ten different internal change operations which can be invoked by the management system.

The model classifies these primitives into three main groups, which have in common the required parameters. *DeploymentActivities* govern the life cycle of the *RuntimeUnits* at the environment. They include installation, activation, update, deactivation, and uninstallation from the runtime environment. In all of them, the target resource is the *Container* which hosts the runtime unit.

ResourceActivities modify the resources of the runtime *Containers*. They allow removing *containerResources* from the host, or creating them by specifying the definition from a *ConfigurableContainerResource*, plus a set of additional parameters. The input received by these activities is the complete resource that will need to be created, or deleted. As it was the

case with the previous set of changes, the target of those activities is the *Container* where the resource will be created or removed.

Finally, *ConfigurationActivities* modify the configuration of the existing resources at the environment. Their input parameter is a set of *Properties* that will be applied to the selected resource. Depending on the type of activity, the targeted resource will either be a *Container Resource* (modifying its properties), or a *RuntimeUnit*, configuring either its properties or one of its *Bindings*.

Up to this point the contents of the plan have been described, but the mechanism to express the restrictions on their execution must still be provided. Before detailing the selected approach I will provide a brief overview of the alternatives found in the literature, which were previously discussed in the state of the art section.

The most classical approach for restricting the execution of multiple activities is through temporal planning [4]. This way, each activity is scheduled for execution at a specific time interval. Time ordering techniques require knowing with a high degree of confidence the estimated time to complete each activity. There are several proposed techniques for addressing this problem, including the use of SAT solvers [15]. However, enterprise environments have a large degree of uncertainty that complicates this process; the diversity in types of containers (and vendor implementations of the same standards), the variable load of the systems and the difference in the actual hardware resources render this approach unfeasible. That type of solution is adequate for homogeneous, predictable environments, but cannot be adopted in this case.

Other approaches model the graph as an executable process, composed by a set of plan activities with several dependencies. In order to express them, a possible alternative is adopting Gantt-like semantics for defining the types of dependencies between primitives: FS (Finish to Start), SF (Start to Finish), FF (Finish to Finish), SS (Start-to-Start), as it is described in CHAMPS [56]. Although with less expressivity, these concepts are already present in the deployment and configuration of enterprise production systems. Manually defined changes are applied through configuration scripts that support simple dependency expression mechanisms, such as Ant target dependencies [70]. In this model, each target (representing a single operation) declared what other targets from the script must be successfully executed before it is invoked; Ant dependencies are semantically equivalent to FS Gantt Dependencies - cannot start until the other one has finished. It must be noted that neither mechanism defines the exact execution order of the activities, as non-dependent ones can be correctly executed in any order, or possibly simultaneously in parallel, for trying to improve the performance.

The difference between these two approaches lies in the required expressivity for the dependency definition, as Gantt-style semantics allow to express small optimizations in the simultaneous execution of the two activities. Depending on the nature of the operations, the simple script dependency model can be enough, or the Gantt more expressive semantics can be used instead. In this case the more simple approach has been selected, as those additional semantics were not applicable to the constraints that must be reflected between the activities. This way, in the model, each activity can express any number of dependencies to other activities. At execution time, one activity cannot be executed until all their dependant activities have finished executing correctly. This mechanism ensures a correct execution order of the

activities, while at the same time supports different interpretations. Activities can be sequentially ordered and executed, or could be split into several, independent sub processes, and executed in parallel for an optimized performance.

Once the change plan model has been completely characterized, the rest of details about internal change communications can be easily described. The core management system defines a set of changes in the form of a plan, which is delivered to the agent infrastructure for it to be executed on the specific environment. Once its execution finishes, the agents must also provide a report on the execution, informing about the successful execution the activities or the detected problems.

6.4.2. Environment Instrumentation Agents communication model

The agent infrastructure must be able to efficiently instrument enterprise environments, while at the same time being prepared to support the variability among different instances. This way, the infrastructure must be flexible enough to automatically adapt to the distribution of the resources over the environment, as well as their complexity and heterogeneity. With those requirements in mind, I have selected as the base input the agent infrastructure described in [16]. This proposal consists of a tiered, multi agent infrastructure that separates the technology-specific communications from the environment topology and coordination roles of the information. Over the following paragraphs I will provide an overview on the specific agent types, describing their role and their internal communications for collaboration and coordination.

The low level information from the environment resources is collected by the gatherer agents. They directly communicate with the runtime elements through their management-specific protocols. Gatherers function at each node of the environment. There are two base types of low level agents, the Container Gatherers detect the running containers, and abstract the specific information as *Container* elements from the runtime model, including all the internal services and *RuntimeUnits*. Similarly to them, Resource Gatherers collect information about the node resources. There can be multiple gatherers providing different kinds of information, such as operating system details (version, name, or installed libraries) or hardware resources (static capacities and available free resources).

The information collected by all the Gatherers of a node is aggregated by the Node Manager**,** which builds a *Node* model instance with all the combined information. Whereas Gatherers are tied to the specific technology they monitor, the node-level agent is completely generic. This separation allows the instrumentation layer to support different environments without the complete system being bound to the specific details of each one.

The central element is the Environment Manager, which coordinates the activities of the monitoring agents and communicates with the core management systems. The main role of this agent is to build an instance of the *Environment* model by aggregating the node information from each Node Manager. In order to do so, this agent must be aware of the running Node manager instances. One possibility would be that an administrator defined manually the environment topology, and stored into a configuration database. However, this is a costly task. Moreover, those descriptions would need to be updated whenever a change altered the environment composition. Instead of that, a Discovery Service is used to enable automatic discovery between the Node Managers with the central Environment Manager

agent. This mechanism simplifies the provision of the agents to the managed environments, as well as reducing operation and maintenance costs, as it automatically adapts to runtime changes. However, in some restricted environments a completely automated discovery is not possible, because of the use of internal firewalls that filter the broadcasting messages. In order to support those scenarios, the Environment Manager also supports manual configuration. The Environment Manager monitors the discovered Node Managers, and in case the communication with one of them is lost, the node information is removed from the general model and the core management system is notified if the communications are not restored in after a number of retries.

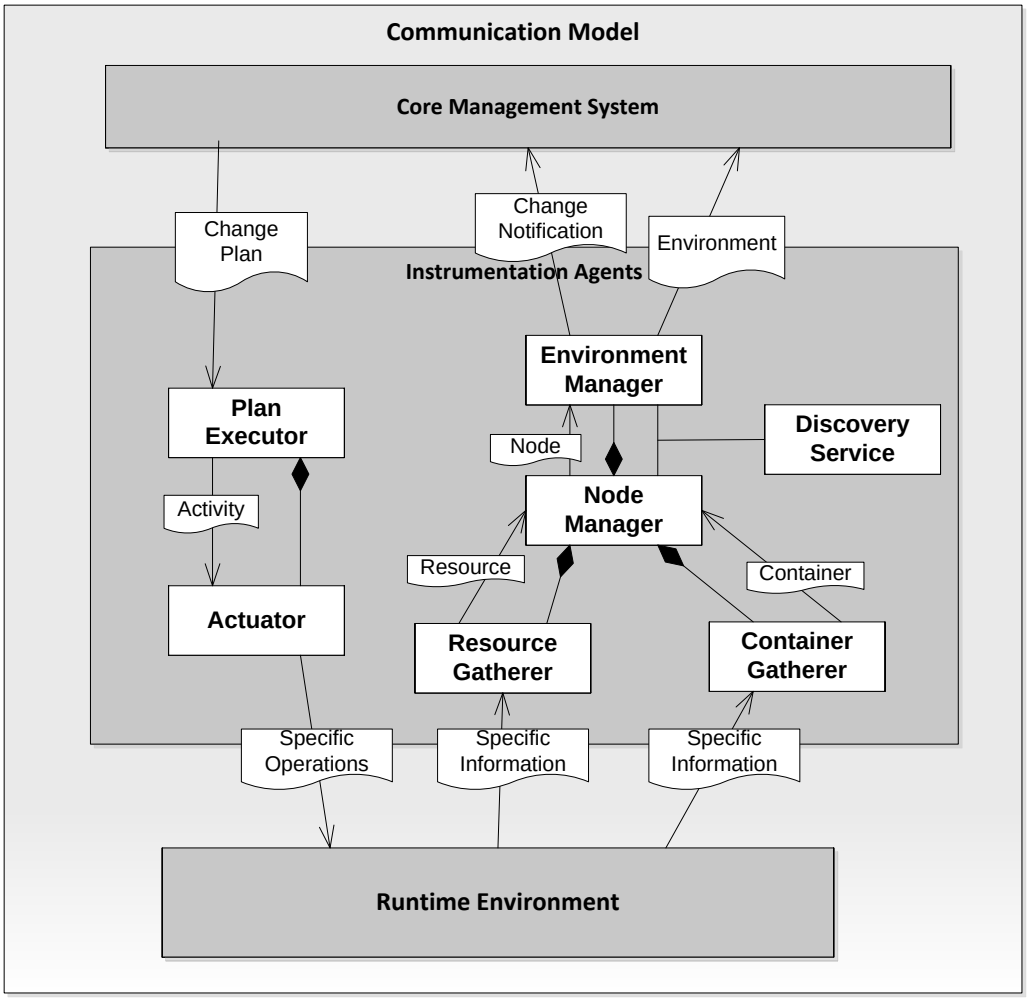

Figure 58 Reference Architecture Communications Model

Regarding the change execution agents, a simpler approach has been applied. As the environment topology is already known because of the instrumentation agents, a simpler structure can be adopted. Every activity is executed either over a *Container* or, a *RuntimeUnit* running over a *Container*, thus increasing its homogeneity for change execution purposes. With these factors in mind, the following agents have been defined: The Plan Executor is the main element. It receives change plans, and dispatches the plan activities to different Actuator agents, which translate the generic operation into protocol-specific operations to the targeted *Container*. The Plan Executor performs the matching between activities and specific Actuators

through a simple mechanism: the Actuators register on the Plan Executor declaring which container types and operations they support. This way, the Executor can obtain compatible Actuators only by looking at the registry information. It must also be mentioned that, the change execution agents do not necessarily have to be bound to one specific environment. First, change execution agents do not need to store state information between one execution and the next. Secondly, the technology specific management interfaces allow for remote invocation of the operations, decoupling the Actuators from the Environment topology. The combination of these two factors allows reusing a single instance of both the Plan Executor and the Actuators for specific technologies among all the managed environments.

The general view of the communications model is shown in Figure 58, which describes the communications between the three main elements, as well as the internal detail of the exchanged information between the different members of the instrumentation agents. Both monitoring and change execution communications are described.

6.5. Functional Model

The functional model describes how the architecture of the system supports the required set of management functions. Whereas the details of the instrumentation subsystem were provided by the previous models, this view focuses on the internal structure of the core of the management architecture. The description of this model will follow a top down approach, first providing the general view and describing the main characteristics of each element. The main entities of this view are the components - or functional blocks - that provide the functionality of the management architecture. The model will detail those elements, describing their responsibilities, their collaboration for fulfilling the supported scenarios, and the managed information by each of them.

Figure 59 provides a high level view of the functional architecture. The core is composed by eight components which collaborate to provide the required management operations. Those entities can be seen in the central part of the picture, as well as the internal communications among them. At the top and bottom of the figure, the main elements of the domain knowledge are represented, detailing their relationships with the described components (creation, storage, manipulation). Once the general view has been established the specific responsibilities and relationships will be detailed for each element:

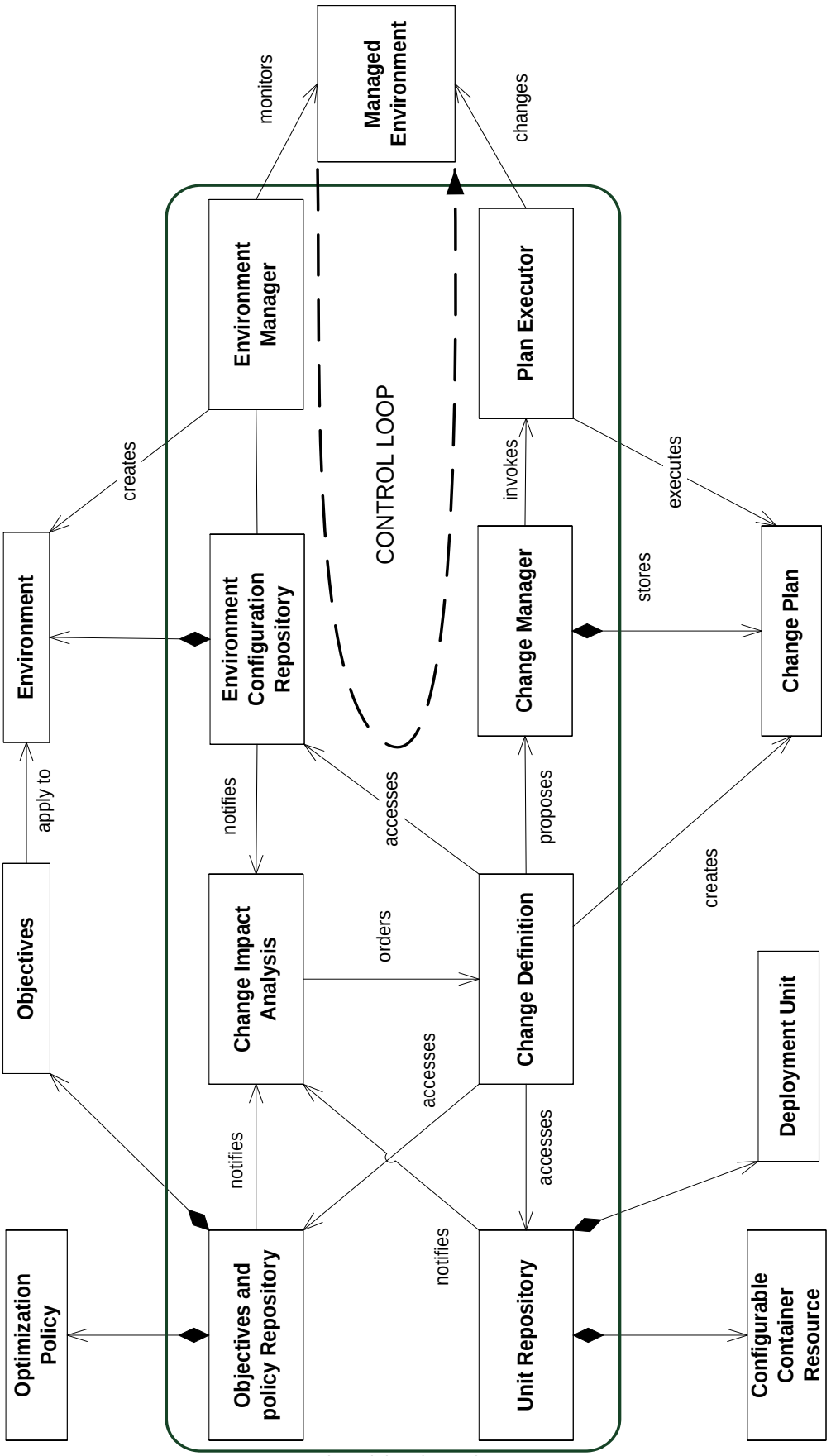

Figure 59 Functional Model High Level View

The Unit Repository is the component which centralizes the logical information known as the LRB in the previous chapters. Its main function consists of storing the definitions of all the *DeploymentUnits*, logical *Resources*, and *ContainerResourceConfigurations*. This information is made available to the rest of core components through access interfaces. The repository allows contributing, modifying or removing logical definitions from the repository through a CRUD (Create-Read-Update-Delete) interface. This access interface is used by the organization's development infrastructure, so that the just developed *DeploymentUnits* can be automatically registered at the repository as the final step of the development cycle. This way, both systems of the internal organization infrastructure are connected. External repositories from service providers can also be connected to the repository, as long as they provide logical descriptions for the repository elements. Logical descriptions are persistently stored by the repository, while physical elements are managed by traditional artifact repositories.

The repository provides one additional function which is invoked during the different repository update operations previously described: Internal repository stability checking. This operation analyzes the set of defined units at the repository, and checks that every logical dependency of the internal elements can be satisfied by at least one from the remaining set. Broken dependencies would cause the affected units to be unstable, thus rendering them unfeasible for any change operation. Because of that, newly uploaded elements will be preliminary checked so the overall stability of the repository is kept after the change.

The Environment Configuration Repository is the component where the environment model information is stored. This block retrieves the runtime information by communicating with the Environment Manager agents (multiple ones as the architecture manages multiple environments, and each one is monitored by its own manager). As it was mentioned in the previous section, the exchange of information can both be requested by the Repository, and provided to it through a publish-subscribe mechanism. Information for each environment is persistently stored, not only the latest snapshot but the complete evolution over time of the environment. Finally, it provides an access interface for the rest of core services to browse, query and retrieve the runtime information.

The Objectives and Policy Repository captures all the high level knowledge governing the service configuration change processes, in a way that it can be processed by the remaining functional blocks. The stored information specifies the role of each runtime environment (the management objectives) and how it should do it (optimization policies). As the objectives are specific to each environment, they will be not only defined but also matched in the OPR with the environments where they are operational. On the other hand, optimization policies, which influence the change identification process, can either be specific to one environment or global. The component persistently stores both types of information, provides manipulation interfaces for their modification, and access interfaces for consulting them by the rest of the architecture modules.

The Change Impact Analysis component controls the actions of the management architecture. Its main function consists of evaluating the severity of the external changes occurring at the managed domain. In order to do so, this component receives notifications from the three repository components, each time a change in the defined deployment units, the runtime information or the defined objectives occurs. These changes are analyzed, estimating their

criticality. If as a result of the change the managed environment is no longer at a correct state, the CIA will order the Change Definition component to prepare a change plan that restores the domain to its intended functionality. This component enables an autonomic management of the domain. However, it must also be designed to support enterprise domains with restrictive policies disallowing the completely automated execution of internal changes. Those cases are supported by the definition of decision points where human validation can be requested after the impact analysis has been completed.

In addition to the automated analysis of external change this component also provides runtime impact estimation tools, which are used to aid the decision of whether to apply changes that affect the environment. This function, starting from a *RuntimeUni*t which will be modified, obtains the *Binding* graph of units potentially affected by the change. An algorithm for automatically obtaining these graphs was described in Section 5.4.6.

The Change Definition component analyzes the current state and creates a change plan which can restore the system to a correct state. In order to obtain the required changes it communicates with the repositories to retrieve the domain information (defined deployment units, configurable container resources, objectives, optimization policies and environment information). Internally, it configures a pseudo-boolean SAT solver, converts the domain information into variables and clauses and obtains a solution for it. These results are finally analyzed and interpreted as an internal change plan. The Change Definition functionality is the most complex from the management subsystems. Its foundations have been completely detailed in the chapter 5.4.

The Change Manager governs the execution of internal change operations to the managed environments. The component provides a plan execution service, which allows ordering the execution of a defined change plan to the target environment. The execution can either be instant or scheduled to a later time, depending on the characteristics of the environment and the internal policies (e.g. in an integration testing environment changes should be applied as soon as possible whereas changes to production environments are usually scheduled at concrete times). After a plan has been completely executed the change manager collects and stores all the information about the execution outcome, enabling internal traceability on the changes initiated by the management system.

In order to prevent plan expiration problems (whenever the interval of time between plan creation and execution increases), and protect the runtime environment stability, every plan dispatched to the manager will be validated to verify that it is consistent with the actual state of the environment. In case there are some warnings or errors during the validation, the execution will be either aborted or suspended for manual review.

The Plan Executor receives a change plan and executes it over the designated environment. Its internal details were described in the communications model. However in addition to environment adaptation, there are additional requirements that further complicate the internal function of plan execution. The change plan model only defines a set of partial orders, but provides some flexibility about the exact way plan activities are executed. In the case of this architecture there will be no parallel execution of activities, instead the executor will focus on preserving the stability of the environment as much as possible. In order to do so, plan execution will be transactional, automatically taking back the environment to its initial state in

case there is a problem during the execution of the changes. On top of that, for auditing purposes, the result of every change operation result is registered and provided back to the Change Manager in the form of an execution report.

The executor will be able to ensure transactional execution as long as only one plan is interpreted simultaneously and the following characteristics are also met. The execution of each single activity must be performed atomically, and its correct execution can be validated. If these conditions are met, after the execution of each activity the state will either be the initial one (if execution failed) or a new state with the operation successfully applied. The transactional executor will keep a stack with the successfully applied activities. In case of failure detection, the executor will automatically obtain a compensation activity for each one of the successfully executed activities, and will execute them in reverse order to cancel the changes applied by the partial execution of the plan.

6.6. Scenario View

Once the main characteristics of the service management architecture have been described over the four presented views I will describe two representative scenarios of the use of the service configuration change management system. They will be used as reference to clarify how the architecture operates, and the way the different previously described models allow supporting the operation of real use cases. Each of them will first introduce the general context where it is defined, followed with the scenario description and the explanation on how the architecture supports it.

6.6.1. Critical Enterprise Service Update Process

A banking company bases their business processed on a renewed core banking system, based on service-oriented architecture principles. This system supports every company service, including B2B (business to business), bank staff services, end user internet banking and cashier operations. The specific services, such as credit concession, account management, bank transfers, loan request and concession, mortgage, or financial services are provided by components and services internally developed by the company personnel, under the guidelines of the internal SOA infrastructure. Services are deployed over the runtime infrastructure of the organization, consisting of multiple tiered environments, ranging from integration testing environments to the publically accessible production one. Different control and change policies apply to each one, with increasingly harder criticality and stability constraints, the closer the environment is to the end users.

In this general context, the internal account transfer service that is being used at the bank offices presents some non critical usage problems during its operation. After detecting those issues, the end users report them to the company service desk. The incidences are reported to the change management team, which after successfully reproducing them, initiates an update development process for correcting the detected bugs on the affected services. Once the development of the updated deployment units finishes, those artifacts are provisioned to the integration environment and tested comprehensively to ensure that they address the problems and at the same time keep the previous functionality. After those validations have been made, the service is promoted to the pre-integration environment, where the compatibility with the final production resources is verified. Once this process also finishes

with a positive result, the change must be accepted by the production environment administrator, which will initiate the schedule of its execution at the pre set time for minimizing the impact of the change in the runtime operation. After this maintenance process is completed, the updated service is available for the users, and the problem incidence can be closed.

The described scenario involves several service configuration change processes applied to the different tiered environments. The technical details on how the architecture supports each one of them would be very similar, only varying in the control and change policy settings. I have selected as a representative of the three the update process at the pre-production environment, because the physical environment is virtually identical to the final one, and at the same time, the process can be executed with a greater degree of automation thanks to the lesser criticality.

As regards the organizational aspects of this scenario it presents a standard structure. The core functional blocks of the management system are centralized, managing all the described environments. The unit repository is integrated with the company software assets repositories, being automatically updated whenever a new or updated deployment unit has been released by the development staff. The pre production environment is instrumented by the agent infrastructure. Each physical node of the environment is monitored by a Node Manager and specific Gatherers, while a single Environment Manager controls these agents and communicates with the main system.

In this scenario, the appearance of the updated version of the service is reflected at the domain as an endogenous external change, originated on the logical entities (the repository and the defined objectives for the pre production environment). Over the scenario description I will detail the complete update process since the instant the change has been detected until the environment has been modified. The focus will be on describing how the change is handled, analyzed and applied to the environment. The following sequence diagram shows the main participant blocks and the exchange of calls and information that allows detecting the change, analyzing it, evaluating the required corrections and applying them to the environment.

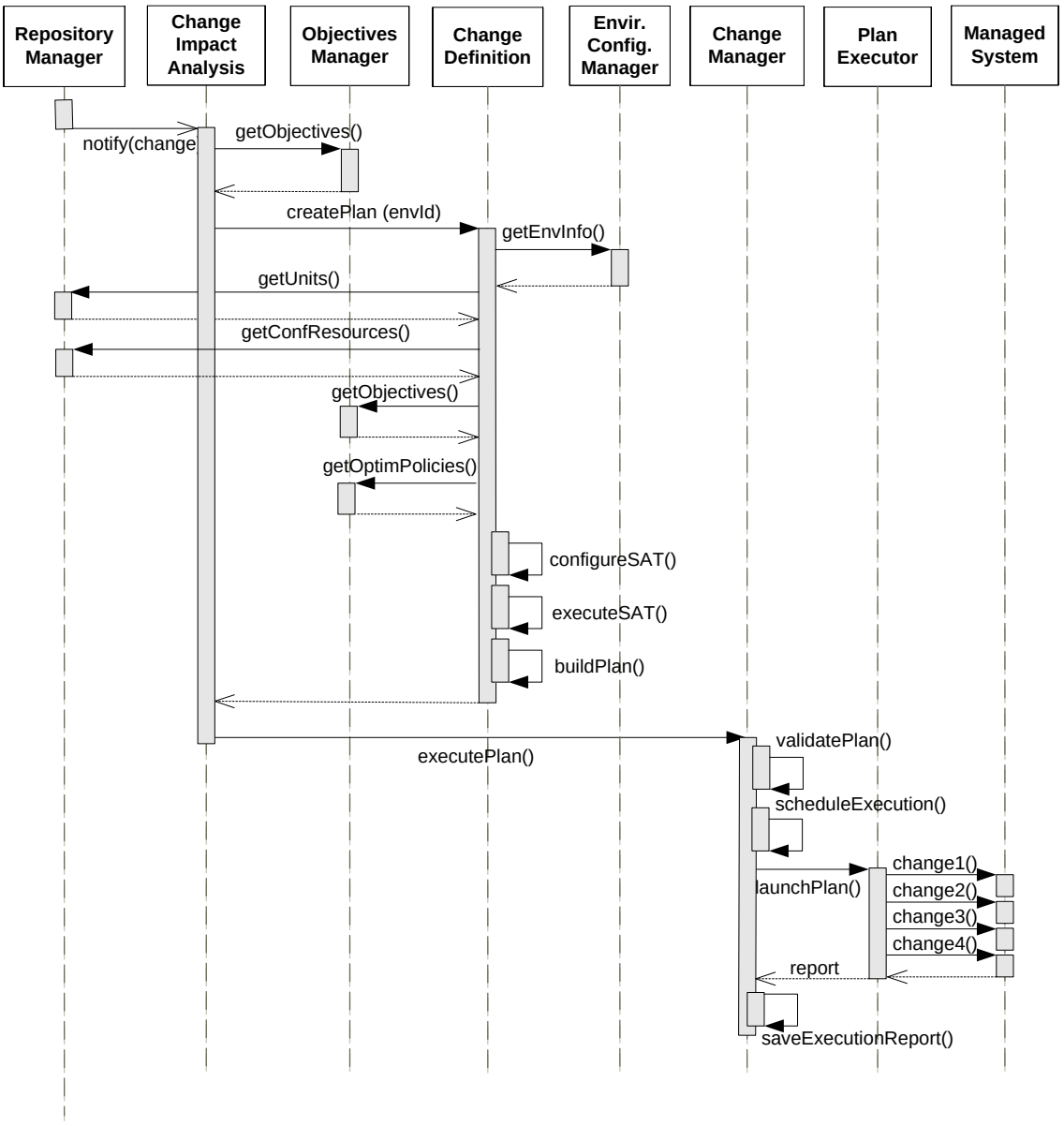

Figure 60 Sequence Diagram of change management of an update process

The objectives and policies defined for the preproduction environment mandate that the bank transfer service must be available at the environment always at the latest version. Whenever the unit repository is updated with the latest version of the unit, the event is notified to the Change Impact Analysis component. After receiving the notification, this component retrieves the established objectives by accessing the Objectives and Policies repository. Because of the previously described objective, this component determines that the environment must be updated to supply the latest version of the service, and in order to do so invokes the Change Definition module to obtain a change plan which satisfies the desired management objectives

When the Change Definition component is invoked, it first retrieves the complete information about the management domain, by communicating with the three repositories. This way, the list of available deployment units, the potentially configurable resources for the runtime containers, the management objectives that must be supported by the running system, and the optimization functions for selecting the final configuration, are obtained. Once all the domain knowledge has been retrieved, it is processed to obtain a set of PBSAT variables and

functions, with the algorithm described in the previous sections. Once the PBSAT problem has been defined and the engine is completely configured, it is invoked to find the desired solution for the system state. The provided restrictions ensure that the obtained solution is both stable and desirable. The results from the solver are boolean assignment to the variables, which are later interpreted in order to calculate the required changes that must be applied to the current environment configuration, as well as identifying potential dependencies between the different activities of the change plan.

After the change plan has been created the Change Impact Analysis component hands it to the Change Manager for its execution. In this environment, there is currently no other plan being executed, and no constraint about the execution times, so it is scheduled to be immediately applied to the environment. However, before executing it, the Change Manager validates the plan so that there are no inconsistencies in the plan definition as regards to the current environment state. Once this has been completed successfully, the plan is handed to the Plan Executor to be applied to the environment. This element finds a set of compatible Actuators, and invokes them to apply in the correct order the required internal changes. Once this process is complete, the updated service is operating correctly at the environment.

6.6.2. Reaction to Runtime Change

An SME company specialized in market research and financial reports offers a set of services which can be consumed by external clients. The portfolio includes stock market reports, customized requests and additional analysis tools and services. The organization manages their own execution environment, which is composed by several nodes with application servers of similar nature. Each running service is operating on a different physical machine to minimize the impact of incidences and avoid competition for the available resources.

After a long period of correct functioning of the system, one of the computing nodes providing the services goes down because of a hardware malfunction. As a consequence of that, part of the company services stop working. After a short period, the management architecture detects this incidence and reconfigures the available elements of the environment in an attempt to automatically restore the missing functionality, instantiating and configuring the affected services into other compatible elements from the platform (as the base infrastructure is highly homogeneous). This way, the downtime for the services is minimized while the IT support team analyzes the incidence, restores the faulty hardware element and returns the system back to its original configuration. In the following paragraphs I will describe how the management system can support this scenario. Over this explanation I will emphasize the similarities and differences with the previous scenario, in order to reflect the adaptability of the proposed architecture.

As regards the communications setup, it does not present significant differences with other cases. The central elements of the management architecture control the environment by communicating with the agent infrastructure, composed by an Environment Manager and a set of Node Managers and Actuators. Those entities automatically find each other through a local network Discovery Service. Once they are bound, the Environment Manager contacts them periodically to update the environment information, and verify no changes have occurred to the environment topology.

The following diagram describes how the management architecture detects the change of the environment and reacts to it in order to restore the system stability.

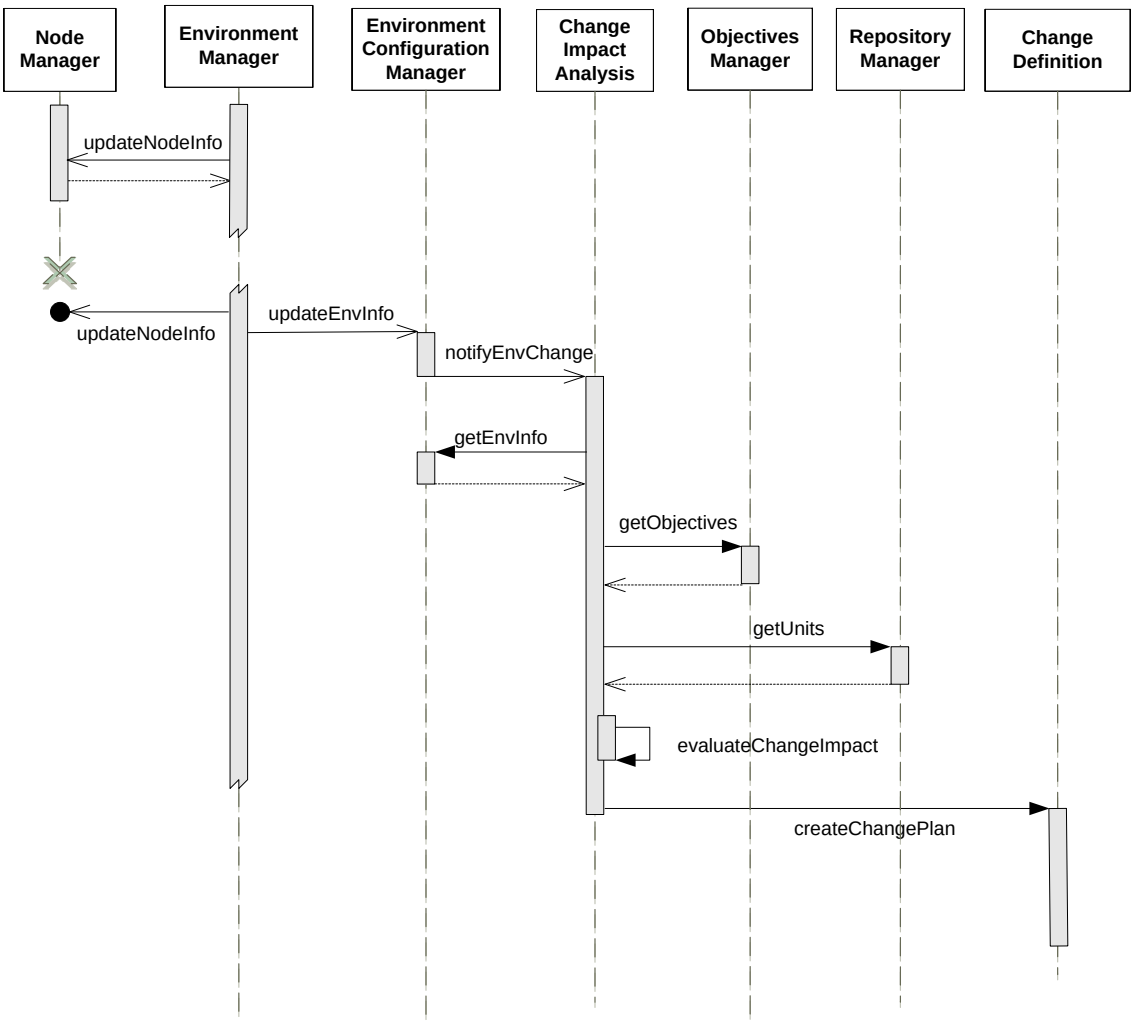

Figure 61 Sequence Diagram on Environment Change Response

The described scenario starts when one of the synchronization messages sent by the Environment Manager to a Node Manager does not obtain a response. The governing agent tries several times to contact the missing Node agent, and after those attempts are also unsuccessful it executes network diagnosis operations and determines that the physical node managed by the agent is no longer reachable. After that, it requests from the remaining nodes an updated snapshot of the actual environment configuration, which is aggregated and immediately sent to the Environment Configuration Manager.

The Environment Configuration Manager receives the updated information, and after detecting the severity of the change, sends a notification to the Change Impact Analysis component. This diagnosis element receives the updated environment snapshot, and obtains the latest known environment information in order to detect the affected elements. Over this process it internally obtains runtime binding graphs of the affected services, in order to estimate the potentially affected elements from the complete environment, not only the vanished node. It also retrieves the environment objectives and available unit definitions, in order to verify whether those components could potentially be replaced. If this is the case, it

finally invokes the Change Identification Component, with the objective of obtaining a change plan that will try to restore the damaged system functionality and restore its stability.

The technical activities for the Change identification and execution of the plan in this scenario would be analogous to the one previously described, so they won't be repeated here. However, because of the severity of the incidence described here, there are multiple additional actions that will have to be addressed, which should be briefly mentioned in spite of them not being directly managed by the proposed architecture. As this is a grave incidence, once a human administrator is notified, he/she will have to analyze the restored environment, and potentially apply additional configuration changes to elements outside of the control of the management architecture (e.g. it might be necessary to reconfigure a DNS server so that requests are transparently handled to the new hosted service). It must be taken into account that the hardware fault might have caused the corruption of application data, as well as the loss of process state, neither of which can be restored by the service management system. In addition to those quick response actions, the damaged element will have to be diagnosed for the originating error, and restored to bring back the system to its original capability.

6.7. Architecture Conclusions

This chapter has described an architecture of a service management system which leverages the previously presented information models and algorithms. The proposed solution can automatically reason over enterprise environments, with the architecture detailing how the system interacts with both the logical knowledge repositories of the company and the physical elements from the runtime. The architecture has been described using the OSI architecture view model, which focuses on one aspect at a time, improving the clarity of the description.

The architecture has been designed with the domain-specific characteristics of enterprise environments. In order to clarify how the proposal can address real management use cases, two reference scenarios have been selected. The description of each scenario shows how the architecture elements collaborate to provide the require functionality to satisfy these use cases. These scenarios will also be referred back over the next chapter in order to derive a set of validation test cases for the contributions presented in this dissertation.

7. Validation

The objective of this chapter is to present the experiments that have been carried out to validate the feasibility of the proposed modeling abstractions and algorithms for addressing enterprise service change management activities. In order to execute the tests, a prototype of the described architecture, based on these concepts has been built, and has been put to test in a series of experiments based on the requirements obtained at the ITECBAN project for a SOA core banking system. The description of the validation results explains by example the internal functioning of the proposed solution, and tests their limitations and degree of fulfillment over a selection of scenarios.

In order to appropriately introduce the set of experiments, I will start with a description of the general context, covering both, the reference domain for the validation and the characteristics of the initial data for all the tested scenarios. On top of that, in order to provide a complete overview on the context of the experiments, some details about the prototype implementation and the platform were the validation was executed will be provided.

The discussion of the validation results will start with an initial, representative case, whose execution will be thoroughly explained, in order to provide a clear insight on how the process works. After that, a study on the scalability and sensibility of the proposed solution will be presented. Although change management processes do not carry real time constraints, it is interesting to measure the performance of the proposed solution both in space (memory occupied) and time, with increasingly larger sets of data, in order to estimate its applicability to large scale situations.

Following those experiments, the results on the execution of a set of varied experiments will also be presented. The intent of that set of scenarios is to simulate a representative amount of the possible change situations that can appear over a service management process, and evaluate the correctness of the proposed changes by the change identification module. Finally, some general conclusions about the result of the complete set of validation tests will be described.

7.1. Scenario description

As the basic reference scenario for the set of validation experiments I have selected a slightly modified version of the target platform which is being used as reference for the works of the ITECBAN project. The objective of this project is to propose a complete core banking solution completely based on the SOA / BPM paradigm. This way, the complete service portfolio of the organization will be provided as SOA services. This includes client services (internet banking, cashiers), internal services (for company workers at the bank offices) and B2B services for inter bank transactions. As those services capture the company knowledge, they are internally developed and provided, with no third party dependencies because they constitute the core of the company business, and consequently must be internally controlled. In spite of its internal nature, in order to cope with the complexity, they are architected in an SOA / BPM approach. These elements are running at a service execution platform composed by multiple application servers, BRM (Business Rule Manager) servers, Business Process servers, mediation servers,

databases and additional infrastructure required to provide the aforementioned functionality with the adequate degrees of efficiency and robustness.

In this context a single final service will involve multiple, distributed elements, that will have to be properly configured and deployed in order to provide the required functionality, making this scenario a good candidate for validating the proposed models and architecture. In order to describe how the proposed solution can support this scenario I will first describe a reference set of deployment units that will participate in the provision of a final service, as well as the definition of the runtime environment where the services will be provisioned.

7.1.1. Deployment Units Definition

Over the complete set of the validation tests I will use the same set of logical resources, which represent an end user internet banking service. It is a complex service, with multiple transitive dependencies. Internally, the functionality is achieved by the composition of multiple lower level services, which will be distributed over the different server types from the runtime environment. The design of the final service is based on typical enterprise layered architectures, with services representing the data access layer, business logic layer and presentation layer. Each indivisible element to be deployed at one of those layers will be modeled as a resource, and will be provided by a *DeploymentUnit*. This way, resources represent participating artifacts such as business processes, business rules, EJB (Enterprise Java Beans) service components, information from the database management systems - both DDL (Data Definition Language) scripts and DML (Data Modification Language), remote functionality exposed through Web Services or corporate image material such as logos and fonts. The inherent differences between those artifacts are reflected through the resource typing system. Each logical *Resource* and enclosing *DeploymentUnit* includes versioning information, allowing a fine track on the evolution of the provided services.

The relationships between those units and resources are expressed through *Dependencies*, demanding the accessibility of the required resources in order to function correctly. It won't be enough to consider the existence or not of the involved elements over the environment, but also their distribution over the environment, as they have different degrees of visibility. As it should be the case in a heavily distributed architecture, most of them are accessible remotely, except the data access services from the business logic, which can only be accessed inside the same container.

Another factor which must be addressed over those scenarios is the units' configuration with respect to the established *Bindings*. Depending on the type of service technology, some of them will be automatically configured when the *Binding* has been established, whereas other types require from the consuming unit additional configuration, as is the case of Web Services communications. In order to connect to the provider, Web Service clients need to know the URL where the service or the service description can be accessed. Moreover, that URL cannot be known beforehand, as obviously it will differ depending on the place in the topology where the providing service is deployed. The model expressivity addresses both concerns, allowing to automate the configuration of the consuming element. This will be supported by the combination of two automatic configuration mechanisms: First, the provider service will automatically configure its service URL, by defining it as a *ContextAwareProperty*, with an expression value of *http://${ip}:${servicePort}/ContextPath*. Over the change process, the

correct value for the Property will be obtained according to that expression, and will be properly configured. On top of that, the depending unit will define a *BoundProperty* attached to the *Dependency*, mandating that the correct value for the *connectionURL Property* must be automatically provided from the value of the *serviceURL* property from the bound *Resource*. This way, it can be seen how both automatic configuration mechanisms are combined to correctly establish the configuration derived from the dynamic binding between the two units communicating through web services, as it is described in the next picture.

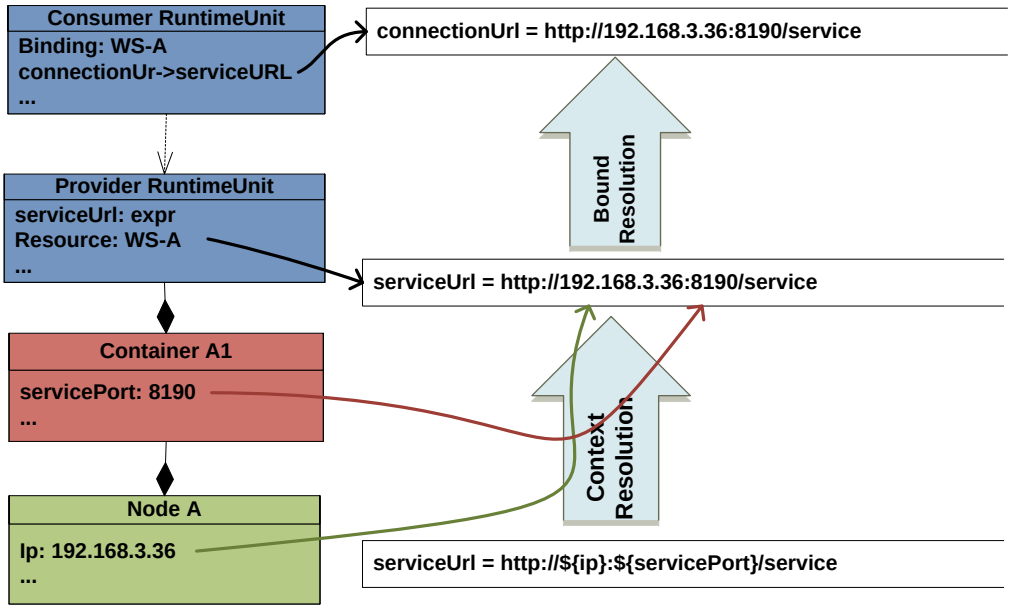

Figure 62 Automatic Binding Configuration through Context-Aware and Bound Properties

With those considerations in mind, the logical elements that participate in the validation have been modeled as a set of *DeploymentUnits*. The logical knowledge base defined at the repository is composed by 22 units. The units belong to seven different types (*zip.apache, war, ear, bpel, zip.drl, zip.ddl, zip.dml*), which must be supported by the different types of *Containers* present at the runtime environment.

As regards the actual structure between these units, they can be divided into two types. By applying transitive dependency analysis, 19 of them must be present at the environment in order to fulfill the complete requirements of the high level service, while the remaining three represent another service which is completely independent from the main one subject to the validation tests. The reason to include non participating units as part of the knowledge base is to validate that the solutions proposed by the change identification service do not involve unnecessary changes to not relevant resources of the environment. Next picture shows the dependency graphs present at the logical knowledge base. Each *DeploymentUnit* is represented by a blue rectangle, with the exported resources depicted inside. Dashed arrows signal satisfied logical dependencies. For clarity's sake, the rest of information about those units has been omitted.

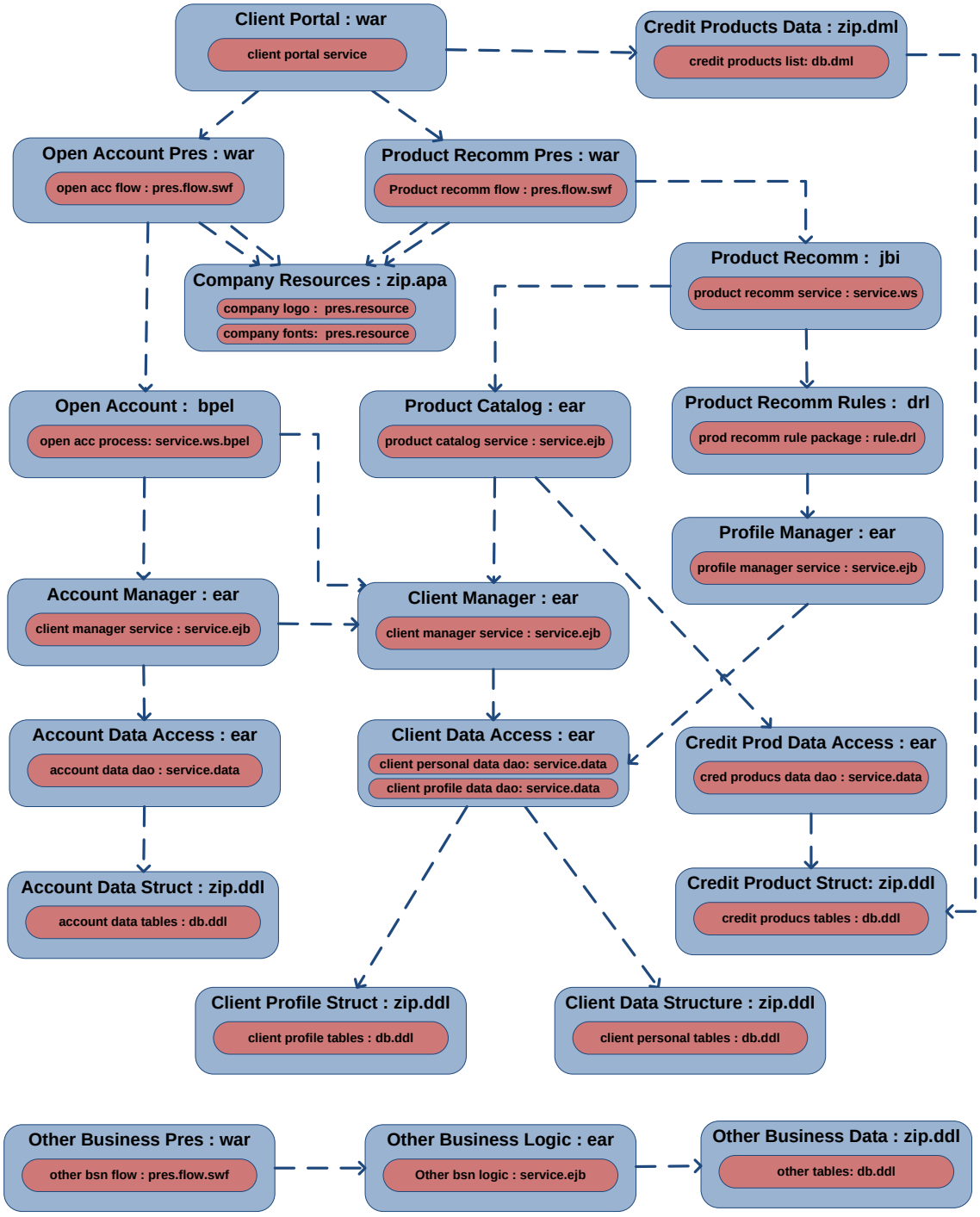

Figure 63 Dependency Graph of the Units involved in the Validation cases

In total, the 22 units define 25 Dependencies, which are satisfied by the exported resources of the enclosed units. Those resources belong to eight different types, and are visible environment wide, except the *"service.data"* resources, which can only be accessed inside the same *Container*. In addition to *Dependencies*, the units providing the data access services define additional *Constraints* on the execution platform, demanding the existence of a resource of type "*datasource*" and a specific name at the selected *Container,* in order to be able to access the information from the DBMS without additional configuration. In total, there are three *Constraints* of this type, one per data access unit.

7.1.2. Environment Definition

The experiments that have been executed for the validation have taken as the reference environment a distributed runtime similar to the integration environments used at enterprises. The computing elements are powerful nodes, hosting the range of application servers described previously. The network topology is very simple, with every node interconnected through a high speed local area network, and restricted external communications. The physical elements of the environment were instrumented by the agent infrastructure described in the architecture chapter. This way, a snapshot of the current status of these resources was captured, modeled under the abstractions defined at the environment model.

The environment is composed by ten Nodes, each of them with a network interface configured. On top of them there are twelve *Containers* available, which cover among their supported types all the unit types of the elements defined at the logical repository. Depending on the specific type the number of compatible containers ranges from one (database and static web units) to four, for business services. It must be noted that in this model DBMS and Apache Web Servers are modeled as *Containers*, although in principle they seem to be very different entities from typical enterprise JEE application servers. The service configuration change management system controls them, and can deploy over them artifacts that provide resources to the environment, i.e. *DeploymentUnits*, so they are identical to traditional *Containers* for the purposes of this system. In the default case, no *RuntimeUnits* will appear at the environment configuration, although this will change for some of the tests.

The environment information also shows a set of node *Resources* that constitute a very important part of an enterprise environment, such as LDAP authentication providers or network firewalls. As the management system does not have control over them, they are better modeled as resources instead of *Containers* in this abstraction.

The next picture shows a graphical representation of the base entities contained in the environment model. The ten nodes are depicted as rounded rectangles, detailing in each case the hosted *Containers*. Each *Container* is described by its name, type and supported types (represented with dark ovals). In addition to that, the additional environment infrastructure modeled as node resources, such as firewalls, or batch process servers is also shown by the picture.

In addition to the current status of the *Environment* and the *DeploymentUnits*, the domain definition also contains a *ConfigurableContainerResource* for the Glassfish servers (*Containers* of type “jee.gf”). This template allows the creation of resources of type “*datasource*” as part of the identified change operations. These *Container* resources encapsulate connections to the databases, and are used by applications to abstract from the implementation details and actual location of the information.

Finally, it must be mentioned that over all the defined cases I will also apply the optimization and customization preferences which were described at the algorithm section. This way, the solutions proposed at each scenario will have a maximum of one instance of each *DeploymentUnit*, and will try to minimize the number of changes by respecting if possible the already existing *RuntimeUnits*.

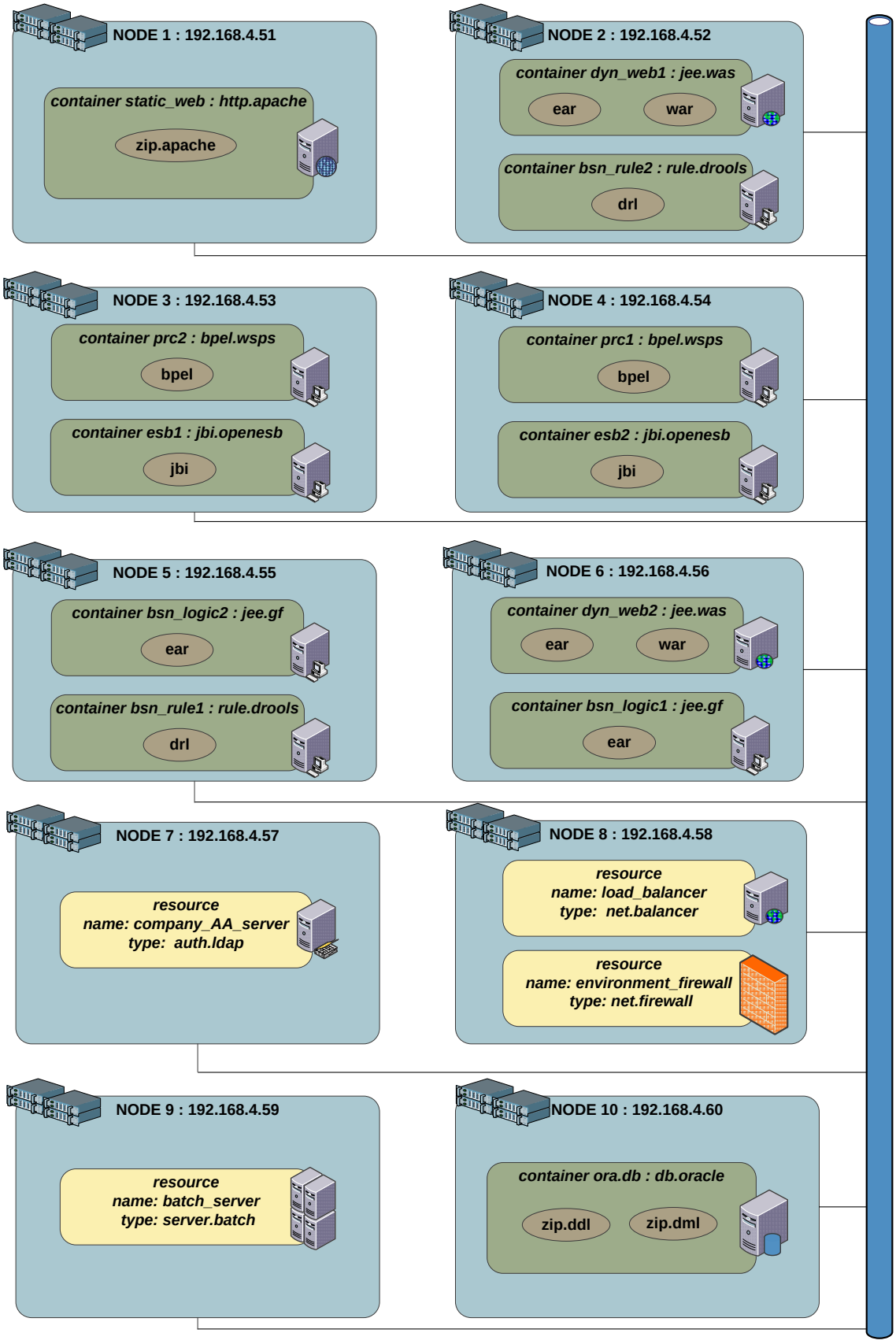

Figure 64 Validation Reference Environment Description

7.1.3. Validation Environment Details

Finally, in order to provide the complete context information on the executed experiments, I will describe the relevant aspects of the hardware and software platform where they were carried out.

The Change Identification Service has been executed on a Dell Latitude D630 computer, with the following hardware and software capabilities: hardware processor Intel Core 2 Duo T 7700 (2.4Ghz speed), 2 Gigabytes of RAM Memory and 160 GB of maximum hard drive space. The machine was running with the OS Windows XP Professional, with Service Pack 3 installed. The prototype was developed in the Java language, and executed over a Java VM of the version Java SE Runtime Environment 1.6.0_13.

Finally, I will mention the dependencies on third party libraries which were used for developing the prototype. As logical end runtime models were defined in ECore, the Change Identification service used the EMF Runtime v2.5.0 for the de/serialization of model instances. As for the core implementation of the service, SAT4j 2.1 Pseudo Boolean SAT engine was the selected implementation. This pseudo boolean solver provides several optimization strategies to the base algorithm such as implementing heuristics derived from the objective function, or constraint learning from conflicts [48]. The library's efficiency and maturity have been validated over the latest two years with its inclusion in Eclipse to power component dependency resolution processes [66].

7.2. Detailed Scenario Execution

Once the context for the validation experiments has been described I will first detail the execution of one of the cases, providing a step by step explanation about the internal functioning of the algorithm. The scope of the validation covers only the Change Identification Service, as the intent is to validate the models and algorithm proposed, which are fundamentally implemented by this element of the service management architecture.

The execution description will follow the following structure: First, the specific characteristics of the scenario will be analyzed. After the input data has been described, an interpretation of the current scenario will be provided, in order to diagnose the current state and predict what the outcome of the change identification service should be. After that, the results from the service execution will be described. The Change Identification Service will be invoked, and the set of defined changes identified will be analyzed, as well as an evaluation of the internal workings.

The logical repository contains the definitions of the 22 *DeploymentUnits* that were described at the previous section, with a total of 25 *Dependencies* and 3 *Constraints* defined by the units. The environment is also the same which has been described, with no *RuntimeUnits* at the initial state. The domain contains one defined management objective: the existence of the resource "*client portal service*" in the environment. This resource represents the end use service, which is only provided by the unit named "*Client Portal*" of type *war*.

The described scenario is a freshly installed environment, ready to start functioning after the hardware elements have been provisioned with the required servers and services. In order for it to start providing the desired functionality, a management objective is defined so that the

environment provides the client portal service. In this context, the change identification process should install the unit providing the required service and all the required units for a stable configuration in compatible runtime *Containers*. On top of that, the *Bindings* among those components must be correctly configured, including the *BoundProperties*, and unit *Constraints* must also be respected. None of those changes are explicitly expressed by the initial input, but they must appear at the set of changes obtained by the execution algorithm.

In order to clearly reflect the details of the algorithm I will focus each step on the same unit, Client Data Access. This unit has been selected because it is one of the most complex in terms of *Dependency* and *Constraint* relationships. Figure 65 shows an expanded view of the selected unit and its first level relationships (relationships with the remaining elements are reflected in the previous general picture). The unit exports two Resources (*client personal data dao*, and client *profile data dao*), which are required by other two units (Client Manager and Profile Manager). Those resources are only visible in the same *Container* Client Data Access is deployed. In order to provide the functionality enclosed in those two resources, the unit requires access to two database tables, which will be provided by the *ddl* definitions contained in other two units (*Client Profile Structure* and *Client Data Structure*). The detailed analysis will only focus on the relationships directly involving the selected unit, as the rest of units will be processed exactly the same way. Finally, the unit also declares a *Constraint* in the runtime environment, requiring a resource named "*DSClient*" of type "*jdbc.datasource*" to exist in order to work properly

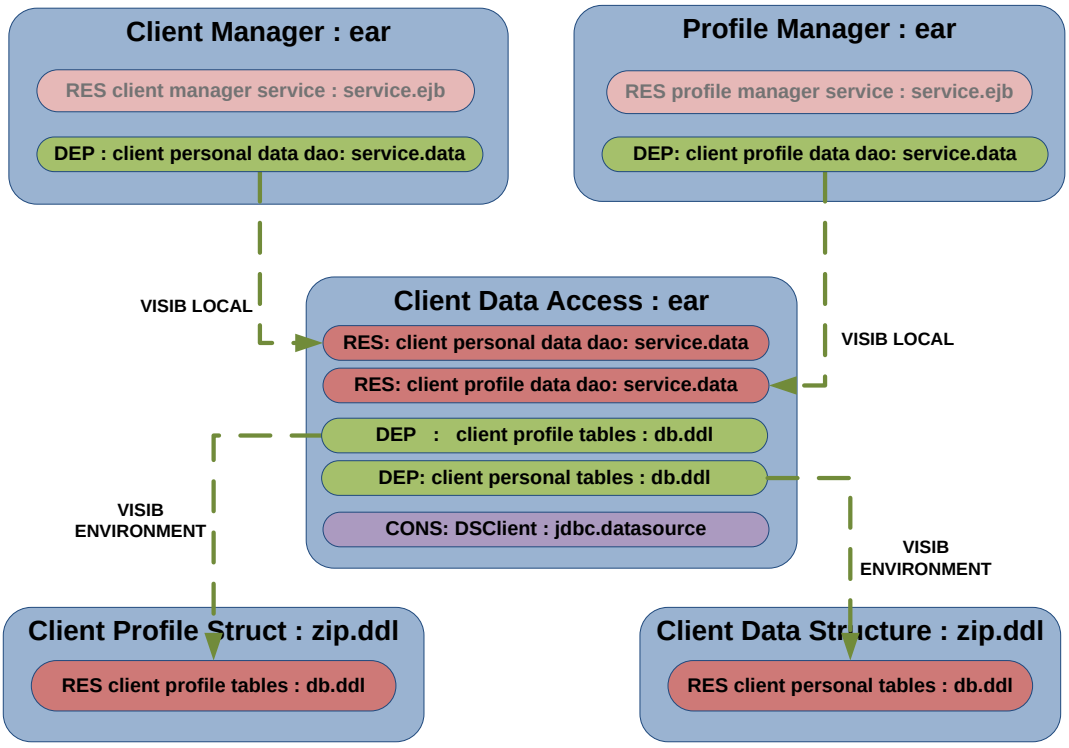

Figure 65 Resource, Dependency and Constraint Details of the Client Data Access Deployment Unit

The description of the case will be divided into two main blocks: first, the operations executed before the SAT resolution engine is invoked will be described, and finally the interpretation of the results will be explained.

7.2.1. Input definition

When the Change Identification Service is invoked it retrieves from the repositories the complete domain information (deployment units, environment state and management objectives), and processes it into SAT variables and constraints.

The first step applied by the algorithm consists of the definition of the SAT literals. The first identified variables are the *RuntimeUnit* literals, which represent the decision of a unit appearing or not at the updated environment. In order to define these literals, the set of *DeploymentUnits* is compared against the set of containers. If there were no additional restrictions a total of 22 x 11 = 232 *RuntimeUnit* literals would be defined. However, the actual number is much lower because not every combination would be stable. In this comparison the first filtering factor is the *supportedType* field of the *Containers*, which greatly restrict the defined literals. In the case of the analyzed units, three of them are of type *ear*, which can be deployed over four environment *Containers* (*bsn_logic1, bsn_logic2, dyn_web1, dyn_web2*). The other two units are of type *zip.ddl*, which can only be deployed over one container (*oraDB*). Clearly, from the worst case of 12 *RLit* literals per *DeploymentUnit* definition, a significant simplification is achieved by this filtering.

In addition to the type comparison, the declared unit constraints will further restrict the generated literals. From the set obtained at the previous comparison, Non-constrained units will be added as literals, while constrained units must either be satisfied by the current container and node *Resources* or be potentially satisfied by configurable container resources. In the case of CDA (Client Data Access), requiring a resource of type '*datasource'*, and name *DSClient*, the *dyn_web* containers must also be discarded as that *Constraint* cannot be satisfied. For the *bsn_logic* containers the resource is not initially present but it can be created as a configurable resource, as there is a *ContainerResourceConfiguration* definition which can be applied to those *Containers*. This way, both the RU literal and a Container Resource literal, representing the decision on creating the configurable resource at the container, will be defined. I will use for the remainder of this explanation the following notation for the Runtime Unit literals *<duname><contname>,* and for CR literals *<resname><contname>*. In total, after this process has finished, 43 RU literals and 6 CR literals have been defined, from which the ones under the detailed analysis are:

$$CMdyn1, \quad CMdyn2, \quad CMbsn1, \quad CMbsn2$$
$$PMdyn1, \quad PMdyn2, \quad PMbsn1, \quad PMbsn2$$
$$CDAbsn1, \quad CDAbsn2, \quad CPSora, \quad CDSora$$
$$DSClientbsn1, \quad DSClientbsn2$$

After the former process has finished, the set of RU Literals originated from units with *Dependencies* will be iterated again, in order to identify the possible *Binding* configurations for each one, represented as Binding literals. For each potential binding the rest of RU literals will be analyzed to check whether it satisfies the dependency and is accessible to the dependant one (by checking visibility and relative topology). If both conditions are true a Binding Literal will be generated representing the potential configuration. If after going over all the remaining RU literals there is no valid configuration for one *Binding*, the dependant RU literal will be set to false. In our focused variables, first the bindings from CDA to CPS and CDS will be

established. In the case of CM and PM bindings to CDA, the local visibility of the exported resources will discard every binding configuration where the requiring unit is not at the same container as the CDA, greatly reducing the possibilities. This shows the large impact that environment visible resources have over the complexity of the complete process. In total, 96 Binding literals have been defined. B Literals will be referenced in the text with the following notation *<rudep>to<ruprovider>*. In the analyzed segment, 4 literals have been valued to false, and eight Binding Literals have been defined:

$$CMdyn1 = false, \quad CMdyn2 = false$$
$$PMdyn1 = false, \quad PMdyn2 = false$$
$$CMb1toCDAb1, \quad CMb2toCDAb2$$
$$PMb1toCDAb1, \quad PMb2toCDAb2$$
$$CDAb1toCPSo, \quad CDAb2toCPSo$$
$$CDAb1toCDSo, \quad CDAb2toCDSo$$

At this point, all the literals which will be provided to the SAT have been defined. The next step is to define the functions which will restrict the potential solutions, ensuring that the results are stable and desirable. The first aspect that will be taken care of is the desirability, by translating the management objectives into SAT constraints. In this case, the only objective was the existence of the client portal service in the environment. As it is provided by the Client Portal unit, the constraint will mandate that at least one of its RU literals appears at the solution, producing the following constraint:

$$CPdyn1 \vee CPdyn2$$

As initially the environment does not contain *RuntimeUnits*, no optimization function will be defined. However, the restriction about a maximum of one instance of each *DeploymentUnit* must be enforced through a set of clauses. In the analyzed subset, three of the units (CM, PM, CDA) have multiple possible locations so one clause must be defined for each one of them. This way, the three following clauses have been defined:

$$CDAbsn1 + CDAbsn2 \leq 1$$
$$CMbsn1 + CMbsn2 + CMdyn1 + CMdyn2 \leq 1$$
$$PMbsn1 + PMbsn2 + PMdyn1 + PMdyn2 \leq 1$$

The rest of the functions ensure that the proposed solution is stable and structurally coherent. As regards stability there are three types of relationships which must be properly restricted through clause definition: dependant RU to Bindings, Binding to bound RU, and Configurable Resource to Constraining Resource. The first type ensures that the dependencies expressed by the units are satisfied and correctly configured with bindings in the proposed solution. There will be one clause for each B Literal, implicating the bound resource to the existence of the binding. This way, in the analyzed subset, the following clauses will be defined:

$$CMb1toCDAb1 \rightarrow CDAbsn1, \quad CMb2toCDAb2 \rightarrow CDAbsn2$$
$$PMb1toCDAb1 \rightarrow CDAbsn1, \quad PMb2toCDAb2 \rightarrow CDAbsn2$$
$$CDAb1toCPSo \rightarrow CPSora, \quad CDAb2toCPSo \rightarrow CPSora$$

$$CDAb1toCDSo \rightarrow CDSora, \qquad CDAb2toCDSo \rightarrow CDSora$$

The second type of clauses are called structural functions, as they enforce that proposed solutions are coherent with the indivisible relationship between one RU and its bindings. In total three clauses will be defined for each group of bindings (the group is composed by the set of Binding literals that express alternatives for one dependency of the same RU). These restrictions mandate that no bindings are present if the dependant RU is not part of the solution, and that exactly one binding literal must be true of the dependant is evaluated to true. In the selected subset all binding groups are composed by one element, simplifying the resulting clauses as the at most one clause is unnecessary, and the disjunctions become single literals:

$$CDAbsn1 \leftrightarrow CDAb1toCPSo, \qquad CDAbsn1 \leftrightarrow CDAb1toCDSo$$
$$CDAbsn2 \leftrightarrow CDAb2toCPSo, \qquad CDAbsn2 \leftrightarrow CDAb2toCDSo$$
$$CMbsn1 \leftrightarrow CMb1toCDAb1, \qquad CMbsn2 \leftrightarrow CDAb2toCDAb2$$
$$PMbsn1 \leftrightarrow PMb1toCDAb1, \qquad PMbsn2 \leftrightarrow CDAb2toCDAb2$$

In order to show the complete set of clauses which can be generated at this step, I will also focus on the CP (Client Portal) and OAP (Open Account Presentation) literals. In the previous steps two RU Literals were defined for each of them, and four B Literals, from each CP to each one of the OAP Literals. In this case, the following functions will respect to structure and binding stability have been defined:

$$CPdyn1 \leftrightarrow (CPd1toOAPd1 \vee CPd1toOAPd2)$$
$$CPdyn2 \leftrightarrow (CPd2toOAPd1 \vee CPd2toOAPd2)$$
$$CPd1toOAPd1 + CPd1toOAPd2 \leq 1$$
$$CPd2toOAPd1 + CPd2toOAPd2 \leq 1$$

Finally, Constraint-related clauses are defined to ensure that if the constrained unit is part of the solution, the Container Resource will also be configured. From the analyzed elements, there will be two Constraint functions, one for each *CDA RU* Literal:

$$CDAbsn1 \rightarrow DSClientbsn1, \qquad CDAbsn2 \rightarrow DSClientbsn2$$

To sum up, after all those processes 280 clauses have been defined, representing the desirability and stability conditions that must be met by the proposed solution by the SAT. With no further operations required, the set of literals and clauses are provided to the SAT and a solution is obtained, which will consist on a true or false value for each one of the 145 variables defined.

7.2.2. Results interpretation

Once the literals and clauses have been defined at the SAT, the results from the solver are obtained and translated. In this scenario, the solver engine has found a solution that evaluates to true 45 of the 145 variables, the remaining 100 being evaluated as false. From those true literals, 19 of them are RU Literals, 23 of them are Binding Literals, and three of them are CR Literals. Those numbers match exactly the figures of the complete dependency graph of the objective *DeploymentUnit*. In fact, when the specific literals are analyzed, not only the numbers are the same but also they belong to exactly those components, ignoring the

remaining three components defined at the repository, as they are neither part of the business objectives nor relevant for the stability of the involved units.

Once those values have been collected they will be interpreted the following way. First, the set of positive values is processed. As in this case the environment is initially empty each one of them implies either the instantiation and activation of a *RuntimeUnit* at the selected *Container*, the configuration of a *Binding* between two *RuntimeUnits* or the creation of a container *Resource*. In the case of the analyzed subset, the three *ear* units will be deployed at the same container, *bsn_logic2*. This is coherent with the local visibility of the resources from the CDA unit, demanding the consumers to be deployed at the same container. The two *zip.ddl* units providing the required resources for CDA will be deployed on the only feasible container, *oraDB*. In addition to the five positive RU literals, another five positive B literals contain the *Binding* configuration among those elements. Finally, the CR literal for creating the *DSClient jee.datasource* resource at the *bsn_logic2* container is also set to true, so this resource will be configured to satisfy the constraint of the CDA Runtime units. Overall, from the 22 related literals of this subset of the complete problem, a total of 11 have been evaluated as true. It is a high percentage (50%) compared to the overall statistic (30%). This is the case because of the restricted options present at this specific subset, thanks to the *Constraints*, existence of only one *Container* for the database units, and local visibility of some of the resources. Those factors greatly reduce the number of possibilities and, as the number of positive literals is related with a valid solution and not with the alternatives, consequently impact the ratio of true variables.

Along the process of evaluating those positive literals, there are some special cases that will be handled with additional processing. First, every positive RU literal belonging to a RU with defined *ContextAwareProperties* will be identified and processed in order to obtain the correct value for the property. Once it has been found, an additional configuration activity will be defined to set up the correct value. After all those values have been identified, a similar process will be carried out for the *BoundProperties*, retrieving the adequate configuration values in function of the *Binding* configuration. This is processed after the context aware values have been obtained in order to allow for those updated properties to be reflected on the bound ones. In this case there were three *ContextAwareProperties* and three *BoundProperties* participating in the final solution, contributing six additional activities.

After the positive values have been analyzed negative values of RU literals belonging to *RuntimeUnits* already existing at the environment should be evaluated. However, as in this case the environment is initially empty, no additional result is obtained from it.

The execution of these checks has identified 70 changes in total. 38 are derived from the positive RU literals (installation and activation), 23 from B literals, 3 from the CR literals and the remaining 6 are related to configurable properties. After applying all those changes the environment ends at the state shown in the next picture. In order to improve its clarity, the figure only shows *Containers* hosting at least one *RuntimeUnit*. The 19 participating units have been deployed over 8 containers from the total of 12 belonging to the environment, distributed over seven nodes from the environment. The distribution is not even as one Container hosts seven units whereas multiple ones host only one. This is a correct solution as there has been no explicit preference about an even or unbalanced spread of the

RuntimeUnits among compatible containers. In case that was mandatory, those factors should have been modeled as factors from the optimization formula. The picture also shows the bindings which have been configured for the *RuntimeUnits*. This clearly reflects the actual complexity of the distributed applications, which for a single end service spans over seven nodes of the environment. As regards memory and footprint consumption of the described process, the Change Identification Service was executed in 84ms, with a memory footprint of 6.2 Mbytes.

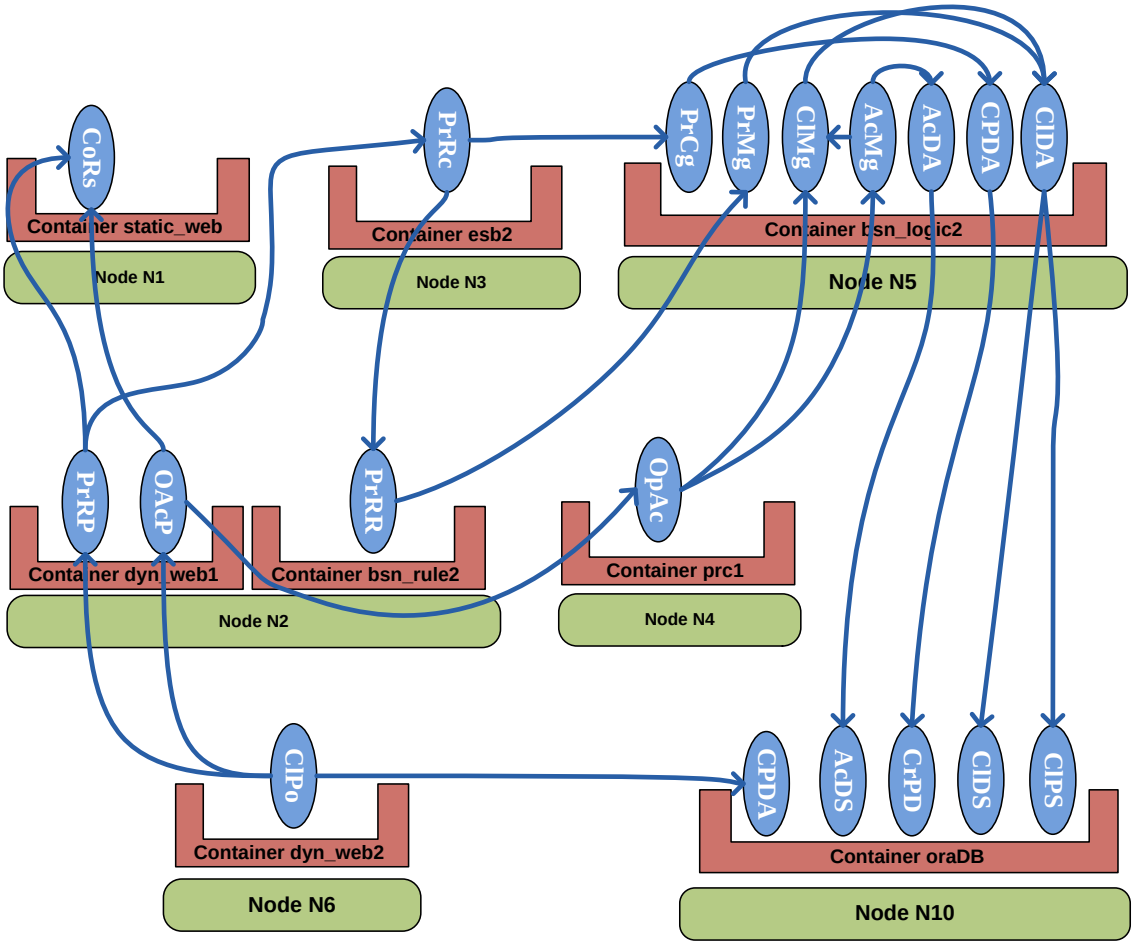

Figure 66 Updated Environment With the Identified Changes

This section has explained the process followed by the Change Identification Service to find a correct solution, which clearly matches the initial interpretation of the required changes initially provided. In the following sections I will present the results of the additional validation cases which have been executed, analyzing both its scalability when the domain data is larger than the one managed here, or the correctness of the solution when presented with some changes of the domain. Nonetheless, in all of them the same internal processing is applied, consisting of literals definition, constraints definition, SAT invocation, boolean results retrieval, positive values interpretation and negative runtime units check.

7.3. Scalability Analysis

After the base case has been successfully tested, the aim of this set of experiments is to test the scalability of the proposed algorithm. The tests applied in this section will be based on the

previous one, starting with an empty environment, and the same defined objective, mandating to instance the *client portal service*. There will be only two aspects of the input data which will be altered: the number of defined units at the logical repository and the size of the runtime environment.

The set of experiments will be divided into three categories, depending on what part of the domain is modified (altering only the size of the environment, altering only the size of the logical repository and altering both of them). The definition of the modified sets is not only a matter of increasing the cardinality of those sets. As it has been discussed over the dissertation, and shown over the detailed execution of a validation case, there are multiple, interrelated aspects that determine the complexity of each example. Because of that, the rationale behind the definition of the test cases, the predicted impact on the results of modifying the base elements, and the conclusions on the obtained results will be detailed for each set of experiments.

As regards the numeric data obtained for each experiment three categories of data will be collected. First, initial statistics on the size of the logical deployment units and the environment resources will be provided. The second category will focus on the amount of SAT literals and clauses which have been defined from that input. This information is usually more representative of the actual complexity of the case when compared to the numbers of input data. Finally, each experiment will report the average time and memory consumption over a set of five executions of the case.

7.3.1. Variation on the size of the environment

The first set of experiments will repeat the initial case with increasingly larger runtime environments than the one presented, which will be at the same time compatible with the required types and resources expressed by the *DeploymentUnits*. The results should be equivalent to the ones obtained in the initial case, with a set of literals belonging to the same *DeploymentUnits* correctly instanced and configured at the runtime environment. Clearly, the distribution of those *RuntimeUnits* may be different as the set of nodes will change.

In order to define those additional environments, some considerations about the effect of the different characteristics of the environment in the actual complexity must be taken into account. First, the number of nodes is not explicitly related to the complexity of service change identification operations. This is clearly the case as *Containers* and not *Nodes* are the base execution platform, which fundamentally determined the possibilities. However, *Nodes* can satisfy unit *Constraints*, reducing the complexity of the problem. However, that factor is actually part of the *DeploymentUnit* characteristics more than a characteristic of the runtime environment.

This way, the main factor of the environment affecting the complexity is the set of available *Containers*. In the worst case scenario, the set of potential *RuntimeUnits* would be the product of the number of *DeploymentUnits* by the number of *Containers*. This can happen at domains with a single type of *DeploymentUnit* and non additional *Constraints* over the valid *Containers*. However, the actual complexity will be restricted by two related factors: the number of Containers with each *supportedType,* and the number of units belonging to each of those types. Clearly, with the same number of *Containers* and *supportedTypes* the actual complexity can be very different depending on how those factors are set. E.g. let's suppose a logical base

with 19 units of one type, and 1 unit of a different one. In an environment with five *Containers* the amount of possible options will greatly increase if the large majority of *Containers* support the dominating type, whereas more *Containers* supporting the second type will have a much lesser impact.

With those considerations in mind, two additional environments have been defined in order to test the scalability of the Change Identification Service with respect to environment information. Both of them replicate a number of times (three in the first case, nine times in the second case) the distribution of the original environment. This way, the ratios of number of instances for each type are preserved, allowing for a better comparison with the base results.

Before executing the experiments, I will estimate the predicted impact on the size of the SAT problem. If N is the number of times the original environment has been replicated, the increase in the problem size would be close to N^2, as that is the factor which will affect the B literals, which constituted the majority of literals in the original case. This way, SAT input statistics should be roughly nine times more in the second case, and roughly eighty times more in the third one, whereas the impact in time and memory cannot be estimated beforehand. The following table shows the obtained metrics over the executions of the base experiment (Case I), and the additional experiments with larger environments (Case II and Case III).

Table 3 Sensibility Analysis Results with Larger Environments

Input data statistics	Case I	Case II	Case III
Number of units	22	22	22
Number of dependencies	25	25	25
Number of constraints	3	3	3
Number of nodes	10	30	90
Number of containers	12	36	108
SAT input statistics			
Number of variables	145	963	7673
RuntimeUnit variables	43	129	387
Binding variables	96	816	7232
ConfRes variables	6	18	54
Number of functions	280	1952	15452
Time and memory statistics			
Consumed memory	6.2MB	7,6MB	18,5MB
Total exec time	83ms	265ms	5892ms
Preparation time	59ms	206ms	5350ms
Execution time	18ms	42ms	246ms
Interpretation time	6ms	17ms	296ms

Before discussing the obtained measurements from the experiments, it must be mentioned that the execution of both cases II and III did successfully provide a correct set of changes for the base input, each of them composed by 45 positive values which were interpreted as 70 activities, analogous to the changes identified in the initial case (although with a different distribution of the units).

By looking at the execution statistics the first conclusion from looking at the SAT input defined is that, as it was predicted, the increase in both the number of clauses and literals was roughly exponential, because of the dominating factor of the *Bindings*. The statistics also show the impact of the increased size in both the memory consumption and required time for completion. Both show an exponential growth, although at this stage the numbers in the most complex case are very manageable (6 seconds of total execution time is perfectly reasonable for the kind of calculation, and the memory footprint was also kept at very manageable levels. Also, the experiments have been executed with a machine considerably less powerful (a two year old laptop) than the available infrastructure for an enterprise management system.

However, it must not be ignored that the size of the environment impacts heavily to the complexity of the process, so the applicability of this approach should be tested for other domains, where the size of the environment can be or an order of magnitude larger than those experiments. However, that is not the case for enterprise domains, were an environment with over a hundred of containers would already be a very large execution platform).

7.3.2. Variations on the number of logical units

After the previous set of experiments, I will execute an additional set of tests where the environment definition will be fixed while the logical repository elements will change. This way, the difference between those cases will be in the set of defined *DeploymentUnits*.

The number of units is an important factor in the complexity, but will be more affected by the actual characteristics of the units, as well as its relationship will the environment. *Dependencies* are the main factors for unit complexity. Dependant units are more complex than non dependant ones, whereas the impact of exported *Resources* is not so clear. First, some of the *Resources* exported by a *DeploymentUnit* might not be required by any other logical element. If that is the case, although processing the resource will inflict a small performance penalty over the initial step, it will not actually increase the complexity of the SAT problem. However, resources which are required by other units do have an impact on the complexity, as they open up additional possibilities. The initial case also showed that the complexity derived from these required resources greatly changes depending on the *Resource* visibility. Environment-wide visibility imposes a very large cost to the solver when compared to local visibility.

DeploymentUnits can also define *Constraints*, which actually decrease the complexity of the SAT problem as they eliminate potential options for the RU Literals, at the cost of increasing the processing time. However, that is not true in every case, as some Constraints can only be satisfied by resources created from *ContainerResourceConfigurations*, ending up with additional defined literals and clauses than if the unit did not have any *Constraint*.

In addition to the characteristics of the *DeploymentUnits*, an additional factor which should be evaluated for the added elements is whether the additional units should be part of the proposed solution or not. For the selected cases I have opted for the latter, increasing the number of non related units to the ones required for the environment objectives. This way, the outcome from the SAT invocation should be equivalent to the results of the initial case. The experiments will analyze the interference of additional deployment units in the execution of the base case.

Nonetheless, I will select a set of additional units which increase considerably the problem complexity. In order to achieve that, all the additional units will be of the same type. The selected type has been *ear*, as there are four possible containers which can host those elements. The additional units will define Dependencies forming a 'dependency chain', with each one of them but the first additional unit depending on a resource provided by the previous. All resources will be environment visible in order to not limit the potential complexity. With those characteristics in mind, the number of RU literals should increase by the number of additional units multiplied by the number of compatible containers (4), while the increase in binding literals will be the number of new binding times the square product of the number of containers (4^2=16).

With those considerations, three additional tests have been executed, contributing to the initial repository 19, 99 and 1000 *DeploymentUnit* definitions, respectively. No additional unit is related to the original elements, so they should not appear at the proposed solution. The following table shows a comparison with the collected execution statistics of the initial case and the expanded ones.

Table 4 Validation Results with an increasing number of non participating units

Input data statistics	Case I	Case II	Case III	Case IV
Number of units	22	41	121	1001
Number of dependencies	25	43	123	1003
Number of constraints	3	3	3	3
Number of nodes	10	10	10	10
Number of containers	12	12	12	12
SAT input statistics				
Number of variables	145	509	2109	19709
RuntimeUnit variables	43	119	439	3959
Binding variables	96	384	1664	15744
ConfRes variables	6	6	6	6
Number of functions	280	1000	4200	39400
Time and memory statistics				
Consumed memory	6.2MB	7.0MB	10.3MB	51.7MB
Total exec time	83ms	160ms	657ms	40698ms
Preparation time	59ms	126ms	598ms	40426ms
Execution time	18ms	27ms	41ms	155ms
Interpretation time	6ms	7ms	18ms	117ms

In the three additional experiments the SAT engine obtained correct sets of changes, analogous to the ones proposed in the first one. As for the execution statistics, it can be seen that the growth in SAT variables follows the expected progression. Memory consumption and execution time increase accordingly, although the exact numbers are still in reasonable levels taking into account the frequency of change processes in enterprise infrastructures, with the largest case taking more than 40 seconds to obtain a solution. It must be noted that the largest scenario represents a managed repository of more than 1000 artifacts, which is already a considerable size, and is applied to a non trivial environment.

However, in case the characteristics of other domains make the execution time of this process unacceptable there is one fundamental optimization activity which could be applied to domains with very large logical repositories. Over these examples it has been mentioned that all the additional units are not part of the desired solution, thus they just pollute the problem with useless variables. In case that would be a problem, it would be possible to initially apply a logical dependency resolution process, where every unit related to the business objectives, or to an existing runtime unit, would be identified. The change identification process can be safely executed only against this subset, greatly improving its efficiency (the evolution in time and memory consumption reflect the large impact of reducing the number of involved units).

7.3.3. Variations on both Logical and Environment Size

Finally, the scalability tests were completed with a set of additional experiments increasing both the environment and the logical repository size. Both sets increase the size of the problem, so its interference should test the limits of the applicability of the current algorithm. This way, the following two experiments were applied: Case II invoked the change identification service with the environment three times the original size, and the logical repository with 20 additional units, while Case III did combine the largest environment (nine times the size) with the repository of 100 additional units. As in the other cases, the following table collects the main results from these experiments.

Table 5 Sensibility Analysis Results with More Deployment Units and Larger Environment

Input data statistics	Case I	Case II	Case III
Number of units	22	41	121
Number of dependencies	25	43	123
Number of constraints	3	3	3
Number of nodes	10	30	90
Number of containers	12	36	108
SAT input statistics			
Number of variables	145	3783	138245
RuntimeUnit variables	43	357	3951
Binding variables	96	3408	134240
ConfRes variables	6	18	54
Number of functions	280	7568	276524
Time and memory statistics			
Consumed memory	6.2MB	11.9MB	204.3MB
Total execution time	83ms	1598ms	2611720ms
Preparation time	59ms	1501ms	2609667ms
Execution time	18ms	63ms	1168ms
Interpretation time	6ms	34ms	885ms

The execution of these two experiments did also obtain correct results, identifying the same set of changes as in the other tests. The execution statistics show the multiplicative effect of both factors when both are increased simultaneously. The last one provides a rough estimation of the limits of the algorithm implementation, with more than 200Mbytes of problem size, and a total execution time of 43 minutes. Nonetheless, the change identification

service has been proved to be able to provide a solution for problems with a size of more than a hundred thousand variables and double that amount of functions.

In case problems of that complexity needed to be supported by the management system, the previous section did already detail how discarding the non-relevant *DeploymentUnits* can address those bottlenecks. It must also be taken into account that neither the preparation nor processing algorithms were completely optimized, nor the execution platform was state-of-the-art equipment, which would have improved the overall performance. However, the specific measurements described here allow an estimation of the relative increment in memory consumption and time execution.

7.4. Validation of reaction to different kinds of changes

After the base scenario has been successfully executed, and the scalability of the change identification service has been tested, the last set of validation tests will define different domain situations which demand for a different set of changes in each experiment. The base input which will be modified by each test will be the one of the initial case. Neither the expanded repositories nor the environments defined at the scalability tests will be used for those tests. As each test represents a different type of change to the domain (by modifying either the environment state or the objective), the objective of this set of tests is to verify if the provided Change Identification Service can address correctly each scenario.

For every case, the internal details on the execution will be identical to the sequence explained in the first case, with one important exception. All the scenarios selected for those tests have at least one *RuntimeUnit* available at the environment. Because of that, the optimization function will be defined in those tests, in order to preserve the already existing *RuntimeUnits* as long as they contribute to the desirability and stability of the environment. This way, the described optimization function will also be tested by these experiments. The different tests have been classified depending on the type of variations applied compared to the base validation scenario.

7.4.1. False positive validation

The first two tests which have been executed modify the state of the environment by including a correctly configured distribution of runtime instances of the nineteen required *DeploymentUnits*, as well as the required *Container Resources* for the "*datasource*" connection. For both tests same logical definitions and management objectives of the initial case will be applied. The difference among both tests lies on the exact distribution of the *RuntimeUnits*: The first one has an input the environment snapshot resulting from applying the proposed changes of the initial test to the empty environment, while the second one represents another stable and desirable configuration, but with a different distribution of the *RuntimeUnits*, as it can be seen in the next picture. In both cases the initial domain state is correct, so the result of the service execution should be a solution implying zero changes to the environment. The reason for selecting both the proposed solution and an alternative one as base inputs is for also validating whether the defined optimization function respects if possible the initial state of the environment.

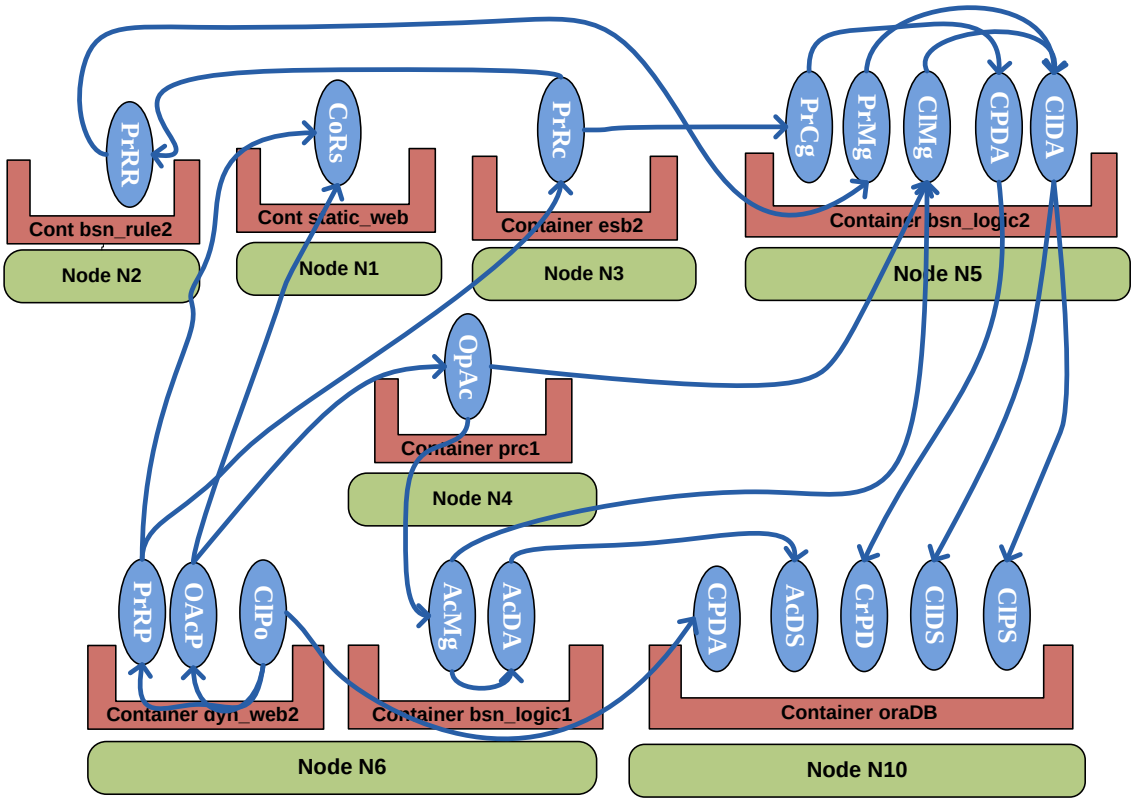

Figure 67 Alternative Correct Configuration of the Environment

For both experiments the obtained results were almost identical. In both cases 142 variables were defined, which equals to the 145 from the empty environments, minus three CR Literals, as three '*datasource*' resources were already created at the business logic containers. From those set, a total of 42 were evaluated to true (again, the same as the 45 from before, minus the three CR Literals). In each case the set of changes reflected the exact same configuration which was already at the environment, thus no changes were generated for either experiment. These experiments verify that false positives in the Change Analysis Service do not originate unnecessary changes to the system. In addition to that, it has also been shown how the use of the optimization function respects the current state of the environment, instead of proposing unnecessary changes.

7.4.2. Change in objectives validation

The next set of experiments repeats the previous two scenarios with a fundamental change: the business objective mandating that the client service is removed. This way, none of the available runtime units are contributing to the desirability of the current configuration, so they could be safely removed.

The execution results in both cases were the same. From the initial 142 variables no one was evaluated as true. Because of that, during the interpretation process 38 activities were defined, consisting of stopping and uninstalling all the *RuntimeUnits*. This way, the change identification service correctly interprets in both cases that as the management objective of providing the Client Portal Service is no longer active, there is no purpose met by those *RuntimeUnits*, so they should be removed from the environment.

7.4.3. Unstable environment validation

The next experiment shows how the Change Identification Service can react to initially unstable environment configurations. The scenario represents the original environment with the proposed changes applied, after a hardware malfunction brings down Node N2. The updated environment snapshot can be seen in Figure 68. In total, two *Containers* have disappeared (*dyn_web1* and bsn_*rules1*) and three *RuntimeUnits* are missing. The remaining *RuntimeUnits* are shown in the picture, with the broken *Bindings* represented as loose connections and another tone. With that initial input, the Change Identification Service is invoked to try to find an updated, correct state.

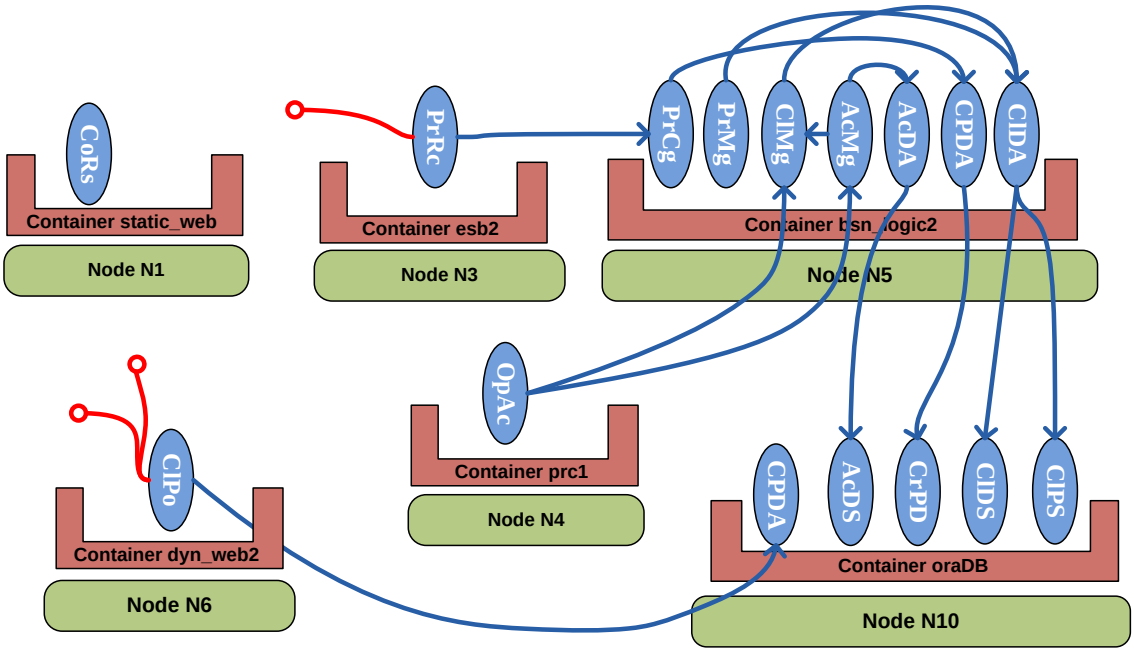

Figure 68 Environment Status After a Malfunction in Node N2

The first noticeable difference over the execution of this test is that the number of SAT literals has been reduced to 113. The decrease in the total number is logical, as the disappearance of two *Containers* has reduced the distribution options for two types of *DeploymentUnits*. However the same number of positive variables has been obtained, as all of them are required for a desirable and stable configuration. After the variables have been analyzed, a total of 28 activities have been defined. The set of changes consists of deploying the missing three components to the only possible options (the *war* units to *dyn_web2*, and the *drl* unit to *bsn_rule1)*, configuring both the broken bindings to those units (initially stopping the broken units for proper configuration), as well as the *Bindings* and *BoundProperties* from these three newly created units.

This case shows how the proposed algorithm can react to unexpected changes to the runtime environment and restore the intended system functionality. The proposed solution also reuses the already available *RuntimeUnits*, in order to minimize the set of required changes to the environment.

7.4.4. Irresoluble Case Validation

Finally I will define a test case which does not have a solution, and test whether the algorithm correctly reaches to the same conclusion. I have selected a simple variation of the base scenario, with the only different being the removal of the *ContainerResourceConfiguration* definition from the Domain information. This way, there will be three *DeploymentUnits*, (*Client Data Access, Account Data Access* and *Credit Product Data Access*) which have a *Constraint* unsatisfiable by any of the possible environment *Containers*. Those elements cannot be stably provisioned to the runtime configuration neither can any element transitively dependant from them. Therefore, it is impossible to satisfy the established objective.

The execution of the case proceeds as expected, correctly failing to obtain a possible solution to the problem. Looking at the internal details of the algorithm it can be seen that instead of the standard 145 literals of the base case, this time the execution of the algorithm, did only obtain 117 (27 less literals). This is coherent with the differences in the domain definition. The missing literals consist of 6 *Clits* (as there is no *ContainerResourceConfiguration* definitions anymore), 6 belonging to the *RLits* of the three constrained units, 8 belonging to the binding literals of these inexistent elements, and the final 8 coming from the bindings of the Account Manager, Client Manager and Profile Manager literals, which in this case could not be established. When looking at the explanation provided by the PBSAT for the problem unsatisfiability the found unsolvable conflict comes from the false value assigned to the Product Catalog literals (consequence of no stable binding found for their dependencies).

7.5. Validation Conclusions

This chapter has described the results of the execution of a set of validation tests for the prototype implementation of the SAT-based Change Identification service. The description of the test executions provide a clear insight on how the base data is processed into SAT literals and clauses, fed to the solver and finally how the results are interpreted and translated into a set of changes.

The results from the validation execution have been very positive, obtaining a correct solution for every defined scenario. Without any explicit indication of what type of changes were required, it has been shown how the combination of stability and desirability restrictions can correctly diagnose the current state of the system, and propose a set of changes which takes into account the already available units thanks to the optimization function. The input data for defining these tests has been taken from a real enterprise context, which increases the relevance of the results. Finally, an additional set of experiments have shown the scalability of the presented solution, which has performed correctly with much larger sets of input data. The collected metrics over these stress tests show the impact of those changes in the actual execution time, and provide a rough estimation on the algorithm upper limits.

8. Conclusions

The challenges for automating the change operations to enterprise services include managing the complexity and heterogeneity of the execution environments, reasoning about all elements running over the distributed platform, taking care of the existing dependencies, relationships and constraints between all of them and ensuring that the managed environment complies with the business objectives for which it has been designed. This dissertation has tried to address these challenges by combining a modeling approach with a selection of reasoning techniques over the available information. Over the previous chapters the base concepts, relationships and restrictions which must be addressed by the management system have been identified and modeled. In addition to that, an enterprise service management architecture, based on a set of algorithms which can reason automatically over the defined models, has been proposed and validated with a set of industrial case studies extracted from the ITECBAN project. The case studies address management problems which are not exclusive of the banking domain, but are common to every enterprise infrastructure. This way, the contributions of this thesis are aligned with the ITEA Roadmap objectives, described in [51].

I will close this dissertation providing a reflection on the main contributions which have been described, and assessing the objectives which were identified at the beginning of this work have been sufficiently achieved. After these conclusions have been discussed I will also mention several future research lines which were identified during the execution of this work and could lead to interesting future works.

8.1. Main Contributions

The main contribution of this dissertation is the definition of a set of models and techniques which enable the automated execution of technical service management operations. The proposed solution automatically controls the managed domain, adapting to its physical topology, characteristics of the runtime elements and established operating objectives for the infrastructure. In addition to that, the presented reasoning algorithm is not tailored to concrete use cases (initial deployment, service update, unit configuration) but instead supports a generic process consisting of evaluating the correctness of the environment starting state and obtaining the set of required changes for restoring the domain to a correct state. The flexibility of the proposed solution is supported by the two main original aspects of this work: the complete characterization of the managed system information (including the runtime state, the available logical definitions which can be instantiated and the defined objectives for the environment), and the definition of stability and desirability conditions which can be evaluated only with the modeled information. By building over those factors, an algorithm has been defined which expresses the service management problem as a boolean function processable by a specialized resolution engine; the function results being interpreted as the required management operations.

In order to detail the main contributions of this work I will review the list of detailed objectives which were identified at the beginning of the dissertation, and will explain how the contributions of this thesis address these objectives

- To define a set of models which completely characterize the problem of managing enterprise distributed services

A set of models have been proposed which allow to characterize the elements from the runtime managed environment, the available logical resources which can be instantiated into the environment and the management objectives which must be supported by the runtime infrastructure. The presented models not only represent the managed elements (the information model) but can also capture the types of relationships, dependencies, and constraints which can occur between the managed elements. This allows an automatic evaluation of the stability of any configuration by analyzing the model instances.

As regards the standards alignment, the models have been defined based on the common base between the analyzed information modeling standards described at the State of the Art analysis (mainly CIM, SDD, WSDM and OMG D&C). This way, the models are build over the concept of resources. In addition to that, the D&C abstractions of node and environment (deployment target in their terminology) have been selected as the base of the runtime model. The proposed models focus on capturing the technical information required for the management activities but they have been defined so that that information can be easily complemented with the additional information thoroughly modeled by standards such as CIM.

One of the main contributions of the proposed models is that, instead of defining independent views of the system, a unified representation of all the managed information is provided. In order to achieve that, the proposed models that represent all the domain information have clear correspondence rules. Logical elements such as deployment units can be instantiated to the runtime model as runtime units, and guidelines for this traceability are provided. On top of that, the management objectives have been defined directly related to those previous models (as existence constraints of those elements). The combination of those factors allows handling at the same level the technical requirements and the business requirements of a managed domain. This way, a single condition, that the environment must be stable and desirable, is enough to determine the correctness of the system, and the same reasoning techniques can be applied to evaluate all the modeled information.

- To support the lifecycle of enterprise services

In parallel to the managed domain characterization the dissertation has provided a definition of the role of the service management system, as a correcting factor which reacts to the external changes occurring at the domain (i.e. a hardware malfunctions causes a node to disappear or a new management objective is defined for the environment) and brings back the system to a correct state. In order to do so, the management system can apply a set of internal changes, which represent the set of allowed management operations. The set of operations which have been identified for the management system cover the complete services lifecycle, ranging from the initial provisioning of the service to the runtime environment, to its proper configuration and activation, until the final retirement of the element when is no longer needed.

- To support the complete automation of the service management

It has already been mentioned how the proposed models allow to completely define the system, and to establish whether the system is at a correct state (by means of stability and

desirability conditions). Based on this, an algorithm has been defined which can take as input that initial data, and in case the current state is not correct, formulate it as a pseudo boolean SAT problem, where all the base information is encoded into a set of variables and functions which represent the possible reachable states of the configuration and the restrictions for it to be a correct solution. With all that information, the SAT solver obtains a correct solution which represents a reachable state, and the set of required changes to achieve it are obtained (from the allowed operations of the management system) and aggregated into a change plan.

This process is completely automated, only requiring the initial domain state for a diagnosis and a solution to be proposed. However, this could also imply that the algorithm obtains a correct solution, but not the preferred one. In order to address that, the followed approach allows expressing additional constraints on the desired solution, in the form of an optimization function or additional clauses. This way, additional policies can be supported. The dissertation provides some examples of this additional customization, such as restricting the maximum number of times a deployment unit appears, or minimizing the number of changes.

- To propose a reference architecture and validate the proposed models and algorithms

A reference architecture for an enterprise service change management system, based on the previous contributions has been described based on the OSI reference model, and its role illustrated through some selected scenarios. Over this process the valuable input obtained from the work at the ITECBAN project has allowed to propose an architecture which can be integrated with the rest of the elements of the enterprise infrastructure and support all the identified management functions.

Finally, a set of validation cases have been defined and executed in order to assess the feasibility of the proposed modeling and reasoning solution, obtaining a positive outcome from all the experiments. First, the models have been proven to be expressive enough to represent the information from an enterprise system, including their specific constraints and relationships. On top of that, the reasoning algorithm has performed correctly all the defined tests, which represented a range of different use cases. Finally, additional experiments were executed to test the solution's scalability, showing the feasibility of the proposed solution for the targeted enterprise domains.

8.2. Future work

Over the development of this work a set of new research lines have been identified, which could not be fully pursued because of time constraints. However, because of their potential interest, they will be briefly mentioned in this section:

Although the focus of the dissertation was supporting enterprise service management, a set of generic management abstractions were initially defined. Those generic modeling definitions were not tied to the constraints of modeling enterprise services, and could also be applied to define other specific management models. The approach and defined abstractions were the consequence of the scope of the management system established in the Objectives chapter (represented graphically in the Figure 20), but it would be possible in the future to expand the management space supported by the solution by applying some modifications to the models and techniques.

The abstractions described in this dissertation aim at representing the complete management information about a service-based organization. Because of that, the presented validation experiments reason only about a set of internally developed services. However, in order to better support the challenges of an open services ecosystem it is necessary to also reason about the services provided by third parties (neither under the control of the organization nor the management system), and adapt automatically to their changes [89] [41]. In principle, the existing abstractions could be improved to support those elements: External service registries could be integrated in the runtime model, containing the external runtime services which can be consumed by the internal elements. On top of that, in order to restrict what bindings can be established, Service-Level Agreements and conditions of use could be represented as defined objectives and constraints, which would be later on expressed as SAT clauses and terms of the optimization function.

Another interesting case is the management of virtualized environments [27], which shares a lot of similarities with the abstractions taken for service management (hardware hosts being comparable to containers, and virtual appliances being the equivalent to deployment units). Because of that, this proposal could be adapted to that environment. Moreover, if that was the case, it would be possible to greatly extend the scope of the management system with network and system management capabilities, by combining both service and virtualization management systems. This way, it would control not only what is or not deployed to the environment but also the actual topology of the managed system, being able to instantiate new nodes or containers as part of the solution. This would be a very interesting development of this work thanks to the recent interest in the cloud computing paradigm. Nonetheless, in order to support the runtime management of virtual instances several new challenges which were not covered by this work should be addressed. Runtime changes at node level require migrating both the persistent data as well as the volatile process data. The low-level support of these aspects is being integrated into the main virtualization technologies [43], but the way to manage those concerns in a generic management system should be evaluated.

Another open topic of research is the specification of additional resource subtypes in the enterprise information metamodel, with an attached inherent behavior. Enterprise environments present some resource interaction patterns that appear in most scenarios, such as clustering and load balancing, intended to improve the reliability and efficiency of the services, or resource proxies and adaptors, such as the integration and transformation components deployed at an Enterprise Service Bus. If those special cases were identified and their special behavior was strictly defined it would be possible for the management system to automatically configure them correctly, further reducing the required manual operation of the domain.

Another interesting aspect which was identified over the execution of this work had to be discarded was the support the vigilance of Service-Level Agreement over the managed environment. This would provide advanced reconfigurability capabilities to the management system, implementing the autonomic computing self-optimization principle. However, it is not possible to adequately support those behaviors with the defined management abstractions. Although it would be possible to define SLAs as LOCAL checks over runtime properties (e.g. the average response time of a service), clearly the values of those elements cannot be directly altered by the management system, as they are a consequence of external factors, including

the incoming traffic from the service clients, the hardware capabilities of the underlying execution platform or the parameters from the network load balancer configuration. Because of those reasons, this is a very complex problem to solve in a generic way, although it should be explored because of its relevance for a complete automation of the operations.

The presented algorithm has shown the applicability of SAT solvers to reason about the problem of finding correct domain environment configurations. Although the results have been satisfactory, there is still room for improvements of the proposed algorithm. It has already been mentioned how some optimization clauses were introduced to the solver to further refine and find the preferred solution. This work should be extended, exploring how multiple preferences can be expressed, and how relative weights should be defined to prioritize competing or conflicting policies. Another possible refinement of the algorithm would be to opt for a different strategy, so that instead of taking the first valid solution received, a timeout was specified and multiple SAT invocations were executed in order to find increasingly better solutions.

9. References

[1] Agrawal D., Giles J., Lee K., Lobo J., "Autonomic Computing Expression Language". IBM developerWorks 2005, Apr 2005.

[2] Agrawal D., Kang-Won L., Lobo J. "Policy-based management of networked computing systems", Communications Magazine, IEEE 2005, Vol. 43 Issue 10, pp:69-75.

[3] Agrawal D., Calo S., Lee K., Lobo J., "Issues in Designing a Policy Language for Distributed Management of IT Infrastructures". Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on, 2007, pp:30-39.

[4] Arshad N., Heimbigner D., Wolf A., "Deployment and dynamic reconfiguration planning for distributed software systems". Software Quality Journal 2007, Vol. 15, Issue 3, pp:265-281.

[5] Bandara A., Lupu E.C., Sloman M., "Policy-Based Management" Handbook of Network and System Administration 2007,pp.507-564.

[6] Baude F., Contes V.L., Lestideau V., "Large-Scale Service Deployment—Application to OSGi. Autonomic and Autonomous Systems", Third International Conference on Autonomous Systems, ICAS07.

[7] Brenner M., Dreo Rodosek G., Hanemann A., Hegering H., Koenig R., "Service provisioning: challenges, process alignment and tool support. Handbook of Network and System Administration 2007, pp:855-904.

[8] Burgess M., Kristiansen L., "On the Complexity of Change and Configuration Management", Handbook of Network and System Administration 2007, pp:567-622.

[9] Case J.D., Fedor M., Schoffstal,l M.L., Davin J. "Simple Network Management Protocol" (SNMP). 1990.

[10] Clark J., DeRose S., XML Path Language (XPath) Version 1.0, World Wide Web Consortium (W3C), 1999.

[11] Computing IBMA. "An architectural blueprint for autonomic computing". Fourth Edition, June 2006.

[12] Conradi R., Westfetchel B.,"Version models for software configuration management", ACM Computing Surveys (CSUR),Volume 30 ,Issue 2,June 1998, pp. 232 – 282, ISSN:0360-0300.

[13] Couch A., "System Configuration Management", Handbook of Network and System Administration 2007, pp.75-134.

[14] Couch A., Chiarini M., "A Theory of Closure Operators", International Conference of Autonomous Infrastructure, Management and Security, AIMS 208, Lecture Notes in Computer Science 5127, pp.162-174, 2008.

[15] Crawford J.M., Baker A.M., "Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems", Proceedings of the Twelfth National Conference on Artificial Intelligence, 1994.

[16] Cuadrado F., Dueñas J.C, Garcia R., Ruiz J.L., "A model for enabling context-adapted deployment and configuration operations for the banking environment", Fifth International Conference on Networking and Services (ICNS), Valencia, Spain. April 2009.

[17] Danciu V.A., Felde N., Sailer M., "Declarative Specification of Service Management Attributes", IM '07. 10th IFIP/IEEE International Symposium on Integrated Network Management, 2007.

[18] Davis M., Logemann G., Loveland D., "A machine program for theorem-proving" Journal of ACM Communications, Vol.5, No 7 pp:394–397, 1962.

[19] Dean M., Connolly D., van Harmelen F., Hendler J., Horrocks I., McGuinness D., Patel-Schneider P.F., Stein L. A., Web Ontology Language (OWL) W3C Reference version 1.0, Feb 2004.

[20] Desertot M., Escoffier C., Donsez D, "Autonomic management of J2EE edge servers" MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing New York, NY, USA: ACM; 2005.

[21] Desertot M, Escoffier C, Lalanda P, Donsez D. "Autonomic Management of Edge Servers" Self-Organizing Systems 2006, pp:216-229.

[22] Di Nitto E., Ghezzi C., Metzger A., Papazoglou M., Pohl K, "A journey to highly dynamic, self-adaptive, service-based applications", Automated Software Engineering, Ed. Springer, Vol. 15, pp. 313-341, 2008, DOI 10.1007/s10515-008-0032-x.

[23] DMTF (Distributed Management Task Force) Common Information Model (CIM) specification v2.19. 2008.

[24] DMTF (Distributed Management Task Force) WS-Management specification v1.0.0. 2008.

[25] Dubus J, Merle P., "Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-Based Systems". Models in Software Engineering, Lecture Notes in Computer Science 2007, pp: 242-251.

[26] Dueñas J.C., Ruiz J.L., Santillan M. "An end-to-end service provisioning scenario for the residential environment". Communications Magazine, IEEE 2005, Vol.43, Issue 9, pp: 94-100.

[27] Dueñas J.C. Ruiz, J.L., Cuadrado F., García B., Parada, H.A., "System Virtualization Tools for Software Development". Internet Computing Magazine, IEEE, Sep-Oct 2009.Vol 13. Issue 5.

[28] Efftinge, S., Voelter, M., "oAW xText: A framework for textual DSLs", Eclipse Summit Europe, Esslingen, Germany, October 2006.

[29] Eilam T., Kalantar M.H., Konstantinou A.V., Pacifici G., Pershing J., Agrawal A. "Managing the configuration complexity of distributed applications in Internet data centers". Communications Magazine, IEEE 2006, Vol.44, Issue 3, pp: 166-177.

[30] Fallside D., Walmsley P., XML Schema Part 0: Primer Second Edition, Word Wide Web Consortium, 2004.

[31] Fiore M., Plotkin G., Turi D., "Abstract Syntax and Variable Binding", Proceedings of 14th Annual IEEE Symposium on Logic in Computer Science, LICS'99, Trento, Italy, 2–5 July 1999, IEEE CS Press, Los Alamitos, CA, 1999, pp. 193–202.

[32] Fleury M., Lindfor J, JBoss Group, "JMX: Managing J2EE with Java Management Extensions", Ed. SAMS Publishing, 2002, ISBN: 0-672-32288-9.

[33] Fowler M. "Language Workbenches: The Killer-App for Domain Specific Languages." Accessed online from: http://www.martinfowler.com/articles/languageWorkbench.html 2005.

[34] Frenken T., Spiess P., Anke J, "A Flexible and Extensible Architecture for Device-Level Service Deployment", ServiceWave 2008, Towards a Service-Based Internet, pp. 230-241.

[35] GB921 T. Enhanced Telecom Operations Map®(eTOM) The Business Process Framework - eTOM ITIL application note – Using eTOM to model the ITIL processes. Approved Version 2004;4.

[36] Ghallab M., Howe A., Knoblock C., McDermott D., Ram A., Veloso M., Weld D., Wilkins D., "PDDL — The Planning Domain Definition language". Tech. rep., Yale Center for Computational Vision and Control, October 1998.

[37] Goldsack P., Guijarro J., Lain A., Mecheneau G., Murray P, Toft P. "SmartFrog: Configuration and Automatic Ignition of Distributed Applications". HP OVUA 2003.

[38] González J.M., Lozano J.A., López de Vergara J., Villagrá V., "Context aware services offering for residential environments". Proceedings of the First IEEE Workshop on Autonomic Communication and Network Management (ACNM), Munich, Germany 2007 May 25 2007, pp: 48-55.

[39] Graham S., Karmarkar A., Mischkinsky J., Robinson I., Sedukhin I.. Web Services Resource Framework (WS-RF) 1.2. OASIS Standard 2006.

[40] Graham S., Treadwell J., Web Services Resource Properties Specification (WS-ResourceProperties) 1.2. OASIS Standard 2006.

[41] Gu Q., Lago P., "Exploring service-oriented system engineering challenges: a systematic literature review", Service-Oriented Computing and Applications, Vol. 3, No. 3, pp. 171-188, Sep. 2009.

[42] Hegering H.G., Abeck S., Neumair B." Integrated management of networked systems: concepts, architectures, and their operational application". San Francisco, CA, USA: Morgan Kaufmann Publishers Inc; 1998.

[43] Hines, M.R., Deshpande, U., Gopalan, K. "Post-Copy Live Migration of Virtual Machines" SIGOPS Operating Systems Review, Volume 43 Issue 3, July 2009.

[44] Hochstatter I., Dreo G., Serrano M, Serrat J, Nowak K, Trocha S. "An architecture for context-driven self-management of services". Computer Communications Workshops, 2008. INFOCOM. IEEE Conference on 2008.

[45] Hoffmann A., Neubauer B. "Deployment and Configuration of Distributed Systems". System Analysis and Modeling 2005, pp: 1-16.

[46] Holzner S., Nehren D., Galbraith B. "Ant: the definitive guide" Ed. O'Reilly & Associates, Inc., 2005, Sebastopol, CA, USA, ISBN: 0596006098.

[47] Horrocks I., Patel-Schneider P.F., Boley H., Tabet S., Grosof B., Dean M. "SWRL: A semantic web rule language combining OWL and RuleML", May 2004.

[48] Hutter F., Hoos H.H., Stutzle T. "Automatic algorithm configuration based on local search," in AAAI '07: Proc. of the Twenty-Second Conference on Artifical Intelligence, 2007, pp. 1152–1157.

[49] International Telecommunication Union – Telecommunication Standardization Sector (ITU-T) Information Technology – Open Systems Interconnection – Systems management Overview. X.701 Recommendation, 1997.

[50] International Organization for Standarization. ISO/IEC 20000-1:2005 Information Technology - Service Management - Part 1: Specification. ; 2005.

[51] ITEA2 Core Roadmap Team, "ITEA technology roadmap for software-intensive systems and services", 3rd Edition, Released on February 2009, available at http://www.itea2.org.

[52] Jellife, R., The Schematron Assertion Language 1.5 Specification Version, 2002.

[53] Jennings B., van der Meer S., Balasubramaniam S., Botvich D., Foghlu M.O., Donnelly W. "Towards autonomic management of communications networks" Communications Magazine, IEEE 2007, Vol.45, Issue 10, pp:112-121.

[54] Johnson M., Hately A., Miller B., Orr R. "Evolving standards for IT service management". IBM Systems Journal 2007, Vol.46, Issue 3.

[55] Keller A., Brown A.B., Hellerstein J.L. "A Configuration Complexity Model and Its Application to a Change Management System". Network and Service Management, IEEE Transactions on 2007, Vol.4, Issue 1, pp:13-27.

[56] Keller A., Hellerstein J.L., Wolf JL, Wu K-, Krishnan V. "The CHAMPS system: change management with planning and scheduling". Network Operations and Management Symposium, 2004. NOMS 2004. Vol.1. pp: 395-408

[57] Keller A., Badonnel R. "Automating the Provisioning of Application Services with the BPEL4WS Workflow Language". Utility Computing 2004, pp: 15-27.

[58] Kephart J.O., Chess D.M. "The vision of autonomic computing" Computer 2003, Vol.36, Issue 1, pp: 41-50.

[59] Klepper A., Warmer, J., Bast, W., "MDA Explained. The Model Driven Architecture™: Practice and Promise", Ed. Addison Wesley, 2003, ISBN: 0-321-19442-X.

[60] Klerer S.M., "The OSI Network Management Architecture. An overview". IEEE Network, March 1988, Vol.2 No. 2.

[61] Kolari P., Finin T., Yesha Y., Lyons K., Hawkins J., Perelgut S. "Policy management of enterprise systems: a requirements study". Policies for Distributed Systems and Networks, 2006. Seventh IEEE International Workshop on 2006

[62] Kon, F., Marques, J.R., Yamane, T., Campbell, R.H., Mickunas, M.D., "Design, implementation and performance of an automatic configuration service for distributed component systems", Software Practice and Experience, issue 35, pp. 667-703, 2005.

[63] Kreger H, Studwell T. "Autonomic Computing and Web Services Distributed Management" IBM DeveloperWorks 2005 June 2005.

[64] Kruchten P., "The 4+1 View Model of Architecture," IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.

[65] Le Berre D., Parrain A, "On SAT Technologies for dependency management and beyond" Limerick, First Workshop on Analyses of Software Product Lines, ASPL September 2008.

[66] Le Berre D., Rapicault P. "Dependency Management for the Eclipse Ecosystem. Eclipse p2. ,metadata and resolution". Proceedings of the International Workshop on Open Component Ecosystems 2009, IWOCE, August 2009, Amsterdam, the Netherlands.

[67] López de Vergara J.E. "Especificación de modelos de gestión de red integrada mediante el uso de tecnologías y técnicas de representación del conocimiento", Tesis Doctoral, 2003. Universidad Politécnica de Madrid.

[68] López de Vergara J.E., Villagrá V.A., Berrocal J., "On the Formalization of the Common Information Model Metaschema", Proceedings of the 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, (DSOM 2005), Barcelona, Spain ,2005.

[69] López de Vergara J.E., Villagrá V.A., Fadón C., González J.M., Lozano J.A., Álvarez-Campana M. "An autonomic approach to offer services in OSGi-based home gateways" Computer Communications, 2008, pp: 3049-3058.

[70] Loughran S., Hatcher E., "Ant in Action". Ed. Manning Press, 2007. ISBN: 193239480X.

[71] Machado G.S., Daitx F.F., Cordeiro Weverton L., Both C.B., Gaspary L.P., Granville L.Z. "Enabling rollback support in IT change management systems". Network Operations and Management Symposium, 2008. NOMS 2008. IEEE 2008, pp: 347-354.

[72] Maghraoui K., Meghranjani A., Eilam T., Kalantar M., Konstantinou A. "Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools". Middleware 2006, pp: 404-423.

[73] Marques-Silva, J., "Practical Applications of Boolean Satisfiability", International Workshop on Discrete Event Systems, WODES08, Gothenburg, Sweden, May 2008.

[74] McCloghrie M., Perkins D., Schönwalder J. Conformance Statements for SMIv2. RFC 2580. 1999.

[75] McCloghrie M, Rose M. "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II.", RFC1213, 1991.

[76] Microsoft Corp., "Devices Profile for Web Services", http://schemas.xmlsoap.org/ws/2006/02/devprof, 2006.

[77] Miller J., Mujerji J., "MDA Guide, version 1.0.1." Document number: ormsc/06-09-03 2006.

[78] Mendoza A. Utility Computing. Technologies, Standards and Strategies. 1st ed.: Artech House; 2007.

[79] Miller B, McCartht J., Dickau R., Jensen M. OASIS Solution Deployment Descriptor (SDD) 1.0. OASIS Standard Sep 2008.

[80] Moore B., Ellesson E., Strassner J., Westerinen A. Policy Core Information Model—Version 1 Specification. 2001.

[81] Murray P., Goldsack A., "Fully distributed service configuration management" HotDep'07: Proceedings of the 3rd workshop on Hot Topics in System Dependability Berkeley, CA, USA: USENIX Association; 2007.

[82] Oberle D. "Semantic Management of Middleware, volume 1 of Semantic Web and Beyond: Computing for Human Experience": Springer, Berlin; 2006.

[83] Object Management Group. Deployment and Configuration of Distributed Component-based Applications Specification. Version 4.0. April 2006.

[84] Office of Government Commerce. Service Delivery, IT Infrastructure Library. : 2001.

[85] Office of Government Commerce. Service Support, IT Infrastructure Library. : 2000.

[86] Open Services Gateway Initiative (OSGi) Service Platform, Core Specification, Release 4.2, available online at http://www.osgi.org, 2009.

[87] Pandit B., Popescu V., Smith V., Service Modeling Language, Version 1.1 (SML). World Wide Web Consortium, 2009.

[88] Pandit B., Popescu V., Smith, V., Service Modeling Language Interchange Format, Version 1.1 (SML). World Wide Web Consortium, 2009.

[89] Papazoglou M.P., Traverso P., Dustdar S. , Leymann F., Krämer B.J. "Service-Oriented Computing: A Research Roadmap" International Journal of Cooperative Information Systems, Vol. 17, No. 2 pp.223–255, 2008.

[90] Pras A., Martin-Flatin J.P. "What can Web Services bring to integrated management?" Handbook of Network and System Administration 2007, pp:241-294.

[91] Prasad M., Biere, A., Gupta, A, "A survey of recent advances in SAT-based formal verification", International Journal on Software Tools for Technology Transfer, Vol.7 No 2, April 2005 Ed. Springer-Verlag.

[92] Rudin W, "Principles of Mathematical Analysis", International series in pure and applied mathematics. Ed. Mc Graw Hill, 3rd Edition, 1976. ISBN 0070856133.

[93] Ruiz J.L. "A policy-driven, model-based, software and services deployment architecture for heterogeneous environments". Tesis doctoral, Universidad Politécnica de Madrid; 2007.

[94] Ruiz J.L., Dueñas J.C., Cuadrado, F. "Model-based context-aware deployment of distributed systems" *Communications Magazine, IEEE* , vol.47, no.6, pp.164-171, June 2009.

[95] Russell S.J., Norvig P. "Artificial Intelligence: A modern Approach", Ed. Prentice Hall, 3rd Edition., ISBN: 0136042597.

[96] Sahai A., Pu C., Jung G., Wu Q., Yan W., Swint G.S. "Towards Automated Deployment of Built-to-Order Systems" Ambient Networks 2005, pp: 109-120.

[97] Schaaf T., Brenner M., "On tool support for Service Level Management: From requirements to system specifications," *Business-driven IT Management, 2008.* BDIM 2008. 3rd IEEE/IFIP International Workshop on , vol. 7, no.7, pp.71-80,April 2008.

[98] Scheffer P., Strassner J, "IT Service Management". Handbook of Network and System Administration, Ed. Elsevier, 2007. pp: 905-928.

[99] Schönwälder J. "Internet management protocols" Handbook of Network and System Administration Ed. Elservier, 2007. pp:295-328.

[100] Schonwalder J. "Characterization of SNMP MIB modules." Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on 2005, pp: 615-628.

[101] Schwartzberg, S., Couch, A., "Experience in Implementing an HTTP Service Closure", Proceedings of the Eighteen Systems Administration Conference, LISA 2004. USENIX Association, Atlanta, USA, November, 2004.

[102] Sidor D.J.: "TMN Standards; Satisfying Today's needs while preparing for tomorrow." IEEE Communications Magazine, March 1998.

[103] Sipse M. "Introduction to the Theory of Computation*",* Second Edition, 2006,Thomson Course Technology, Boston, MA, ISBN-13: 978-0-534-95097-2

[104] Sloman M. "Policy driven management for distributed systems." Journal of Network and Systems Management 1994 12/30, pp:333-360.

[105] Srivastava B. "AutoSeek: A Method to Identify Candidate Automation Steps in IT Change Management" Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on 2007, pp: 864-867.

[106] Steinberg D., Budinsky F., Paternostro F., Merks E., "EMF: Eclipse Modeling Framework", 2nd edition, Ed. Addison-Wesley, 2008. ISBN: 978-0321331885.

[107] Strassner J. "Knowledge Engineering Using Ontologies", Handbook of network and service administration. Ed. Elsevier, 2007. pp. 425-455.

[108] Strassner J., Samudrala S., Cox G., Liu Y., Jiang M., Zhang J., et al. "The Design of a New Context-Aware Policy Model for Autonomic Networking" Autonomic Computing, 2008. ICAC '08. International Conference on 2008, pp: 119-128.

[109] Sun Y., Couch, A. "Complexity of System Configuration Management", Handbook of network and service administration. Ed. Elsevier, 2007. pp. 623-651

[110] Tele-Management-Forum. Enhanced Telecom Operations Map®(eTOM) The Business Process Framework - eTOM ITIL application note – Using eTOM to model the ITIL processes. Approved Version April 2004.

[111] Swint G.S., Gueyoung J., Pu C., Sahai A. Automated Staging for Built-to-Order Application Systems. Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP 2006, pp: 361-372.

[112] Talwar V, Milojicic D, Qinyi Wu, Pu C., Yan W., Jung G. "Approaches for service deployment". Internet Computing, IEEE 2005, Vol.9, Issue 2, pp: 70-80.

[113] Tucker C., Shuffleton D., Jhala R., Lerner S., "OPIUM: Optimal Package Install Uninstall Manager", 29th International Software Engineering Conference, ICSE07, Minneapolis, USA.

[114] UPnP Forum, "UPnP Device Architecture", 2006. Available at http://www.upnp.org/specs.

[115] Vambenepe W., Bullard W. Web Services Distributed Management: Management using Web Services (MUWS 1.1) Part 1. OASIS Standard Sep. 2006

[116] Verma D.C. "Simplifying network administration using policy-based management" Network, IEEE 2002, Vol.16, Issue 2, pp: 20-26.

[117] Warmer J., Kleppe A., "The Object Constraint Language: Getting Your Models Ready for MDA". Ed. Addison-Wesley, 2nd Edition, Sept. 2003, ISBN: 0321179366.

[118] Wilson K., Sedukhin I. Web Services Distributed Management: Management Of Web Services (MOWS 1.1). OASIS, Standard 2006.