The AdCIM framework: extraction, integration and persistence of the configuration of distributed systems

Iván Díaz Álvarez

Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems
University of A Coruña, Spain

PhD THESIS

The AdCIM framework: extraction, integration and persistence of the configuration of distributed systems

Iván Díaz Álvarez

May 2010

PhD Advisor:
Juan Touriño Domínguez

Dr. Juan Touriño Domínguez
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

CERTIFICA

Que la memoria titulada "*The AdCIM framework: extraction, integration and persistence of the configuration of distributed systems*" ha sido realizada por D. Iván Díaz Álvarez bajo mi dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Informática.

A Coruña, 24 de Marzo de 2010

Fdo.: Juan Touriño Domínguez
Director de la Tesis Doctoral

Fdo.: Juan Touriño Domínguez
Director del Dpto. de Electrónica y Sistemas

Resumen de la Tesis

Este resumen se compone de una introducción, que explica el enfoque y contexto de la Tesis, seguida de una sección sobre su organización en partes y capítulos. Después, sigue una enumeración de las contribuciones recogidas en ella, para finalizar con las conclusiones y trabajo futuro.

Introducción

Los administradores de sistemas tienen que trabajar con la gran diversidad de hardware y software existente en las organizaciones actuales. Desde el punto de vista del administrador, las infraestructuras homogéneas son mucho más sencillas de administrar y por ello más deseables. Pero, aparte de la dificultad intrínseca de mantener esa homogeneidad a la vez que progresa la tecnología y las consecuencias de estar atado a un proveedor fijo, la propia homogeneidad tiene riesgos; por ejemplo, las instalaciones en monocultivo son más vulnerables contra virus y troyanos, y hacerlas seguras requiere la introducción de diferencias aleatorias en llamadas al sistema que introduzcan diversidad artificial, una medida que puede provocar inestabilidad (ver Birman y Schneider [7]).

Esto hace la heterogeneidad en sí casi inevitable, y una característica de los sistemas reales difícil de obviar. Pero de hecho conlleva más complejidad. En muchas instalaciones, la mezcla de Windows y derivados de Unix es usual, ya sea en combinación o divididos claramente en clientes y servidores. Las tareas de administración en ambos sistemas son diferentes debido a las diferencias en ecosistema y modo de conceptualizar los sistemas informáticos acaecidas tras años de divergencia en interfaces, sistemas de configuración, comandos y abstracciones.

A lo largo del tiempo ha habido muchos intentos de cerrar esa brecha, y algunos lo hacen emulando o versionando las herramientas Unix, probadas a lo largo de muchos años. Por ejemplo, la solución de Microsoft, Windows Services for Unix [60] permite el uso de NIS, el Network File System (NFS), Perl, y el shell Korn en Windows, pero no los integra realmente en Windows, ya que está más orientado a la migración de aplicaciones. Cygwin [70] soporta más herramientas, como Bash y las Autotools de GNU, pero se centra en la traslación directa a Windows de programas Unix basados en POSIX usando gcc. Outwit [94] es un port muy interesante del conjunto de herramientas Unix que integra los pipelines de Unix en Windows y permite acceder al Registro, los drivers ODBC y al portapapeles desde los shells de Unix, pero los scripts desarrollados para este sistema no son usables directamente en sistemas Unix. Por lo tanto, la separación sigue a pesar de dichos intentos.

En esta Tesis presentamos un *framework*, denominado AdCIM, para la administración de la configuración de sistemas heterogéneos. Como tal, su objetivo es integrar y uniformizar la administración de estos sistemas abstrayendo sus diferencias, pero al mismo tiempo ser flexible y fácil de adaptar para soportar nuevos sistemas rápidamente. Para lograr dichos objetivos la arquitectura de AdCIM sigue el paradigma de orientación a modelo [41], que propone el diseño de aplicaciones a partir de un modelo inicial, que es transformado en diversos "artefactos", como código, documentación, esquemas de base de datos, etc. que formarían la aplicación. En el caso de AdCIM, el modelo es CIM, y las transformaciones se efectúan utilizando el lenguaje declarativo XSLT, que es capaz de expresar transformaciones sobre datos XML. AdCIM realiza todas sus transformaciones con XSLT, excepto la conversión inicial de ficheros de texto plano a XML, hecha con un párser especial de texto a XML. Los programas XSLT, también denominados *stylesheets* ("hojas de estilo"), enlazan y transforman partes específicas del árbol XML de entrada, y soportan ejecución recursiva, formando un modelo de programación declarativo-funcional con gran potencia expresiva.

El modelo elegido para representar los dominios de administración cubiertos por el framework es CIM (Common Information Model) [23], un modelo estándar, extensible y orientado a objetos creado por la Distributed Management Task Force (DMTF). Usando esquemas del modelo CIM, los múltiples y distintos formatos de configuración y datos de administración son traducidos por la infraestructura de AdCIM en instancias CIM. Los esquemas CIM también sirven como base para generar formularios web y otros esquemas específicos para validación y persistencia de los datos.

El desarrollo de AdCIM como un framework orientado al modelo evolucionó a partir de nuestro trabajo previo [89], que extraía datos de configuración y los almacenaba en un repositorio LDAP utilizando scripts Perl. En sucesivos trabajos, como [22], se empezó a trabajar con la orientación a modelo y se demostró la naturaleza adaptativa de este framework, mediante adaptaciones a entornos Grid [20] y a Wireless Mesh Networks en [21].

El enfoque e implementación de este framework son novedosos, y usa algunas tecnologías definidas como estándares por organizaciones internacionales como la IETF, la DMTF, y la W3C. Vemos el uso de dichas tecnologías como una ventaja en vez de una limitación en las posibilidades del framework. Su uso añade generalidad y aplicabilidad al framework, sobre todo comparado con soluciones ad-hoc o de propósito muy específico.

A pesar de esta flexibilidad, hemos intentado en todo lo posible definir y concretar todos los aspectos de implementación, definir prácticas de uso adecuadas y evaluar el impacto en el rendimiento y escalabilidad del framework de la elección de las distintas tecnologías estándar.

Organización de la Tesis

Esta Tesis está organizada en dos partes, precedidas por una introducción general en el Capítulo 1, y seguidas por un capítulo con conclusiones y trabajo futuro (Capítulo 10), la bibliografía y un glosario, que creemos muy necesario en un trabajo con gran cantidad de términos técnicos y acrónimos.

La primera parte introduce el framework AdCIM en el Capítulo 2, y luego lo detalla por capas de diseño, comenzando por la Capa de Modelado en el Capítulo 3, la Capa de Datos de Sistema en el Capítulo 4, y luego la de Persistencia de Datos en el Capítulo 5. Finalmente, la Capa de Aplicación se trata en el Capítulo 6.

Una parte considerable de la investigación y desarrollo del framework concierne a su adaptación a escenarios y casos de uso complejos. La segunda parte de la Tesis se debe a dicha naturaleza del trabajo, con el objetivo de ponerlo en contexto y entender su uso y alcance. Empieza con una pequeña introducción, seguida por tres ejemplos de aplicación. El primero, en el Capítulo 7, es una extensión de la Capa de Datos de Sistema para soportar el formato del fichero de configuración Sendmail, seguido de una adaptación al middleware de Grid Globus en el Capítulo 8, en concreto para poder utilizar información CIM en el el servicio de publicación de información de administración de Globus. Dicha parte de la Tesis se cierra con el ejemplo más complejo de los tres en el Capítulo 9, una aplicación a las Wireless Mesh Networks (un tipo especial de red inalámbrica), que sirve también como demostración de las capacidades de razonamiento ontológico del framework.

Metodología de Trabajo

El framework presentado en esta Tesis se basa en el paradigma de la orientación a modelo, por lo que el desarrollo se enfocó en la especificación del modelo y en sus transformaciones a diversos artefactos que especifican o implementan la diversa funcionalidad del framework. Por ello, para cada dominio determinado se siguieron los siguientes pasos:

- Determinar, describir y establecer requisitos del dominio modelizado.
- Adaptar y extender el modelo CIM para el dominio modelizado.
- Especificar los artefactos necesarios (Soporte de persistencia, esquemas, interfaces de usuario, código).
- Determinar el formato de extracción de la configuración y establecer los métodos usados.
- Transformar e integrar los elementos anteriores.

En el caso de usar ontologías para la implementación de procesos de razonamiento, se requiere también:

- Establecer formalmente las reglas tanto explícitas como implícitas del dominio.
- Formular las limitaciones semánticas y lógicas del dominio.
- Especificar una ontología y taxonomía que complemente al modelo del dominio e incluya los dos puntos anteriores.
- Suplementar con reglas lógicas para políticas y limitaciones del tratamiento ontológico.

Contribuciones

Debido a su naturaleza flexible y a la complejidad del problema, las contribuciones que aporta la Tesis son múltiples. En resumen, AdCIM provee infraestructura para:

- Representación eficiente y extensión del modelo CIM usando un formato especial de CIM (denominado miniCIM), que ocupa poco espacio y es fácil de procesar.
- Extracción, basada en gramáticas, de información de configuración y administración como instancias miniCIM. Como caso complejo de estudio, hemos adaptado este proceso a la configuración del agente de correo Sendmail.
- Persistencia transparente de datos miniCIM en un repositorio (un directorio LDAP, aunque AdCIM es independiente del repositorio), incluyendo mapeado de esquema. Hemos conseguido un escalado lineal en sistemas multicore del rendimiento de esta capa de persistencia.
- Consulta y modificación de la información miniCIM, soportada por el directorio mediante servicios web REST.
- Pregeneración y asignación de estilo de interfaces web de usuario XForms para crear aplicaciones de administración que pueden comunicarse directamente con los servicios web antes mencionados.
- Especificación formal de ontologías para dominios de administración con soporte para procesos de razonamiento y delineación de políticas de administración.

Conclusiones

En este trabajo se describe en detalle AdCIM, un framework orientado al modelo basado en CIM y destinado al desarrollo de aplicaciones para la integración de la administración de sistemas distribuidos heterogéneos. Su naturaleza orientada al modelo significa que es capaz de hacer prototipado y adaptación rápida a escenarios complejos, como aquellos descritos en los Capítulos 7, 8 y 9. AdCIM aporta hojas de estilo XSLT que producen varios artefactos a partir del esquema, por lo que las extensiones y modificaciones del modelo pueden transformarse rápidamente en esquemas de bases de datos, documentos XML o formularios web. Dichos artefactos también respetan las restricciones semánticas, ya que se derivan directamente del esquema.

AdCIM también soporta la extracción de datos CIM de varias fuentes de configuración, y especialmente desde ficheros de texto plano, prevalentes en sistemas Unix, pero también soporta el subsistema WMI de Windows, configuraciones de firmware y otras fuentes. En el caso de ficheros de texto plano, soporta configuraciones complejas como la de Sendmail (ver Capítulo 7).

AdCIM también deriva esquemas de bases de datos como parte de los artefactos soportados y, usando las plantillas XSLT que proporciona, soporta la persistencia de datos CIM en repositorios LDAP. El rendimiento de esas hojas de estilo XSLT se muestra como escalable y adaptable a arquitecturas multicore. Parte de este rendimiento es debido al uso de nuestro formato miniCIM, que minimiza la sobrecarga debida a utilizar datos CIM codificados en XML. Esta codificación es posible utilizando un esquema XML derivado del esquema original de CIM.

Los formularios XForms se generan con el objetivo de manipular instancias CIM de datos de configuración y administración de un modo general y simple para el usuario. Estos formularios son fácilmente prototipables, pero al mismo tiempo soportan personalización con CSS y su visualización en navegadores comunes. Su acceso a los datos CIM es intermediado por una interfaz de servicios web REST que expone estos datos para clientes web y aplicaciones externas.

Por último, la naturaleza semiformal de CIM se complementa con restricciones semánticas adicionales con el objeto de crear una especificación formal ontológica OWL, con la cual se demuestran capacidades de razonamiento formal y la delineación de políticas basadas en reglas de Horn.

Todos estos componentes software se ensamblan para abstraer, integrar y asociar todos los datos de configuración y administración en un todo coherente. La tarea de desarrollo se complementa con hojas de estilo XSLT proporcionadas por el framework y con el uso de tecnologías de integración, como servicios web REST, XForms y XSLT. Aunque los recursos consumidos en los nodos dependen de la implementación elegida para el repositorio, del procesador XSLT y del intérprete XForms, hemos conseguido un uso de memoria y procesador que apenas interfiere con otros procesos en la máquina.

Trabajo Futuro

Nuestro trabajo futuro incluye el estudio de otros repositorios distribuidos, el uso más completo de ontologías para agregar y derivar nuevo conocimiento a partir de información adquirida en los nodos administrados, la extensión a nuevos dominios de administración como Cloud Computing, y el desarrollo de aplicaciones para la diagnosis y recuperación automática de condiciones de error distribuidas en sistemas con gran heterogeneidad.

International Publications

1. Jesús Salceda, Iván Díaz, Juan Touriño, Ramón Doallo. CIM Modeling for System Management (invited talk).*Large Scale System Configuration Workshop at the 17th Usenix Large Installation Systems Administration Conference*, LISA'03. San Diego, USA, October 2003.

2. Jesús Salceda, Iván Díaz, Juan Touriño, Ramón Doallo. A Middleware Architecture for Distributed Systems Management.*Journal of Parallel and Distributed Computing*, volume 64, number 6, pages 759-766, June 2004.

3. Iván Díaz, Juan Touriño, Jesús Salceda, Ramón Doallo. A Framework Focus on Configuration Modeling and Integration with Transparent Persistence. In*Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005, Workshop on System Management Tools for Large-Scale Parallel Systems*, page 297a, Denver, USA, April 2005.

4. Iván Díaz, Juan Touriño, Ramón Doallo. Towards Low-Latency Model-Oriented Distributed Systems Management. In*Proceedings of the 10th Asia-Pacific Network Operations and Management Symposium, APNOMS 2007*, Lecture Notes in Computer Science volume 4773, pages 41-50. Sapporo, Japan, October 2007.

5. Iván Díaz, Gracia Fernández, María J. Martín, Patricia González, Juan Touriño. Integrating the Common Information Model with MDS4. In*Proceedings of the 9th IEEE/ACM International Conference on Grid Computing, Grid 2008*, IEEE Computer Society, pages 298-303. Tsukuba, Japan, September 2008.

6. Iván Díaz, Cristian Popi, Olivier Festor, Juan Touriño, Ramón Doallo. Ontological Configuration Management for Wireless Mesh Routers. In*Proceedings of the 9th International Workshop on IP Operations and Management, IPOM 2009*, Lecture Notes in Computer Science, volume 5843, pages 116-129. Venice, Italy, October 2009.

7. Iván Díaz, Juan Touriño, Ramón Doallo. AdCIM: a System Administration Framework for Heterogeneous Environments. December 2009 (submitted for Journal publication).

8. Iván Díaz, Gracia Fernández, María J. Martín, Patricia González, Juan Touriño. Extending the GT4 Information Service with the Common Information Model. March 2010 (submitted for Conference publication).

Abstract

System administration is a complex field constantly growing in scale and complexity in tandem with new hardware and software developments. There are many solutions that enable scalable administration but they are generally proprietary and non-interoperable in nature, causing vendor lock-in and data fragmentation into unrelated information silos. There is also a need to manage and abstract an increasing heterogeneity in networks. This Thesis describes a management application framework, named AdCIM, that unifies representation, integration and persistence of management and configuration data into CIM model objects. CIM is an object oriented, extensible model for management and configuration. The framework presented can integrate data from existing agents, registries or configuration files, as complex as those of Sendmail, and is specialized in heterogeneous networks, such as Grids and Wireless Mesh Networks. The framework also supports ontological reasoning procedures and policy enforcement, uses standard and open technologies like XSLT, XForms and LDAP and has a declarative and model-driven nature.

Dedicado a. . .

Mis padres.

Cris, por tener tanta paciencia.

Miguel, por tener visión de futuro.

Acknowledgements

The completion of a Thesis is not a task to undertake alone, so there are people whose support and dedication I wish to acknowledge. First, I want to acknowledge my PhD supervisor Juan for the confidence, support and attention placed in me. I must also not forget to mention my colleagues in the Department of Electronics and Systems and the good working environment they provided.

I gratefully thank the following institutions for funding this work: *Department of Electronics and Systems of A Coruña* for the human and material support, *University of A Coruña* for financing my attendance at several conferences, and *Xunta de Galicia* and *Spanish Government* for the projects *Soluciones Middleware y Hardware en Computación de Altas Prestaciones: Aplicación a Códigos Multimedia y de Simulación* (TIN2004-07797-C02-02), *Arquitectura Software y Marco Metodológico basados en Modelos Formales para la Generación Sistemática de Herramientas de Administración de Sistemas* (PGIDIT06PXIB105228PR) and *Soluciones Hardware y Software para Computación de Altas Prestaciones* (TIN2007-67537-C03-02).

I also thank the *Spanish Government* for funding my work with FPI grant BES-2005-8758, and the INRIA LORIA Group MADYNES and its Research Director Olivier Festor for its collaboration and support in the work done in Chapter 9 during a research visit financed by the aforementioned FPI grant.

Iván Díaz

"A thousand mile journey begins with one step", Lao-Tse

Contents

List of Tables

List of Figures

Chapter 1

Introduction

System administrators have to take into account the great diversity of hardware and software existing nowadays in organizations. From the point of view of administrators, a homogeneous infrastructure would be much easier to manage, and definitely desirable. But, besides the difficulty of preserving homogeneity against the force of progress and the fear and consequences of vendor lock-in, homogeneity also presents risks. Monoculture, for example, makes sites more vulnerable to viruses and trojans, and making them secure requires the introduction of artificial diversity measures, which affect system stability greatly (see Birman and Schneider [7]).

This makes the existence of heterogeneity almost unavoidable, and a feature of real systems that can not be easily factored out. But it brings added complexity. In many sites, the combination of Windows and Unix-like machines is not unusual, whether mixed or in either side of the client/server divide. System administration tasks in both systems are different due to the divergence in ecosystem and culture formed during several years that created a great variety of interfaces, configuration storage, commands and abstractions.

There have been many attempts to bridge the gap between Windows and Unix, and some emulate or port the time-proven Unix toolset. For example, Windows Services for Unix [60] are Microsoft's solution, enabling the use of Network Information Service (NIS), Network File System (NFS), Perl, and the Korn shell in Windows, but it is not really integrated with Windows as it is more a migration-oriented toolset. Cygwin [70] supports more tools, such as Bash and the GNU Autotools, but it is centered in porting Portable Operating System Interface for Unix (POSIX) compliant code to Windows using gcc. Outwit [94] is a very interesting port of the Unix tools that integrates Unix pipelines in Windows and allows accessing the Registry, Open Database Connectivity (ODBC) drivers, and the clipboard from a Unix shell, but the resulting scripts are not directly usable in Unix. Therefore, the divergence is still not solved.

In this Thesis we present a framework, named AdCIM, for the management of the configuration of heterogeneous systems. As such, its aim is to integrate and uniformize the management of these systems abstracting the differences, but at the same time being sufficiently flexible and easy to adapt to support new systems quickly. To achieve these objectives, the architecture of AdCIM follows the model-driven paradigm [41], which advocates the design of applications originating from an initial model, which is later transformed into several "artifacts", like code, documentation, database schemata, and others which comprise the application. In AdCIM's case, the model is CIM, and the transformation is achieved using the eXtensible Sheet Language Transformations (XSLT), which is declarative and capable of expressing transformations of XML data. AdCIM performs all its XML transformations with XSLT, except the initial conversion of flat text files to XML, done with a specialized text-to-XML parser. XSLT programs, also called stylesheets, match and transform specific parts of the input XML tree, and support recursive execution, conforming a very flexible functional-declarative programming model.

The model chosen to represent the administration domains covered by the framework is CIM (Common Information Model) [23], a standard, object-oriented, and extensible model created by the Distributed Management Task Force (DMTF). Using CIM model schemata, the multiple and disparate formats of configuration and management data are translated by AdCIM's infrastructure into CIM instances. CIM schemata also serve as the basis to generate web interface forms and other specific schemata for validation and persistence purposes.

The development of AdCIM as a model-driven framework was evolved from our previous work in [89], which extracted configuration data and stored them in a LDAP repository using Perl scripts. In our subsequent work in [22], we started to implement this model-oriented approach, and demonstrated its adaptive nature in later works, extending AdCIM for the management of Grids in [20] and Wireless Mesh Networks in [21].

This work is novel in scope and implementation, but it uses some technologies defined as standard by international organisms such as the Internet Engineering Task Force (IETF), Distributed Management Task Force (DMTF) and World Wide Web

Consortium (W3C). We see the use of these technologies as an advantage rather than a liability. Their use not only adds generality and applicability to the framework compared with hacks or ad-hoc limited purpose solutions, but also offers a number of implementations and choices that make the framework flexible and adaptable to multiple scenarios.

Despite this flexibility, we have tried to the utmost extent to define and make all the implementation aspects concrete, to stablish best-use practices, and measure the impact on performance and scalability of the choice and use of different standard technologies.

These technologies are integral to the comprehension of the framework, so they are briefly documented. Due to their transversal use, CIM and XSLT are covered in the next chapter. Other technologies, like Lightweight Directory Access Protocol (LDAP) and XForms, are covered in appropriate chapters.

Organization of the Thesis

This Thesis is organized into two parts, preceded by this introduction, and followed by a chapter with Conclusions and Future work (Chapter 10), the bibliography and a glossary, which we feel is very important in a work full of technical terms and acronyms.

The first part introduces the framework in Chapter 2, and then details its design layer by layer. The Modelling Layer, which defines schemata to represent this information and does transformations between them, comes first in Chapter 3. The System Data Layer, focused on the extraction of configuration information, is detailed next in Chapter 4, and the Data Persistence Layer, which stores and preserves these data, in Chapter 5. Finally, the Application Layer, concerned with user and application interfaces, is covered in Chapter 6.

A considerable part of the research and development of the framework was concerned with its adaptation to complex scenarios and use cases. The second part was motivated by this peculiarity of the work, to put it in context and understand its use and scope. It begins with a small introduction, followed by three examples of application. The first one, in Chapter 7, is an extension of the System Data Layer to support the Sendmail configuration file format. Next, an adaptation to integrate CIM information into the discovery and monitoring services of the Grid middleware

Globus is shown in Chapter 8. The Part closes with the most complex scenario in Chapter 9, an application to Wireless Mesh Networks (a special type of wireless network), which also doubles as a demonstration of the ontological reasoning capabilities of the framework.

Part I

The AdCIM Framework

Chapter 2

Framework Overview

In this chapter, we give a high-level overview of the AdCIM framework. This overview shows the framework from the layering (Section 2.1) and deployment views (Section 2.2). AdCIM uses the CIM model as representation for the configuration and management data. Data and model transformations are done using XSLT stylesheets. Both technologies are covered in Section 2.3. Finally, we compare our framework with some related works in the literature.

2.1. Framework Layers

The AdCIM framework is structured following a multi-layered organization, in which the communication between layers is accomplished using different protocols and interchange formats. Additionally, each component can be deployed among a large number of network nodes.

Figs. 2.1 and 2.2 show these two different views of the AdCIM framework: an architectural/layered view in Fig. 2.1, and a deployment view in Fig. 2.2.

The architectural view consists of four layers:

- **Modelling Layer**: This layer includes CIM schemata and instances, and transformations that involve both. It is particularly interesting, since the modelling of new domains begins with these schemata, from which the rest of artifacts are derived. This modelling and the different XML representations of CIM are detailed in Chapter 3.

- **System Data Layer**: This layer includes extraction processes that collect, interpret, collate and translate data to a CIM-XML representation (step `[A]` in Fig. 2.1). It adapts the framework to existing management data, and must be extended for new domains. In the case of text configuration files, this adaptation is done using text-to-XML grammar-based parsers. This layer is the subject of Chapter 4. As an example of a complex configuration, we also modelize the configuration of the Sendmail mail agent in Chapter 7.
- **Data Persistence Layer**: The scope of this layer is the specification of data formats and transformations for data persistence. This is a transparent layer, and requires no adaptation effort to develop new administration applications. It is covered in Chapter 5.
- **Application Layer**: It comprises the interfaces visible to the external applications and end users of the framework. For applications, AdCIM provides web services for the acquisition of XML data, and for users an XForms- and XSLT-based solution to develop full-featured, responsive web applications, as shown in Chapter 6.

In brief, AdCIM provides:

- A more space-efficient XML format to represent CIM data (named miniCIM) which is used in all data transfers.
- XSLT stylesheets to transform XML data:
  - From XML configuration and management data to miniCIM instances (step `[B]` in Fig. 2.1) (Section 4.2).
  - From miniCIM instances to LDIF, for persistence in an LDAP directory (step `[C]`) (Section 5.2.1).
  - From DSML directory query data back to miniCIM instances (step `[D]`) (Section 5.2.2).
  - From DMTF CIM-XML schema to miniCIM schema (step `[H]`, Section 3.3), and from miniCIM schema to LDAP schema (step `[G]`) (Section 5.2.3), used by LDAP directories for validation and performance purposes.
- Templates and web styles to generate XForms from arbitrary CIM class definitions (step `[E]`) (Section 6.2).

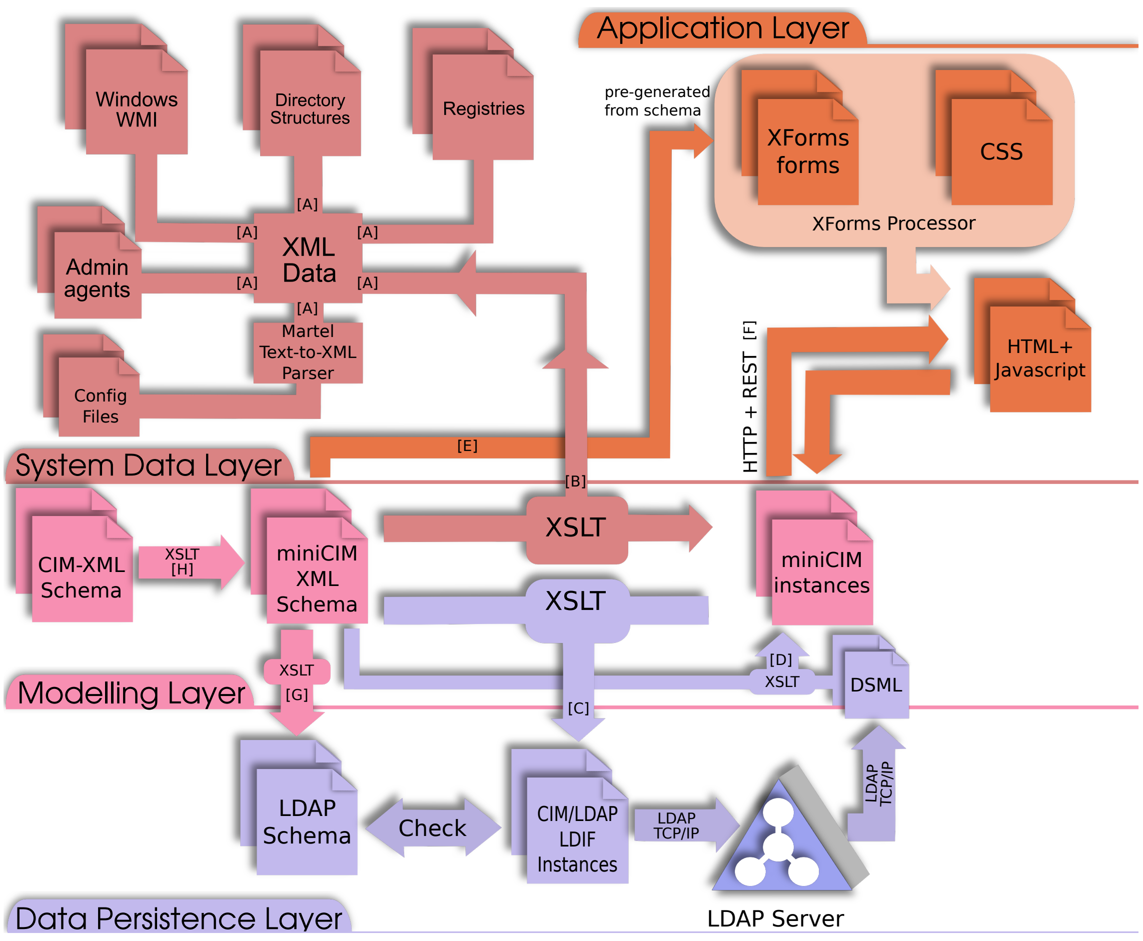


Figure 2.1: AdCIM layer diagram. Transformations shown in brackets

- A web service to serve and update miniCIM data to/from the repository (step `[F]`) (Section 6.3).

2.2. Framework Deployment

The AdCIM default deployment (Fig. 2.2) follows a three-tiered architecture (comparable to the Model-View-Controller pattern), with a system data acquisition and persistence tier which gathers the administration data of the network nodes, and persistence servers which store these data. The network nodes monitor changes and send raw data (configuration files, performance parameters, etc.) to the administration servers to minimize management overhead in the network nodes.

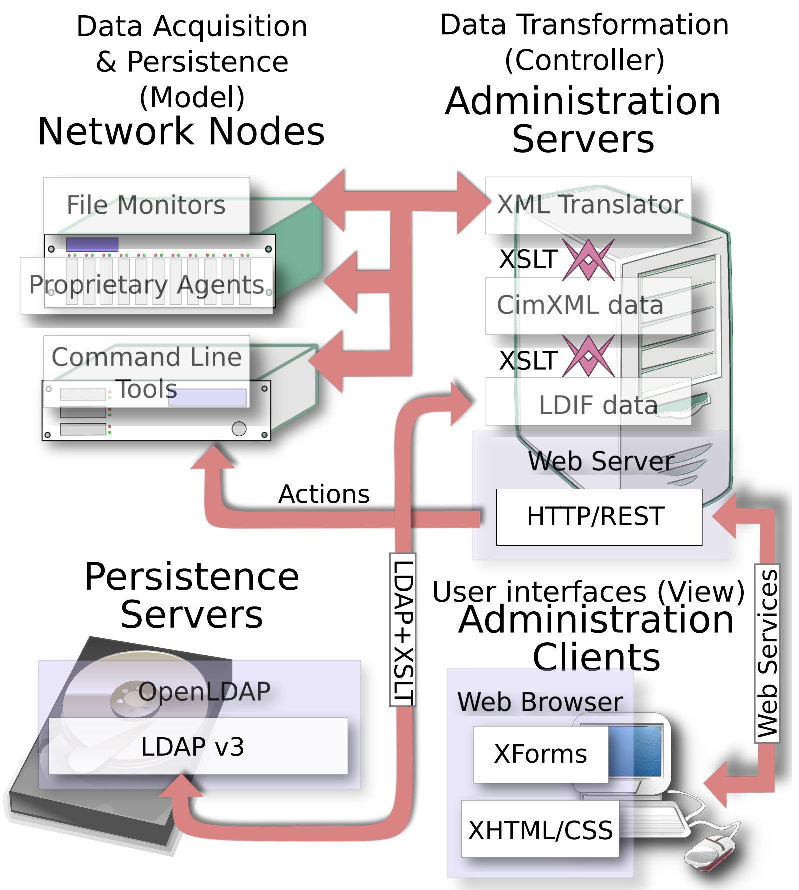


Figure 2.2: Framework default deployment view

Other kinds of deployment are supported by AdCIM; for example, management data transformations and other administration server tasks support being partitioned into several servers to balance the load, or moved to the client nodes (displacing the load to them).

The currently widespread web browser support for XSLT transformations also makes possible to offload some of the work to clients, so that they are able to directly access the persistence servers or view network node data bypassing the central repository, but generally processing the XSLT code in the server is preferable to ease the scalability and provisioning of the system – since load is aggregated and controllable – and to avoid security problems derived from the possibility of modifying the XSLT code in the client with added templates that introduce information maliciously (e.g., to activate a service with a known vulnerability).

The CIM model and its XML formats to represent CIM instances are covered in Chapter 3. The administration servers translate data to these CIM instances in a process detailed in Chapter 4. These data are stored and retrieved using LDIF and DSML in LDAP servers (Chapter 5), and later exposed by the application server transparently as XML data using a REST [30] web service interface. Administration clients use XForms [107] styled with CSS (Cascading Style Sheets) [103], running in a web browser (detailed as part of the Application Layer in Chapter 6).

2.3. Base Technologies

This section gives a technical and historical overview of the CIM model and the XSLT language, focused on the aspects useful to the AdCIM framework.

2.3.1. The CIM Model

CIM [23] is a management standard which defines an object-oriented and user-extensible information model for all management data. These data are divided into physical and logical classes and attributes. Physical classes represent tangible concepts such as location, cabling, component placing, temperature and voltage. Logical classes embody functional, abstract and intangible properties related with purpose, operation and category such as users, software packages and their dependences, administration policy and hardware capabilities. In Figure 2.3, both types of classes are shown in different colors. Both share the top of their inheritance hierarchy and are related at various levels via associations, such as `Realizes`, which relates one or several logical entity instances with physical instances that form their implementation.

CIM is organized into several divisions called “models”. There are models to represent hardware, logical devices, products, networks (with several submodels) and other domains. The main model from which others are derived is called the “Core model”. Inside this model, CIM information is represented as a class hierarchy based on inheritance with the class `ManagedElement` at the top (see Fig. 2.4). Parent classes represent broader and more general entities and children classes in the hierarchy inherit or refine properties from their ancestor, add specialized attributes and cover narrower concepts. For instance, class `CacheMemory` (from the Device model) has the properties and methods of its parent `Memory`, plus specialized properties like

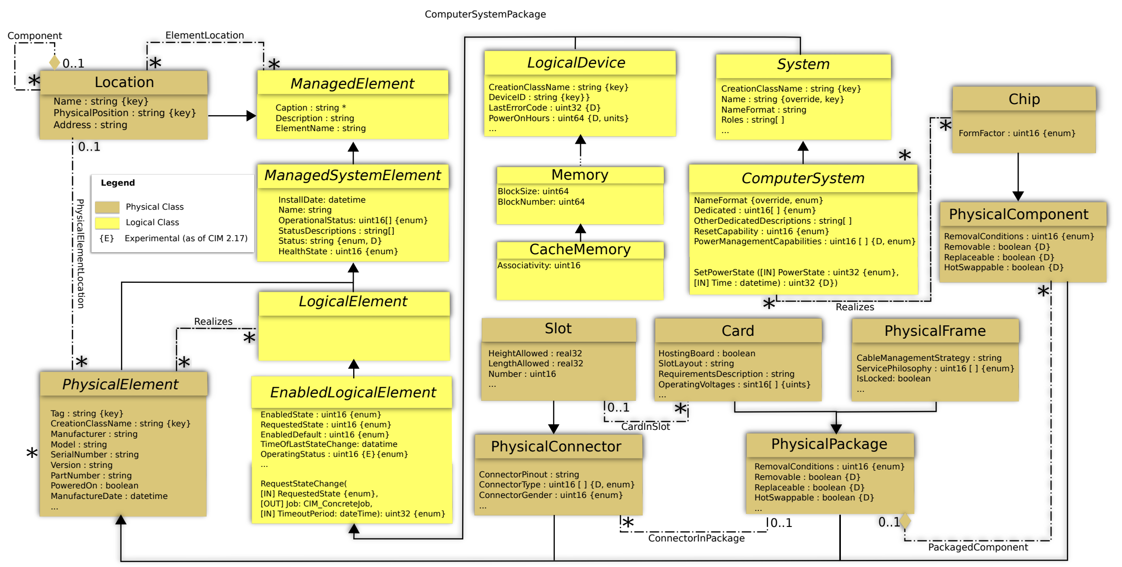

ComputerSystemPackage
Component
ElementLocation
0..1
*
Location
Name : string {key}
PhysicalPosition : string {key}
Address : string
ManagedElement
Caption : string *
Description : string
ElementName : string
PhysicalElementLocation
Legend
Physical Class
Logical Class
{E} Experimental (as of CIM 2.17)
ManagedSystemElement
InstallDate: datetime
Name: string
OperationalStatus: uint16[] {enum}
StatusDescriptions : string[]
Status: string {enum, D}
HealthState : uint16 {enum}
LogicalElement
Realizes
PhysicalElement
Tag : string {key}
CreationClassName : string {key}
Manufacturer : string
Model : string
SerialNumber : string
Version : string
PartNumber : string
PoweredOn : boolean
ManufactureDate : datetime
...
EnabledLogicalElement
EnabledState : uint16 {enum}
RequestedState : uint16 {enum}
EnabledDefault : uint16 {enum}
TimeOfLastStateChange: datatime
OperatingStatus : uint16 {E}{enum}
...
RequestStateChange(
[IN] RequestedState {enum},
[OUT] Job: CIM_ConcreteJob,
[IN] TimeoutPeriod: dateTime): uint32 {enum}
LogicalDevice
CreationClassName : string {key}
DeviceID : string {key}}
LastErrorCode : uint32 {D}
PowerOnHours : uint64 {D, units}
...
System
CreationClassName : string {key}
Name : string {override, key}
NameFormat : string
Roles : string[ ]
...
Chip
FormFactor : uint16 {enum}
Memory
BlockSize: uint64
BlockNumber: uint64
CacheMemory
Associativity: uint16
ComputerSystem
NameFormat {override, enum}
Dedicated : uint16[ ] {enum}
OtherDedicatedDescriptions : string[ ]
ResetCapability : uint16 {enum}
PowerManagementCapabilities : uint16 [ ] {D, enum}
SetPowerState ([IN] PowerState : uint32 {enum},
[IN] Time : datetime) : uint32 {D})
PhysicalComponent
RemovalConditions : uint16 {enum}
Removable : boolean {D}
Replaceable : boolean {D}
HotSwappable : boolean {D}
Realizes
Slot
HeightAllowed : real32
LengthAllowed : real32
Number : uint16
...
Card
HostingBoard : boolean
SlotLayout : string
RequirementsDescription : string
OperatingVoltages : sint16[ ] {uints}
...
PhysicalFrame
CableManagementStrategy : string
ServicePhilosophy : uint16 [ ] {enum}
IsLocked: boolean
...
CardInSlot
PhysicalConnector
ConnectorPinout : string
ConnectorType : uint16 [ ] {D, enum}
ConnectorGender : uint16 {enum}
...
PhysicalPackage
RemovalConditions : uint16 {enum}
Removable : boolean {D}
Replaceable : boolean {D}
HotSwappable : boolean {D}
...
ConnectorInPackage
PackagedComponent

Figure 2.3: CIM Logical and Physical classes

`Associativity`. Each instance of a CIM class is identified by a unique name, formed from an aggregation of the value of a set of class-designated properties, called *key* properties.

The CIM model, though, not only categorizes all these data in an object hierarchy, but it also defines an association model which relates instances of all these classes. Associations are first-class CIM classes which contain regular properties and two *referential properties*. These properties work as “pointers” to other CIM instances, referring to them by their key properties, and are usually named `Antecedent` and `Dependent`. These referential properties are always key properties, and their values must be the unique name of an instance of some given class or its descendants, called *ends* of the association. For example, the association `MemberOfCollection` (see Fig. 2.4) expresses the association between two ends: an abstract collection (`Collection`) and its components, which are instances of any class (i.e., children of `ManagedElement`).

Associations can have 1–to–1, 1–to–many and many–to–many cardinality. The 1-to-many subtype includes *weak* associations, which represent asymmetric associations where one end has no separate existence/name without the other; i.e., a file is not separable from a file system without changing its identity, as a file system is not separable from a partition. It must be noted that changing the key properties of a CIM instance (i.e., its name) means a change of its very nature, so a copy of a

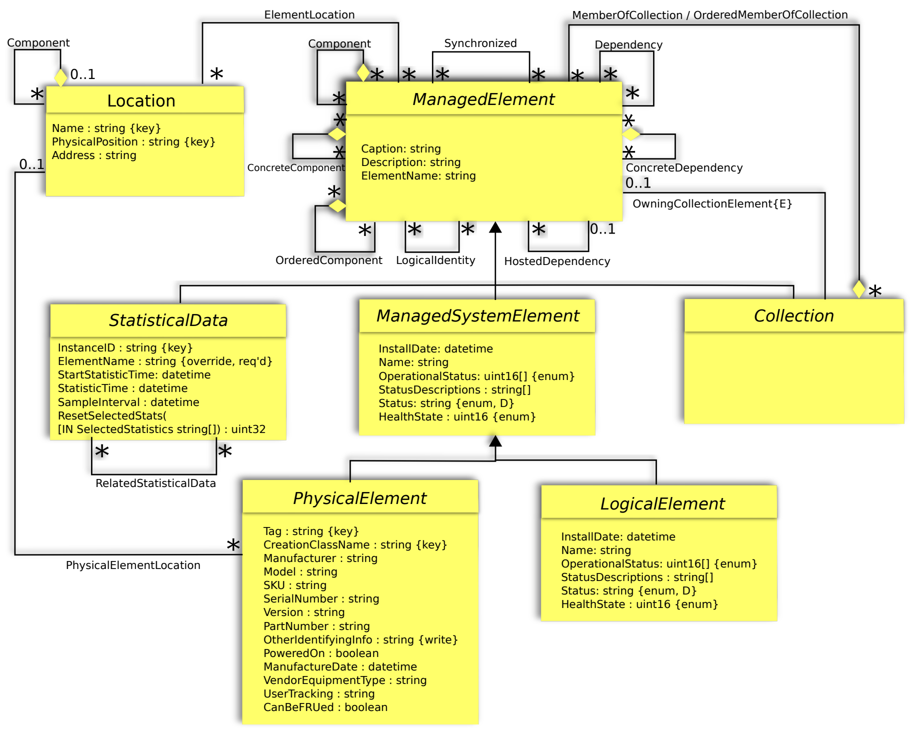


Figure 2.4: CIM Core model – top level hierarchy

file in another file system is conceptualized as another independent file.

The association model is the basic pillar of CIM: it renders CIM object trees into navigable networks of management data, enabling very powerful queries. For instance, a decrease in the throughput of a web application can be traced through associations, tracing from the web server that hosts the application, via a file system in a Redundant Array of Independent Disks (RAID), to one of the disks associated with the RAID with a mechanical failure condition as the root cause of the problem. For illustrative purposes, Fig. 3.2 of the next chapter shows part of the hierarchy of the CIM classes representing system services and network interfaces. Specifically, `InetdService` represents Inetd Unix services, which are used as case study throughout the Thesis.

CIM constructs

The representation of the CIM model is based on several constructs. Among these are the just exposed *instances*, *properties* and *associations* and, additionally,

Type	Description
uint8	Unsigned 8-bit integer.
sint8	Signed 8-bit integer.
uint16	Unsigned 16-bit integer.
sint16	Signed 16-bit integer.
uint32	Unsigned 32-bit integer.
sint32	Signed 32-bit integer.
uint64	Unsigned 64-bit integer.
sint64	Signed 64-bit integer.
char16	UCS-2 character.
string	UCS-2 string.
boolean	Boolean.
real32	IEEE 4-byte floating point.
real64	IEEE 8-byte floating point.
datetime	A string containing a date.
reference	A reference to another instance.

Table 2.1: CIM property types

qualifiers.

Instances All non-abstract classes can be instantiated. The identity and name of their instances is determined by the value of its key properties, and, in the case of instances of a class subject to a weak relationship, the instance must also include as its own the key properties of the other end of the association.

Properties Properties can have simple types, such as integer, boolean and string (shown in Table 2.1), and the reference type, which represents pointers to other instances. They can also be defined as arrays of simple types. Optionality and cardinality can also be specified with qualifiers, and subclasses can override their definition inherited from parent classes and even redefine their meaning using the `Override` qualifier. This mechanism is part of the CIM model design principle of refining abstract concepts to more concrete representations.

Associations Associations are represented as children classes of the `Dependency` class instead of `ManagedElement`. Associations also contain referential properties which point to other instances called association ends. These ends are related in a concrete way by the association without needing to be modified, so that the association of two instances is represented independently of the identity and intrinsic properties of the end of the association. These referential properties are also key properties, so usually only normal properties can be modified without losing identity. The main classification of associations used in this Thesis is by the cardinality of their ends and their use of non-referential properties:

1-to-1. 1-to-1 associations have end with a fixed cardinality of 1. They are usually meant to represent identity associations.

1-to-many. 1-to-many associations relate one end of cardinality 1 with an unlimited number of instances in the other end, optionally with non-referential properties. A subset of this group are the weak associations, which model entities subsumed into another (i.e., like files in a filesystem). Usually, 1-to-many associations model formal inclusion, containment and connectivity relationships (i.e., producer-consumer relationships).

In Fig. 2.5, a 1-to-many association is shown. In this case, it models a physical location. Semantically, a physical entity can not occupy two different places, so limiting the cardinality of the relationship in the location end is coherent with physical world configurations.

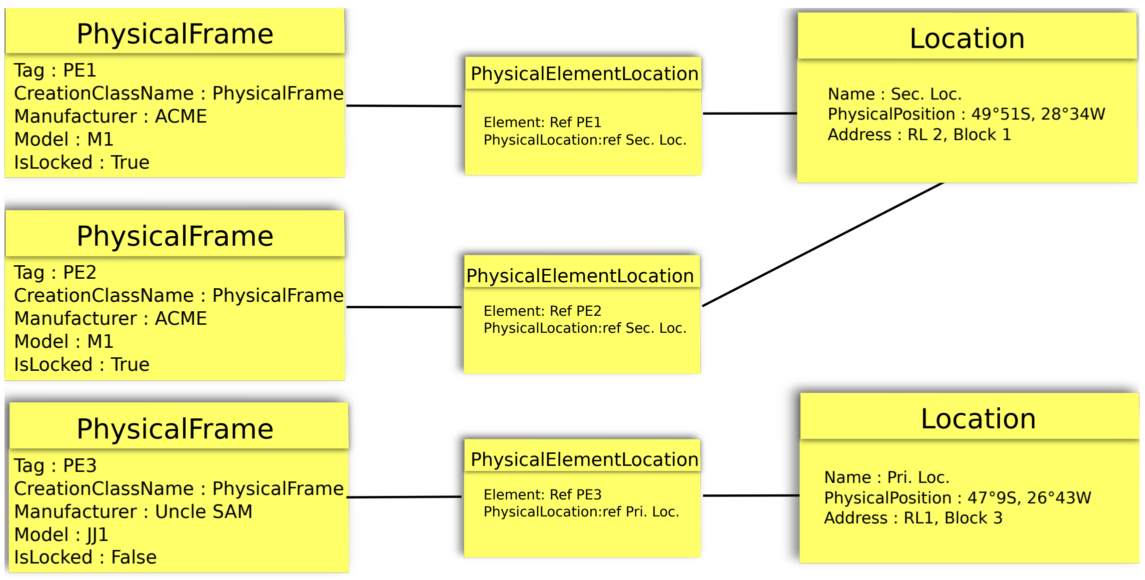


Figure 2.5: 1-to-many CIM association with properties

Many-to-many. Many-to-many associations have no cardinality limitations on either end, so the representation and attribute partition is the same for both. In Fig 2.6, a many-to-many `Synchronized` association is shown. Since this relationship is symmetric, and each end can have any number of associated synchronized elements, it is natural to represent it and peer-to-peer relationships with many-to-many associations.

Qualifiers Qualifiers are modifiers which control the expression of other CIM constructs. They have a name, type (like properties), flavor, and a scope which determines the type of construct affected. Some qualifiers are exclusive to classes, others to properties and some are applicable to all constructs. Class qualifiers control, for example, abstractness; property qualifiers control cardinality, overriding and optionality, and there are also those applicable to all constructs, such as `Description`. The flavor determines if the qualifier can be overridden in subclasses, whether is inherited by them or not, or if it is restricted only to that class. Table 2.2 shows some of the most often used qualifiers, their scope and purpose.

2.3.2. XSLT

eXtensible Sheet Language Transformations (XSLT) is a base technology for AdCIM since it is used for all model and data transformations. In this section, we

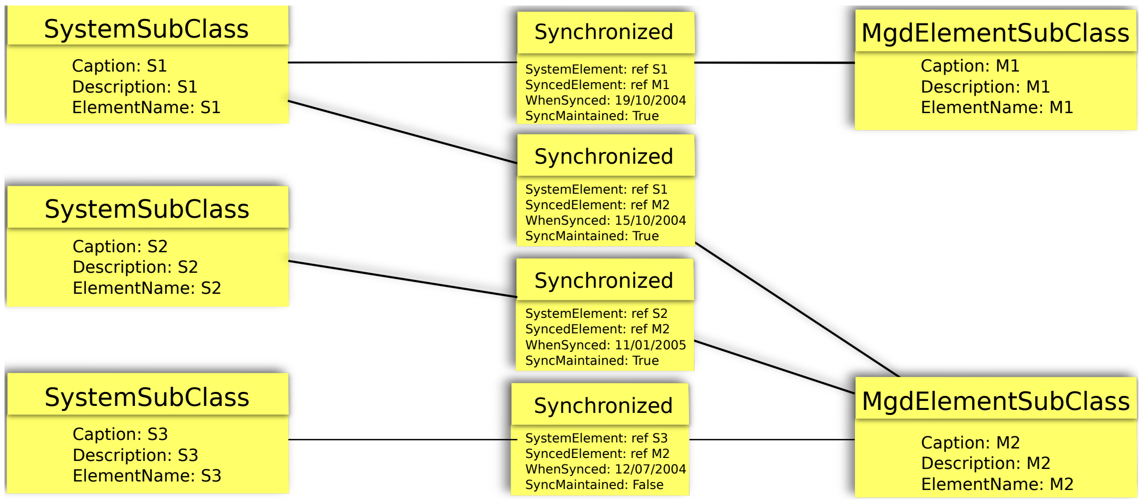


Figure 2.6: Many-to-many CIM association with properties

Name	Scope	Description
Abstract	Class, Association	Determines if the construct is instantiable.
Description	Any	A natural language description of the construct.
Deprecated	Class, Property	Superseded construct. Not recommended for new developments.
Displayname	Property	The name used in user interfaces.
Key	Property	Defines a property as a key property used for naming.
Min and Max	Property	Define the minimum and maximum cardinality of the property.
Override	Property	Allows to change the format or semantics of an inherited property.
Propagated	Property	Weak-association inherited properties are implicitly assumed in subclasses.
Required	Property	The property must have a non-null value.
ValueMap	Property	Determines the valid values of a property.
Values	Property	Maps the allowed values of a property to fixed strings.
Version	Class, Association, Schema	Versioning information.

Table 2.2: Common CIM qualifiers

cover its conception, purpose, characteristics and implementations.

History and conception

The genesis of XSLT is a direct result of the evolution and inheritance of XML, derived from the Standard Generalized Markup Language (SGML) [46] W3C [102] standard for the specification of documents. SGML was also a tag-based metalanguage which decoupled the document representation from the processes applied to it.

Document Style Semantics and Specification Language (DSSSL) [45] was developed to specify in a device-independent way the representation of SGML documents, specially for typographical and print applications. DSSSL supported the assignment of styles depending of context and semantic purpose and included a transformation engine to support the reordering of the document (i.e. to support tables). It was declarative since it used and was based on the syntax of Scheme Lisp.

With the advent of XML as a web-oriented simplification of the markup specification language SGML, soon a DSSSL-equivalent was needed. XSL [104] was proposed by the W3C to fill this niche. It was also composed of a transformation and a formatting language, but since most transformations were to HTML or XML, the first implementations only covered the transformation language. In time, this part of the specification was proven to be good enough for web formatting and later for XML transformation. This separation became the norm, and XSL was split into XSLT (the transformation part) and XSL-FO (the formatting part), used mostly for print applications. This separation was still palpable in XSLT 1.0, in which variables with XML output were called *fragment result trees*, which were to be later formatted by the XSL-FO part.

XSLT defined a sublanguage to specify nodes in an XML tree which verified some conditions. This language was fused with the then in-development XPointer specifications and became XPath 1.0 [12]. The use and scope of XPath for XML can be compared with that of Structured Query Language (SQL) for relational databases.

XSLT is used, among other things, to publish XML data as HTML in the web. Dunkel et al. [26] compare XSLT favorably with Java Server Pages (JSP), especially in its model-view separation and that XSLT is not restricted to pure web environments.

XSLT 2.0 was released much later, and supposed a dramatic change, specially for its adoption of XPath 2.0, which introduced control structures, sequence evaluation and a new type system. This new paradigm made XSLT development more complex, which was already considered intimidating by traditional programmers. In this Thesis we have not adopted XSLT 2.0 as core component since there are still very few implementations to date.

Characteristics

Due to its conception and design, XSLT has characteristics that set it apart from most programming languages. Most of these are related to its declarative nature and purpose:

- XML syntax. The most obvious characteristic is that XSLT uses an XML syntax. Although it makes XSLT programs much more verbose, the XML syntax was justified to reduce syntactic mismatch with XML, reuse XML parsers and employ XSLT stylesheets as inputs of other stylesheets, or *metatemplating*. In Section 6.2, a form of this technique is used.

- Rule-based. An XSLT program is organized around templates, which are rule-based. A template is a program fragment which describes how an expected input is transformed. Templates do not need to be ordered and can implicitly call each other, by recursing and pattern-matching over an XML tree. This means that the execution order is not predefined and depends on the input and the posterior addition of new templates. General rules can also be overridden by more concrete or prioritary ones.

- No side-effects. Since template evaluation order can change, it is important that the evaluation of templates is side-effect free. This means that a template can be reevaluated multiple times with the same input and produce the same output without modifying the state of the program. To guarantee this property, variable values can not be changed after the initial assignment. Instead of traditional modifiable variables, tail recursion and pattern-matching can be used to obtain the same results.

The elimination of side-effects opens the door to a series of mathematical optimizations of the code and promotes parallelism. It also avoids the need to recalculate the output of a program from scratch when only part of the input is changed.

In Table 2.3, most XSLT elements and their functions are shown. These elements are frequently used in the XSLT stylesheet listings in the rest of this Thesis.

Transformation example

In Fig. 2.7, a small XSLT stylesheet is shown. It begins in line 1 with the XML preamble, and lines 2–13 are the stylesheet, inside an `<xsl:stylesheet/>` element. Line 3 establishes the output of the document as an XML file – this creates a valid preamble in the result, text output is also supported – and line 4 declares all non-significant spaces to be stripped of every element.

The template at lines 5–9 matches all attributes, elements and text nodes with its `match` expression. Since it matches the root node of the XML tree, it is executed

XSLT element	Description
`<xsl:apply-templates/>`	Gives control to templates matching child elements or elements pointed by the `select` attribute.
`<xsl:call-template/>`	Calls a template by name.
`<xsl:choose/>`	Multiple condition section, comparable with C `switch`.
`<xsl:for-each/>`	Does a loop across a set of XML nodes.
`<xsl:if/>`	Executes the content if the condition in the attribute `test` is true.
`<xsl:import/>`	Includes code from another stylesheet file.
`<xsl:otherwise/>`	Executed in `<xsl:choose/>` if no `<xsl:when/>` was taken.
`<xsl:output/>`	Changes output file.
`<xsl:param/>`	Defines a stylesheet or template parameter.
`<xsl:stylesheet/>`	Declares a stylesheet, and works as the root node.
`<xsl:template/>`	Defines a template either by the matched expression or by name.
`<xsl:text/>`	Content is copied verbatim to the output.
`<xsl:value-of/>`	Evaluates and prints its `select` attribute.
`<xsl:variable/>`	Defines a variable with its value either in the `select` attribute or inside the element.
`<xsl:when/>`	In an `<xsl:choose/>` marks a branch taken if its `test` attribute is true.
`<xsl:with-param/>`	Inside `<xsl:call-template/>` identifies a passed parameter and its value.

Table 2.3: XSLT elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml"/>
    <xsl:strip-space elements="*"/>
    <xsl:template match="@*|node()">
        <xsl:copy>
            <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="elementA[@flavour='xyzzy']">
    </xsl:template>
</xsl:stylesheet>
```

Figure 2.7: Example of an XSLT stylesheet

first. The `<xsl:copy/>` element denotes a shallow copy, which is applied to all XML nodes and attributes, following the `select` attribute of the `<xsl:apply-templates/>` element.

The “shallow copy” term means that all matching children input nodes and attributes are copied to the output, and the first template is called for each of them, performing a recursive copy. Nevertheless, the second template (lines 11-12) matches all elements in the input named `elementA` with the condition (marked by brackets) of having a `flavor` attribute with the *xyzzy* value. By the matching rules in XSLT, templates with narrower matching expressions take priority (since they usually cover special cases), and the action inside the template is taken. Since the element is empty and does not indicate any action, the matching elements are simply discarded from the output. Thus, the final effect of the stylesheet is eliminating the elements matched by the second template.

It must be noted that the stylesheet does not directly control program flow. If a new template were to be added, the result and evaluation order could be completely changed. Also, the declarative style means that the stylesheet is centered on describing possible inputs and desired outputs, rather than explicitly controlling iteration and covering cases. This coding paradigm needs markedly different mindset and coding practices.

Support

XSLT 1.0 is supported in a large number of computing platforms and devices. Since it is implemented in Java, it works on the wide range of platforms supported by the Java Runtime. Among these implementations, Saxon [48] and Xalan XSLTC [5] are the most frequently used. Saxon uses a highly optimized Document Object Model (DOM)-like tree representation, and XSLTC precompiles stylesheets into native Java code. There is also the Intel XSLT Accelerator (also known as PaNaPa), which automatically parallelizes XSLT code and supports multicore systems [98].

There are also hardware implementations, like the one in [18], that uses dedicated network hardware that can transparently filter XML traffic in the network at wire speed.

2.4. Related Work

AdCIM is a novel approach in the field of configuration management, and not a derivative work or application of an existing system. Of course, there are common points with other tools and frameworks, since they have similar purposes. Therefore, we approach the related literature in a piecemeal fashion, covering concrete areas of our framework with which to compare with other works that by goal, scope or design are divergent in other aspects.

In this section, we begin with general categorizations. In later chapters, works related with more specific topics are cited in the appropriate sections.

The use of CIM

Since CIM is a well-established standard there has been copious literature – though it must not be confused with the one about the also popular International Electrotechnical Commission (IEC) CIM [44] (Common Information Model) standard of the electric power industry to modelize electric transmission. Here we are specially interested about the works touching CIM's objective of integrating management information and the reduction of data fragmentation and "silos".

This increasing need of integration has been explored in the work by Schott et al. [92], which advocates CIM to avoid "information overload". There are also authors that propose the use of CIM as an integral part of the development of management applications. Mehl et al. [58] propose that application and system developers integrate a "management infrastructure" based on CIM in the normal development process of the applications. They explicitly discourage the development of this infrastructure by third parties, which has the effect of ultimately shutting off system administrators from management application development. Of course, the costs of this development activity for the system administrators can be insurmountable for most organizations and it is part of what this work endeavours to avoid.

Formalization of configuration

The integration of management data, the verification of the validity of configurations, and the reasoning based on ontologies are high-level processes, but these data originate in most cases from low-level, application-dependent formats. Thus, there is a need to bridge this gap.

There are several works that modelize or formalize existing configurations. Sinz et al. [93] model the Apache configuration file to perform the formal testing of the correctness of configurations. A later work, by Post and Sinz [83], can verify the correctness of whole configuration models (in this case, the Linux kernel configuration), by translating them to logic programs. Dolstra and Löh [25] prove the feasibility of a purely functional (non-imperative) configuration system by making an entire functionally-configured Linux distribution, with immutable configuration files which are evaluated lazily, and with the possibility of retrieving any past configuration state. Nevertheless, this distribution is more experimental than production-oriented, and using its capabilities would mean to install it in all nodes, compared with the platform-independence of AdCIM.

Configuration workflows

Configuration changes can be the result of manual system administrator intervention, or a higher-level process like the ones mentioned above. In both cases, the modification process can be complex when macro languages, server restart, validation and data transfer are involved, specially when these tasks are distributed among several machines.

Some works try to support and enhance traditional configuration workflows: for example, Bajohr and Margaria [6] automate the modification and deployment of configuration files on complex systems so the configuration process can be abstracted, streamlined and organized as a workflow for remote operators. The system developed by Su et al. [97] can speculate about changes in configuration and make causal analysis of the changes made by an administrator, but they do not represent and integrate these changes in an extensible management framework like AdCIM.

Transformations in system management

There are many works in system management that transform data: for adaptation of legacy systems, between systems with different configuration formats or for expediency, if there is a format that is more convenient or easier to modify. The transformation of management data to XML is seen as particularly useful, because of the tool support and interchange potential. This is covered, for example, by Strauss and Klie [96], and Yoon et al. [110]. Both works translate Simple Network Management Protocol (SNMP) data to XML, but the structure is translated as is from SNMP MIBs, without the integration approach that AdCIM provides. SNMP has a very flat structure that does not represent aspects like associations as flexibly as CIM.

XSLT transformations concerning translations between general models have been explored by Peltier [82], which generates XSLT stylesheets from a high-level language.

Large scale administration frameworks

These frameworks are geared toward systems that are homogeneous by design and hence not directly comparable with AdCIM, but have common points, like the abstract representation of network nodes and the manipulation of configuration files.

LCFG [3] and Cfengine [10], for example, are based on templating solutions. The former is oriented towards initial installation of network nodes using a template language, and the latter towards simplifying and replicating common administration operations across several network nodes. Both are limited frameworks that address a fixed set of operations which can not be easily expanded. Thus, LCFG is only capable of installing or updating network nodes but, for instance, it can not extract configuration information from the administered nodes like AdCIM does.

SmartFrog [40] is a Java-based framework for distributed configuration management that supports user extension via a component model and a runtime which controls the life cycle of the components. Its language avoids the duplication usual in configuration files, but it is still a flat file mapping, so it is not useful to derive ontologies and detect misconfigurations preemptively like AdCIM.

The language used by bcfg2 [19] is more advanced than the one in SmartFrog, and it is based on clauses for installing packages, modifying configuration files, and creating directories and symlinks. It also allows merging configurations with manual changes, but it does not represent the configuration state or detect misconfigurations distributed among several preexisting configured nodes as our framework does.

Chapter 3

Modelling Layer

This chapter describes the Modelling Layer, which comprises the adaptation of the CIM model to management domains and the XML representations of CIM used in the framework. Figure 3.1 shows an overview of this layer. In the left, we can see a CIM-XML schema that can be provided by the DMTF as part of the domains normally covered by CIM, or added as an extension. From this schema, an XSLT transformation derives a special XML Schema that defines our miniCIM format, used by the remaining layers in AdCIM, and validates for syntactical or low-level semantic errors (such as range or cardinality).

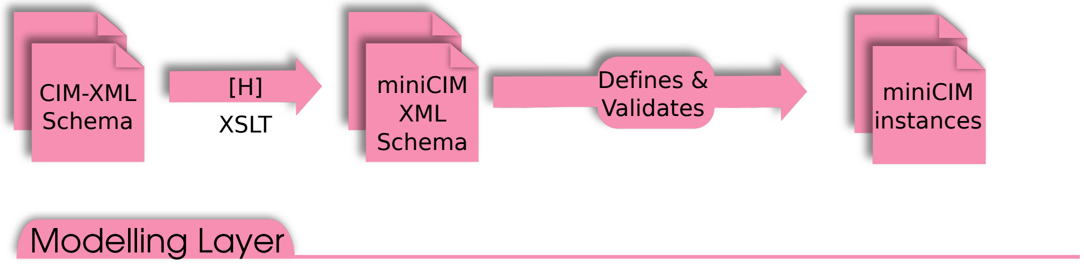


Figure 3.1: AdCIM Modelling Layer

3.1. Application and Extension of the CIM Model

The CIM model covers a vast range of management data but, naturally, not all possible manageable entities. Thus CIM is fully extensible through inheritance: a new concept is derived from a class adding new properties or constraints. For example, in Fig. 2.3, the `ComputerSystem` class adds information about power management, role and nomenclature to the `System` class. In rare cases, the new entity or concept is not assimilable to an existing class; then, an extension of the top class `ManagedElement` is needed.

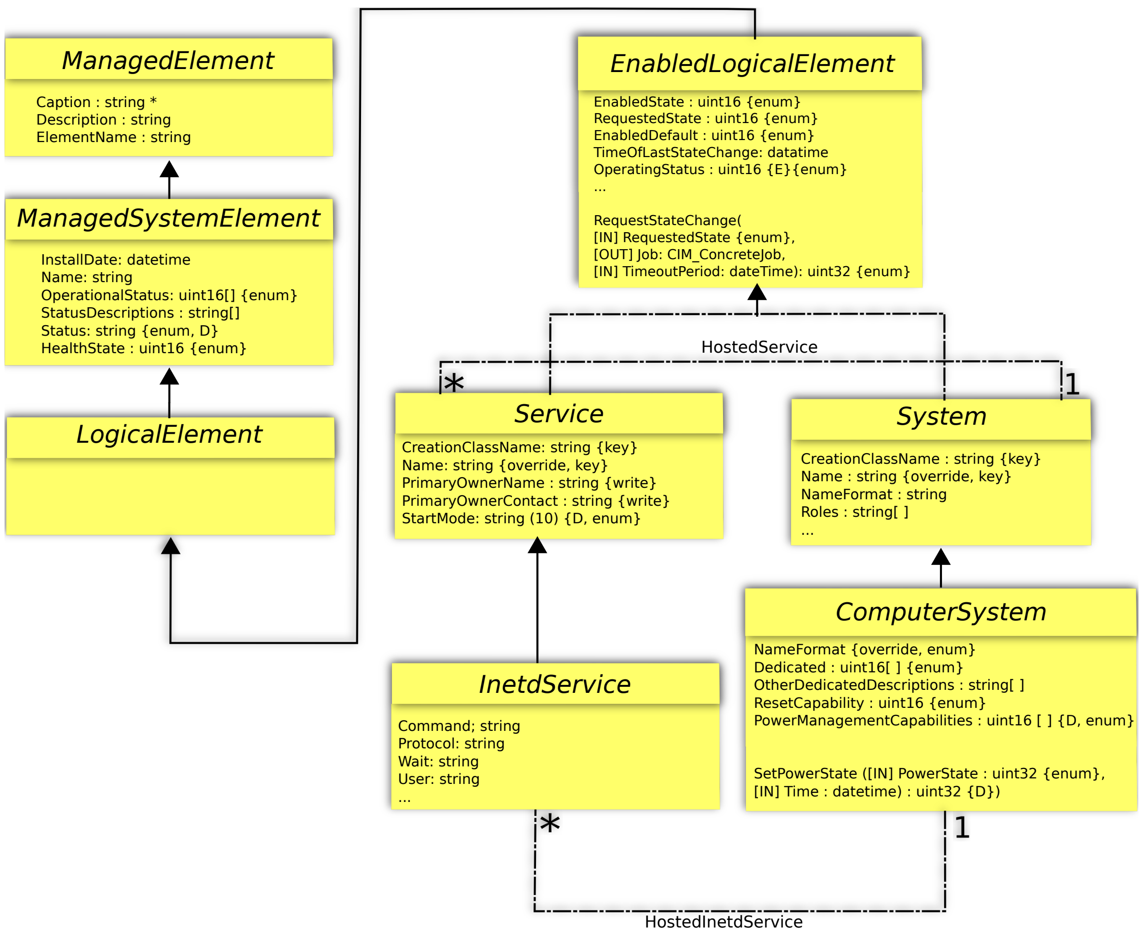

ManagedElement
Caption : string *
Description : string
ElementName : string
ManagedSystemElement
InstallDate: datetime
Name: string
OperationalStatus: uint16[] {enum}
StatusDescriptions : string[]
Status: string {enum, D}
HealthState : uint16 {enum}
LogicalElement
EnabledLogicalElement
EnabledState : uint16 {enum}
RequestedState : uint16 {enum}
EnabledDefault : uint16 {enum}
TimeOfLastStateChange: datatime
OperatingStatus : uint16 {E}{enum}
...
RequestStateChange(
[IN] RequestedState {enum},
[OUT] Job: CIM_ConcreteJob,
[IN] TimeoutPeriod: dateTime): uint32 {enum}
HostedService
*
1
Service
CreationClassName: string {key}
Name: string {override, key}
PrimaryOwnerName : string {write}
PrimaryOwnerContact : string {write}
StartMode: string (10) {D, enum}
System
CreationClassName : string {key}
Name : string {override, key}
NameFormat : string
Roles : string[ ]
...
ComputerSystem
NameFormat {override, enum}
Dedicated : uint16[ ] {enum}
OtherDedicatedDescriptions : string[ ]
ResetCapability : uint16 {enum}
PowerManagementCapabilities : uint16 [ ] {D, enum}
SetPowerState ([IN] PowerState : uint32 {enum},
[IN] Time : datetime) : uint32 {D})
InetdService
Command; string
Protocol: string
Wait: string
User: string
...
*
1
HostedInetdService

Figure 3.2: InetdService place in CIM hierarchy

Descendant classes are not merely syntactic sugar; they can substitute parent classes, specially as association ends. The benefits of this substitution are twofold: first, it gives a simplified, abstract view of instances; and second, it eases the categorization and query of objects. For example, CIM can represent detailed hardware information of all physical interfaces in a network, being also able to abstract them as generic network adapters without data duplication.

An example of extension which is used as case study in the subsequent chapters is illustrated with the class `InetdService` in Fig. 3.2. This class is a descendant of `Service`, the CIM class that describes generic services with high-level properties (e.g., a service started state). `Service` also encapsulates more abstract properties from its parent `LogicalElement`, and properties derived from the weak association `Hosted-InetdService` with the `System` class (such as `SystemName`). `InetdService` inherits all these properties and includes additional properties of interest, such as packet protocol, process owner and command line parameters. The CIM model is designed to be extensible, but this extension must be designed to represent the domain closely. There are many scenarios, the easiest ones the modelling of specific vendor models or a small extension of preexisting classes. In these cases, the extension can be done with a subclass that adds some vendor-specific or additional attributes.

A harder category of problems requires the creation of new associations. They can be accomplished with subclasses of preexisting associations or with totally new classes. Care must be taken to limit the classes that can be related by the association if it is not representing a generic relationship.

The hardest cases are the modelling of whole new domains, with their own set of classes and associations. In this case, the classes used should be subclasses of the most concrete CIM class possible. This ensures that the resulting classes are integrated in the CIM model and honor abstract relationships, specially `Realizes`. At most, a derived class should be child of `EnabledLogicalElement` (see Fig. 2.3) for logical entities (since it supports attributes to indicate if a logical device is activated), or `PhysicalComponent` (see also Fig. 2.3) for physical entities (since it supports the notion of packaging). In Fig. 7.2 of Chapter 7 and Fig. 9.4 of Chapter 9, CIM extensions for concrete domains are explained (the Sendmail mail agent configuration and Wireless Mesh Networks, respectively). In Chapter 7, all new classes are derived from the `SettingData` class, representing generic settings. The domain of Chapter 9 also derives from this class, but it also uses at their fullest preexisting classes, modelling network entities like `WirelessPort` or `SwitchService`.

3.2. XML Representations of CIM

Inside our framework, exchanges of CIM data employ an XML representation transformable with XSLT which enables web service technology and XForms to directly read CIM data, as will be seen in Chapter 6. Nevertheless, there are many possible XML representations.

CIM-XML [24] is the official DMTF representation of CIM in XML, devised to represent CIM schemata and instances. CIM-XML uses very long uppercase names

CIM-XML	miniCIM XML Schema	miniCIM format
`/CIM/DECLARATION-/DECLGROUP-/VALUE.OBJECT-/CLASS with name aClass`	`<xsd:complextype name="aClass"/> and <xsd:element name="aClass" type="aClass"/>`	`<aClass/>`
`SUPERCLASS="sClass"`	`<xsd:complexContent> <xsd:extension base="sClass"/> </xsd:complexContent>`	implicit
`PROPERTY with name aProperty and type aType`	`<xsd:element name="aProperty type ="aType"/>`	`<aClass> <aProperty/> </aClass>`
`REFERENCECLASS`	`<xsd:attribute name="adc:range"/>`	`<Antecedent adc:range = "aClass"/>`
`<QUALIFIER NAME="Abstract"/>`	`<xsd:attribute name="abstract"/>`	`<aClass abstract = "true"/>`
`<QUALIFIER NAME="Key"/>`	`<xsd:key name="aClass"> <xsd:selector xpath="aProperty"/> </xsd:key>`	implicit
`<QUALIFIER NAME="Min"/>`	`xsd:minOccurs`	implicit
`<QUALIFIER NAME="Max"/>`	`xsd:maxOccurs`	implicit
`<QUALIFIER NAME="Required"/>`	`xsd:minOccurs=1`	implicit
`<QUALIFIER NAME="Values"/>`	`<xsd:restriction> <xsd:enumeration value="aValue"/> </xsd:restriction>`	implicit

Table 3.1: Mapping of CIM-XML Schema to miniCIM XML Schema

and convoluted naming schemes; hence, it is very cumbersome to use for representing instances. To shorten this representation some information can be merged from an external schema file, but there are unavoidable overheads; e.g., key properties must be present twice, in the naming part of the instance and in its properties. As a consequence, a simple declaration of the network services of a managed node in CIM-XML takes up 305Kb. Our XML representation of CIM, called miniCIM, reduces

```
<AdCIM_InetdService namespace="dc=udc">
    <SystemCreationClassName>CIM_ComputerSystem</SystemCreationClassName>
    <SystemName>shalmaneser</SystemName>
    <CreationClassName>AdCIM_InetdService</CreationClassName>
    <Name>ftp</Name>
    <SocketType>stream</SocketType>
    <Protocol>tcp</Protocol>
    <Wait>nowait</Wait>
    <User>root</User>
    <Command>root/usr/sbin/tcpd</Command>
</AdCIM_InetdService>
```

Figure 3.3: Example of miniCIM Inetd service instance

this size to 3Kb, by removing redundant schema and key information. The miniCIM schema is constructed by an AdCIM stylesheet from the original DMTF CIM-XML schema (step `[H]` in Figs. 2.1 and 3.1). This stylesheet (covered in more detail in the next section) implements the transformations shown in Table 3.1. The left column shows CIM-XML concepts, the middle one its translation to XML Schema, and the rightmost one shows an example of the concept in miniCIM format, in which "implicit" means that this concept is controlled by the schema and does not need to be explicitly indicated in instances. The miniCIM format is used for the rest of stylesheets as input and output. In Fig. 3.3 we can see a miniCIM instance of the `InetdService` class seen in Fig. 3.2.

The schema language chosen to validate miniCIM instances is XML Schema, the first schema format standardized by the W3C for XML. XML Schema [108] has a very extensive set of features, like type inheritance, abstract classes and keys. This makes it unwieldy for simple tasks, but very suited to represent CIM instances in XML. These features offload much of the complexity of a representation of CIM as XML in the schema. Other schema languages, like RELAX NG, were not used because they are designed to represent XML document structure rather than data typing and inheritance (see Lee and Chu [52]). An excerpt of an XML Schema describing the miniCIM representation of the class `InetdService` and its parent class `Service` is shown in Fig. 3.4; miniCIM instances are composed of property-value pairs, and semantic information is confined to the schema. The format of the miniCIM instances validated with this schema can be seen in Fig. 3.5. It shows the association class `HostedInetdService`, derived from the `HostedService` association of Fig. 3.2, representing Inetd network services belonging to a computer system, and a declaration of its key properties. Invalid or key-less instances, or associations pointing to a non-extant instance, are detected by an XML validator.

The use by AdCIM of XSLT code to obtain a miniCIM XML Schema from the CIM-XML schema greatly eases the transformation problem: to create the miniCIM

```
1  <xsd:complexType name="InetdService">
2   <xsd:complexContent>
3    <xsd:extension base="Service">
4     <xsd:sequence>
5      <xsd:element name="SocketType" type="xsd:string"/>
6      <xsd:element name="Protocol" minOccurs="1" type="xsd:string"/>
7      <xsd:element name="Wait" type="xsd:string"/>
8      <xsd:element name="WaitInstance" type="xsd:string"/>
9      <xsd:element name="User" type="xsd:string"/>
10     <xsd:element name="Command" type="xsd:string"/>
11    </xsd:sequence>
12   </xsd:extension>
13  </xsd:complexContent>
14 </xsd:complexType>
15 <xsd:complexType name="Service" abstract="true">
16  <xsd:complexContent>
17   <xsd:extension base="EnabledLogicalElement">
18    <xsd:sequence>
19     <xsd:element name="SystemCreationClassName" minOccurs="1" type="xsd:string"/>
20     <xsd:element name="SystemName" minOccurs="1" type="xsd:string"/>
21     <xsd:element name="CreationClassName" minOccurs="1" type="xsd:string"/>
22     <xsd:element name="Name" minOccurs="1" type="xsd:string"/>
23     <xsd:element name="StartMode" type="xsd:string"/>
24     <xsd:element name="Started" type="xsd:string"/>
25    </xsd:sequence>
26   </xsd:extension>
27  </xsd:complexContent>
28 </xsd:complexType>
29 <xsd:key name="InetdService">
30  <xsd:selector xpath="InetdService"/>
31  <xsd:field xpath="SystemCreationClassName"/>
32  <xsd:field xpath="SystemName"/>
33  <xsd:field xpath="CreationClassName"/>
34  <xsd:field xpath="Protocol"/>
35  <xsd:field xpath="Name"/>
36 </xsd:key>
```

Figure 3.4: miniCIM XML Schema defining Inetd services

schema, CIM-XML schema classes are inspected for abstractness, parent class and association status. Then CIM-XML concepts are translated to XML Schema constructs following Table 3.1. Classes are translated as a complex type (lines 15–28 in Fig. 3.4) which represents their possible attributes. Superclass information is defined using type extensions, so it is implicit, as opposed to abstractness, which is represented with an attribute (like in line 15). Properties are children elements of the class elements with a normal or restricted type, and their qualifiers `Min`, `Max` and `Required` are mapped as schema cardinality restrictions (see lines 5–10 and 19–24). Simple data types are translated directly to their XML Schema equivalents, and enumerations based on the `Values` qualifier are defined as restrictions of a string base type — later, in Section 6.2, these enumeration values, which represent the possible string values of a CIM property, are used to fill a selection box in a user

```
<HostedInetdService namespace="dc=udc">
 <StartMode>Automatic</StartMode>
 <Antecedent>
  <ref classname="ComputerSystem" namespace="dc=udc">
   <CreationClassName>ComputerSystem</CreationClassName>
   <Name>shalmaneser</Name>
  </ref>
 </Antecedent>
 <Dependent>
  <ref classname="InetdService" namespace="dc=udc">
   <SystemCreationClassName>ComputerSystem</SystemCreationClassName>
   <SystemName>shalmaneser</SystemName>
   <CreationClassName>InetdService</CreationClassName>
   <Name>ftp</Name>
   <Protocol>tcp</Protocol>
  </ref>
 </Dependent>
</HostedInetdService>
```

Figure 3.5: Inetd service association instance expressed in miniCIM (`HostedInetd-Service`)

web interface to both control the possible values that the user can input and also to give hints of these available values.

The `Key` qualifier is translated by the AdCIM schema mapping stylesheet to the XML Schema `<xsd:key/>` operator (see also lines 29–36 in Fig. 3.4), which marks certain parts of an element (its key, which must be unique). To find the key properties of a CIM class, a recursive search of them in the class and in all its superclasses must be done but, since keys are defined independently, modifying key properties of a class does not change its miniCIM representation (but rather changes its CIM-XML representation).

The next section discusses the stylesheet used to reach this result.

3.3. Schema Transformation

This section details the stylesheet that translates the CIM-XML schema into an XML Schema (step `[H]` in Figs. 2.1 and 3.1), using the translation covered in Table 3.1. This XML Schema is then used to validate the miniCIM XML syntax, which keeps all semantic constraints and helps to reduce latency and transfer times in our framework.

This XSLT stylesheet is partly shown in Figs. 3.6–3.7. It begins outputting the

```
 1  <xsl:template match="/">
 2    <xsd:schema>
 3      <xsl:for-each select="/CIM/DECLARATION/DECLGROUP/VALUE.OBJECT/CLASS[not(contains(@NAME,'PRS_'))]">
 4        <xsl:for-each select="PROPERTY[QUALIFIER[@NAME='Values']] | PROPERTY.ARRAY[QUALIFIER[@NAME='Values']]">
 5          <xsd:simpleType name="{current()/ancestor::CLASS/@NAME}-{current()/@NAME}">
 6            <xsl:apply-templates select="QUALIFIER[@NAME='Values']"/>
 7          </xsd:simpleType>
 8        </xsl:for-each>
 9        <xsd:complexType name="{@NAME}">
10          <xsl:if test="QUALIFIER[@NAME='Abstract']/VALUE='true'">
11            <xsl:attribute name="abstract">true</xsl:attribute>
12          </xsl:if>
13          <xsl:if test="PROPERTY.REFERENCE">
14            <xsl:choose>
15              <xsl:when test="current()/PROPERTY | current()/PROPERTY.ARRAY and not (current()/PROPERTY.REFERENCE[
                        QUALIFIER[@NAME='Max']])">
16                <xsl:attribute name="adc:dependency">withnonref</xsl:attribute>
17              </xsl:when>
18              <xsl:otherwise>
19                <xsl:attribute name="adc:dependency">bare</xsl:attribute>
20              </xsl:otherwise>
21            </xsl:choose>
22          </xsl:if>
23          <xsl:choose>
24            <xsl:when test="@SUPERCLASS">
25              <xsd:complexContent>
26                <xsd:extension base="{@SUPERCLASS}">
27                  <xsl:apply-templates select="."/>
28                </xsd:extension>
29              </xsd:complexContent>
30            </xsl:when>
31            <xsl:otherwise>
32              <xsl:apply-templates select="."/>
33            </xsl:otherwise>
34          </xsl:choose>
35        </xsd:complexType>
36      </xsl:for-each>
37      <xsd:element name="CIM">
38        <xsd:complexType>
39          <xsd:sequence>
40            <xsl:for-each select="/CIM/DECLARATION/DECLGROUP/VALUE.OBJECT/CLASS[not(contains(@NAME,'PRS_'))]">
41              <xsl:if test="not (QUALIFIER[@NAME='Abstract'])">
42                <xsd:element name="{@NAME}" type="{@NAME}"/>
43              </xsl:if>
44            </xsl:for-each>
45          </xsd:sequence>
46        </xsd:complexType>
47        <xsl:call-template name="key"/>
48      </xsd:element>
49    </xsd:schema>
50  </xsl:template>
```

Figure 3.6: CIM-XML to miniCIM XML Schema stylesheet excerpt

root element of the schema (line 2 of Fig. 3.6) and iterating across all classes (lines 3–36), and creates complex types for each one. Inside this loop, the `abstract` attribute is set in lines 10–12, and those representing the type of association in lines 14–21. In lines 23–34, classes are declared as an extension type of their superclass.

In lines 37–48, the document is declared as having a `CIM` root element, inside of which there can be a unordered succession of arbitrary length of elements named after the classes, with their corresponding complex type.

The stylesheet continues in Fig. 3.7 with a template processing the properties of each class in line 51. The loop in lines 53–70 iterates over the non-referential ones, and assigns a minimum cardinality of 1 to required or key properties (lines 55–57),

```
51  <xsl:template match="CLASS">
52     <xsd:sequence>
53        <xsl:for-each select="PROPERTY[@CLASSORIGIN=ancestor::CLASS/@NAME] | PROPERTY.ARRAY[@CLASSORIGIN=
                ancestor::CLASS/@NAME]">
54           <xsd:element name="{@NAME}">
55              <xsl:if test="QUALIFIER[@NAME='Required' or @NAME='Key']/VALUE='true'">
56                 <xsl:attribute name="minOccurs">1</xsl:attribute>
57              </xsl:if>
58              <xsl:choose>
59                 <xsl:when test="QUALIFIER[@NAME='Values']">
60                    <xsl:attribute name="type"> <xsl:value-of select="current()/@CLASSORIGIN"/> <xsl:text>-</xsl:text>
61                    <xsl:value-of select="current()/@NAME"/> </xsl:attribute>
62                 </xsl:when>
63                 <xsl:otherwise>
64                    <xsl:attribute name="type">
65                       <xsl:apply-templates select="@TYPE"/>
66                    </xsl:attribute>
67                 </xsl:otherwise>
68              </xsl:choose>
69           </xsd:element>
70        </xsl:for-each>
71        <xsl:for-each select="PROPERTY.REFERENCE[@CLASSORIGIN=ancestor::CLASS/@NAME]">
72           <xsd:element name="{@NAME}">
73              <xsl:attribute name="adc:reference">true</xsl:attribute>
74              <xsl:if test="@REFERENCECLASS">
75                 <xsl:attribute name="adc:range">
76                    <xsl:value-of select="@REFERENCECLASS"/>
77                 </xsl:attribute>
78              </xsl:if>
79              <xsl:if test="QUALIFIER[@NAME='Override']/VALUE='Antecedent'">
80                 <xsl:attribute name="adc:antecedent">true</xsl:attribute>
81              </xsl:if>
82              <xsl:if test="QUALIFIER[@NAME='Override']/VALUE='Dependent'">
83                 <xsl:attribute name="adc:dependent">true</xsl:attribute>
84              </xsl:if>
85              <xsl:if test="QUALIFIER[@NAME='Required' or @NAME='Key']/VALUE='true' and not(QUALIFIER[@NAME='Min'])">
86                 <xsl:attribute name="minOccurs">1</xsl:attribute>
87              </xsl:if>
88              <xsl:if test="QUALIFIER[@NAME='Min']">
89                 <xsl:attribute name="minOccurs">
90                    <xsl:value-of select="QUALIFIER[@NAME='Min']/VALUE"/>
91                 </xsl:attribute>
92              </xsl:if>
93              <xsl:if test="QUALIFIER[@NAME='Max']">
94                 <xsl:attribute name="maxOccurs">
95                    <xsl:value-of select="QUALIFIER[@NAME='Max']/VALUE"/>
96                 </xsl:attribute>
97              </xsl:if>
98           </xsd:element>
99        </xsl:for-each>
100    </xsd:sequence>
101 </xsl:template>
```

Figure 3.7: CIM-XML to miniCIM XML Schema stylesheet excerpt (cont.)

and an enumeration or simple type to all properties (lines 58-68).

The processing follows with the referential properties. In lines 72–73 an element is created for each one with the `adc:reference` attribute (indicating that it is a referential property) set to `true`. The classes that the association can point at are defined next using `adc:range` (lines 75–77). The rest of the template (lines 79–97) is concerned with the translation of qualifiers. For example, lines 85–87 detect a `Required` or `Key` qualifier, and set the minimum cardinality to 1 (since the `Key` qualifier also implies that the property is required).

The stylesheet also has parts (not shown) to translate key properties to `<xsd:key/>` sections (as covered by Table 3.1), and to map attribute types.

3.4. Conclusions

We have seen in this chapter the use and extension of CIM, the mappings of XML used, and the space benefits of miniCIM. The process of transformation of the CIM-XML schema into a miniCIM XML Schema was also covered, including the mapping of its constructs to XML Schema and part of the XSLT transformation to implement this mapping. All these topics are of great importance not only for the extension of AdCIM to new domains, but also for its performance; so, in a sense, the Modelling Layer of AdCIM can be viewed as its core.

Chapter 4

System Data Layer

In this chapter we cover the System Data Layer of the AdCIM framework. This layer is tasked with the extraction of data from the managed nodes. This extraction is required because management frameworks have no purpose without real data of the managed entities. These data come from numerous sources, such as configuration files, registries, and proprietary repositories, so these sources are covered in Section 4.1. Data are converted from these sources to an intermediate XML format (step `[A]` in Figs. 2.1 and 4.1) and then translated by AdCIM to miniCIM instances using XSLT (step `[B]`) stylesheets provided by AdCIM that greatly simplify this task. Obtaining XML data from configuration sources can be especially difficult in the case of semi-structured flat text files, but it is simplified by text-to-XML grammar parsers that "bootstrap" flat text to XML, covered in Section 4.2.1. These text files are the norm in Unix systems, but rarely used in Windows. The differences between Windows and Unix systems are great; there also are divergences among Unix flavors, but they are minor compared with those of Windows incarnations and Unix-like systems. Thus, we have used another extraction method for Windows systems which is detailed in Section 4.2.2.

The chapter continues in Section 4.3 with a discussion of how Common Object Request Broker (CORBA) and Web Services are used in AdCIM for data transport between the managed nodes and management servers (see Fig. 4.1). Then, Section 4.4 explains our experiments to determine which technology is better in the context of our framework in terms of the transfer latency, total transfer time and message size for the messages generated in three typical scenarios. Finally, Section 4.5 details our conclusions.

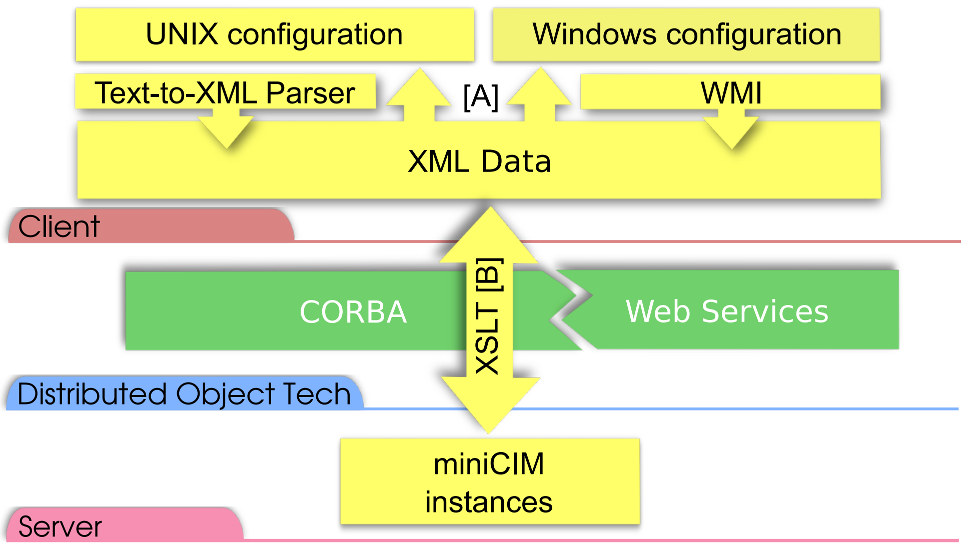


Figure 4.1: Overview of the configuration extraction process

4.1. Sources of Configuration and Management Data

Configuration and management data in production network nodes are dispersed among numerous and diverse locations, ranging from semi-structured flat files to registries, proprietary repositories and even firmware.

"Flat file" is a term that denotes a database that is "flattened" into a file, as opposed to a hierarchical or relational organization. These files have no overt structure, but a covert one, based on conventions on delimiters and use of line separators that must be respected by all editors, thus the "'semi-structured" term. The structure is actually consensual and external to the file.

Proprietary repositories are by nature found in many forms, but it is usual to store the configuration on embedded databases, separate disk partitions or, more rarely, as overlays on binary or library files.

Configurations on firmware are very widespread, since every PC stores vital information to its operation in Flash memory accessible by the BIOS configuration menu. Network cards can have self-booting programs in EEPROMs, and routers, wireless cards and even printers store their configuration in some sort of firmware. The main problem with firmware is the access to it, which can be proprietary, poorly documented and/or unsafe.

While most of these data sources are readily available through standard interfaces and OS calls, some of these interfaces are of proprietary nature, and ridden with platform and version differences, which makes keeping track of these data a difficult task. The system manager has the responsibility of locating all relevant information, so AdCIM tries to lighten it, by isolating, expressing and categorizing this knowledge. For this, system manager intervention is necessary for semantic interpretation and translation of the data, particularly for in-house software.

In the case of Windows machines, the Windows Management Instrumentation (WMI) subsystem [61] catalogs many software abstractions using a custom adaptation of CIM to Windows entities, so CIM-XML instances can be created easily with small scripts invoking directly the WMI API. UNIX and derivatives such as Linux rely instead on configuration files for expressing most configuration data, so their transformation to CIM data is more complex. Other important entities are the directory structures, such as the `/proc` filesystem, which represents many real-time management data and is also writable; and the output of administration agents (particularly logs and alert reports) can be handled like configuration files by processing them as text. Registries (other than Windows' own one) usually have proprietary APIs that require a case-by-case approach based on customized scripts.

4.2. Configuration Data Extraction

This section describes the methods used both in Unix and Windows systems to extract configuration data from system files and other text file-based sources. Configuration information is mainly collected from two sources: flat text sources, such as files and internal commands, and the WMI subsystem present in all Windows systems since Windows 2000.

4.2.1. Text File Configurations

Usually, Unix-based OS codify almost all configuration data in flat files and directory structures that are not available in directly parseable formats like XML, so our framework parses and transforms them to XML to facilitate further processing. Semi-structured flat files have been ubiquitously used to express configuration data [29], mostly for simplicity and integration with existing operating system capabilities — i.e. the Unix philosophy of 'everything is a file'. Their disadvantages are fragmentation, poor version tracking and text coding issues. From the point of view of data integration, the worst problems are text formatting and spacing: while many text file formats ignore white spaces, tabs and line distribution, some, like the Makefile format (which requires strict tabulator use) are very stringent.

Converting flat text files into CIM instances thus implies capturing these nuances in a formal and automated manner. Human intervention is needed to properly specify the semantics, largely external to the format. Despite this, many configuration formats are generalizable to line-oriented (e.g., `inetd.conf, sendmail.cf`) or section-record based (e.g., Apache's `httpd.conf`), and special cases are generally derived from these base cases.

The parsing of these text files is implemented in AdCIM through grammar rules, written using Martel [17], a Python module to parse text files to Simple API for XML (SAX) events, then directly transcribable to XML data. Fig. 4.2a shows an example of a Martel program that produces a structured XML file from the Inetd network services (described in Section 3.1) configuration file `/etc/inetd.conf` that can be seen in Fig. 4.2b. Table 4.1 shows some Martel operators. For example, `Martel.Re` and `Martel.Alt` represent the "*" and "|" regular expression operators, respectively: operator `Martel.Group` aggregates its second argument into a single XML element, and `Martel.ToEol` matches any text before the next end of line.

Martel command	Description
`Martel.Alt`	Matches only one of its arguments.
`Martel.Group`	Defines its contents as a group to be placed inside an XML element.
`Martel.Opt`	Matches its argument zero or one times.
`Martel.Re`	Tries to match a regular expression (Python flavor) into the text.
`Martel.Rep`	Matches its argument zero or more times.
`Martel.Str`	Renders its value with no provision to enter or modify data.
`Martel.ToEol`	Matches all text to the end of the line inclusive.
`Martel.UntilEol`	Matches all text until the end of the line.

Table 4.1: Martel operators

The Martel code follows the `inetd.conf` file structure, composed of lines of three types: off lines (see line 4 of Fig. 4.2a), commentaries (line 5), and normal service lines (line 6). Every normal line maps to an enabled service, and off lines to temporarily disabled services. The program also has to discriminate between commentaries and the `#<off>#` sequence that begins an off line. Each line is then sectioned, forming a list of items that are mapped to predetermined properties in

```
import Martel; from xml.sax import saxutils
def Item(name): return Martel.Group(name,Martel.Re("\S+\s+"))
fields=Item("name")+Item("socktype")+Item("proto")+Item("flags")+Item("user")+Martel.ToEol("args")
offline=Martel.Re("#<off>#\s*")+Martel.Group("off",fields)
commentary=Martel.Re("#")+Martel.Group("com",Martel.ToEol())
serviceline=Martel.Group("service",fields)
blank=Martel.Str("\n")
format=Martel.Group("inetd",Martel.Rep(Martel.Alt(offline, blank, commentary, serviceline)))
parser = format.make_parser()
parser.setContentHandler(saxutils.XMLGenerator())
parser.parseFile(open("inetd.conf"))
```

(a) Martel program used for parsing `inetd.conf` to XML

```
#echo stream tcp nowait root internal
ftp  stream  tcp nowait root /usr/sbin/tcpd /usr/sbin/proftpd
#<off># sgi_fam/1-2  stream rpc/tcp wait root /usr/sbin/famd
```

(b) Sample lines from the original `inetd.conf` format

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
   <commentary> <com>#</com>echo stream tcp nowait root internal </commentary>
   <line><id>ftp</id><ws> </ws><id>stream</id><ws> </ws><id>tcp</id><ws> </ws>
         <id>nowait</id><ws> </ws><id>root</id><ws> </ws><id>/usr/sbin/tcpd</id>
   <ws> </ws><id>/usr/sbin/proftpd</id></line>
   .........
   <off> <com>#</com>&lt;off&gt;#<ws> </ws>
      <line><id>sgi_fam/1-2</id><ws> </ws><id>stream</id><ws> </ws><id>rpc/tcp</id>
      <ws> </ws><id>wait</id><ws> </ws><id>root</id><ws> </ws>
      <id>/usr/sbin/famd</id><ws> </ws><id>fam</id></line>
   </off>
   </doc>
```

(c) Excerpt of the transformation of `inetd.conf` to XML

Figure 4.2: Parsing of the `inetd.conf` file to XML

the resulting CIM instances. The output of this transformation, in Fig. 4.2c, is still a direct representation of the original data in Fig. 4.2b, now structured.

The specification of a configuration file with grammar-based rules has another benefit, as this documents the configuration format formally. The detail can be high, following the original file format very closely, or be more general and describe the high-level format of the document (e.g., line-oriented with space separators). The latter approach makes it easier to process many formats by specifying general rules (and a more lenient matching when files do not conform to format), but the former has the benefit of early-on error checking and validation of configuration formats. Since Martel supports backtracking, multiple versions of the same file can be specified. If the file fails at the end of the parsing process, the backtrack makes the process resume with the next matching rule that would represent another version.

This feature can be also used to determine file version and mismatches.

The output of Fig. 4.2c is converted to miniCIM via an XSLT stylesheet (step `[B]` in Figs. 2.1 and 4.1). Since this stylesheet creates miniCIM XML instances, its output is simple to write (i.e., XML element-value pairs for each property).

The task of converting configuration and management data to CIM instances is perhaps the most difficult and error-prone stage in the use of AdCIM, due to the format and semantic mismatches, and the usual problems in using regular expressions, such as matching unintended strings or failing intended matches. Nevertheless many tools and languages used traditionally by system administrators, such as Perl or Python, use regular expressions extensively, so this problem is not particular to our framework.

Figure 4.3a shows a more complex example of grammar rules that parses the `/var/log/messages` log file, composed of messages, warnings and errors from various system processes and the kernel. This log format (shown in Fig. 4.3b), very representative of Linux flavors, is line-based (entries correspond with lines in the file). Each entry contains a date (parsed in line 5 of Fig. 4.3a) and a process identifier (line 9). In lines 11-12, the format of entries is specified as consisting of a date, followed by a space, followed by a host name, another space, a process identifier and a free-form message until the end of the entry. This free-form message can be further parsed to detect alerts and notifications of individual processes. The objects obtained from a log file are semantical alerts (not configurations) and are thus read-only, but in tandem with configuration data enable the detection of misconfigurations.

4.2.2. WMI Data

Windows moved its system configuration repository from files to the Registry with the introduction of Windows 95. Thus, to extract configuration data it would seem necessary to manipulate Registry data. Instead, we have used the Windows WMI subsystem [61], which provides comprehensive data of hardware devices and software abstractions in CIM format, exposed using the Component Object Model (COM), the native Windows component framework. WMI is built-in since Windows 2000, but it is also available for previous versions. Queries can also be made remotely using Distributed COM (DCOM). Its coverage varies with the Windows version, but it can be extended by users.

```
import Martel; from xml.sax import saxutils
def Group(x,y): return Martel.Group(x,y)
def Re(x): return Martel.Re(x)
def Item(name): return Martel.Group(name,Martel.Re("\S+"))
def Date(name): return Martel.Group(name,Martel.Re("\S+\s+\d+\s+[0-9:]*"))
def Space(): return Martel.Re("\s*")
def Colon(): return Martel.Re(":\s*")
def Origin(): return Group("origin",Re("\w+"))+Col()
def OriginPid(): return (Group("origin", Group("name", Re("[\w()_-]+")) +Re("\[")+
        Group("pid", Re("[0-9]+")) +Re("\]")+ Colon()))
fields=(Date("date") +Space()+ Item("host") +Space()+ Martel.Alt(OriginPid(),Origin(),Space()) +
        Martel.UntilEol("message") + Martel.ToEol())
format=Group("file",Martel.Rep(fields))
parser = format.make_parser()
parser.setContentHandler(saxutils.XMLGenerator())
parser.parseFile(open("m"))
```

(a) Martel program used for parsing `/var/log/messages` to XML

```
Nov 25 17:40:01 host1 cron[8698]: (root) CMD (/usr/bin/giis -u > /dev/null )
Nov 25 17:46:18 host1 sshd[8714]: Accepted keyboard-interactive/pam for user1 from 283.154.60.211 port 58598 ssh2
```

(b) Sample lines from `/var/log/messages`

Figure 4.3: Parsing `/var/log/messages` file to XML

```
1  import sys, win32com.client, pythoncom, time; from cStringIO import StringIO
2  locator = win32com.client.Dispatch("WbemScripting.SWbemLocator")
3  wmiService = locator.ConnectServer(".","root\cimv2")
4  refresher = win32com.client.Dispatch("WbemScripting.SWbemRefresher")
5  services = refresher.AddEnum(wmiService, "Win32_Service").objectSet
6  refresher.refresh()
7  pythoncom.CoInitialize()
8  string = StringIO()
9  for i in services:
10   (string.write((
11   "<SystemCreationClassName>"+unicode(i.SystemCreationClassName)+"</SystemCreationClassName>"+
12   "<CreationClassName>"+unicode(i.CreationClassName)+"</CreationClassName>"+
13   "<Name>"+unicode(i.Name)+"</Name>"+"<State>"+unicode(i.State)+"</State>"+
14   "<StartMode>"+unicode(i.StartMode)+"</StartMode>"+"</CIM_Service>").encode("utf8")))
15 print string.getvalue()
```

Figure 4.4: Python script to extract service information from WMI

WMI data can be uniformly retrieved using simple code, such as the one shown in Fig. 4.4, which uses the COM API and directly writes XML data of miniCIM instances representing Windows services (which use a derived class of `CIM_Service` with custom properties). The code uses a locator to create a WMI COM interface named `SWbemRefresher` (see line 4) which makes possible to update WMI instance data without creating additional objects. Line 5 declares Win32 services as the class to listen for instances. In the next lines, instances contained in the refresher interface are queried and their data written as miniCIM instances. Line 7 is needed to avoid problems accesing DCOM objects in multi-threaded contexts. A `StringIO` Python object (line 8) is used to avoid string object creation overheads. Since Python, like Java, has immutable strings, using them directly would generate a number of unneeded temporary objects. Hence, we use `StringIO`, which is a mutable object, mimicking a file interface and designed for efficient string concatenation. Aside from these minor technical points, the conversion of WMI data to miniCIM instances is a much easier process than the parsing of flat configuration files.

4.3. Distributed Client Data Transport

The data in managed nodes must be transferred to the servers and updated back in the clients via a transport mechanism. Since the arrangement and number of managed nodes could vary widely over time, a network-transparent distributed transport system is needed to support large numbers of nodes, allowing them to communicate in a standard and decentralized manner.

Thus, this section describes the use of both CORBA [72] and Web Services [9] solutions for distributed data transport in our framework. As can be seen in Fig. 4.1 the purpose of this middleware is to efficiently transfer miniCIM instances or raw XML data between clients and servers.

CORBA achieves interoperability between different platforms and languages by using abstract interface definitions written in the Interface Description Language (IDL), from which glue code for both clients and servers is generated. This interface is a “contract” to be strictly honored by both parties. This enforces strict type checking, but clients become “brittle”: any change or addition in the interface breaks their code and implies their recompilation and/or readaptation.

Using an XML Schema validated dialect has two benefits: first, it promotes flexibility, since changes in format can be safely ignored by older clients and, second, it preserves strict validation in document exchanges. Both aspects are important due to the extensibility of the CIM model, which induces more frequent updates, but in very time-critical instances a direct mapping of a CIM class to IDL is still possible.

To pass XML data via CORBA they are flattened to a string. This solution is not optimal, since some time must be lost in serialization and de-serialization. A more efficient solution would be to pass the data as a CORBA DOM Valuetype [73], which is passed by value with local methods. Then, the parsed XML structure would not be flattened, so that clients could manipulate the XML data without remote invocations. Unfortunately, this is a feature not yet well supported in most production-grade ORBs, so the ultimate impact of this solution on serialization performance is yet to be measured. Our requirements for the solution were opennes, performance, and, above all, small footprint, so our chosen implementation of ORB after reviewing existing implementations was omniORB [42], a high-speed CORBA 2.1 compliant ORB with bindings for both C++ and Python.

In contrast with CORBA, Web Services (WS) solutions provide an interoperation layer that can be both tightly coupled (using methods with strict typing and procedures) or loosely coupled (XML document-centric). Gradually, WS are being more oriented to support web-based service queries than to offer distributed transport, but there is a significant overlap between the two approaches.

WS use XML dialects, such as WSDL for interface definition, and Simple Object Access Protocol (SOAP) for transport. It may seem that using XML dialects would promote synergy, but the use of XML as both "envelope" of the message and for representing it does not offer additional synergy. In fact, it can be a hindrance, since the message must be either sent as an attachment (which implies Base64, a binary codification scheme that codifies 3 binary bytes into 4 ASCII characters and has roughly a 37% penalty in size), or with its XML special characters encoded as character entities to avoid being parsed along with the XML elements of the envelope. Additionally, two XML parsings (and the corresponding encoding) must be done, causing high latency. There seems to be no plans for supporting platform-independent parsed XML representations like DOM Valuetype in WS.

As implementation of Web Services we have chosen the Zolera SOAP Infrastructure [90], the most active and advanced WS library for Python. It was chosen to minimize software and resource requirements in the managed nodes, and offers good performance at the same time.

4.4. Data Transport Experimental Results

We have proceeded to evaluate the performance of the System Data Layer for various representative tasks and the impact of the transport technology (Web Services vs CORBA). The tests have been performed using Athlon64 3200+ nodes connected by Gigabit Ethernet cards.

One objective of this framework is to support monitoring applications which consume structured data and gather and update them with small footprint and low-latency. To achieve multiplatform interoperability and low-latency network messaging, both CORBA and Web Services are assessed.

The subject of performance and low-latency issues in system administration is discussed in other works. Pras et al. [84] find Web Services more efficient than SNMP for bulk administration data retrieval, but not for single object retrieval (i.e. monitoring), and conclude that data interfaces are more important for performance than encoding (Basic Encoding Rules (BER) vs XML). Nikolaidis et al. [69] show great benefits by compressing messages in a Web Service-based protocol for residential equipment management, but only use Lempel-Ziv compression, which is outdated, and patent-encumbered. Yoo et al. [109] implement zlib compression – using the patent-free Deflate method – and other mechanisms to optimize the NETwork CONFiguration (NETCONF) protocol using SOAP (Web Services), Blocks Extensible Exchange Protocol (BEEP) and Secure SHell (SSH) messaging, but their method gives similar response times for all three protocols, which indicates that the bottleneck is elsewhere.

We have tested three different cases of use of system data extraction, both in Windows and Unix (Linux). The parameters measured for these cases have been total time, latency and message size, with and without compression. Total time is defined as the round-trip time elapsed between a request is sent and the response is completely received. Latency is the round-trip time when the response is a 0-byte message. Two different algorithms have been used for compression: zlib and bzip2. The three cases tested are:

- CPU load retrieval, (results are shown in Fig. 4.5). This case is representative of monitoring applications, which would be typically invoked several times per second, and need specially fast response times and low load on the client.
- Service information discovery, shown in Fig. 4.6. This case represents queries for the discovery of services or machines, invoked with a frequency ranging from minutes to hours.
- Log file information retrieval and parsing (data mining), shown in Fig. 4.7. This case represents bulk data requests invoked manually or as part of higher-level diagnostic processes. These requests have unspecified total time and data size, so they are invoked ad-hoc, with little or no regularity.

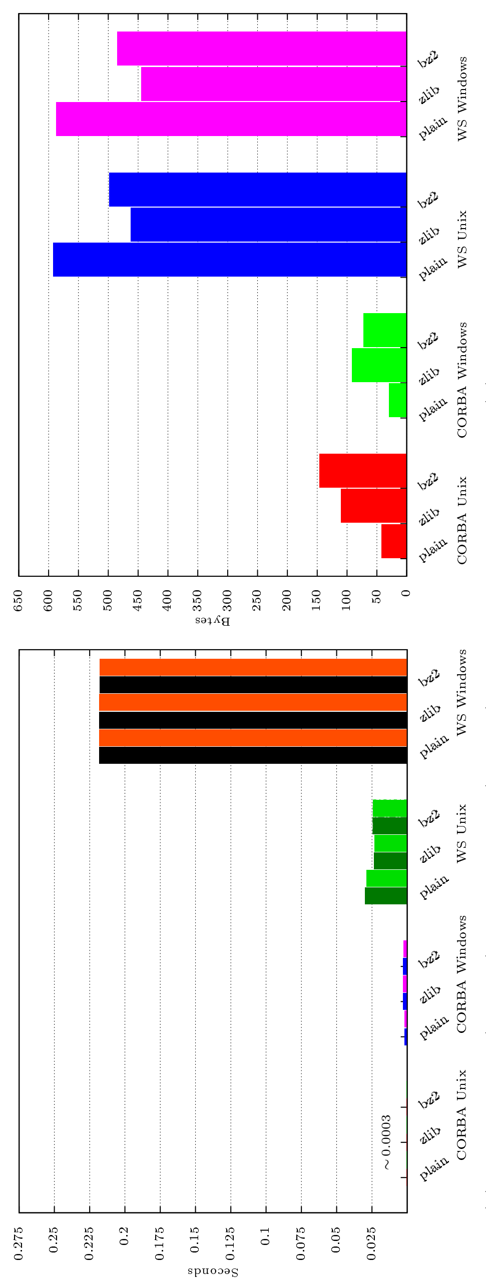
The code examples shown in Section 4.2, specifically Figs. 4.2a and 4.4 for network services, and 4.3a for log parsing have been used in their corresponding cases.

The first test in Fig. 4.5 shows lower latency and total time for CORBA vs WS in both platforms. Base latency for CORBA is roughly 0.2 ms, whereas it rises to 20-30 ms when using WS, in great part due to the overhead of parsing the envelope and codifying the message. Compression benefits greatly WS but slows down CORBA performance. Fig. 4.5b shows the cause: WS messages are large enough to benefit from compression, but CORBA messages (less than 50 bytes long) are actually doubled in size. In the second test (Fig. 4.6), WS times are very similar to those obtained in the previous case, since parsing overhead dominates total time. CORBA times are longer than in the first case, but still shorter than the WS counterparts.

In these two test cases, Windows times are higher than those of Unix, due to the overhead of operating with COM objects. Nevertheless, these overheads are smoothed over in the third test, since each object carries more data. From Figs. 4.5 and 4.6, it is clear that message size determines total time, affecting WS much more, due to the use of an XML codification and envelope. The envelope size, which introduces an almost fixed penalty affects significantly only the first test with short message size, but codification introduces a 20% message size overhead on average.

The third test (Fig. 4.7) shows a much narrower spread of values due to message size (1Mb+); thus, total time is dominated by transfer time, instead of by protocol overheads. In this test compression in WS achieves times comparable to those of uncompressed CORBA. This would be more noticeable with less bandwidth, as WS compression ratios of 40:1 are reported in Fig. 4.7b.

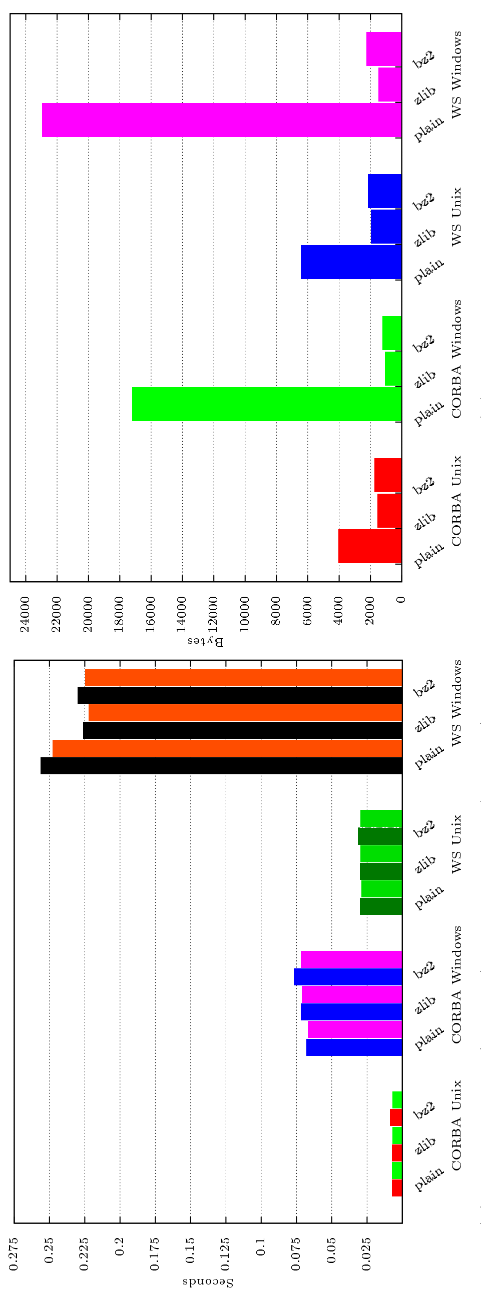
In general, all times are very acceptable, although CORBA has a clear advantage. The benefits of compression are dubious, except in the third test for WS. bzip2 compresses better, but it is slower and is oriented to larger data sets than zlib, which is a better choice for the tested cases. Although both WS and CORBA are acceptable solutions for information exchange in our framework, monitoring and low-latency applications strongly favor CORBA over WS due to its message compactness and better processing time, so it is the transport solution chosen for AdCIM.



(a) Total time (darker) and latency (lighter)

(b) Message size

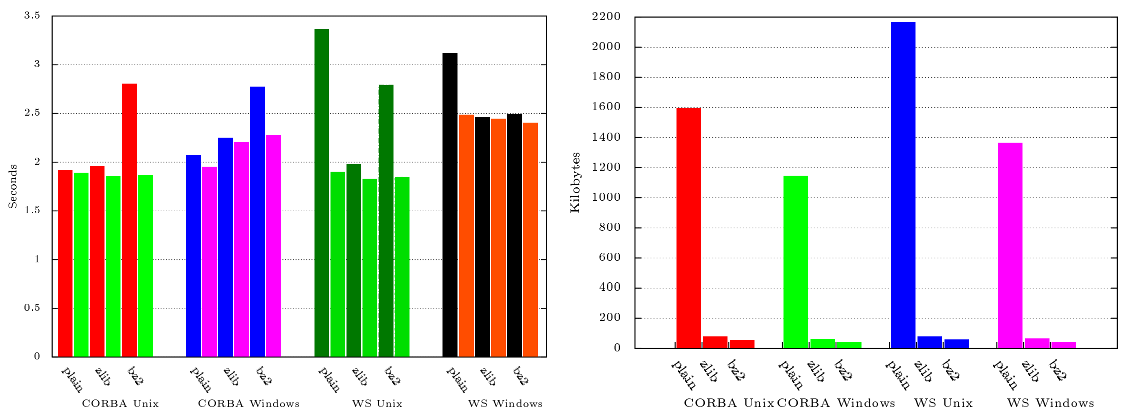
Figure 4.5: Performance measurements for CPU Load test



(a) Total time (darker) and latency (lighter)

(b) Message size

Figure 4.6: Performance measurements for Service Discovery test



(a) Total time (darker) and latency (lighter) (b) Message size

Figure 4.7: Performance measurements for Log Parsing test

4.5. Conclusions

We have explored the configuration data extraction in the AdCIM framework, focusing on the extraction of service and log data from Windows and Unix into mini-CIM instances. This is achieved using different techniques (text-to-XML parsing grammars, WMI scripts) due to the different management approaches supported by each OS.

We have also discussed methods and alternatives to implement multiplatform and low-latency transport methods using two different approaches: CORBA and Web Services technologies. Finally, we have assessed the implementations by defining a testing framework for measuring total time, latency and message size of the transport of messages for three usual scenarios on network management.

The chosen fields of network services and log data analysis have illustrated the use of these methods for administration domains particularly dissimilar between operating systems. But the scope of the System Data Layer is not limited to such domains, as will be shown in Part II of the Thesis, which shows adaptations to the Sendmail mail agent (Chapter 7), and Wireless Mesh Networks (Chapter 9).

Chapter 5

Data Persistence Layer

This chapter details the implementation of the persistence in the AdCIM framework. Section 5.1 presents an overview of the LDAP directories used to store management data, their protocols and interchange formats, which represent the main interface between AdCIM and the LDAP directory. Section 5.2 covers the transformations shown in Figs. 5.1 (an excerpt of Fig. 2.1), that is, the transformation from miniCIM instances to LDAP repository data (step `[C]`), from LDAP data to miniCIM instances (step `[D]`), and from miniCIM Schema to LDAP Schema (step `[G]`). Finally, Section 5.3 summarizes this chapter.

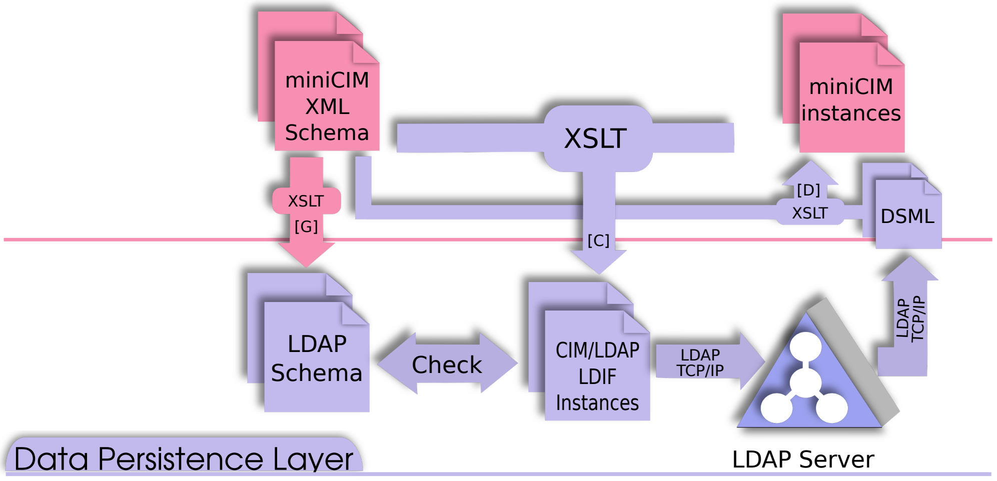


Figure 5.1: AdCIM Data Persistence Layer

5.1. Directory Databases

The AdCIM framework requires a persistence layer with replication and scalability to support efficient decentralized management, but it does not require strong transaction enforcement or triggers. The flexibility and extensibility of the logical schema and the query language are also important to ease storage and translation of miniCIM instances to/from the repository. Several repository solutions were considered, finally settling on LDAP directories.

Here we consider "directory" broadly to include hierarchical databases optimized for queries, such as Domain Name System (DNS) or LDAP. The data inside them are usually organized on a tree of nodes. These nodes have a type which determines their attributes. Nodes (also named entries) can be accessed directly from the root or via a query which searches the tree for nodes that satisfy a condition.

Directory databases were developed primarily to store user, naming and organizational information for mail exchange services, which led naturally to their hierarchical, tree-like structure. Directories are highly optimized for read operations, support multivaluated attributes and multi-master replication. They often implement the LDAP access protocol [54].

The greatest benefit of directories lies in their having centralized and fast access and update to frequently used information. This logically centralized access makes it much easier to implement access control to sensible information. Directories also allow defining and introducing new types of entries and can be reorganized more easily than other types of databases. They have disadvantages in applications that require transactional processing.

5.1.1. The LDAP Protocol: Origin and Design

X.500 was developed by the CCITT (Telegraph and Telephone Consultative Committee) – now ITU (International Telecommunications Union) – as a proposal to standardize directory services. X.500 is notable because it was the first directory system designed to be open, general-purpose and extensible, and supported a more powerful search model and replication mechanisms than contemporary systems. Nevertheless, implementing the standard was difficult, and these implementations were unstable and slow in part due to the use of the Open Systems Interconnection (OSI) network stack instead of the TCP/IP one.

The protocol used to access X.500 directories was called DAP (Directory Access Protocol), so when a lighter protocol was implemented it was named LDAP (Lightweight Directory Access Protocol). Since it was easier to implement, it gradually displaced X.500.

LDAP was designed in the early 90's and later standardized by the IETF [47]. It defines an open and standard interface for directory access which simplified several points of DAP and used the TCP/IP stack with minimal functionality loss. Over the years, LDAP-only directory implementations appeared and the benefits carried over to the server side. Nowadays LDAP is a critical part of the computer services in many organizations.

LDAPv3, the last revision, brought several improvements, such as internationalization support, client referrals, strong cryptography support, extensible operations, and the possibility of retrieving schema and implementation information from the repository.

Information Model

Tipically, the entries in a directory share a common domain, and there is a strong relationship between entries. Hence, it is natural to group these semantic restrictions in a common schema, to avoid duplication and promote reusability, and ultimately standardize common types into an inheritance hierarchy of object classes.

The schema is enforced strictly, so non-explicitly defined attributes or class inclusions are disallowed. Attributes can be declared as multivaluated, and be inherited between classes. Entries – which represent entities – are defined as instances of object classes. These classes can be divided in three different types:

- *abstract*: Classes on the top of the information model hierarchy. Entries can not have directly an abstract class – i.e. these classes are not instantiable – because their purpose is to define common attribute sets inheritable by normal classes, and make searches easier. The `top` class, ancestor of all the classes, is their representative.
- *structural*: Instantiable classes that can be represented by actual entries in the repository. All must inherit from an abstract class such as `top`.
- *auxiliary*: Auxiliary classes do not represent entities, but secondary attribute sets which are transversal. For example, a telephone number can be associated

to a person, a company or a dial-up service. It is used also to add optional attributes to an entry without having to change its schema.

In this work, auxiliary classes are used extensively for the representation of CIM associations.

Naming Model

The naming model is the most visible, because it determines the name of entities. The model has a tree topology, as can be seen in Fig. 5.2, but in contrast with the information model, it has no common root for all the entries in a repository. This allows a server to have several separate trees with local roots which are effectively in separate namespaces. It is also possible to fuse these trees in a super-hierarchy if needed.

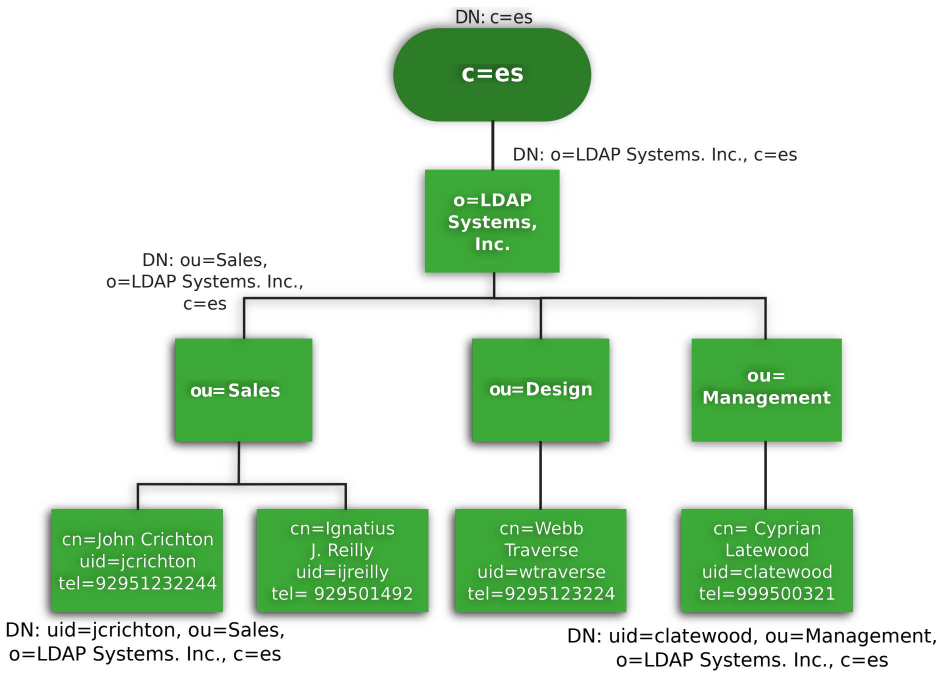

DN: c=es
c=es
DN: o=LDAP Systems. Inc., c=es
o=LDAP Systems, Inc.
DN: ou=Sales, o=LDAP Systems. Inc., c=es
ou=Sales
ou=Design
ou= Management
cn=John Crichton uid=jcrichton tel=92951232244
cn=Ignatius J. Reilly uid=ijreilly tel= 929501492
cn=Webb Traverse uid=wtraverse tel=9295123224
cn= Cyprian Latewood uid=clatewood tel=999500321
DN: uid=jcrichton, ou=Sales, o=LDAP Systems. Inc., c=es
DN: uid=clatewood, ou=Management, o=LDAP Systems. Inc., c=es

Figure 5.2: LDAP tree showing naming convention

Entry names are formed concatenating the name of each node, or Relative Distinguished Name (RDN), constructed using one attribute designed as identifier, with the RDN of its parent, until reaching the local root. The resulting name, or Distinguished Name (DN), is constructed backwards compared with that of a file in a filesystem. In Fig. 5.2, nodes representing people are identified using the attribute `uid` (user identifier), and the node at the top by the attribute `c` (country). Other usual attributes are `cn` (common name), `ou` (organizational unit), and `o` (organization). Since each entry can use different attributes as identifiers, the name of the attribute is also included in the name.

Functional Model

The functional model represents the query and modification operations that can be performed on constructs defined in the information or naming models. LDAP operations are considered atomic, so if some step in an operation is not completed satisfactorily, the previous state is restored.

Most queries are performed using the `search` operator, which can search the entire tree, a subtree or a concrete node. To select only nodes which verify some conditions, LDAP filters are used. These filters are composable, but only filter out nodes from the subtree. They can not join entries, direct the search or perform arithmetic operations.

The `compare` operator is used to verify if an entry has an attribute with an expected value. The client sends a triplet `<DN>,<attribute_name>,<value>`. The server then sends an affirmative or negative answer. This operator stems from X.500, where the search operator did not discriminate the case where an attribute was not present from the case where the same attribute had a value not satisfying the filter.

The entry operator `delete` accepts a DN and removes the referred entry if it is present, does not have children, and the user is authorized.

The perhaps non-intuitive `rename` operator can not only change the name of entries, but also copy and move them. It accepts four parameters, the DN of the entry to be changed, its new RDN, optionally the DN of its new parent, and a boolean value that, when true, instructs the server to keep the old entry, effectively making a copy.

These operators are tree-level operators that work with whole entries. There are also entry-level operators that work with attributes. In DAP, it was possible to add and delete attributes and add or remove values to an existing attribute, but when adding a value to an attribute, this setup required the client to first check if the attribute was present or not and use a different operation in each case.

LDAP simplified entry modification semantics using three operators: `add`, `delete` and `replace`. The first one, `add`, appends new values for both existing or non-existing attributes. In a similar manner, `delete` erases values, and attributes if they lose their last values. Finally, `replace` overwrites the previous set of values. As shown in the next section, these semantics pose a problem to our framework persistence, since adding a preexisting value to an attribute is considered as an error.

5.1.2. LDAP Interchange Formats

There are a great number of LDAP clients which facilitate access to directories, but there are some tasks, such as data interchange between different vendor implementations, bulk import of entries, database dumps and bulk edit automation, that can be unfeasible using most clients directly. These cases are covered by the interchange formats. The most prominent are LDIF (LDAP Data Interchange Format) and DSML (Directory Services Markup Language).

LDIF was part of the initial LDAP ecosystem, and later underwent a standardization process that ended with LDAPv3, specified in Request for Comments (RFC) documents included in [54]. It is text-based and supported universally by LDAP clients. DSMLv2 [78] is basically an XML mapping of LDIF, with the same capabilities, designed much later and standardized by OASIS (Organization for the Advancement of Structured Information Standards) [77].

LDIF

LDIF is a plain text-based format structured in multi-line records separated by blank lines. Each record can either describe a directory entry (entry record) or any change that can be made to it within LDAP (operator record). These changes are specified with directives, which are equivalent to the operators seen before.

The types of possible lines in an LDIF file are shown in Table 5.1. Records are separated by blank lines, and separator lines are used when there are multiple operators in the same entry. Content lines are of the format '`<x>: <y>`', in which the meaning of `x` and `y` depends on the context. For example, the first line of either record begins with a '`dn: <DN of entry>`' directive, which specifies the DN of the entry described or modified by the record. In the case of entry records, the following lines have the format '`<attribute_name>: <attribute_value>`'. There is at least one line for each attribute, and also one for each value of multivaluated attributes.

Operator records also begin with the `dn` directive, but followed by the metadirective `changetype: <directive>`, where `directive` can be one of `add`, `delete`, `modify` and `modrdn`. Three of these directives (the "node directives") refer to entire entries: `add` adds a new entry, `delete` removes an existing entry, and `modrdn` changes the DN of an entry, with the same options seen for the operator `replace` at the end of Section 5.1.1.

Type	First character	Description
Content line	any except <space>,'`-`' or '`#`'	A line containing part of an entry or operation.
Wrapped line	<space>	Continuation of the previous line, can be repeated.
Blank line	`\n` (newline)	A line used to end records.
Comment line	'`#`'	A comment, ignored by clients.
Separator line	'`-`'	Used to separate operators.

Table 5.1: Types of lines in an LDIF file

The `changetype: modify` directive is used when, instead of entries, attributes need to be changed. This directive must be followed by a number of subdirectives in the form `<directive>: <attribute_name>`, where `directive` can be one of `add`, `replace` and `delete`. All of these subdirectives should be followed by argument pairs of the form `<attribute_name>: <attribute_value>`, where `attribute_name` must match the one following the directive. These directives should not be confused with the node directives just seen, as they operate exclusively with attribute values. Consequently, `add` adds a new value to an attribute, creating it if necessary; `replace` changes whatever values the attribute had by the ones in the argument, and `delete` can delete some values of an attribute, or the entire attribute if no values are specified. When adding new values to an attribute, it is possible to specify a Base64 encoded string or a file URL as input.

It is possible to indicate several operations to be performed in the same entry. Since directives can have an arbitrary number of arguments, it is required to put a separator line (a lone '`-`' symbol) between directives.

Figure 5.3 shows an example of LDIF file codifying instances of the `InetdService` and `HostedInetdService` classes shown in Fig. 3.2, and which employ the organization and directives just explained: the first block of lines (lines 1–12) before the blank line form an entry record. This record begins with a `dn` directive (line 1), followed by the DN of the entry, formed by two RDNs (the value of the `orderedCimKeys` attribute and `dc=udc`, the top entry of the tree) separated by a comma. The next attribute, `orderedCimKeys` (line 2) is the identifier attribute. Which attribute is the identifier and its exact content is not important, but it has to be a unique name amongst its siblings. The `objectClass` line (line 3) is the most important one, since it determines the class of the entry, and thus, the type of attributes that the entry can hold, and its mandatory attributes. An entry can have one primary objectclass and an arbitrary number of auxiliary classes which enable the attachment of new attributes. The next lines (lines 4–12) are pairs attribute name – attribute value. The attribute in lines 9–10 appears two times, because it has two values. The blank line in line 13 ends the record.

The second block in lines 14–23 is an operator record which codify several additions to the entry represented in the first block, so its DN is repeated in the first line (line 14). Afterwards, a `changetype: modify` directive indicates that the rest of the record is composed of attribute and value operations. All of them are `add` operations, with separator lines in between. The first `add` operation (lines 16–17) assigns a new objectclass to the entry, which permits the addition of the following two attributes (lines 19–23). LDAP operations are atomic, but some implementations do not support adding a new objectclass and attributes allowed by that objectclass in the same operator record.

Summarizing, LDIF files can represent both entries and operations on these

```
 1  dn: orderedCimKeys=AdCIM_InetdService.CreationClassName\3Dccname\2CName\3Dftp\2CProtocol\3Dtcp\2CSystemCreationClassName\3
        Dscname\2CSystemName\3Dsname,dc=udc
 2  orderedCimKeys: AdCIM_InetdService.CreationClassName=ccname,Name=ftp,Protocol=tcp,SystemCreationClassName=scname,SystemName
        =sname
 3  objectClass: AdCIM-InetdServiceInstance
 4  CreationClassNameCIM-Service: ccname
 5  SystemCreationClassNameCIM-Service: ccname
 6  SystemNameCIM-Service: scname
 7  NameCIM-Service: ftp
 8  SocketTypeAdCIM-InetdService: stream
 9  ProtocolAdCIM-InetdService: tcp
10  ProtocolAdCIM-InetdService: udp
11  UserAdCIM-InetdService: nowait
12  CommandAdCIM-InetdService: root/usr/sbin/tcpd
13
14  dn: orderedCimKeys=AdCIM_InetdService.CreationClassName\3Dccname\2CName\3Dftp\2CProtocol\3Dtcp\2CSystemCreationClassName\3
        Dscname\2CSystemName\3Dsname,dc=udc
15  changetype: modify
16  add: objectClass
17  objectClass: AdCIM-HostedInetdServiceAuxClass
18  -
19  add: StartModeAdCIM-HostedInetdService
20  StartModeAdCIM-HostedInetdService: Automatic
21  -
22  add: AntecedentAdCIM-HostedInetdService
23  AntecedentAdCIM-HostedInetdService: orderedCimKeys="CIM_ComputerSystem.CreationClassName=CIM_ComputerSystem,Name=
        shalmaneser", dc=udc
```

Figure 5.3: Example of an LDIF file representing `InetdService` and `HostedInetdService` instances.

entries. These capabilities are useful for both initial directory population and the automation of operations. The use of interchange formats such as LDIF decouples the transformation of data to directory format from the actual repository updating, which can then be made in the most appropriate way. It also makes possible to rollback changes easily and log them incrementally using versioning systems. Finally, LDIF files can serve as full backups of the repository data that are both human-readable and implementation independent.

DSMLv2

As mentioned before, DSMLv2 has the same capabilities as LDIF. Since LDIF already covers the full spectrum of data representation and manipulation on LDAP directories, the necessity of another interchange format could be questioned. The motivation behind DSML was to develop an XML-based LDIF equivalent. This would mean an easy integration with XML-based tools and applications and the updating and consulting of directories across firewalls, which customarily blocked LDAP ports, using HTTP as transport protocol.

DSMLv1 was a preliminary specification released by OASIS [77] in 1999, and only covered directory representation. DSMLv2 adds support for data operations, making it a true LDIF-equivalent. DSML can also specify queries and represent the schema of a directory, areas not covered by LDIF, and usually non interoperable between LDAP implementations.

By design, DSML is a client format. Clients read DSML and communicate with the server using standard LDAP commands. Therefore, no support is required in the server. The drawback is that there must be an intermediate conversion step from the LDAP response to DSML. Nevertheless, there are no clear benefits of implementing this support in the server, and the design of DSML makes the conversion straightforward.

In Figure 5.4 a small snippet of DSML data is shown. The first line is the XML preamble that all well-formed XML documents must have. The `batchResponse` and `searchResponse` elements (lines 2–38) contain the result of a search with one result entry contained in the `searchResultEntry` element (lines 4–36). The DN, which in an LDIF file would be the first line of the record, here is an XML attribute, but using the same format as in line 1 of Fig. 5.3.

Inside the entry, the objectclass is defined as an `objectclass` attribute (lines 20–

```xml
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <batchResponse xmlns="urn:oasis:names:tc:CIML:2:0:core">
 3    <searchResponse>
 4      <searchResultEntry dn="orderedCimKeys=AdCIM_InetdService.CreationClassName\=AdCIM_InetdService\,Name\=ftp\,Protocol\=
            tcp,dc=udc">
 5        <attr name="CommandAdCIM-InetdService">
 6          <value>root/usr/sbin/tcpd</value>
 7        </attr>
 8        <attr name="SocketTypeAdCIM-InetdService">
 9          <value>stream</value>
10        </attr>
11        <attr name="UserAdCIM-InetdService">
12          <value>nowait</value>
13        </attr>
14        <attr name="orderedCimKeys">
15          <value>AdCIM_InetdService.CreationClassName=AdCIM_InetdService,Name=ftp,Protocol=tcp</value>
16        </attr>
17        <attr name="ProtocolAdCIM-InetdService">
18          <value>tcp</value>
19        </attr>
20        <attr name="objectClass">
21          <value>AdCIM-InetdServiceInstance</value>
22          <value>AdCIM-HostedInetdServiceAuxClass</value>
23        </attr>
24        <attr name="CreationClassNameCIM-Service">
25          <value>AdCIM_InetdService</value>
26        </attr>
27        <attr name="AntecedentAdCIM-HostedInetdService">
28          <value>orderedCimKeys=CIM_ComputerSystem.CreationClassName\3DCIM_ComputerSystem\2CName\3Dshalmaneser,dc=udc</
                value>
29        </attr>
30        <attr name="NameCIM-Service">
31          <value>ftp</value>
32        </attr>
33        <attr name="StartModeAdCIM-HostedInetdService">
34          <value>Automatic</value>
35        </attr>
36      </searchResultEntry>
37    </searchResponse>
38  </batchResponse>
```

Figure 5.4: Example of a DSML file representing an `InetdService` instance with a `HostedInetdService` association attached.

23), with several `value` subelements which hold the classes associated with the entry. In DSML attributes are represented as `attr` elements with a `name` XML attribute, and their values are represented as `value` subelements.

DSML has also a complete operation model, but AdCIM uses LDIF for directory modification operations, so it is not covered in this section. The main benefit of DSMLv2 is that its XML syntax makes possible the use of XSLT to transform the directory data back to CIM objects. The use of LDIF would require an intermediate parsing of the LDIF output not needed with DSML.

5.2. AdCIM Persistence in LDAP

Despite the repository-independent design of AdCIM, LDAP directories are specially suited to store its miniCIM information due to their scalability, schema flexibility, hierarchical structure and implantation. There are other possibilities, the most straightforward is to do a flat mapping to a relational database, such as in the works by Finke [31, 32]. The problem with this approach is the inflexibility of relational tables and schemata. Automatic XML persistence in relational databases is a more current proposal addressed in the works surveyed by Nambiar et al. [66]. It offers transparent persistence, but to date has to cope with severe performance penalties. These shortcomings are in great part due to their generality. In AdCIM's case, we have additional knowledge about the structure of the stored data which allows a more efficient and natural mapping.

The transparent storage of miniCIM data on LDAP repositories is ensured by three XSLT stylesheets provided by AdCIM. Two of these stylesheets carry out the transformation of miniCIM to and from directory data (steps `[C]` and `[D]` in Figs. 2.1 and 5.1) according to the CIM schema, and are invoked by the Application Layer when an appropriate query is made to the web service interface (step `[F]` in Fig. 2.1). The third stylesheet translates the CIM schema to an LDAP schema that checks equivalent constraints on directory data (step `[G]` in Figs. 2.1 and 5.1).

The following subsections describe these stylesheets in detail, and a final subsection presents benchmark results. For clarity, the term *attribute* is used within LDAP contexts, and the term *property* within a CIM context.

5.2.1. miniCIM to LDIF Stylesheet

This stylesheet maps miniCIM information to the LDIF input format (step `[C]` in Fig. 5.1). Even if LDIF is a standard, there are variations on the strictness of its parsing in its implementations. OpenLDAP [75], the open-source LDAP implementation used by AdCIM, imposes strict spacing and line breaking, so the stylesheet has to handle these requirements. The next difficulty is mapping the CIM instance structure to LDAP; our implementation is loosely based on the DMTF recommendations in [101], which divides CIM classes into three categories depending on how they are processed: regular, auxiliary associations and structural associations.

Regular instances are the easiest to map. Each property is transposed directly to an LDAP attribute, and a new attribute, `orderedCimKeys`, is constructed by a concatenation of its class name, the names and values of key attributes (ordered alphabetically and comma-separated), and used as the entry RDN. The local root of the LDAP tree is named after the CIM namespace. The first line of Fig. 5.6b shows a DN built in this manner.

```
<!-- Auxiliar -->
<xsl:when test="contains($schemaClass//@adc:dependency,'bare')">
  <xsl:for-each select="$schemaClass/xsd:complexContent/xsd:extension/xsd:sequence/xsd:element[@adc:reference]">
    <xsl:variable name="atref" select="current()"/>
    <xsl:variable name="ock">
      <xsl:call-template name="instance2orderedCimKeys">
        <xsl:with-param name="instance" select="$instance/*[local-name()=current()/@name]"/>
        <xsl:with-param name="schema" select="$schema"/>
        <xsl:with-param name="ref">true</xsl:with-param>
        <xsl:with-param name="escaped">true</xsl:with-param>
      </xsl:call-template>
    </xsl:variable>
    <xsl:text>dn: orderedCimKeys=</xsl:text>
    <xsl:value-of select="$ock"/>
    <xsl:text>,</xsl:text>
    <xsl:value-of select="$namespace"/>
    <xsl:text>
</xsl:text>
    <xsl:text>changetype: modify
</xsl:text>
    <xsl:text>add: objectClass
</xsl:text>
    <xsl:text>objectClass: </xsl:text>
    <xsl:value-of select="translate(local-name($instance/.),'_','-')"/>
    <xsl:text>AuxClass
-

</xsl:text>
    <xsl:text>dn: orderedCimKeys=</xsl:text>
    <xsl:value-of select="$ock"/>
    <xsl:text>,</xsl:text>
    <xsl:value-of select="$namespace"/>
    <xsl:text>
</xsl:text>
    <xsl:for-each select="$instance/*">
      <xsl:if test="(local-name(current())!=$atref/@name) and (($schemaClass//*[@name=local-name(current())]) or (not($atref[@minOccurs]))))">
        <xsl:variable name="name" select="translate(concat(local-name(current()),local-name($instance/.)),'_','-')"/>
        <xsl:text>changetype: modify
</xsl:text>
        <xsl:text>add: </xsl:text>
        <xsl:value-of select="$name"/>
        <xsl:text>
</xsl:text>
        <xsl:if test="local-name($schemaClass//*[@name=local-name(current()) and (not(@adc:reference))])">
          <xsl:for-each select="current()/value">
            <xsl:value-of select="$name"/>
            <xsl:text>: </xsl:text>
            <xsl:value-of select="normalize-space(current())"/>
            <xsl:text>
</xsl:text>
          </xsl:for-each>
        </xsl:if>
        <xsl:if test="local-name($schemaClass//*[@name=local-name(current()) and @adc:reference])">
          <xsl:value-of select="$name"/>
          <xsl:text>: </xsl:text>
          <xsl:text>orderedCimKeys=</xsl:text>
          <xsl:call-template name="instance2orderedCimKeys">
            <xsl:with-param name="schema" select="$schema"/>
            <xsl:with-param name="instance" select="current()"/>
            <xsl:with-param name="ref">true</xsl:with-param>
            <xsl:with-param name="escaped">true</xsl:with-param>
          </xsl:call-template>
          <xsl:text>,</xsl:text>
          <xsl:value-of select="translate($namespace,'_','-')"/>
          <xsl:text>
</xsl:text>
        </xsl:if>
        <xsl:text>-
</xsl:text>
      </xsl:if>
    </xsl:for-each>
    <xsl:text>
</xsl:text>
  </xsl:for-each>
</xsl:when>
```

Figure 5.5: miniCIM to LDIF stylesheet for processing an auxiliary association

```
<HostedInetdService namespace="dc=udc">
 <StartMode>Automatic</StartMode>
 <Antecedent>
  <ref classname="ComputerSystem" namespace="dc=udc">
   <CreationClassName>ComputerSystem</CreationClassName
        >
   <Name>shalmaneser</Name>
  </ref>
 </Antecedent>
 <Dependent>
  <ref classname="InetdService" namespace="dc=udc">
   <SystemCreationClassName>ComputerSystem</
        SystemCreationClassName>
   <SystemName>shalmaneser</SystemName>
   <CreationClassName>InetdService</CreationClassName>
   <Name>ftp</Name>
   <Protocol>tcp</Protocol>
  </ref>
 </Dependent>
</HostedInetdService>
```

(a) Input auxiliary instance (same as Fig. 3.5)

```
dn: orderedCimKeys=InetdService.CreationClassName\=ccname
        \2CName\=ftp\2CProtocol\=tcp,dc=udc
changetype: modify
add: objectClass
objectClass: AdCIM−HostedInetdServiceAuxClass
−
dn: orderedCimKeys=InetdService.CreationClassName\=ccname
        \2CName\=ftp\2CProtocol\=tcp,dc=udc
changetype: modify
add: StartModeAdCIM−HostedInetdService
StartModeAdCIM−HostedInetdService: Automatic
−
changetype: modify
add: AntecedentCIM−HostedService
AntecedentCIM−HostedService: orderedCimKeys="
        ComputerSystem.CreationClassName=ComputerSystem,
        Name=shalmaneser", dc=udc
−
```

(b) Resulting LDIF file

```
<searchResultEntry dn="orderedCimKeys=ComputerSystem.CreationClassName\=ComputerSystem\,Name\=shalmaneser,dc=udc">
 <attr name="CreationClassNameCIM−System"> <value>ComputerSystem</value> </attr>
 <attr name="NameCIM−System"> <value>shalmaneser</value> </attr>
 <attr name="DependentCIM−HostedService">
  <value>orderedCimKeys=InetdService.CreationClassName\=ccname\2CName\=ftp\2CProtocol\=tcp,dc=udc</value>
  <value>orderedCimKeys=InetdService.CreationClassName\=ccname\2CName\=skserv\2CProtocol\=tcp,dc=udc</value>
 </attr>
 <attr name="objectClass">
  <value>CIM−ComputerSystemInstance</value>
  <value>AdCIM−HostedInetdServiceAuxClass</value>
 </attr>
 <attr name="orderedCimKeys"> <value>ComputerSystem.CreationClassName=ComputerSystem,Name=shalmaneser</value> </attr>
</searchResultEntry>

```

(c) DSML data representing a regular instance with appended auxiliary association

Figure 5.6: Representations of CIM auxiliary associations in (a) miniCIM, (b) LDIF, output from the miniCIM to LDIF stylesheet, (c) DSML data extracted from the LDAP repository

Associations have relational and non-relational attributes, the former having DN values. The simplest way of mapping associations would be to allocate a node for each association and add new attribute values for all its properties. This approach fails because LDAP does not guarantee any ordering for the values of a multivaluated attribute. The consequence is that when there is more than one instance of a given association, their non-referential attributes are mixed and become inseparable from one another, what we call "attribute aliasing".

To solve this problem associations are mapped as LDAP auxiliary classes for both ends, antecedent and dependent, pointing to each other and which can have different cardinalities. These association attributes are partitioned between those ends. The auxiliary classes for each end are attached to the associated instances adding a new value to their `objectClass` attribute. These new values define new association attributes to be used in the associated instances, but the two ends must also follow an attribute partitioning convention. Non-referential properties are mapped to the end with cardinality greater than one, usually the dependent. This avoids attribute aliasing since the attributes prone to attribute aliasing are allocated to different nodes. This mapping method is restricted to 1-to-many associations and many-to-many associations without non-referential properties, since referential attributes do not suffer from aliasing.

Fig. 5.5 shows the XSLT code to map auxiliary associations. It begins iterating over the referential properties. Each referential property points to an entry which receives a subset of the attributes. The DN of this entry is determined in lines 5–12, and a new objectClass with support for the attributes of the association is added to the entry in lines 13–25. A new record with the same DN is codified in lines 29–34. Line 35 then iterates over the properties. Line 36 is the "partition condition" that determines which attributes go to which end of the association. The condition states that if a property exists in the schema for the class, and the referential property of this end has no cardinality limitations (that is, it is not the end with maximum cardinality 1), then the attribute is added in lines 37–67. Lines 37–43 print the correct attribute name concatenating the originating class of the property and its name, and lines 44–67 process the properties depending on the type. In lines 57–62 referential properties are processed with the same `instance2orderedCimKeys` function that obtains DNs from entries.

The remaining case, many-to-many with non-referential properties, produces aliasing with the auxiliary mapping, so it is instead mapped by structural associations, which are regular instances with referential properties. Auxiliary classes with referential attributes are attached to both ends of the association for navigation help. Both of these helper attributes point to the association instance. This method of mapping could be used by all associations, but auxiliary mapping meshes better with LDAP persistence, allowing to retrieve association data without the additional queries required by structural mapping to access the separate association node.

Figure 5.6a shows an example of a miniCIM representation of the association `HostedInetdService` (shown in the CIM hierarchy of Fig. 3.2), and Fig. 5.6b part of the LDIF representation as an auxiliary association, with two entries for the Dependent association end (lines 6–10 and 11–14); the other end would differ by having only a referential attribute.

5.2.2. DSML to miniCIM Stylesheet

This stylesheet (corresponding to step `[D]` in Fig. 5.1) transforms information retrieved from the LDAP repository to miniCIM format. This process is the opposite to the one performed by the preceding stylesheet, except that the input format is not LDIF, but DSML. Fig. 5.6c shows an auxiliary association (`HostedInetdService`) attached to a CIM regular instance (`ComputerSystem`) coded in DSML. As covered in Section 5.1, it represents the same entities as LDIF, such as attributes, values and DNs using a different format.

Due to the mapping used in Section 5.2.1, each node in a DSML file can correspond to several miniCIM instances (e.g., a regular instance with several auxiliary associations attached), and the information from auxiliary associations is also broken into several nodes. Each class in a node has one corresponding `objectClass` value, with at least one regular or structural instance, and possibly several auxiliary class definitions.

The stylesheet in Fig. 5.7, similar to the previous subsection, shows the XSLT code for converting an auxiliary association from DSML to miniCIM. Lines 5–6 get the names of the referential properties of each end of the association, which are used in lines 7–22 to iterate first over the nodes pointed by one referential property, and then once more over the referential property in these nodes (since they are auxiliary associations, each node has exactly one referential property). Line 23 performs a sanity check, only passed if both referential properties point to each other. Next, lines 24–35 aggregate and consolidate the attributes from both ends, emulating the input expected from a structural class (which does not have its attributes distributed over several entries), and the function `map-result`, which processes structural entries with the aggregated attributes is called in lines 36–39. The end result of the processing of this stylesheet are miniCIM instances, which were the input of the last stylesheet, so round-tripping and the persistence of all the data represented is achieved.

5.2.3. miniCIM Schema to LDAP Schema Stylesheet

Constructing an LDAP schema (step `[G]` in Fig. 5.1) is a final step to integrate directories into the AdCIM framework. LDAP schemata, like any database schema, specify constraints about the allowed entry and data types, ensuring data integrity and semantic validity. Schemata also improve performance, and make possible more

```xml
<!--*********************** map-result-aux (class,result) ***************************-->
<xsl:template name="map-result-aux">
  <xsl:param name="class"/>
  <xsl:param name="result"/>
  <xsl:variable name="ref_prop_name_1" select="translate(concat($class/xsd:complexContent/xsd:extension/xsd:sequence/xsd:element[
        @adc:reference][1]/@name,$class/@name),'_','-')"/>
  <xsl:variable name="ref_prop_name_2" select="translate(concat($class/xsd:complexContent/xsd:extension/xsd:sequence/xsd:element[
        @adc:reference][2]/@name,$class/@name),'_','-')"/>
  <xsl:if test="$result[not (dsml:attr[@name=$ref_prop_name_1])]">
    <xsl:for-each select="$result/dsml:attr[@name=$ref_prop_name_2]/dsml:value">
      <xsl:variable name="current">
        <xsl:call-template name="escapedn">
          <xsl:with-param name="dn" select="current()"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:variable name="result_1" select="/dsml:batchResponse/dsml:searchResponse/dsml:searchResultEntry[@dn=$current or @dn=
            current()]"/>
      <xsl:variable name="pos_1" select="position()"/>
      <xsl:for-each select="$result_1/dsml:attr[@name=$ref_prop_name_1]/dsml:value">
        <xsl:variable name="current2">
          <xsl:call-template name="escapedn">
            <xsl:with-param name="dn" select="current()"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:variable name="result_2" select="/dsml:batchResponse/dsml:searchResponse/dsml:searchResultEntry[@dn=$current2 or @dn=
              current()]"/>
        <xsl:if test="$result_1 and $result_2">
          <xsl:variable name="merged_result">
            <dsml:searchResultEntry>
              <xsl:copy-of select="$result_1/dsml:attr[not(@name=$ref_prop_name_1)]"/>
              <xsl:copy-of select="$result_2/dsml:attr[not(@name=$ref_prop_name_2)]"/>
              <dsml:attr name="{$ref_prop_name_1}">
                <xsl:copy-of select="$result_1/dsml:attr[@name=$ref_prop_name_1]/dsml:value[position()]"/>
              </dsml:attr>
              <dsml:attr name="{$ref_prop_name_2}">
                <xsl:copy-of select="$result_2/dsml:attr[@name=$ref_prop_name_2]/dsml:value[$pos_1]"/>
              </dsml:attr>
            </dsml:searchResultEntry>
          </xsl:variable>
          <xsl:call-template name="map-result">
            <xsl:with-param name="class" select="exsl:node-set($class)"/>
            <xsl:with-param name="result" select="exsl:node-set($merged_result)"/>
          </xsl:call-template>
        </xsl:if>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:if>
</xsl:template>
```

Figure 5.7: DSML to miniCIM stylesheet for processing an auxiliary association

efficient indices. Their format is tightly coupled to directory implementations, so each vendor has formats incompatible with each other, despite representing almost equivalent information. In this section the process and examples of the code for obtaining a valid schema for the OpenLDAP open-source implementation is shown.

CIM and OpenLDAP schemata differ in structure, as the latter defines attributes and classes separately. Therefore, attribute names must be prepended with their original class names, and associated with a universal identifier, called Object Identifier (OID), assigned by the IANA (Internet Assigned Numbers Authority). The OpenLDAP schema format is a line-oriented flat text file, using the format shown in Fig. 5.8a. This figure shows the codification in LDAP schema format of the `InetdService` class shown in Fig. 3.2 and used as case study for this chapter. Each attribute or class definition of this example has its own line and OID. Class defi-

```
attributetype (1.3.6.1.4.1.5657.20.2.2.1.823 NAME 'CommandAdCIM-InetdService' DESC 'The command for running the server' SYNTAX
        1.3.6.1.4.1.1466.115.121.1.15 EQUALITY caseIgnoreMatch SUBSTR caseIgnoreSubstringsMatch SINGLE-VALUE)

objectclass (1.3.6.1.4.1.5657.20.2.2.3.3739 NAME 'AdCIM-InetdService' DESC '' SUP CIM-Service ABSTRACT MAY (orderedCimKeys $
        SystemCreationClassNameCIM-Service $ SystemNameCIM-Service $ CreationClassNameCIM-Service $ NameCIM-Service $
        SocketTypeAdCIM-InetdService $ ProtocolAdCIM-InetdService $ WaitAdCIM-InetdService $ WaitInstanceAdCIM-InetdService $
        UserAdCIM-InetdService $ CommandAdCIM-InetdService))

objectclass (1.3.6.1.4.1.5657.20.2.2.3.3743 NAME 'AdCIM-HostedInetdServiceAuxClass' DESC '' SUP AdCIM-HostedInetdService
        AUXILIARY MAY (AntecedentCIM-HostedService $ DependentCIM-HostedService))
```

(a) Example of LDAP schema mapping CIM instances

```
1  <!-- Auxiliary -->
2  <xsl:text>objectclass (1.3.6.1.4.1.5657.20.2.2.3.</xsl:text><xsl:value-of select="(position()-1)*3+1"/>
3  <xsl:text> NAME '</xsl:text><xsl:value-of select="translate(@name,'_','-')"/><xsl:text>' DESC '</xsl:text>
4  <xsl:value-of select='normalize-space(translate(descendant::desc,"&apos;"," "))'/><xsl:text> '</xsl:text>
5  <xsl:if test="@superclass"><xsl:text> SUP </xsl:text><xsl:value-of select="translate(@superclass,'_','-')"/>
6  </xsl:if>
7  <xsl:text> ABSTRACT </xsl:text>
8  <xsl:call-template name="print-must-may">
9     <xsl:with-param name="attributes" select="descendant::attribute[not (@reference)]"/>
10 </xsl:call-template>
11 <xsl:text>objectclass (1.3.6.1.4.1.5657.20.2.2.3.</xsl:text>
12 <xsl:value-of select="(position()-1)*3+2"/>
13 <xsl:text> NAME '</xsl:text><xsl:value-of select="translate(@name,'_','-')"/>
14 <xsl:text>AuxClass' DESC '</xsl:text>
15 <xsl:value-of select='normalize-space(translate(descendant::desc,"&apos;"," "))'/>
16 <xsl:text>' SUP </xsl:text><xsl:value-of select="translate(@name,'_','-')"/><xsl:text> AUXILIARY </xsl:text>
17 <xsl:call-template name="print-must-may">
18    <xsl:with-param name="attributes" select="descendant::attribute[@reference]"/>
19 </xsl:call-template>
```

(b) Excerpt of XSLT stylesheet to create LDAP schema auxiliary class definitions

Figure 5.8: miniCIM schema to LDAP schema

nitions can have the optional (`MAY`) and required (`MUST`) keywords. Attributes are inherited from a superclass with the `SUP` keyword. Auxiliary `objectClass` values are defined with `AUXILIARY`, and abstract classes with the `ABSTRACT` keyword.

Translating the CIM schema to this format is direct once attributes are assigned unique names. The main difference lies in the separation of attributes and classes. The type of attribute is encoded as an OID number following the keyword `SYNTAX`. An example of an attribute definition can be seen in the first entry of Fig. 5.8a. The other two entries show the definition of an auxiliary association, the first one an abstract class with the attribute definitions corresponding to the `InetdService` class, and the second one an instantiable auxiliary class derived from the first class using the `SUP` keyword.

An excerpt from the stylesheet that generates this OpenLDAP schema is shown in Fig. 5.8b. The excerpt corresponds to the auxiliary association mapping section. Line 2 declares an objectclass entity using an OID stemming from the 1.3.6.1.-

4.1.5657.20.2.23 OID branch. Since an abstract class, an auxiliary class and an instantiable class is associated to each auxiliary class in the miniCIM schema, three OIDs are associated to each one. Line 5 translates the superclass information to `SUP` format, and lines 8–10, and 17–19 call the `print-must-may` template, which formats the attributes in `MUST` and `MAY` groups depending on their optionality. Regular classes and structural associations are mapped in a similar fashion. The end schema resulting from this processing is a large number of `attributetype` and `objectclass` definitions such as the ones in Fig. 5.8a.

5.2.4. Persistence Stylesheet Benchmarks

This section details the performance results obtained while benchmarking the three XSLT stylesheets described in the previous sections. These stylesheets use techniques like caching schema files, creating indexes via the `<xsl:key/>` operator, and reducing to a minimum the nodes accessed on XPath expressions, but the greatest gain is due to the functional nature of XSLT, which makes parallelization possible without code changes with reasonably optimized stylesheets. Nevertheless, the only parallelizing XSLT processor currently available, to our knowledge, is the Intel XSLT Accelerator (from now on, PaNapa), discussed in Sun et al. [98].

We performed several benchmarks with this software using the Data Persistence Layer of AdCIM (the most performance critical), both with PaNapa and with the Saxon XSLT processor [48]. The results (measured on a quad-core Core 2 Duo machine at 2.8Ghz) can be seen in Figures 5.9 and 5.10. The graphs measure the bandwidth (given in instances/s and Megabytes/s) depending on the size of the input and the number of cores (for PaNapa, as Saxon is not multicore-aware).

In Fig. 5.9 we can see the results of measuring the throughput of the miniCIM to LDIF stylesheet both using Saxon and PaNapa with 1 to 4 cores. In this and the following graph the input is a set of miniCIM instances of which 36% are regular instances, 53% are auxiliary associations and 11% structural associations, a distribution that is usual in the CIM representation of real data and covers all the stylesheet code paths. Looking at the graph, PaNapa has a nearly linear speed up with the number of cores when the input size is large enough (still good for small sizes). The Saxon and PaNapa 1-core curves show a very similar performance, which suggests that the code is not penalized due to a performance bug present on either XSLT processor.

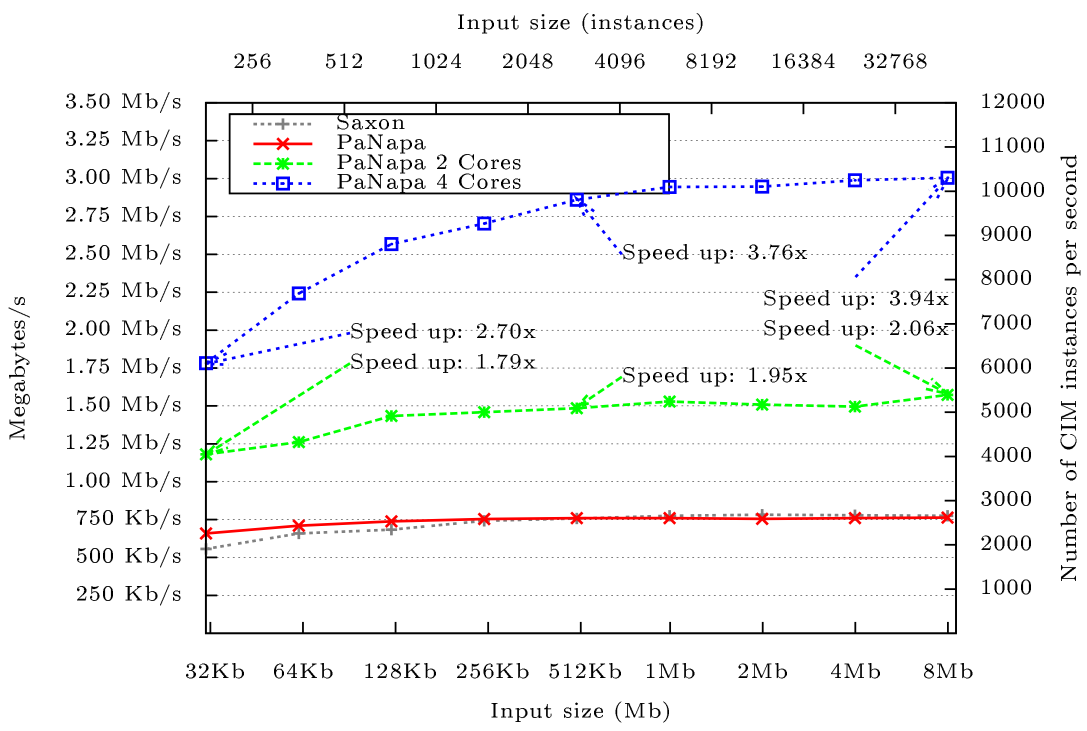


Figure 5.9: miniCIM to LDIF stylesheet efficiency

Figure 5.10 shows the efficiency of the DSML to miniCIM stylesheet. In this case, the throughputs in Mb/s are higher, but the instances/s throughput is very similar to the first graph. The reason is that DSML is more verbose than miniCIM, and the performance limiting factor for the stylesheet is not the I/O, that scales with instance size, but the CPU use, that scales with the number of instances. Saxon is also noticeably slower than PaNapa, due to the higher complexity of the processing compared with the miniCIM to LDIF stylesheet. In conclusion, it can also be seen that PaNapa's throughput scales almost linearly with multiple cores, reaching very high values for XML processing.

The third stylesheet, transforming the miniCIM schema to an OpenLDAP schema, is a transformation step only needed on schema modifications, so we measured the time of completion for the 2.17 CIM schema. This schema has 2.3Mb of size and more than 8000 entries counting attributes and classes, very large compared to the usual LDAP schema sizes (which usually only reach the low hundreds of entries), yet it only takes about 2 seconds to load the full CIM schema into OpenLDAP at restart, and has no measurable penalty on the subsequent performance of OpenLDAP.

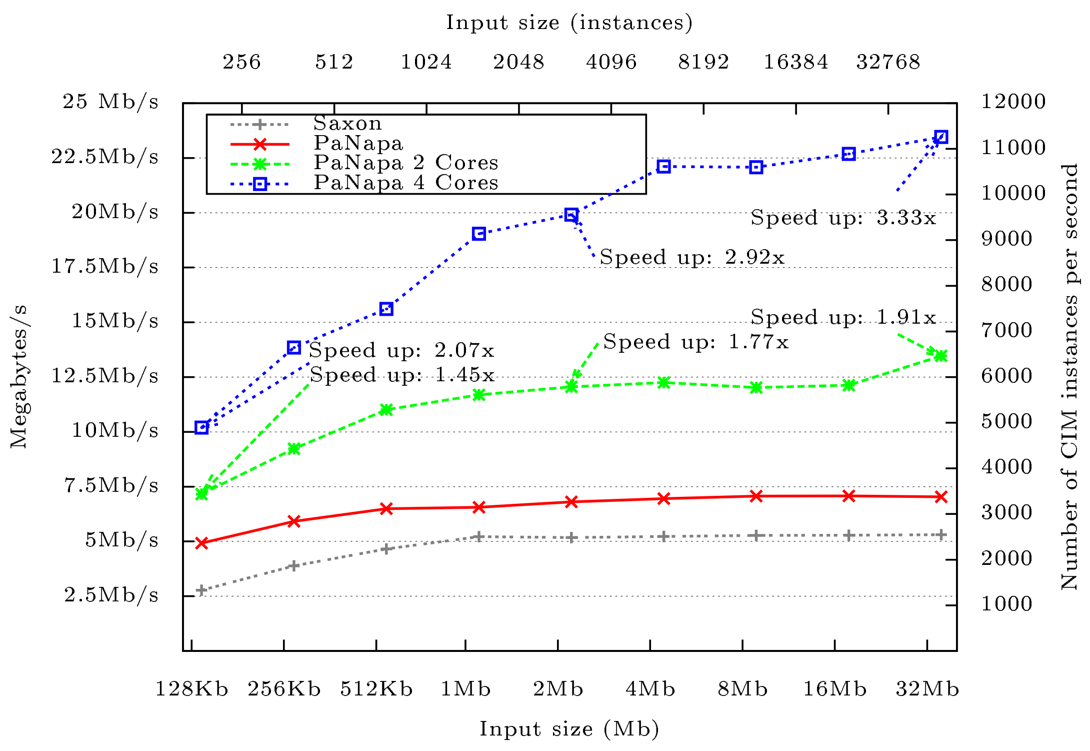


Figure 5.10: DSML to miniCIM stylesheet efficiency

5.3. Conclusions

In this chapter, we have first surveyed some basic concepts on LDAP repositories to understand their differences with “traditional” relational databases and the characteristics that made them desirable to support the Data Persistence Layer of the AdCIM framework. With this background, we have explored the transformations contained in this layer displayed in Fig. 5.1 to translate data between the repository and miniCIM instances and back, and to create an LDAP schema from a miniCIM XML Schema. Finally, we have presented some performance results that show the performance and scalability of this layer and its support of multicore environments.

Chapter 6

Application Layer

The AdCIM Application Layer comprises all AdCIM's APIs and user interfaces. Among these, there are high-level user interfaces implemented using XForms, and there are low-level Representational EState Transfer (REST) web services for exposing miniCIM data to these interfaces and external applications. XForms are created from the miniCIM XML Schema using several XSLT templates (step `[E]` in Fig. 6.1 and Fig. 2.1) and can represent and manipulate any CIM class in a general fashion via REST web services (step `[F]` in Fig. 6.1). They can be displayed in any web browser, styled with Cascading Style Sheets (CSS) and populated with data from the Data Persistence Layer. This chapter is organized as follows: Section 6.1 briefly details XForms and Section 6.2 explains the transformation processes involved in generating the final form. Next, Section 6.3 covers the REST web services and Section 6.4 gathers the chapter conclusions.

6.1. XForms

XForms [107] is a W3C standard designed as a successor to classic HTML forms. XForms improve on classic forms by using XML to represent both input and output data. These data are organized in XML instances associated with "controls", which are normally represented by common operating system widgets, such as text input boxes, combo boxes (a combination of a text input box and a drop-down option list), and other widgets commonly found on web forms. The XML instances are grouped into a "model", which is basically a collation of XML documents. This is in stark contrast to traditional forms, in which data are interspersed between

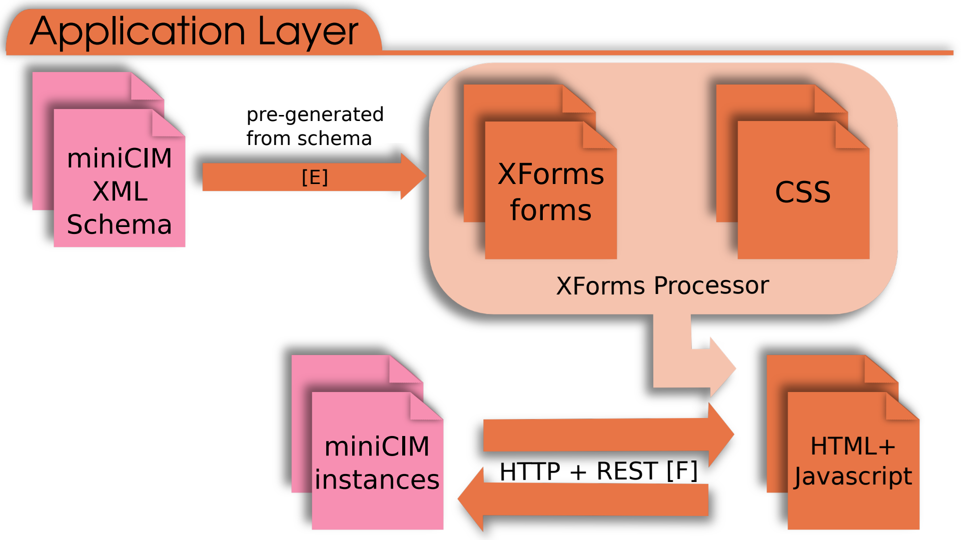


Figure 6.1: AdCIM Application Layer

several controls, a situation prone to mismatches in the data shown, which forces applications to assemble the data from each individual control.

Specific XML elements inside the model are bound to one or more controls, so changes in either one update the other. Elements can also be bound to calculations involving other elements in the model, so a form can have, for example, a control showing a total sum in real time depending on user actions. Classic web forms implement this functionality using Javascript code embedded in the page. In contrast, calculated controls are updated implicitly in XForms, avoiding common scripting errors.

Data validation is another functionality traditionally performed using scripts. The XForms models support the specification of constraints applied prior to submission. Since all output data are collated in an XML instance, there are sets of constraints difficult to perform in Javascript that are much easier and clearer to specify in XForms. The same functionality can be applied to controls dynamically, so it is possible to show a group of controls or modify their editability under a set of constraints. Wizard-dialog functionality is also simple to support, since it can be implemented with groups of controls that are shown in order, sharing the same underlying data model.

XForms have also an event model, and some of these events can be sent from

XForms element	Description
`<xf:alert/>`	Shows a modal message, normally a message box.
`<xf:bind/>`	Binds together an element of a model to a control or another element.
`<xf:insert/>`	Creates more instance data by cloning existing model elements.
`<xf:instance/>`	An explicit declaration of initial instance data in a model.
`<xf:load/>`	Loads the contents pointed by a link in a new page or replacing the current one.
`<xf:model/>`	Defines one model containing XML form data.
`<xf:recalculate/>`	Updates all the data derived by calculations in a model.
`<xf:reset/>`	Resets all models and the form to the original state.
`<xf:send/>`	Submits data to a place determined by a `<xf:submission/>` element. Several asynchronous submissions can be done during the life of a form.
`<xf:setvalue/>`	Directly sets the value of an element in a model.
`<xf:submission/>`	Defines a place to send the data of the form, its serialization, transport method and other parameters.

Table 6.1: XForms model and action elements

controls (instead of being bound to data), affecting the form state, model data or performing a submission operation. Some of these actions are described in Table 6.1.

Controls in XForms (of which a selection is shown in Table 6.2) are not necessarily represented with traditional web widgets, since they are defined by logical function (i.e., "choose an element", "trigger an action"), instead of appearance. Nevertheless, controls have an `appearance` attribute and the type of their bound data can give additional hints to the XForms interpreter to represent controls. For example, the `<xf:select1/>` XForms element specifies a control that lets the user select exactly one of its elements, usually represented as a combo box with "`normal`" appearance, and a list showing several selections with "`full appearance`"; `<xf:alert/>` represents error or warning messages (e.g., shown when validation fails), that can be realized by a message box or a box on an HTML page, and so on. These forms can also have other implementations besides web forms, since XForms provide representation-independent controls; so, for instance, a multiple choice list could be represented by an aural interface like that of an answering machine.

The XForms standard was geared for inclusion into existing web client technology, thus many implementations are plug-ins or extensions for popular browsers such as Internet Explorer or Mozilla Firefox (for example, the official XForms extension of

XForms element	Description
`<xf:input/>`	Enables free-form data entry, normally with a text input area.
`<xf:output/>`	Renders its value with no provision to enter or modify data.
`<xf:range/>`	Accepts input from a sequential range of data.
`<xf:repeat/>`	Creates as many copies of the groups of controls inside as the selected instance element.
`<xf:secret/>`	Represents a control which makes it difficult for third parties to obtain its value.
`<xf:select/>`	Allows the user to select one or more items from a group of choices, or optionally enter another value.
`<xf:select1/>`	The same as `<xf:select/>`, but limits the choice to one element.
`<xf:submit/>`	Initiates the submission of data to the server.
`<xf:textarea/>`	Similar to `<xf:input/>`, but supports multiline content, e.g., to input an email message.
`<xf:trigger/>`	Represents an action directly initiable by the user. Normally represented with a button.
`<xf:upload/>`	Uploads a file or data from external devices, such as cameras or microphones.

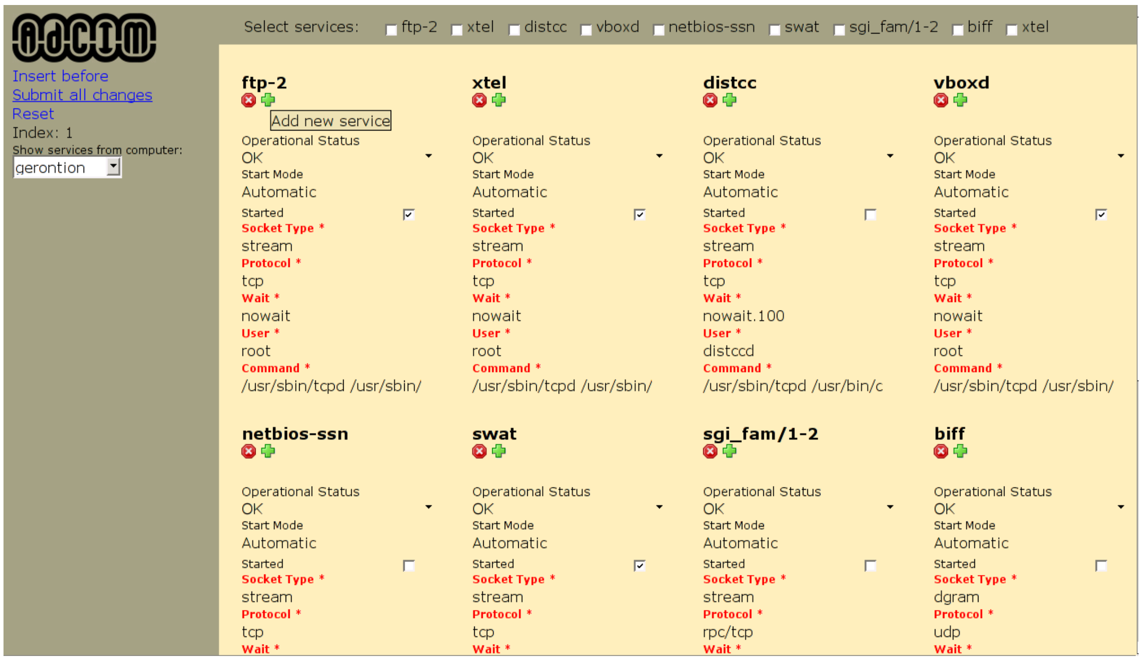
Table 6.2: XForm controls

Mozilla Firefox [64]). Some of these solutions use existing browser technologies; for example, DENG [16] implements XForms and other advanced functionalities over several browsers using client-side Javascript. There are many other client-based implementations of XForms, including those which convert the forms to legacy HTML in the server, like Chiba [11]. The implementation currently used in AdCIM is Orbeon Presentation Server [76], but due to this wealth of implementations new deployment possibilities are always possible.

6.2. XForms Use in AdCIM

XForms was chosen to implement AdCIM user interfaces due to the dynamic nature of CIM data and the need to rapidly prototype and generate applications from the schema, as part of the model-driven approach. As such, user interfaces are just another artifact of the schema which need to be generic and easily populated with miniCIM data.

XForms forms are translated on-the-fly to HTML and styled with CSS. An example of the final appearance of such an AdCIM form for Inetd services generated with XForms is shown in Fig. 6.2. The form is structured as a grid of grouped controls. Each of these groups represents an Inetd service and its associated properties, such as if it is started or its command line parameters. The values of these properties are represented as editable controls styled with CSS to resemble standard text. The icons next to the service name (actually, CSS styled buttons) allow the user to add or remove interfaces. The data used to populate the form are miniCIM data translated from entries in the LDAP repository using the XSLT stylesheet described in Section 5.2.2, previously extracted from the managed nodes using the methods seen in Chapter 4 and persisted in the repository via the stylesheet of Section 5.2.1.



Insert before
Submit all changes
Reset
Index: 1
Show services from computer:
gerontion

Select services: ftp-2 xtel distcc vboxd netbios-ssn swat sgi_fam/1-2 biff xtel

ftp-2
Add new service
Operational Status
OK
Start Mode
Automatic
Started ✓
Socket Type *
stream
Protocol *
tcp
Wait *
nowait
User *
root
Command *
/usr/sbin/tcpd /usr/sbin/

xtel
Operational Status
OK
Start Mode
Automatic
Started ✓
Socket Type *
stream
Protocol *
tcp
Wait *
nowait
User *
root
Command *
/usr/sbin/tcpd /usr/sbin/

distcc
Operational Status
OK
Start Mode
Automatic
Started
Socket Type *
stream
Protocol *
tcp
Wait *
nowait.100
User *
distccd
Command *
/usr/sbin/tcpd /usr/bin/c

vboxd
Operational Status
OK
Start Mode
Automatic
Started ✓
Socket Type *
stream
Protocol *
tcp
Wait *
nowait
User *
root
Command *
/usr/sbin/tcpd /usr/sbin/

netbios-ssn
Operational Status
OK
Start Mode
Automatic
Started
Socket Type *
stream
Protocol *
tcp
Wait *

swat
Operational Status
OK
Start Mode
Automatic
Started ✓
Socket Type *
stream
Protocol *
tcp
Wait *

sgi_fam/1-2
Operational Status
OK
Start Mode
Automatic
Started
Socket Type *
stream
Protocol *
rpc/tcp
Wait *

biff
Operational Status
OK
Start Mode
Automatic
Started
Socket Type *
dgram
Protocol *
udp
Wait *

Figure 6.2: Complete CSS styled XForms form for Inetd services

In order to create a XForms form (step `[E]` in Fig. 6.1 and Fig. 2.1) from the miniCIM XML Schema, a XSLT stylesheet (shown in Fig. 6.3) gets as input an association and the two classes pointed by it, and returns as XForms instances the properties of the dependent class, a blank instance of this class, and the appropriate XForms controls for showing and modifying these class properties. These instances are passed to the XForms form template shown in Figs. 6.4, 6.5 and 6.6. This template has XSLT commands that copy the instances in their correct places, creating a form for the specified class with the appropriate fields to manipulate its properties, and with functionality to add or remove form fields dynamically. We denote the final result of these processes as pregeneration. With it, any instance from a class defined in the schema can be edited automatically by a web form.

Another benefit of pregeneration is that the values allowed for every property are populated from the schema, so that the user is required to input valid values.

```
<instances>
  <view_class><xsl:value-of select="$classname"/></view_class>
  <assoc_class><xsl:value-of select="$assoc_classname"/></assoc_class>
  <related_class><xsl:value-of select="$related_classname"/></related_class>
  <instance_fields>
   <selected/>
   <delselected/>
   <fields>
     <xsl:copy-of select="$ofields"/>
   </fields>
   <fields2>
     <xsl:for-each select="exsl:node-set($ofields)/*">
       <xsl:element name="{local-name(current())}"/>
     </xsl:for-each>
   </fields2>
   </instance_fields>
   <instance_new>
     <xsl:copy-of select="document('input:new_instance')"/>
   </instance_new>
   <binds>
     <xf:bind id="bind_A" nodeset="instance('instance_0')//{$classname}">
       <xsl:for-each select="exsl:node-set($ofields/*)">
         <xf:bind id="{concat('bind_',local-name(current()))}" nodeset="{local-name(current())}">
           <xsl:if test="@type">
             <xsl:attribute name="type"><xsl:value-of select="@type"/></xsl:attribute>
            </xsl:if>
            <xsl:if test="@key">
              <xsl:attribute name="required">true</xsl:attribute>
            </xsl:if>
         </xf:bind>
       </xsl:for-each>
     </xf:bind>
   </binds>
   <controls>
     <xf:repeat id="rep1" class="body" bind="bind_A">
       <html:div class="status">
         <html:div class="items">
           <xf:input ref="./Name" class="title">
             <xf:label class="title-label">Name: </xf:label>
           </xf:input>
           <xsl:for-each select="exsl:node-set($ofields/*)">
             <xsl:variable name="name" select="local-name(current())"/>
             <html:div class="control">
             <xsl:choose>
             <xsl:when test="values">
               <xf:select ref="{$name}" id="input_{$name}">
                 <xf:label id="label_{$name}"><xsl:value-of select="concat($name,': ')"/></xf:label>
                 <xf:itemset nodeset="instance('instance_fields')/fields/{local-name(current())}/values/value">
                   <xf:label ref="."/> <xf:value ref="."/>
                 </xf:itemset>
               </xf:select>
             </xsl:when>
             <xsl:otherwise>
               <xf:input ref="{$name}" id="input_{$name}">
                 <xf:label id="label_{$name}"><xsl:value-of select="concat($name,': ')"/></xf:label>
               </xf:input>
             </xsl:otherwise>
             </xsl:choose>
             </html:div>
           </xsl:for-each>
         </html:div>
       </html:div>
     </xf:repeat>
   </controls>
</instances>
```

Figure 6.3: XSLT code transforming miniCIM XML schema to XForms instances

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/root">
    <html:html>
      <html:head>
        <html:link rel="stylesheet" type="text/css" href="myforms/xforms.css"/>
        <xf:model id="model_0">
          <xf:instance id="instance_0">
            <xsl:copy-of select="instance_0"/>
          </xf:instance>
          <xf:instance id="instance_1">
            <xsl:copy-of select="instance_1"/>
          </xf:instance>
          <xf:instance id="instance_1_sel">
            <instance/>
          </xf:instance>
          <xf:instance id="instance_new_serv">
            <xsl:copy-of select="instances/instance_new"/>
          </xf:instance>
          <xf:instance id="instance_fields">
            <xsl:copy-of select="instances/instance_fields"/>
          </xf:instance>
          <xf:instance id="instance_select">
            <instance/>
          </xf:instance>
          <xsl:copy-of select="instances/binds/*"/>
          <xf:submission action="http://127.0.0.1:8080/chiba-web-1.0.0/jsp/debug-instance.jsp" id="submission_0" method="post
                "/>
          <xf:submission action="http://127.0.0.1:8080/ops/ldap2?assoc={instances/assoc_class}" ref="instance('instance_1')/CIM/{
                instances/related_class}[Name=instance('instance_1_sel')]" id="submission_comp" method="post" replace="
                instance" instance="instance_0"/>
          <xf:message level="modeless" ev:event="xforms-compute-exception">xforms-compute-exception</xf:message>
          <xf:message level="modeless" ev:event="xforms-binding-exception">xforms-binding-exception</xf:message>
        </xf:model>
      </html:head>
      <html:body>
        <xxforms:dialog id="dialog1" appearance="full" level="modeless" close="true" draggable="true">
          <xf:label>Modify Fields</xf:label>
          <html:table>
            <html:tr>
              <html:td>
                <xf:select1 appearance="minimal" ref="instance('instance_fields')/selected" id="select1_field">
                  <xf:label id="label_486">Available Fields:</xf:label>
                  <xf:itemset nodeset="instance('instance_fields')/fields/*
                      [not (local-name()=(for $x in instance('instance_0')//{instances/view_class}[index('rep1')]/* return
                              local-name($x)))]">
                    <xf:label value="concat(local-name(.),if (@key) then '[K]' else'')"/>
                    <xf:value value="local-name(.)"/>
                  </xf:itemset>
                </xf:select1>
              </html:td>
              <html:td>
                <xf:trigger>
                  <xf:label>Insert New Field</xf:label>
                  <xf:action ev:event="DOMActivate">
                    <xf:insert ev:event="DOMActivate" nodeset="instance('instance_0')//{instances/view_class}[index('rep1')
                          ]/*" origin="instance('instance_fields')/fields2/*[local-name(.)=instance('instance_fields')/selected]"
                          at="last()" position="after"/>
                    <xf:refresh/>
                    <xf:rebuild/>
                    <xf:recalculate/>
                  </xf:action>
                  <xf:setvalue value="------"/>
                </xf:trigger>
              </html:td>
            </html:tr>
            <html:tr>
```

Figure 6.4: Generalized XForms form code (part I)

XForms instances are arbitrary XML data, so these forms interface directly with AdCIM's web service infrastructure, via Uniform Resource Locator (URL) directions. The output of the form is guaranteed to be well-formed miniCIM instances, so that the server does not waste bandwidth on validation failures.

```
      <html:td>
        <xf:select1 appearance="minimal" ref="instance('instance_fields')/delselected" id="select2_field">
          <xf:label id="label_586">Shown Fields:</xf:label>
          <xf:itemset nodeset="instance('instance_fields')/fields/*
               [(local-name()=(for $x in instance('instance_0')//{instances/view_class}[index('rep1')]/* return
                    local-name($x)))]">
            <xf:label value="concat(local-name(.),if (@key) then '[K]' else'')"/>
            <xf:value value="local-name(.)"/>
          </xf:itemset>
        </xf:select1>
      </html:td>
      <html:td>
        <xf:trigger>
          <xf:label>Delete Field</xf:label>
          <xf:action ev:event="DOMActivate">
            <xf:delete ev:event="DOMActivate" nodeset="instance('instance_0')//{instances/view_class}[index('rep1')
                 ]/*[local-name(.)=instance('instance_fields')/delselected]" at="1" position="after"/>
            <xf:refresh/>
            <xf:rebuild/>
            <xf:recalculate/>
          </xf:action>
        </xf:trigger>
      </html:td>
    </html:tr>
  </html:table>
</xxforms:dialog>
<html:div class="header">
  <xf:select class="topselect" appearance="full" ref="instance('instance_select')" incremental="true">
    <xf:itemset bind="bind_A">
      <xf:label ref="Name"/>
      <xf:value ref="Name"/>
    </xf:itemset>
  </xf:select>
</html:div>
<html:div class="sidebar">
  <html:div>
    <html:img src="myforms/adcim_logo.png" height="34" width="100"/>
  </html:div>
  <html:div>
    <xf:trigger appearance="minimal" class="link">
      <xf:label class="sidebar-label">Insert before</xf:label>
      <xf:insert ev:event="DOMActivate" nodeset="/CIM/{instances/view_class}" origin="instance('instance_new_serv')/{
            instances/view_class}" at="last()" position="after"/>
    </xf:trigger>
  </html:div>
  <html:div>
    <xf:submit appearance="minimal" class="link" submission="submission_0">
      <xf:label class="sidebar-label" id="label_84">Submit all changes</xf:label>
    </xf:submit>
  </html:div>
```

Figure 6.5: Generalized XForms form code (part II)

The instance generation stylesheet in Fig. 6.3 begins transferring its inputs (CIM association and associated classes) to the ouput (lines 2–4). In line 9 it copies the dependent class properties in variable `$ofields`, and then copies their names to another element of the instance (lines 11–15), and also copies a blank template for that instance (line 18). The stylesheet then proceeds to write `<xf:bind/>` bindings to relate controls with the miniCIM properties they represent (lines 20–33), and creates these controls in lines 34–64 inside a `<xf:repeat/>` element, so there is a group of controls per miniCIM instance. The `Name` property, present in all CIM classes, is put as header of the controls (lines 38–40), and properties with the `Values` qualifier in the schema are represented as a `<xf:select/>` element, normally represented with a combo box and loaded with the allowed values. The rest of the properties are represented with `<xf:input/>` elements, which normally are mapped to a text

```
110                     <html:div>
111                         <xf:trigger appearance="minimal" class="link">
112                             <xf:label class="sidebar-label">Reset</xf:label>
113                             <xf:reset ev:event="DOMActivate"/>
114                         </xf:trigger>
115                     </html:div>
116                     <html:div>
117                         <xf:trigger appearance="minimal" class="show-drop-down" id="show-trigger">
118                             <xf:label>
119                                 <html:span>Modify Fields...</html:span>
120                                 <html:img src="/apps/xforms-sandbox/samples/images/dialog-down.png"/>
121                             </xf:label>
122                             <xxforms:show ev:event="DOMActivate" dialog="dialog1" neighbor="show-trigger" constrain="true"/>
123                         </xf:trigger>
124                     </html:div>
125                     <html:div>
126                         <xf:label class="title">
127                             Standard CIM form:
128                             <xf:output value="index('rep1')"/>
129                         </xf:label>
130                     </html:div>
131                     <html:div>
132                         <xf:select1 ref="instance('instance_1_sel')" incremental="true">
133                             <xf:label class="sidebar-label">Show services from computer:</xf:label>
134                             <xf:itemset nodeset="instance('instance_1')//{instances/related_class}">
135                                 <xf:value ref="Name"/>
136                                 <xf:label ref="Name"/>
137                             </xf:itemset>
138                             <xf:action ev:event="xforms-value-changed">
139                                 <xf:send submission="submission_comp"/>
140                                 <xf:refresh/>
141                                 <xf:rebuild/>
142                                 <xf:recalculate/>
143                             </xf:action>
144                         </xf:select1>
145                     </html:div>
146                 </html:div>
147                 <html:div class="total">
148                     <xsl:copy-of select="instances/controls"/>
149                 </html:div>
150             </html:body>
151         </html:html>
152     </xsl:template>
153 </xsl:stylesheet>
```

Figure 6.6: Generalized XForms form code (part III)

input box, but can vary depending on the property type. For example, properties representing date values are represented with a Javascript-generated calendar.

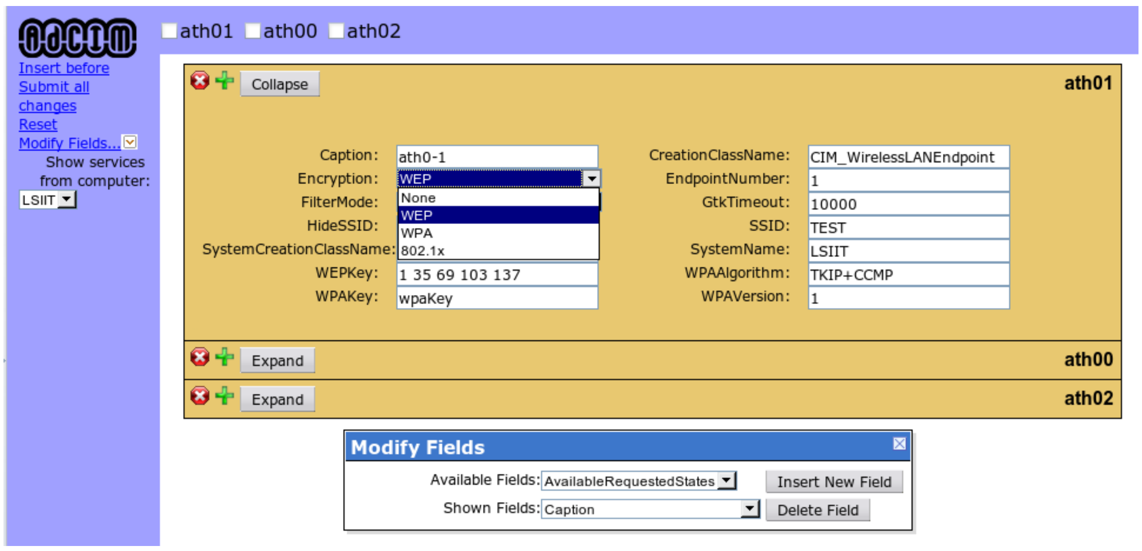
The generated instances are passed to an XForms template with XSLT commands, which starts loading the CSS code of the form (Fig. 6.4, line 7), and declaring the XForms form model in lines 8–32, including the copy of the `<xf:bind/>` instructions created in the last stylesheet (line 27), and a `<xf:submission/>` element to support the change of the shown instances when another antecedent end for the association is selected (line 29).

The template then declares an Orbeon-exclusive dialog container (from Fig. 6.4, line 35 to Fig. 6.5, line 86). This dialog is a modal draggable window dialog. Dialogs are not part of the XForms standard, but they improve the presentation and the controls inside the dialog can still be easily displayed in a standard way, so in the end it is a small cosmetic addition that does not cause real incompatibilities with other

XForms implementations. The controls inside this dialog support the functionality to add and remove form fields (and their associated controls) dynamically. For example, lines 40 to 47 create a `<xf:select1/>` field to select available instance fields, and lines 64–71 a similar list with the displayed fields. Both lists are updated dynamically with field changes. Additionally, lines 50 to 59 define the control to dynamically add fields, and lines 74 to 82 the control to delete them. There is only one control in the header, a `<xf:select/>` control (lines 88–93) that shows one checkbox per miniCIM instance to make their selection faster. There are also controls in the sidebar (see Fig. 6.2), but these are related with adding or deleting miniCIM instances and form state; so, for example, lines 100 to 103 insert a new instance before the current one. Lines 111–114 in Fig. 6.6 define a `<xf:trigger/>` control (see Table 6.2) to reset the form, and lines 117–123 another one that brings up the dialog touched before. The last element of the sidebar, in lines 132–144 creates a `<xf:select1/>` control which allows changing the association antecedent and retrieving the dependent instances using the submission of line 29 in Fig. 6.4. Finally, the most important part of the template is in line 148, which copies the actual controls pregenerated in Fig. 6.3.

The resulting form is styled and given layout with CSS like a standard web page. The main function of CSS in the CSS+XForms combination is customizing the representation of the different possible states of a control. Control representation can also be changed using CSS coupled with the `appearance` attribute. A practical use of this is representing `trigger` controls as hyperlinks instead of buttons. This requires setting the `appearance` attribute to `minimal` and changing the text appearance to mimic a link. A further refinement of this technique is to substitute a `trigger` with a custom image. The resulting code is cleaner and the original `trigger` text is still on the page code, and these images can be modified without any change to the underlying form. Presentation concerns are thus separated from the semantic-centered markup representation, leveraging existing web designer knowledge and promoting integration with web page content.

The form shown in Fig. 6.2 for managing Inetd services was pregenerated using the stylesheets described above, just like the form for managing wireless mesh routers (an AdCIM application scenario covered in Chapter 9) that can be seen in Fig. 6.7.



Show services from computer: LSIIT

ath01

Caption:	ath0-1	CreationClassName:	CIM_WirelessLANEndpoint
Encryption:	WEP	EndpointNumber:	1
FilterMode:	None	GtkTimeout:	10000
HideSSID:	WEP	SSID:	TEST
SystemCreationClassName:	WPA	SystemName:	LSIIT
	802.1x		
WEPKey:	1 35 69 103 137	WPAAlgorithm:	TKIP+CCMP
WPAKey:	wpaKey	WPAVersion:	1

ath00

ath02

Available Fields: AvailableRequestedStates

Shown Fields: Caption

Figure 6.7: Complete CSS styled XForms form for managing wireless mesh routers

AdCIM Interaction Model

The interactivity and look&feel of AdCIM web forms mimic that of typical desktop applications. Another way of achieving this would be using Asynchronous Javascript and XML (AJAX) [81], a web application paradigm based on Javascript and XML data transfers decoupled from the user interface (i.e., page information can be updated without reloading). This relegates servers to provide raw XML data on demand to clients which are responsible for the view and controller roles. The prime disadvantages of AJAX for its use in AdCIM are that the script developer must code data transfers explicitly (or use a third-party library), form data must be converted from and to XML in the page code, and calculated fields and data dependences between controls must be updated manually.

These drawbacks are overcome by the use of XML in XForms and by its data dependency model. Thus, data dependences and constraints are specified in the model and automatically enforced, changes in instances are propagated to controls (and vice versa), and initial field-population can be done from files, URLs or web services. Besides, XForms provide a similar functionality to AJAX by using instance substitution. For example, when the value selected in the left combo box of the form of Fig. 6.2 is changed, an `xforms:submit` event is fired, sending a submission (the name of the selected computer) to a REST -format URL. The result from this submission is used to replace the data bound to the right interface pane and the top check box group (representing service selection in Fig. 6.2), which are updated accordingly, as XForms have the added advantage of declarativeness.

Regardless of the underlying interface and interaction technology, the use of REST and the data-centric web approach of AdCIM guarantees that management data are not locked in a concrete web application, and can be integrated and accessed programmatically.

6.3. Web Data Interface

AdCIM exposes miniCIM data to the forms in the Application Layer and external components via a REST web service (step `[F]` in Fig. 6.1). These services are tasked with receiving commands encoded in URLs, accessing the LDAP repository to retrieve and store the instances affected by the command, and translating these instances into/from miniCIM using the XSLT stylesheets described in Chapter 5.

REST [30] is a type of XML messaging interface that uses exclusively the HTTP protocol operations and semantics [62], and is presented as a simpler alternative to SOAP web services, increasingly perceived as convoluted. In REST, objects and associations are accessed using standard HTTP operations. For example, the URL to retrieve all computer system objects is "`http://localhost:8080/adcim/adcim?-class=ComputerSystem`"; and the URL to retrieve all Inetd managed services from the computer system `pc1` is "`http://localhost:8080/adcim/assoc?association=Hosted-InetdService&end=antecedent&CreationClassName=ComputerSystem&Name=pc1`". This URL signals the service to traverse the `HostedInetdService` association from the `ComputerSystem` end and returns the `Service` instances at the other end (see Fig. 3.2). Objects are modified with the `PUT` HTTP operation, which creates or updates CIM objects stored in the network nodes and afterwards in the repository.

While SOAP represents an opaque, stateful and complex view of web services, in which data resources are not directly addressable and involves the generation of a SOAP envelope (and pre-generated stub code), REST web services assign a unique addressable HTTP Uniform Resource Identifier (URI) to each resource, and employ the `GET`, `POST`, `PUT` and `DELETE` operations for data access. The resultant interface is a noun-based interface in which the nouns are XML-represented entities and the verbs are HTTP operations, as opposed to the method-centric Remote Procedure Call (RPC)-like SOAP approach and its focus on object and type representation. As REST uses HTTP as transport protocol, it is stateless, obviating the need for state keeping in both client and server.

REST renders AdCIM data interfaces very easy to use and maintain, specially since schema changes do not affect them, and invocation using XForms is direct, using only HTTP accesses.

6.4. Conclusions

This chapter has covered the top layer of the AdCIM framework, the Application Layer. This layer covers both user and application interfaces. The user interfaces are implemented using XForms, so the chapter began with background on this technology in which the portable, general and XML-centric nature of XForms was explained. The following section covered the use of XForms in AdCIM, and presented an XSLT stylesheet that can generate web forms “on-the-fly” from CIM Schema definitions. Finally, this chapter touched upon the REST web services used as application interfaces and their invocation. These interfaces unite to give a very flexible, portable, extensible and very customizable Application Layer that makes the development of CIM-enabled management applications much easier.

Part II

Advanced Applications

Introduction to Part II

This Part of the Thesis covers several real world applications of the AdCIM framework. These applications not only validate its usefulness in complex environments and prove its extensibility, but they also give a unique insight into the operation of the framework and its application.

The first application, in Chapter 7, is an extension of the System Data Layer to import Sendmail configuration files. This configuration is notorious for being complex and difficult to manage and, as such, a good case study for our framework.

The second application in Chapter 8 is the application of the AdCIM framework to Grid management and its integration inside the Globus Toolkit, and, concretely, in its Monitoring and Discovery Subsystem. As Grid infrastructures are a good example of heterogeneous distributed systems, it is also an interesting application field.

The last application, in Chapter 9, is the most complex one. It is concerned with Wireless Mesh Networks (WMN), which extend wireless LANs to compete with classical wired LANs, offering self-organizing and self-healing capabilities. Concretely, we have modelled the configuration of routers in such networks. We have chosen this case as a good place to introduce AdCIM's ontology support. Since CIM is a good taxonomy of the management domain of discourse, we have extended it to support logical deduction processes. These processes, capable of configuration diagnosis, validation and prediction, combined with the descriptive properties of the CIM model and the functionality of AdCIM, offer limitless possibilities.

Chapter 7

Advanced Configuration Extraction: the Sendmail File

As covered in Section 4.1, there are many flat file configuration formats which follow a simple line- or section-oriented structure, with only small differences. Nevertheless, in some cases, there are configuration formats that have grown very complex with time, to the point of being largely handled using macro languages.

An extreme case is `sendmail.cf`, the configuration file for the mail delivery system Sendmail. This file is considered to be an example of unfriendliness and complexity, so it has become common to use an intermediate file written in the M4 macro processing language [49] from which to generate a final `sendmail.cf`. The original `sendmail.cf` file is currently considered so low-level that the reference book about Sendmail configuration [15] advises against direct modification, as new versions could radically change its format. The configuration is thus managed with the intermediate M4 macro file, `sendmail.mc`.

This chapter presents an example of extension of the System Data Layer (see Chapter 4) to manage the Sendmail configuration format, shown in Section 7.1. The CIM model extension developed to represent it is detailed in Section 7.2, and the process to transform the configuration into this representation is covered in Section 7.3. Finally, conclusions are drawn in Section 7.4.

```
1  divert(-1)dnl
2  dnl #
3  dnl # This is the sendmail macro config file for m4. If you make changes to
4  dnl # /etc/mail/sendmail.mc, you will need to regenerate the
5  dnl # /etc/mail/sendmail.cf file by confirming that the sendmail-cf package is
6  dnl # installed and then performing a
7  dnl #
8  dnl # make -C /etc/mail
9  dnl #
10 dnl #include(‘/usr/share/sendmail-cf/m4/cf.m4’)dnl
11 divert(0)
12 VERSIONID(‘setup for Red Hat Linux’)dnl
13 OSTYPE(‘linux’)dnl
14 dnl #
15 dnl # default logging level is 9, you might want to set it higher to
16 dnl # debug the configuration
17 dnl #
18 dnl define(‘confLOG_LEVEL’, ‘9’)dnl
19 dnl #
20 dnl # Uncomment and edit the following line if your outgoing mail needs to
21 dnl # be sent out through an external mail server:
22 dnl #
23 define(‘SMART_HOST’,‘smtp.myisp.com’)dnl
24 dnl #
25 .............
26 FEATURE(‘no_default_msa’,‘dnl’)dnl
27 ............
28 LOCAL_RULE_0
29 R$* <@ $=w . $=m> $* $#local $: $1 @here.ourdomain
30 ................
31 dnl # The following causes sendmail to additionally listen to port 465, but
32 dnl # starting immediately in TLS mode upon connecting. Port 25 or 587 followed
33 dnl # by STARTTLS is preferred.
34 dnl #
35 DAEMON_OPTIONS(‘Port=smtps, Name=TLSMTA, M=s’)dnl
36 ................
37 MAILER(smtp)dnl
38 MAILER(procmail)dnl
```

Figure 7.1: Example entries of a `sendmail.mc` file

7.1. Configuration Format

The organization of the `sendmail.mc` file (an example can be seen in Fig. 7.1) is basically line-based. It has two types of constructs: built-ins (in lowercase) and macros (normally in uppercase). The most used built-in instruction is `dnl`, which discards all characters until the next line (including newline characters). Its main purpose is to avoid blank lines in the output, but here is used also as a comment line marker (e.g. in lines 2–10). The `divert` built-in (line 1) also redirects all text following it (and discards it when called with a negative number) and thus serves as a block comment marker.

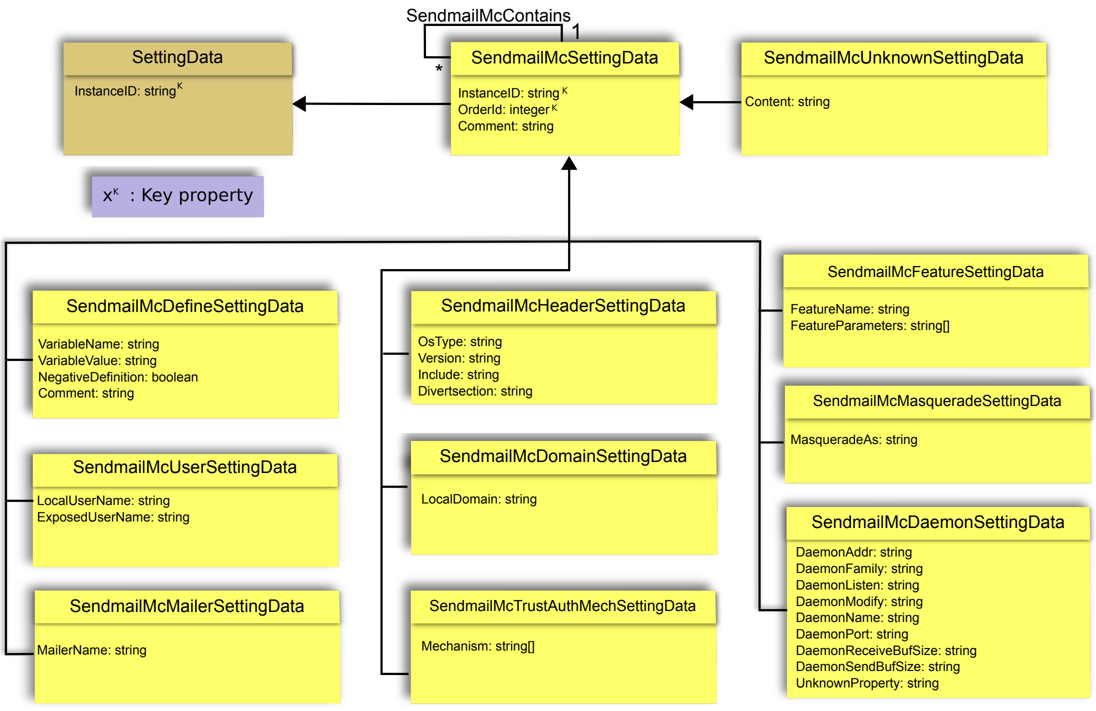


Figure 7.2: CIM representation of the Sendmail configuration

After the comment header comes the mandatory `VERSIONID` macro, which helps to differentiate between configurations, and `OSTYPE`. `define`, the most important M4 built-in, appears in line 23. It is used for setting variables or macro expansions (there is also `undefine` for negative definitions). `FEATURE` in line 26 is similar to `define`, but accepts several arguments depending on the first argument, the feature name. `DAEMON_OPTIONS` (line 35) also accepts many arguments concerned with System Log configuration, but as comma-separated name-value pairs. In a more abstract level of configuration, macros like `MAILER` (lines 37–38) expand to entire blocks of configuration which can be tuned with appropriate `FEATURE` and `define` built-ins.

The `sendmail.mc` format also has macros to write low level, `sendmail.cf` style configuration, like `LOCAL_RULE_0` (lines 28–29). This instruction redefines rule group 0. In Sendmail rule groups are used for address rewriting tasks, and are defined using a special rule language. For example, the rule in line 29 rewrites local addresses to include also the local domain.

7.2. CIM Representation

There are multiple non-trivial ways of representing this configuration meaningfully in CIM. The representation used in this work (shown in Figure 7.2) was developed with these principles in mind:

- Seamless round-tripping: No relevant information should be discarded from the configuration. The process from configuration to CIM and back should not be destructive and should conserve ordering.
- Support for unknown constructs: The mapping should handle and gracefully represent unknown parts of the configuration.
- Security and efficiency: As part of supporting unknown data, these data should be coded to avoid injection attacks and compressed to maintain efficiency.

The representation derives from the CIM standard `SettingData` class, which represents settings, and has a free-form string as key. The top class of the CIM mapping is `SendmailMcSettingData`, which adds a key property that stores the order of a block in the file, and a `Comment` property which stores the `dnl` segments preceding the represented construct. This class contains the rest of the subclasses via the `SendmailMcContains` association. There are two types of subclasses, the class of known components of the configuration and the class for unknown components (`SendmailMcUnknownSettingData`).

Known classes correspond with the constructs mentioned in the previous section. The heading `divert` block and version and operating system are grouped inside the class `SendmailMcHeaderSettingData`; `define` and `undefine` are also aggregated in class `SendmailMcDefineSettingData`, and the rest of constructs are modelled individually, with concrete parameters in the case of `SendmailMcDaemonSettingData` and string arrays in the case of `SendmailMcFeatureSettingData`.

`SendmailMcUnknownSettingData` obeys the three design principles mentioned at the beginning of this section by representing unknown constructs as a string, compressing it and codifying the result in Base64 to be storable in repositories like LDAP (which does not allow strings with arbitrary characters). This encoding also enforces the third design principle, making secure the unknown (and untrustable) information. Representing unknown information in this manner also avoids breakage when new constructs appear in future versions of the configuration or are not recognized

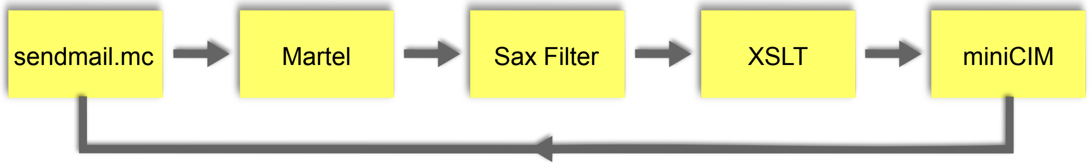


Figure 7.3: Process to transform the Sendmail configuration to miniCIM

```
def divert():
        return Martel.Re("(?P<divert>divert[(]-?\d+[)]((?!divert[(]-?\d[)]).*\R*)*divert[(]-?\d+[)]\s*)")+dnl()
def include():
        return Martel.Re("(?P<include>include[(]`*(?P<include-file>[^)',]*)'*[)])\s*")+dnl()
def versionid():
        return Martel.Re("VERSIONID[(]`*(?P<version>[^')]*)'*[)]\s*")+dnl()
def ostype():
        return Martel.Re("OSTYPE[(]`*(?P<ostype>[^')]*)'*[)]\s*")+dnl()
def define():
        return Martel.Re("(?P<define>define[(]`*(?P<variable>[^)',]*)'*(,\s*`*(?P<value>[^'`)]*)'*)?)[)]\s*")+dnl()
def undefine():
        return Martel.Re("(?P<undefine>undefine[(]`*(?P<variable>[^)',]*)'*[)])\s*")+dnl()
def feature():
        return Martel.Re("(?P<feature>FEATURE[(]`*(?P<name>[^)'`,]*)'*(,\s*`*(?P<parameter>[^,')']*)'*){0,9}[)])\s*")+dnl()
def exposed():
        return Martel.Re("EXPOSED_USER[(]`*((?P<exposed_user>[^'),]*)[,']+\s*)+[)]\s*")+dnl()
def daemon_options():
        return Martel.Re("(?P<doptions>DAEMON_OPTIONS[(]`*(\s*(?P<option_name>[^`),'=]*)=(?P<option_val>[^`),'=]*),?)+'*[)])\
             s*")+dnl()
....
def unknown():
        return Martel.Re("(?P<unknown>(?!^[\s]*$)^.*\R*)\s*")+dnl()
def dnl():
        return Martel.Opt(Martel.Re("(?P<dnl>(dnl.*\R)+)\s*"))

fields=Martel.Rep(Martel.Alt(divert(),include(),versionid(),ostype(),define(),undefine(),feature(),exposed(),daemon_options(),local_domain(),
        local_user(),input_mail_filter(),masquerade_as(),masquerade_domain(),trust_auth_mech(),mailer(),unknown()))
format=Martel.Group("input",Martel.Group("data",fields))
XMLstr= StringIO()
parser = format.make_parser()
parser.setContentHandler(SendmailContentHandler(XMLstr))
parser.parseFile(open(sys.argv[1]))
print(XMLstr.getvalue())
```

Figure 7.4: Excerpts from the Martel program used for parsing `sendmail.mc` to XML

correctly. In these cases, the information is handled and stored transparently as unknown while it is not supported in AdCIM. This encoding process is also applied to `dnl` and `divert` comment blocks.

7.3. Transformation Process

The process to transform the format shown in Section 7.1 to the CIM mapping in Fig. 7.2 is outlined in Fig. 7.3. First, the `sendmail.mc` text file is processed using Martel expressions such as the ones in Fig. 7.4. This step creates an initial XML tree that is refined and tweaked with a SAX filter (shown in Fig. 7.5), which fuses certain types of adjacent nodes (like comment nodes) and implements Base64 codification. Finally, the result is processed by an XSLT stylesheet (partly shown in Fig. 7.6)

```python
eliminate_mixed_content = ['data','define','feature','include','daemon_options','undefine','trust_auth_mech','input_mail_filter']
eliminate_mixed_content_2 = ['version','ostype','exposed_user','local_domain','local_user','masquerade_as','masquerade_domain','mailer','unknown']
b64_elements = ['dnl','divert','unknown']

class SendmailContentHandler(saxutils.XMLGenerator):
    "insert the CIM schema in the <input> element"

    def __init__(self, output=sys.stdout, encoding='UTF-8'):
        self._out = output
        self._encoding = encoding
        self._nomixed = 0
        self._unknownend = 0
        self._b64=0
        saxutils.XMLGenerator.__init__(self)

    def startElement(self,name,attrs):
        prefix, local=SplitQName(name)
        if self._unknownend and local!='unknown':
            self._out.write("</unknown>")
            self._unknownend=0
        if (local!='unknown' or not self._unknownend) and local!='input':
            saxutils.XMLGenerator.startElement(self,name,attrs)
        if local == 'input':
            self._out.write("<input filename='")
            self._out.write(os.path.basename(sys.argv[1]))
            self._out.write("'>")
            with open("cimxml-2.17-schema-opt-pre.xml") as schema:
                for line in schema:
                    self._out.write(line)
        if filter(lambda x: local == x,b64_elements):
            self._b64=1
        if filter(lambda x: local == x,eliminate_mixed_content):
            self._nomixed=1
        elif filter(lambda x: local == x,eliminate_mixed_content_2):
            self._nomixed=0
        elif self._nomixed>=1:
            self._nomixed+=1

    def characters(self,ch):
        if self._nomixed != 1:
            if self._b64:
                saxutils.XMLGenerator.characters(self,base64.b64encode(zlib.compress(ch,6)))
            else:
                saxutils.XMLGenerator.characters(self,ch)


    def endElement(self,name):
        self._emptydnl=0
        prefix, local=SplitQName(name)
        if filter(lambda x: local == x,b64_elements):
            self._b64=0
        if local == 'unknown':
            self._unknownend=1
        else:
            saxutils.XMLGenerator.endElement(self,name)
        if filter(lambda x: local == x,eliminate_mixed_content+eliminate_mixed_content_2):
            self._nomixed=1
        elif self._nomixed>=1:
            self._nomixed-=1
```

Figure 7.5: Excerpts from the SAX filter used after parsing `sendmail.mc` to XML

which outputs miniCIM instances. The final result can be seen in Fig. 7.7.

The Martel expressions in Fig. 7.4 use regular expressions with some Python idioms. For example, named groups such as `?P<divert>` in line 2 are translated as XML elements sharing their name and content. Since they can appear anywhere in the document, the expression scanning `dnl` comments (in lines 22–23) is invoked at the end of the line. The expressions `define` and `undefine` are detected in lines 9–12; `define` has two parameters and `undefine` one. A more complex case is that

```
<xsl:template match="ostype">
  <CIM_SendmailMcHeaderSettingData>
     <InstanceID><xsl:value-of select="$filename"/>-<xsl:value-of select="position()"/></InstanceID>
      <OsType><xsl:value-of select="$current"/></OsType>
    <xsl:if test="preceding-sibling::version">
      <Version><xsl:value-of select="preceding-sibling::version"/></Version>
    </xsl:if>
    <xsl:if test="preceding-sibling::include">
      <Include><xsl:value-of select="preceding-sibling::include"/></Include>
    </xsl:if>
    <OrderId><xsl:value-of select="position()"/></OrderId>
    <xsl:if test="preceding-sibling::divert">
      <DivertSection><xsl:value-of select="codify:encode(preceding-sibling::divert)"/></DivertSection>
    </xsl:if>
    <Comment>
    <xsl:if test="preceding-sibling::dnl">
      <xsl:for-each select="preceding-sibling::dnl">
         <value>
            <xsl:value-of select="codify:encode(current())"/>
         </value>
      </xsl:for-each>
    </xsl:if>
    </Comment>
  </CIM_SendmailMcHeaderSettingData>
</xsl:template>
<xsl:template match="define|undefine">
  <CIM_SendmailMcDefineSettingData>
     <InstanceID><xsl:value-of select="$filename"/>-<xsl:value-of select="position()"/></InstanceID>
    <VariableName><xsl:value-of select="variable"/></VariableName>
    <xsl:if test="value">
       <VariableValue><xsl:value-of select="value"/></VariableValue>
    </xsl:if>
    <OrderId><xsl:value-of select="position()"/></OrderId>
    <NegativeDefinition>
    <xsl:choose>
      <xsl:when test="local-name($current)='define'">FALSE</xsl:when>
      <xsl:otherwise>TRUE</xsl:otherwise>
    </xsl:choose>
    </NegativeDefinition>
    <Comment><value><xsl:value-of select="codify:encode(preceding-sibling::dnl[following-sibling::*[local-name(.)!='dnl'][1]=$current
         ])"/></value></Comment>
  </CIM_SendmailMcDefineSettingData>
</xsl:template>
<xsl:template match="feature">
  <CIM_SendmailMcFeatureSettingData>
     <InstanceID><xsl:value-of select="$filename"/>-<xsl:value-of select="position()"/></InstanceID>
    <FeatureName><xsl:value-of select="name"/></FeatureName>
    <xsl:if test="parameter">
    <FeatureParameter>
       <xsl:for-each select="parameter">
       <value>
          <xsl:value-of select="current()"/>
       </value>
       </xsl:for-each>
    </FeatureParameter>
    </xsl:if>
    <OrderId><xsl:value-of select="position()"/></OrderId>
    <Comment><value><xsl:value-of select="codify:encode(preceding-sibling::dnl[following-sibling::*[local-name(.)!='dnl'][1]=$current
         ])"/></value></Comment>
  </CIM_SendmailMcFeatureSettingData>
</xsl:template>
<xsl:template match="unknown">
  <CIM_SendmailMcUnknownSettingData>
     <InstanceID><xsl:value-of select="$filename"/>-<xsl:value-of select="position()"/></InstanceID>
    <Content><xsl:value-of select="codify:encode(.)"/></Content>
    <OrderId><xsl:value-of select="position()"/></OrderId>
    <Comment><value><xsl:value-of select="codify:encode(preceding-sibling::dnl[following-sibling::*[local-name(.)!='dnl'][1]=$current
         ])"/></value></Comment>
  </CIM_SendmailMcUnknownSettingData>
</xsl:template>
```

Figure 7.6: Excerpts from the XSLT stylesheet to output `sendmail.mc` data to miniCIM

of `FEATURE` (lines 13–14) which takes up to nine optional arguments. Other special case is the `DAEMON_OPTIONS` expression (lines 17–18), with comma-separated attribute value pairs. When no other rule matches, the rule in lines 20–21, which is given the lowest matching priority, classifies input lines as unknown data. This rule uses the regular expression negative lookahead operator `?!` to ignore blank lines. This parsing produces many XML elements of the same type that are adjacent, especially `dnl` and unknown, which should be fused, since they are logically the same and their separation is a side effect of the parsing process. Comment and unknown elements must also be encoded into Base64. This is the function of the SAX filter shown in Fig. 7.5. It begins declaring lists of elements depending on if they need to be fused or encoded (lines 1–3). Line 5 declares the filter as inheriting from the Python base class for SAX filters, `saxutils.ContentHandler`, and initialises its internal variables in the `__init__` initialiser method (lines 8–14).

SAX filters define methods overriding certain XML parsing events. This filter overrides the `startElement` (lines 16–37), `characters` (lines 39–44) and `endElement` methods (lines 47–59). The first and last methods are associated with the opening and closing of elements, and the second one with the parsing of character data inside an element. `startElement` begins getting the prefix and local name of the opened element (line 17), and sets the parser variables accordingly if they appear in the lists declared above. The `characters` method simply encodes the characters with Base64 if the element requires it, such as with `dnl` or `divert`. The `endElement` method is more interesting since it controls element fusion. If the local name of the element and the variables are set correctly it calls the `endElement` method of the superclass (line 55), which prints the closing tag normally. If not, it skips this step, eliminating the tag, and sets variables to prevent printing the opening tag of the following XML element in line 22. The result is that the two elements are fused and appear as one in the output.

After the SAX filter, the resulting XML tree is passed through the stylesheet of Fig. 7.6. Lines 1–25 process the header, including `OSTYPE`, `VERSION` and `include` directives (lines 4–10) and the `divert` section (lines 12–14). Lines 26–42 process the `define` and `undefine` constructs, and the `FEATURE` directive is processed in lines 43–59, including a loop in lines 49–53 to support its variable number of parameters; and, finally, lines 60–67 handle unknown data. Lines 11, 33, 56 and 64 maintain a counter to preserve ordering in the miniCIM data, and lines 15–23, 40, 57 and 65 aggregate all preceding `dnl` comments into a `Comment` attribute.

The result of this transformations of the Sendmail configuration file, in Fig. 7.7, is represented in miniCIM instances that follow the diagram in Fig. 7.2. The original file is reconstructed by ordering the instances by their `OrderId` attribute

```
 1  <SendmailMcSettingData>
 2      <InstanceID>unknown.mc−0</InstanceID>
 3      <OrderId>0</OrderId>
 4  </SendmailMcSettingData>
 5  <SendmailMcHeaderSettingData>
 6      <InstanceID>unknown.mc−1</InstanceID>
 7      <OsType>linux</OsType>
 8      <Version>setup for Red Hat Linux</Version>
 9      <OrderId>1</OrderId>
10      <DivertSection>eJx1UM1OAyEQvvcpJvHQNnEXTfYNP
11          ... ... ...
12      7PUYuqMhc2du0mTlTemad/n7WO/HHc/VE19yw==
13      </DivertSection>
14      <Comment>
15          <value>eJxLycvh</value>
16      </Comment>
17  </SendmailMcHeaderSettingData>
18  <SendmailMcDefineSettingData>
19      <InstanceID>unknown.mc−2</InstanceID>
20      <VariableName>SMART_HOST</VariableName>
21      <VariableValue>smtp.myisp.com</VariableValue>
22      <OrderId>2</OrderId>
23      <NegativeDefinition>FALSE</NegativeDefinition>
24      <Comment>
25          <value>eJxdjjFuAkEMRfuc4kspAlkLbHpEsxJV0s
26          ... ... ...
27          uPzPVK9W3X4dvTTViWA==</value>
28      </Comment>
29  </SendmailMcDefineSettingData>
30  ...
31  <SendmailMcFeatureSettingData>
32      <InstanceID>unknown.mc−22</InstanceID>
33      <FeatureName>no_default_msa</FeatureName>
34      <FeatureParameter>
35          <value>dnl</value>
36      </FeatureParameter>
37      <OrderId>22</OrderId>
38      <Comment>
39          <value>eJxLycvhSsnLUXBzdQwJDXLVSEnNSayMT8
40          5lTc4u1gRKcAEAuv8LKw==</value>
41      </Comment>
42  </SendmailMcFeatureSettingData>
43  ...
44  <SendmailMcUnknownSettingData>
45      <InstanceID>unknown.mc−33</InstanceID>
46      <Content>eJzz8Xd29lkPCvVxjTdQ4ApS0VKwcVBQsS1
47          ... ... ...
48      WpeTnJmbmcQEAsLgSxw==</Content>
49      <OrderId>33</OrderId>
50      <Comment>
51          <value>eJxLycvh</value>
52      </Comment>
53  </SendmailMcUnknownSettingData>
54
55  <SendmailMcDaemonSettingData>
56      <InstanceID>unknown.mc−36</InstanceID>
57      <DaemonPort>smtp</DaemonPort>
58      <DaemonAddr>127.0.0.1</DaemonAddr>
59      <DaemonName>MTA</DaemonName>
60      <OrderId>36</OrderId>
61      <Comment>
62          <value>eJw1jkEKwkAMRfc9xQf3xYrgGdyJelE4k+
63          ... ... ...
64          xTkUNT5USF25+o+QJyyVK/</value>
65      </Comment>
66  </SendmailMcDaemonSettingData>
67  ...
68  <SendmailMcContains namespace="dc=udc">
69      <Antecedent>
70          <ref classname="SendmailMcSettingData">
71              <InstanceID>unknown.mc−0</InstanceID>
72              <OrderId>0</OrderId>
73          </ref>
74      </Antecedent>
75      <Dependent>
76          <ref classname="SendmailMcHeaderSettingData">
77              <InstanceID>unknown.mc−1</InstanceID>
78              <OrderId>1</OrderId>
79          </ref>
80      </Dependent>
81  </SendmailMcContains>
```

Figure 7.7: CIM rendition of `sendmail.mc` data

and decoding sequentially each instance. In this result, we can see a coded Unknown block (lines 44–53), and `dnl` comments also follow the same format (e.g. in lines 14–16). Finally, in lines 68–81 there is an instance of the inclusion association `SendmailMcContains` which associates the container instance (with `OrderId` 0, lines 1–4) with the header instance (lines 5–17).

7.4. Conclusions

In this chapter the Sendmail configuration format was shown as a specially complex case of configuration file. Next, a possible CIM representation of this format with support for round-tripping and unknown constructs was described. A process to translate from the original configuration file to miniCIM formatted data and back was also explained. The principles and techniques shown in this scenario can be applied to a wide scope of configuration formats with ease, widening the range of application of AdCIM and lessening the negative impact of configuration maintenance and version changes.

Chapter 8

Grid Integration

Grids constitute a category of heterogeneous systems, as geographically distributed collections of forcibly heterogeneous machines maintained by different organizations, and only linked through the use of a common middleware that authenticates their users and enables their controlled access to computing resources.

Grid systems thus coordinate resources not subject to centralized control, using standard, open, general purpose protocols and interfaces, to deliver nontrivial qualities of service. There are several middleware tools that provide the needed infrastructure for Grid systems. The Globus Toolkit [33] is the one used in this work due to its widespread adoption and use. It is based on the Open Grid Services Architecture (OGSA) [34], an Open Grid Forum (OGF) architectural specification defining a service-oriented Grid computing environment.

A key aspect in Grid environments is the management and monitoring of the resources. Their heterogeneity and the need of interoperability between different middleware solutions for Grids explain the need of a common model of information in order to allow the exchange of resource information both inside a Grid and across Grids.

In this chapter we present an infrastructure integrating the AdCIM framework and the Monitoring and Discovery System (MDS) component of Globus [91], that allows to access and query CIM information in a Grid system via a new Operation Provider, enabling more expressive queries for improving resource management. The developed infrastructure can be used for Grid resource discovery, fault diagnosis, scheduling, accounting and many other applications. We have also developed a new

Information Provider to retrieve CIM information, and an Index Service backend to store CIM data in persistent storage, instead of into main memory, the default with Globus. The scalability of this backend is assessed, taking comparative benchmark results of memory usage and response times against the default storage method in Globus.

The structure of this chapter is as follows. Section 8.1 details the integration of AdCIM with the MDS information service in Globus and our new Information Provider. Section 8.2 presents benchmarks showing the scalability using the new Information Provider with our database-backed backend compared with using the memory-backed default one. In Section 8.3 some work related with this chapter is covered. Finally, Section 8.4 concludes the chapter.

8.1. Integration of MDS and CIM

This section covers the integration of AdCIM with the MDS component of Globus. It begins with an overview of the Globus MDS component (Section 8.1.1), describes the CIM internal Java representation used in the new components (Section 8.1.2), followed with a description of the new Information Provider (Section 8.1.3) and database backend (Section 8.1.4), and ends describing the queries supported by the new Operation Provider (Section 8.1.5). Figure 8.1 shows an overview of this integration.

8.1.1. Globus MDS

The Globus Monitoring and Discovery System version 4 (MDS4) is the component of the Globus Toolkit tasked with publishing the information needed for resource discovery and monitoring functionality. Unlike previous versions, MDS4 is based on Web Services Resource Framework (WSRF) services instead of on an LDAP directory, so the Data Persistence Layer of AdCIM must be adjusted with this in mind. MDS4 is sometimes defined as having an “hourglass” structure, as can be seen in Fig. 8.2, in which the sources of information at the base communicate with “sinks” (or information users) at the top via a “narrower” middle section which represents both the MDS web service interface and the common information schemata used in the stored information. Thus, MDS4 does not have a standard schema (the information must be encoded in XML, though), but sources and sinks

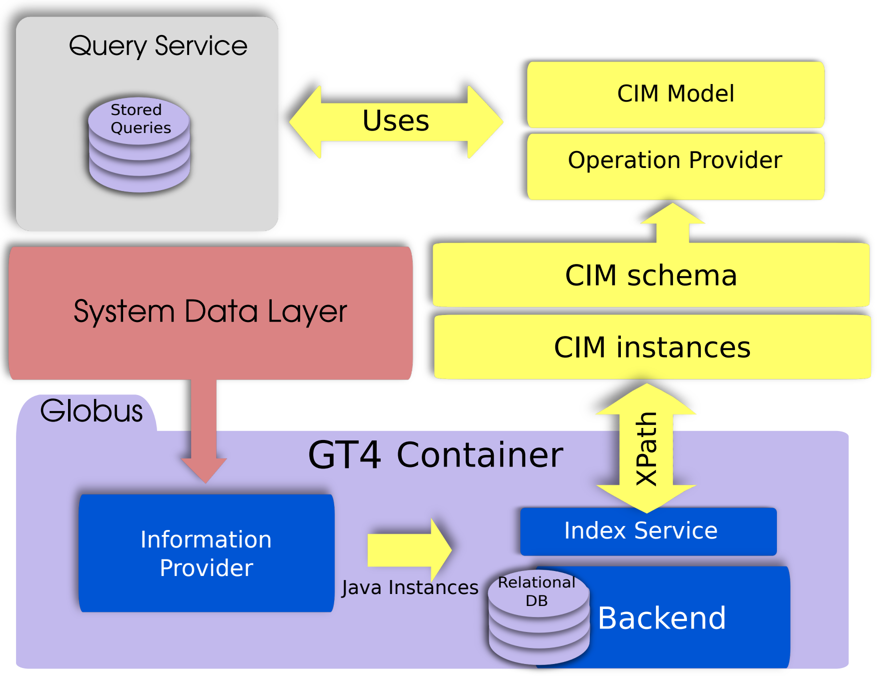


Figure 8.1: Overview of the AdCIM and MDS4 integration

usually need to agree on one schema to communicate. In a sense, this integration makes CIM a standard schema for Grid management applications.

MDS4 comprises two main services: the Index Service publishes, updates and manages the life cycle of resource information, and the Aggregator Framework defines three mechanisms for source applications to publish their data:

- The `QueryAggregatorSource` polls periodically the values of several WSRF properties.
- The `SubscriptionAggregatorSource` registers itself as the listener of a WS-Notification subscription, normally linked to a WSRF property.
- The `ExecutionAggregatorSource` invokes code which periodically returns XML-encoded information.

The last version of Globus (4.2.1) offers the UsefulRP system [38], which enables the gathering of data from various sources and their automatic transformation to Java instances using an XML Schema. This system greatly simplifies the use of

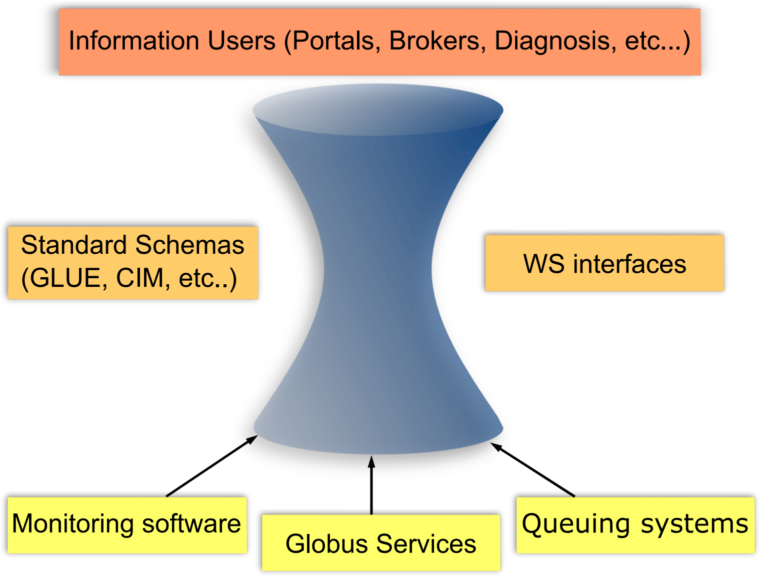

Information Users (Portals, Brokers, Diagnosis, etc...)
Standard Schemas (GLUE, CIM, etc..)
WS interfaces
Monitoring software
Globus Services
Queuing systems

Figure 8.2: MDS4 “hourglass” representation

XML-based sources and integrates them better than the `ExecutionAggregatorSource` by representing them as WSRF objects.

8.1.2. Internal Representation of the Model

Figure 8.3 shows the Unified Modelling Language (UML) model of the CIM internal representation used in the new Operation Provider, Information Provider and backend. As can be seen in the figure, the CIM information is modelled using a metaschema approach, so CIM classes are treated as instances of the `CIMClass-Information`, as opposed to being translated directly as classes in the internal representation. This has the advantage of being much more flexible when new CIM classes are introduced (as only new instances of `CIMClassInformation` have to be considered), and easy to implement. The drawback is the additional overhead, since some implicit object qualities like the number and type of its properties have to be represented as objects and validation has to be explicit. Indications, triggers and methods are excluded, since they are not needed for resource information querying.

Qualifiers are not treated equally. The most important ones (like those indicating the name of the superclass, if a property is a key, if a class is abstract, and so on) are mapped as properties of `CIMClassInformation` or `CIMProperty`. Non-critical qualifiers

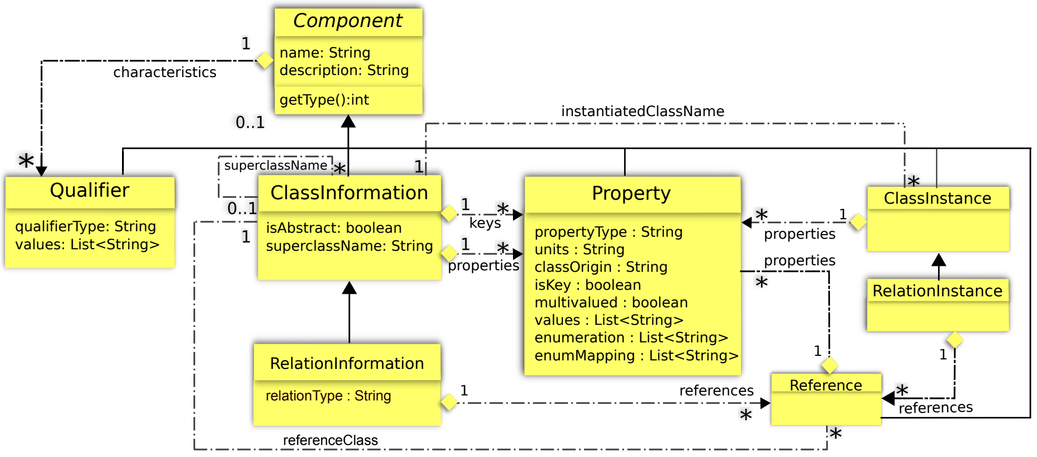


Figure 8.3: UML diagram of the internal CIM representation

are represented with the `CIMQualifier` class. This mapping avoids having to access a separate instance each time an important qualifier is used, but it still allows to define new qualifiers.

References are represented by a special class named `CIMReference` instead of being represented by properties. Inheritance is explicit, with each class inheriting properties and key properties by direct copy from its parent.

The types of the properties are simplified to standard Java types: integer values are converted to Java `long`, real numbers to `double`, strings and characters to the `String` class, and finally, dates and times to `java.util.Calendar`.

Summarizing, the internal representation of the model has been devised trying to improve both performance and flexibility, and also to conserve expressive power.

8.1.3. CIM Information Provider

This section covers the development of an Information Provider for CIM management information. The management information in CIM format is generally provided via non-WSRF querying APIs. The choice is thus limited to use either the `ExecutionAggregatorSource` or the UsefulRP system. The former only provides valid XML documents to Aggregator Sinks, whereas the latter can transform these XML data into Java objects, which can be managed in a programmatic way. Therefore,

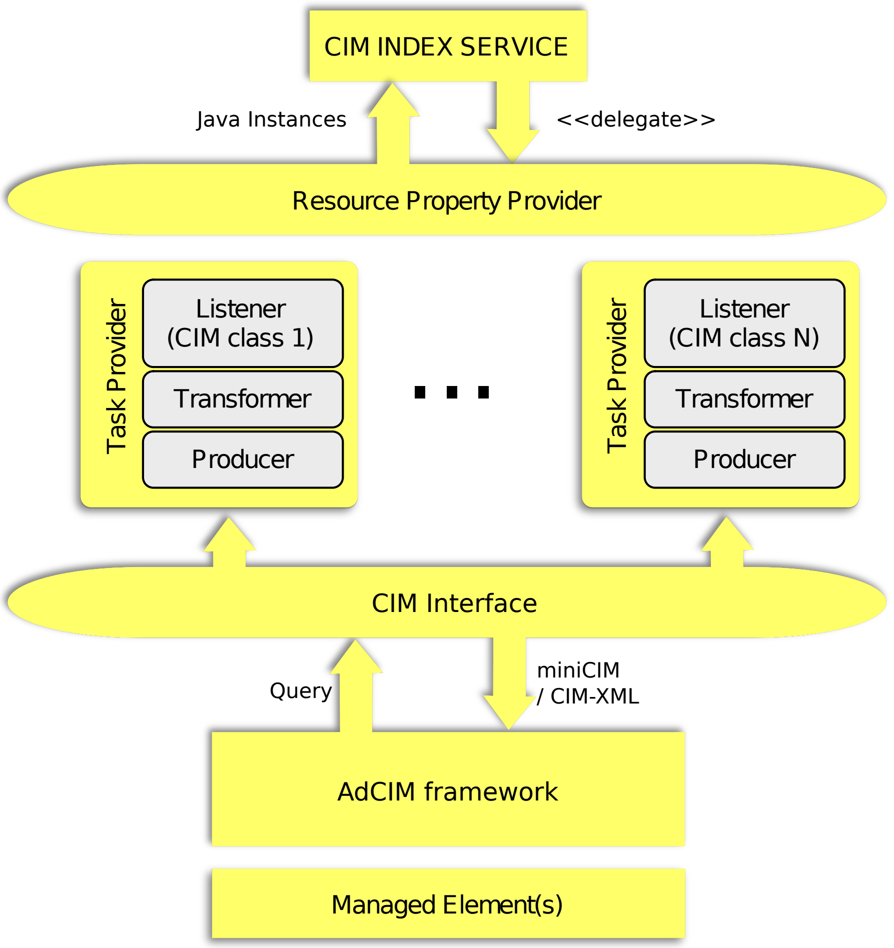


Figure 8.4: CIM Information Provider

UsefulRP was used to implement the CIM Information Provider due to its more extensive functionality. Our implementation of the CIM Information Provider follows the Ganglia Information Provider [57] as a rough guide. This Information Provider is part of GT4 (see Fig. 8.1); it collects information from sources encoded with the XML mapping of the Grid Laboratory Uniform Environment (GLUE) Schema [37].

Figure 8.4 shows a diagram of the proposed Information Provider. Inside it, several components assume different responsabilities:

- The *Producers* serve as mediators between clients and the Information Provider, decoupling them. They allow to transform the data obtained from the client into the format required by both listeners and transformers. GT4 provides an interface for this purpose, which returns a Java DOM representation of an XML Element class. Due to the memory requirements imposed by the DOM tree representation, an Information Producer is defined for each CIM class to obtain its instances. This limits the size of the received XML documents and avoids performance bottlenecks.

- The *Transformers* perform the transformation of CIM-XML into instances of Java object classes defined inside GT4. This transformation is made transparently by the UsefulRP framework if the XML document is defined with an XML Schema. This schema represents CIM instances and their relations including properties and references, involving some modifications needed for the representation of persistent entities such as the insertion of extra fields. It also supports strongly typed queries, not supported with the default MDS4 mechanism. We have also defined an XSLT stylesheet to remove redundancy and normalize data.

- The *Listeners* are notified periodically of the information from the sources to be updated in MDS4. The implementation provided by GT4 maintains the subscripted information in one object attribute and defines methods to query it. Each listener receives the CIM information from a producer and delegates its storage to the CIM Index Service.

- The *Task Providers* link each listener with a producer and invokes them periodically. We have defined our own task provider, which provides producers with the necessary formatted parameters to obtain management information and also notifies the corresponding listener of new CIM instances.

- The *Resource Property Provider* is a front-end class that defines an API to access the information obtained by the Information Provider. It keeps a list of listeners that manage this information. We have defined a CIM specialization to support the new database backend.

8.1.4. Database Backend

The MDS4 Index Service collects monitoring and discovery information from this Information Provider to be queried in a single location. The default GT4 implementation of the Index Service stores the collected data in memory. Maintaining this information in memory entails both performance and scalability problems when large amounts of CIM data are subscribed.

Thus, two backend components have been developed: one of them stores the information in a database and the other maintains the information in memory, like the Globus default solution does. Java Persistent API (JPA) [8], a Sun Microsystems specification, was used for the database access. JPA describes an object/relational mapping API to manage relational data in Java applications. More specifically, we have used the Apache implementation of the standard, OpenJPA [4]. This choice has been motivated by the integration of Globus Toolkit with this implementation since version 4.2, and because OpenJPA provides configurable support for a broad range of databases.

In Section 8.2, the performance and memory consumption of these backends is assessed.

8.1.5. CIM Operation Provider

In order to improve the integration of the CIM model with MDS4, a new Operation Provider was developed, the CIM Operation Provider. It allows to query CIM information subscribed in the CIM Index Service in a simpler and more efficient way than the default GT4 one.

The CIM Operation Provider incorporates the following composable operations, which can be used by the clients to query available CIM instances:

- Obtaining all instances of a particular CIM class.
- Obtaining all instances of a particular CIM class filtered by some property values.
- Obtaining all instances of a particular CIM relation with certain values in some of its references, so CIM associations can be navigated.

An example of composition of these queries can be seen in Fig. 8.5. The expression corresponding to this tree searches computers with a processor speed of 2GHz or more, Linux operating system installed and that are currently running it or have reset capabilities. It is represented as a string using infix operators to make it more intuitive for users. At execution time, each operator is evaluated with the result of the recursive evaluation of its children as argument.

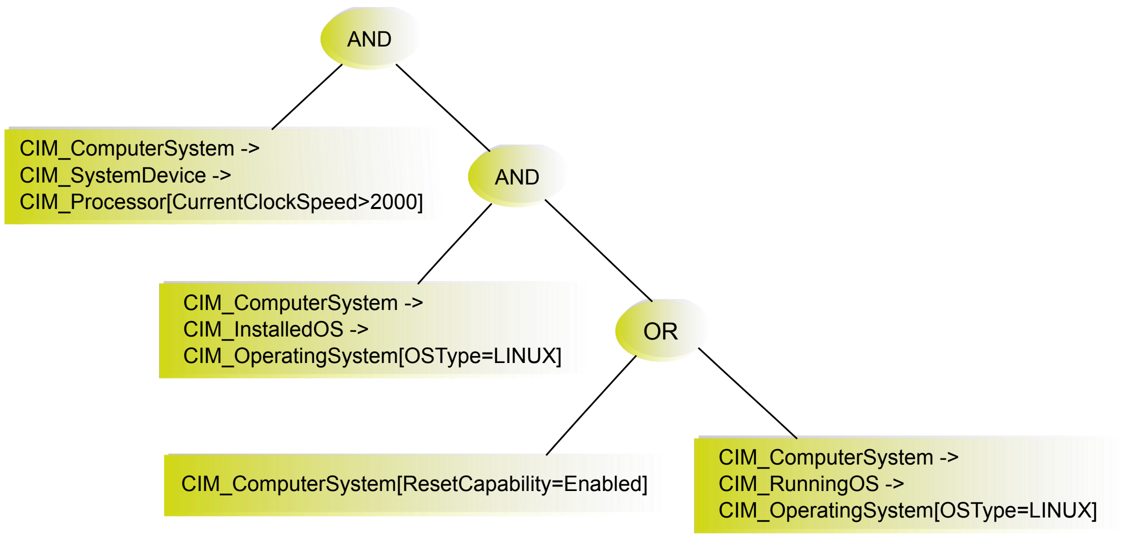

AND
CIM_ComputerSystem ->
CIM_SystemDevice ->
CIM_Processor[CurrentClockSpeed>2000]
AND
CIM_ComputerSystem ->
CIM_InstalledOS ->
CIM_OperatingSystem[OSType=LINUX]
OR
CIM_ComputerSystem[ResetCapability=Enabled]
CIM_ComputerSystem ->
CIM_RunningOS ->
CIM_OperatingSystem[OSType=LINUX]

Figure 8.5: Example query tree

Search paths are evaluated by first finding all the instances at the end of the association that satisfy the conditions on their properties, and then working backwards, navigating through the associations trying to find the origin instance. This method is used to avoid semantic-changing situations in 1-to-N associations that could arise when the search is done breadth-first from the origin node.

The resulting implementation allows arbitrary stacking of conditions on associated nodes, and arbitrarily large expressions based on these predicated nodes.

8.2. Experimental Results

Some experiments were carried out to show the scalability and efficiency of the proposed CIM Information Service. The experiments were run on a Grid in which each node had a 3GHz Intel Core Duo E8400 and 2GB RAM running Linux kernel 2.6.27. Globus Toolkit version 4.2.1 was used.

To ensure that the measures were taken in all cases under the same conditions, the Globus container was executed always with the minimum number of services and without performing unnecessary background tasks. Three different scenarios were considered:

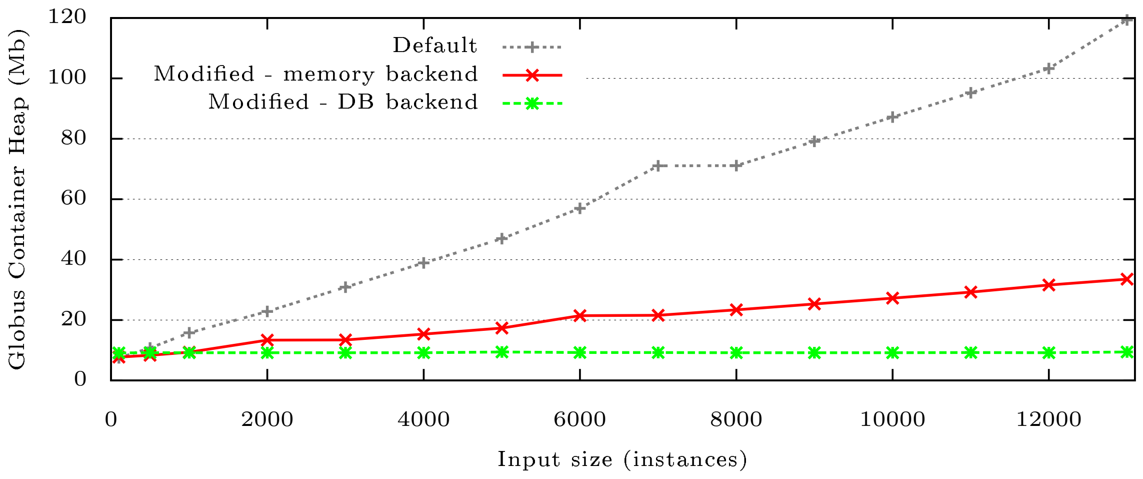


Figure 8.6: MDS memory usage experimental results

1. Default MDS: In this case, the CIM information is obtained and queried using the default mechanisms provided by GT4. The data subscription is carried out with the `ExecutionAggregatorSource`, which executes scripts periodically and provides the resulting XML data to the Index Service. The querying of this information is done with the *wsrf-query* client, which uses the predefined Operation Provider to make XPath queries over the information available in the Index Service.

2. Modified MDS - memory backend: This service obtains the information from our CIM Information Provider and stores it in memory. The information queries are carried out with the *cim-query* client, which was implemented to query the information to the CIM Operation Provider.

3. Modified MDS - database backend: The container is executed with the implemented CIM Index Service. The difference with the previous scenario is that in this case the information is maintained in a database through the use of our database backend.

The first experiment measures the memory consumption in each scenario. The memory consumed is measured as the size of the heap used by the Globus container. Figure 8.6 shows the results when the number of CIM instances increases. It can be observed that in the default solution (scenario 1) memory consumption increases sharply with the number of CIM instances. This can lead to a bottleneck in Grid systems in which the monitoring and discovery of information is organized in a hierarchical form and nodes may have information from several others, collecting too

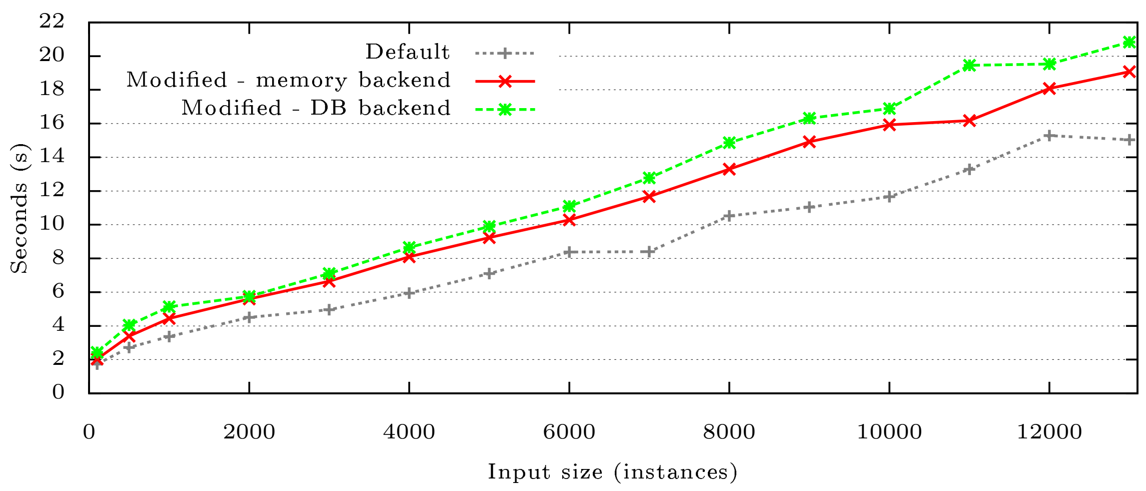


Figure 8.7: MDS response time experimental results

many data. A meaningful improvement is achieved by the solutions that use the UsefulRP providers (scenarios 2 and 3). This is because the information is represented through Java instances instead of through an XML DOM tree. Therefore, in case of memory consumption constraints, any of these two solutions would perform well. The final solution, maintaining the CIM information either in memory or in a database, are chosen conditioned to the requirements imposed by each concrete environment.

Another experiment was carried out to measure the penalties in the response time to query CIM information caused by the use of a database instead of maintaining data in memory. The results obtained are depicted in Fig. 8.7. These results show that the response time to query the information increases when the CIM Information Provider is used to handle the CIM information, especially when the information is maintained in a database. This is because both the search of the instances with certain values in their attributes and the querying of the database (with the consequent creation of the corresponding instances) incur in performance penalties compared with the DOM tree access made by the default Operation Provider of GT4.

This provider uses the XPath 1.0 API included in Xalan-Java to implement the XPath queries over the available information maintained in memory. This implementation of the API uses the DTM (Document Table Model) interface, which was designed specifically to optimize performance and minimize storage of the XPath and XSLT implementations. Nevertheless, the usage of XPath 1.0 means that values are not strongly typed, and operations such as searching value ranges in a data type are difficult to implement. Conversely, since UsefulRP uses an XML Schema to perform the transformation to instances, this representation based on Java objects allows typed queries with more complexity and functionality than XPath queries.

Summarizing, the use of the proposed CIM Information Service offers improved query semantics at the expense of slightly larger delays in queries. The tests have also shown a significant improvement in memory consumption.

8.3. Related Work

Nowadays many efforts within the Globus Alliance are being invested on improving the information subscribed to the Index Service of GT4 by increasing the types of information sources [39]. The latest versions of the toolkit have included a series of Information Providers for the most popular monitoring and management tools (Ganglia, Nagios, Condor, etc.). In this way, the management information for GT4 services like GRAM or RFT is completed with the monitoring and management information coming from several sources.

However, currently the majority of the production Grid projects use some forms of the GLUE schema instead of CIM for the description of grid resources (e.g. EGEE [27], TeraGrid [68], OSG [74]...). This is because they rely on Grid middleware that operates with this schema, such as Globus or gLite. One exception is the NAREGI project [67], which uses its own middleware (NAREGI Middleware) that supports an extension of the CIM schema. The main purpose of the GLUE Schema is to define a common resource information model to be used as a base for the Grid Information Service (GIS), both for resource discovery and for monitoring activities. CIM offers better modelling characteristics and covers other areas of computing not directly related to Grids, but which can be useful to their management.

CIM has already been used as information model to build different monitoring systems for Grid infrastructures. In Mao et al. [56] a monitoring system of web resources, RMCS, is presented. Experimental results have shown that the system provides scalable and flexible capabilities and satisfies the web application requirements. The work presented by Ravelomanana et al. [87] is another example of CIM-based grid monitoring system. Tursunova et al. [99] present a way to develop monitoring applications for Grid systems using CIM. However, unlike our framework, these approaches work independently of the monitoring and management service provided by the Grid middleware, so they must be used separately.

An approach to include CIM information in the Index Service of Globus was proposed by Wang et al. [100]. It implements a transformation from CIM format to the GLUE Schema. The same idea was followed by Nakada et al. [65] to achieve interoperability among different grid middleware that use different resource information schemata. This solution is quite limited since occasionally a correspondence between both models cannot be found, as the CIM model covers much more scope than the GLUE schema and allows its extension with definitions that represent a specific environment.

CIS [59] is another Information Service using CIM as the underlying information model. It can be deployed in UNiform Interface to COmputing REsources (UNICORE) [28] Grids to obtain information about resources and services.

With regard to the persistent storage, Aloisio et al. [2] have developed, within the European GridLab project, a Grid information service (named iGrid) based on the MDS of GT2 that maintains the management information in a database as does the solution presented in this chapter. While the prerequisites of performance and scalability are achieved, this system was designed for GT2, which has a very different architecture and uses different technologies than GT4.

8.4. Conclusions

The application of our AdCIM framework proposed in this chapter enables the use of CIM information in the management and monitoring of Grids. A series of components for the MDS4 have been presented to adapt GT4 to CIM: a new Information Provider using the UsefulRP framework, and a database-backed backend for the Index Service which obtains a good trade-off between memory consumption and response time, compared with the default memory-backed method.

Chapter 9

Application to Wireless Mesh Networks

The focus of AdCIM on heterogeneous infrastructures makes possible its application to very useful and interesting networks, such as Wireless Mesh Networks (WMNs), which leverage existing wireless infrastructures to offer great flexibility and resilience.

This chapter is structured as follows. First, Section 9.1 gives background information about Wireless Mesh Networks and defines the application of AdCIM to them. Next, Section 9.2 contains an analysis of the mesh router configuration and the entities it represents. Section 9.3 elaborates about the mapping of these entities to CIM and its implementation. Setion 9.4 details the transformation of the CIM model to an ontological representation which supports reasoning processes, useful for semantic checking, diagnosis and recovery tasks that are specially suited to WMNs due to their distributed and dynamic nature. This aspect is developed in Section 9.4. Section 9.5 covers related works and, finally, conclusions are drawn in Section 9.6.

9.1. AdCIM meets Wireless Mesh Networks

Wireless Mesh Networks (WMN)s [1] extend the Wireless LAN architecture by replacing the classical wired distribution system with a wireless, self-organizing and self-healing infrastructure. This allows for an easy and cheap deployment of access

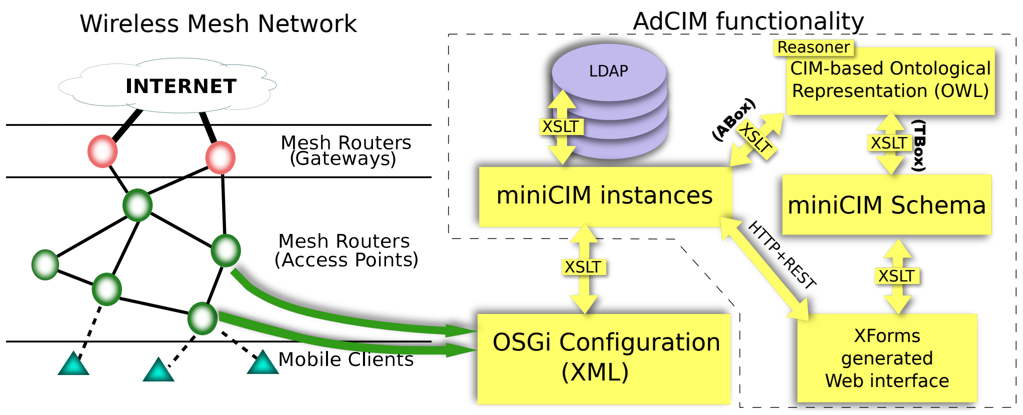


Figure 9.1: Application of the AdCIM framework to WMN management

points where cabling is costly. A WMN consists of access points and client nodes.

The access points (also simply called mesh routers) form a mesh of fixed nodes, known as the network infrastructure or backbone. The backbone can rely on various radio technologies for interconnection (amongst them IEEE 802.11). These access points have a double function: that of providing access to roaming clients, and of relaying data for other routers. Some of the routers act as gateways for connectivity with other wireless mesh networks, or other types of networks, including wired ones (see left side of Fig. 9.1).

The coverage of a wireless mesh network is extended by means of multi-hop communications. Therefore the mesh routers have additional functions, besides those present in regular access points, to support mesh networking (i.e. routing capabilities), functions which are very important to manage for the performance and health of the network.

There is currently a lack of frameworks for the integrated configuration of WMN routers, so this chapter explores the application of the AdCIM framework to the configuration of WMN routers. The chosen approach is shown in Fig. 9.1. The router configuration is mapped to the CIM model and then converted using XSLT into CIM instances (in miniCIM format). Then the functionality of the AdCIM framework can be exploited, including persistence in LDAP directories and the generation of web forms to manipulate these instances. The modified data can then be transformed back to its original format and uploaded to the router. These CIM data can also be transformed to a Web Ontology Language (OWL) [105] based semantic representation, which allows to check the internal consistency of the configuration and infer new data.

We chose the configuration of a modular wireless router developed by the Image Sciences, Computer Sciences and Remote Sensing Laboratory (LSIIT) RP team [51]. This router configuration is managed using the Open Services Gateway initiative (OSGi) [79] Java-based framework. OSGi adds infrastructure for on-the-fly management and deployment of modules, and to support starting, stopping and uninstalling them independently. An overview of the architecture of the router can be seen in Fig. 9.2.

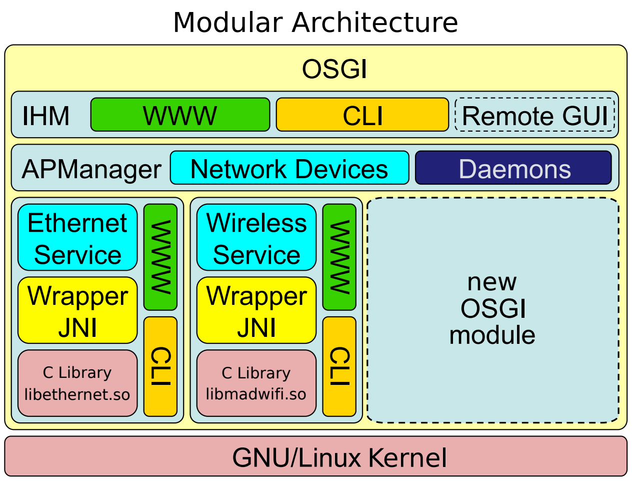


Figure 9.2: Wireless mesh router architecture

Using the CIM model and its OWL representation opens new possibilities to diagnose mesh network problems or to simulate the effect of a proposed configuration change globally. It also provides a higher level view that hides OSGi implementation details and that is generalizable to routers with other architectures which could then be managed homogeneously.

9.2. Configuration Analysis

The router subject to study provides a naming schema and structure for the configuration attributes that conform to the standard OSGi Configuration Service, detailed in [80], which is the component of OSGi tasked with managing the settings of other services and their persistence. It defines configuration objects that contain configuration dictionaries, a collection of name-value pairs that represent the settings of an OSGi service; objects also have a PID (persistent identifier) as primary key.

```
<configurations>
  <level name="device">
    <level name="ethernet">
      <level name="interface">
        <configuration name="br31">
          <String name="ConfigurationType">interface</String>
          <Integer name="UpdateType">0</Integer>
          <String name="device.deviceType">ethernet</String>
          <String name="device.ethernet.broadcast">0.0.0.0</String>
          <Short name="device.ethernet.flags">1</Short>
          <String name="device.ethernet.ip">130.79.91.223</String>
          <Boolean name="device.ethernet.ipDesactivated">false</Boolean>
          <StringArray name="device.ethernet.ipv6addresses"/>
          <Integer name="device.ethernet.mtu">1500</Integer>
          <String name="device.ethernet.netmask">255.255.254.0</String>
          <Boolean name="device.ethernet.usingDHCP">false</Boolean>
          <String name="device.interfaceName">br31</String>
          <String name="device.virtualName">br31</String>
          <String name="service.bundleLocation"> file:ap−bundles/devmng_eth.jar </String>
          <String name="service.pid"> device.ethernet.interface.br31 </String>
        </configuration>
      </level>
    </level>
  </level>
</configurations>
```

Figure 9.3: Wireless mesh router configuration for an IP interface

OSGi services can register themselves to a PID to receive a dictionary, or can also register to a configuration factory to receive an arbitrary number of dictionaries registered in the factory.

The LSIIT router stores its configuration objects as XML data in the format seen in Fig. 9.3. Configuration objects have structured PIDs located in a hierarchy similar to that of Java packages. Properties in the dictionary are represented as subelements named after their type, their actual name translated as an attribute. Configuration objects and factories can also refer to one another using these PIDs.

Configuration objects also contain sets of properties for some OSGi services that map into conceptual entities, sometimes as an exact match, others as several related entities. In the following points, several of these entities are detailed:

Services. There are three services in the router configuration: SNMP, Bridging and Telnet. Each has a very different configuration: SNMP only needs to be set as started or stopped, Bridging needs a list of network interfaces; so the mapping of each service is different in each case.

IP Interfaces. These entities represent various virtual interfaces on top of the wireless interfaces. Their set of properties includes IP address and netmask and Dynamic Host Configuration Protocol (DHCP) status. While their configuration location is named `device.ethernet.interface`, they mostly represent IP configurations, with two properties MTU and Flags) representing transport level properties. IP Interfaces can be related to wireless interfaces or be implemented by the Bridging service.

Logical Wireless Interfaces. They represent the aspects of wireless interface configuration that reside in a higher level than physical configuration. These aspects include encryption algorithms and settings, Remote Authentication Dial In User Service (RADIUS) server configuration, Virtual Local Access Network (VLAN) configuration, Service Set IDentifier (SSID), MAC filtering, and Optimized Link State Routing Protocol (OLSR) related parameters [14] like the link quality level, window size or link hysteresis control. Each one of these entities are generally associated with an IP interface and a physical wireless interface.

Physical Wireless Interfaces. They represent the low-level settings of a wireless interface and are bound one-to-one to a logical interface. These settings include transmission channel (or optionally frequency) and transmission power. They have regulatory restrictions depending on country of usage to avoid interference with other equipment, but these limits are not modelled in the configuration.

Other entities. They include a generic IP routing default gateway setting and password information. There are also entities to represent virtual LAN settings.

These configuration entities are abstracted and mapped to higher-level preexisting entities and extensions from the CIM model in the next section.

9.3. Configuration Mapping to CIM

This section covers the mapping from the original format of the router configuration to the CIM model that is later translated to OWL. According to the classification presented by Rivière and Sibilla [88], this mapping follows the “recast” philosophy, since “concepts” are mapped. It also follows the principle of abstract translation, since redundant information is removed. Finally, the organization is independent, since the resulting model is standard. In Fig. 9.4, a CIM class hierarchy to map the router configuration is proposed. In this section we explore both the mapping and the XSLT templates that implement it.

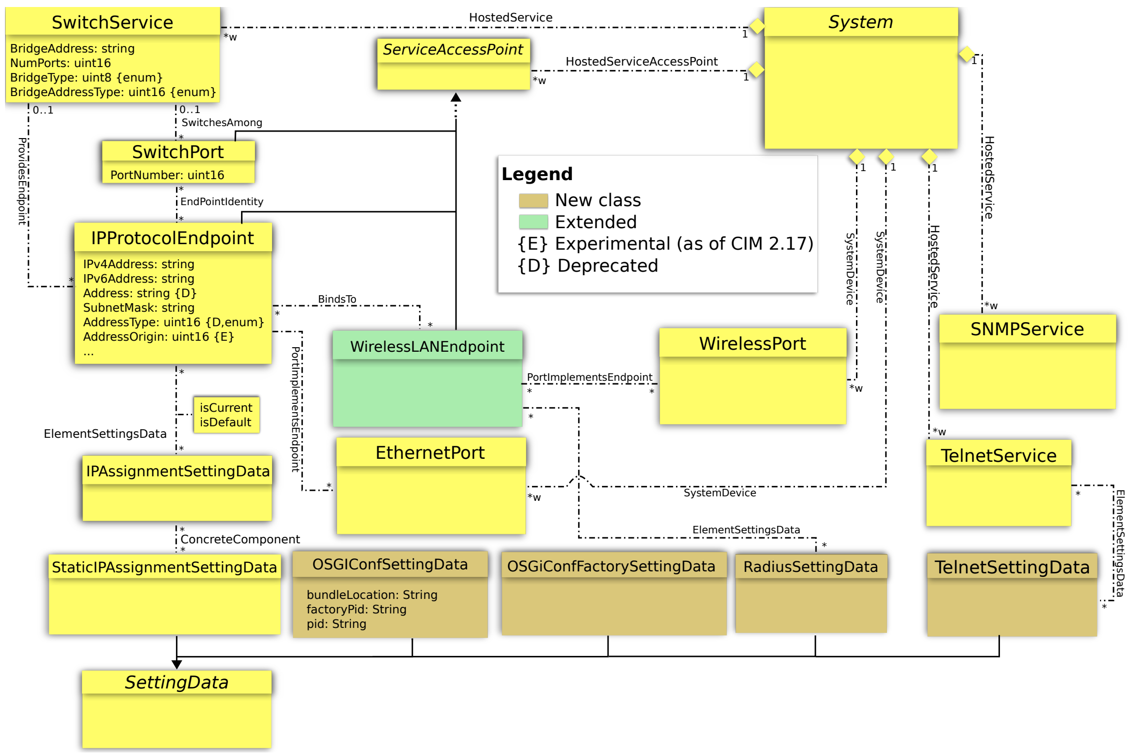


Figure 9.4: CIM mapping class hierarchy for the router configuration

9.3.1. CIM Class Semantics

The CIM mapping presented in Fig. 9.4 can be separated in two abstraction levels, one including the `OSGIConfSettingData` and `OSGIConfFactorySettingData` classes, and representing all OSGi-related structures and identifiers, and the other level representing abstract entities. The latter includes some specialized service classes, such as `SNMPService`. `SwitchService` represents the bridging facility and is associated to a list of `SwitchPort` instances associated to an `IPProtocolEndpoint` instance. The bridging service also generates an IP interface of its own, related with `Provides-Endpoint` instances. These `IPProtocolEndpoints` instances, which represent IP interfaces, are related to an `IPAssignmentSettingData` instance which indicates if the parameter setting is static or via DHCP. In the first case, it is further associated with a `StaticIPAssignmentSettingData` instance which contains the IP data. There are cases in which no IP assignment data are given.

`IPProtocolEndpoint` instances can be associated to a `WirelessLANEndpoint` instance which represents logical wireless interfaces. Each one can have several RA-

DIUS configurations, represented by the `RadiusSettingData` class. The maximum transfer unit value of IP interfaces is moved to the `EthernetPort` class. Finally, physical wireless interface data are included in `WirelessPort` instances, related to `WirelessLANEndpoint` by the `PortImplementsEndpoint` association.

All service and interface classes are weak entities with regard to the containing system, so appropriate `HostedService` and `HostedServiceAccessPoint` associations are made for services and service access points. These associations are also very useful for navigating through the data. The two abstraction levels simplify the recovery of the original configuration file and, at the same time, avoid pollution of OSGi specific attributes on abstract entities.

9.3.2. Mapping Considerations

The properties of an OSGi dictionary are usually directly mapped to CIM properties. CIM property types in the output are specified in a schema. In the case of multiple choice values their numerical representation is changed to an enumeration of schema-restricted string values.

There are some properties that are not mapped at all to CIM, such as boolean values that control the expression of others. For example, `device.ethernet.ip-Desactivated` in line 12 of Fig. 9.3 determines if the IP interface has a valid IP configuration. In the resultant CIM instances, there are no invalid fields, since almost all can be omitted instead of having invalid values.

9.3.3. XSLT Implementation

The implementation of the transformations is done with two XSLT stylesheets: one that transforms the XML OSGi configuration data into CIM data and another one that does the opposite. The result of the application of the first stylesheet to the configuration shown in Fig. 9.3 can be seen in Fig. 9.5.

Internally, the first stylesheet has a general function that can create the correct miniCIM representation of any particular CIM association with only its name and the classes related by it. The input file is read recursively and the default action is to store the OSGi related attributes and instances. If the configuration PID matches certain values, appropriate templates that create CIM classes and associations are invoked. The generated instances are processed in a second pass to add relationships

```xml
<CIM_OSGiConfSettingData namespace="dc=udc">
  <BundleLocation>
    file:ap-bundles/devmng_eth.jar
  </BundleLocation>
  <InstanceID>device.ethernet.interface.br31</InstanceID>
  <ConfigurationType>interface</ConfigurationType>
  <UpdateType>0</UpdateType>
</CIM_OSGiConfSettingData>

<CIM_IPProtocolEndpoint namespace="dc=udc">
  <SystemCreationClassName>
    CIM_ComputerSystem
  </SystemCreationClassName>
  <SystemName>LSIIT</SystemName>
  <CreationClassName>
    CIM_IPProtocolEndpoint
  </CreationClassName>
  <Name>device.ethernet.interface.br31</Name>
  <Caption>br31</Caption>
</CIM_IPProtocolEndpoint>

<CIM_ElementSettingData namespace="dc=udc">
  <IsCurrent>Is Current</IsCurrent>
  <ManagedElement>
    <ref classname="CIM_IPProtocolEndpoint"
         namespace="dc=udc">
      <CreationClassName>
        CIM_IPProtocolEndpoint
      </CreationClassName>
      <Name>device.ethernet.interface.br31</Name>
      <SystemCreationClassName>
        CIM_ComputerSystem
      </SystemCreationClassName>
      <SystemName>LSIIT</SystemName>
    </ref>
  </ManagedElement>
  <SettingData>
    <ref classname="CIM_OSGiConfSettingData"
         namespace="dc=udc">
      <InstanceID>
        device.ethernet.interface.br31
      </InstanceID>
    </ref>
  </SettingData>
</CIM_ElementSettingData>


<CIM_IPAssignmentSettingData namespace="dc=udc">
    <InstanceID>br31</InstanceID>
    <AddressOrigin>Static</AddressOrigin>
</CIM_IPAssignmentSettingData>
<CIM_StaticIPAssignmentSettingData namespace="dc=udc">
    <InstanceID>br31-static</InstanceID>
    <IPv4Address>130.79.91.223</IPv4Address>
    <SubnetMask>255.255.254.0</SubnetMask>
    <GatewayIPv4Address>130.79.91.254</GatewayIPv4Address>
</CIM_StaticIPAssignmentSettingData>

<CIM_ElementSettingData namespace="dc=udc">
  <ManagedElement>
    <ref classname="CIM_IPProtocolEndpoint"
         namespace="dc=udc">
      <CreationClassName>
        CIM_IPProtocolEndpoint
      </CreationClassName>
      <Name>device.ethernet.interface.br31</Name>
      <SystemCreationClassName>
        CIM_ComputerSystem
      </SystemCreationClassName>
      <SystemName>LSIIT</SystemName>
    </ref>
  </ManagedElement>
  <SettingData>
    <ref classname="CIM_IPAssignmentSettingData"
         namespace="dc=udc">
      <InstanceID>br31</InstanceID>
    </ref>
  </SettingData>
</CIM_ElementSettingData>

<CIM_ConcreteComponent namespace="dc=udc">
  <GroupComponent>
    <ref classname="CIM_IPAssignmentSettingData"
         namespace="dc=udc">
      <InstanceID>br31</InstanceID>
    </ref>
  </GroupComponent>
  <PartComponent>
    <ref classname="CIM_StaticIPAssignmentSettingData"
         namespace="dc=udc">
      <InstanceID>br31-static</InstanceID>
    </ref>
  </PartComponent>
</CIM_ConcreteComponent>
```

Figure 9.5: Excerpt from the output of transforming the configuration of Fig. 9.3 into miniCIM format

which are inefficient to add in the first pass, such as the one between `SwitchPort` and `IPProtocolEndpoint`, which would require processing sections of the input several times.

The second stylesheet, which converts CIM data back to the OSGi configuration, uses XSLT functions to follow arbitrary CIM associations and rebuilds the OSGi configuration by retrieving the CIM classes representing OSGi configurations and factories and navigating associations from them. A second pass is required to structure the file in the level hierarchy of OSGi configurations by recursively parsing the PID values of each configuration.

The two stylesheets use the pattern-matching capabilities of XSLT to allow extensibility: recognized elements trigger special case processing and unknown elements would only be mapped at the OSGi level, without stopping the transformation.

9.4. Ontology Representation

If we only transform the wireless configuration data into a CIM model as we do in Section 9.3, we would have a semi-formal representation that includes a taxonomical classification and domain knowledge, but this representation is not formal because many domain constraints and metadata are not expressed explicitly, and it is not possible to infer new information and reason over the data without a priori knowledge of the semantics of the domain (see Quirolgico et al. [86]). For example, in the domain of WMNs, the channel information of a wireless interface really maps into a range of frequencies that could be barred because of national regulation or other equipment used in the installations. The existence of several useable bands, proprietary wi-fi protocols and directional antennas further complicates the issue. The use of an ontology and an off-the-shelf reasoner (a program implementing logical procedures) would allow us to deduce a conflict in those situations.

Another important motivation is configuration semantic checking, as seen in the works of Sinz et al. [93] and Glasner and Sreedhar [36], in which logical constraints are verified in the Apache configuration file. These constraints are in a higher semantic level because they are not concerned with mere well-formedness, but with semantic integrity (e.g. “Every server must have a configuration”, “Virtual hosts addresses must be contained in the address of their server”).

Performing this checking with ontologies has many advantages; for example, problems in higher levels of abstraction can be traced logically by the reasoner to lower level causes, and other configuration formats can be expressed with the same model, so the checking can be applied unchanged to very different formats of configuration.

The rest of this section covers this representation. First, Section 9.4.1 presents some background information about the OWL ontology language and its formalisms. Next, Section 9.4.2 explains the transformation made from CIM concepts and entities to OWL concepts and entities. Finally, Section 9.4.3 gives commented examples of application and implementation details.

Letter	Description Logic Characteristic
$\mathcal{AL}$	Attributive Language, has atomic negation, concept intersection, universal restriction and limited existential quantification.
$\mathcal{C}$	Complement of complex (non-atomic) concepts.
$\mathcal{S}$	$\mathcal{ALC}$ with transitive roles.
$\mathcal{H}$	Role hierarchy (subproperties).
$\mathcal{O}$	Nominals (concepts defined by enumerated individuals).
$\mathcal{I}$	Inverse roles.
$\mathcal{N}$	Cardinality restrictions (maximum, minimum cardinality of a role).
$\mathcal{R}$	Reflexivity, irreflexivity, disjointness and inclusion of roles (implies $\mathcal{H}$).
$\mathcal{Q}$	Qualified cardinality restrictions (maximum and minimum of a role restricted to a particular concept).
$\mathcal{U}$	Concept Union (Implied by $\mathcal{ALC}$ and DeMorgan).

Table 9.1: Expressivity characteristics in description logics

9.4.1. The Web Ontology Language

OWL (Web Ontology Language) [105] is a W3C standard extending its Resource Description Framework (RDF) for the specification of metadata. OWL can represent formally knowledge domains organized as hierarchical classifications and supports reasoning tasks on them, such as concept satisfiability and consistency. OWL is based on the description logics formalism and supports three levels of expressivity: OWL Lite, OWL-DL, and OWL Full, which represent various compromises between expressivity and computability.

Among OWL Lite, DL and Full we have chosen OWL-DL for its better balance between expressivity and efficiency. Lite is too restrictive and Full is undecidable and also less efficient. Expressing the key uniqueness in CIM instances can only be done in OWL Full (since it entails having inverse functional datatype properties), but this constraint does not represent semantic errors in the original document so there is no benefit in expressing it.

An interesting feature in ontological processing is the formal enactment of policies. A rule language like Semantic Web Rule Language (SWRL) [106] allows to specify Horn-like rules of the form $H \leftarrow B_1, \ldots, B_n$, in which the head H is asserted if all the body atoms $B_{1 \ldots n}$ are true. These rules can be used to enact policies on concrete individuals, but not on generic concepts since, as Motik et al. [63] show, the combination of OWL-DL and unrestricted SWRL rules is undecidable (not guaranteed to end in the worst case). Nevertheless, they also show that if the reasoner restricts rule variables to known individuals, they become decidable. SWRL rules can also extend OWL when more expressivity is needed (e.g. role composition like in a hypothetical property *uncleOf*) or there is no reasoner support (e.g. reasoning and mathematical operations with datatypes).

OWL is based on description logics, basically fragments of first order logic (FOL, hereafter), in turn, propositional logic with existential and universal quantifiers. Full FOL is not used because of its undecidability and computational intractability.

Description logics differ in the expressivity retained (or added) from FOL. Expressivity characteristics are symbolised by a letter in Table 9.1. The subset of OWL we have used, OWL-DL, is described as $\mathcal{SHOIN}(\mathbf{D})$ ($\mathbf{D}$ indicates that it operates with integer and string datatypes as domain), so the supported operations are those in the following expression:

$$C \rightarrow A \mid \neg C \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid \exists R.C \mid \forall R.C \mid \geq nS \mid \leq nS \mid a_1, \ldots, a_n \tag{9.1}$$

In description logic, a concept is a set of individuals, and a role is a binary relationship between concepts. Then, C and C_i are concepts, A an atomic concept, $R.C$ describes a role or property, and a_i are named individuals. S is a property name and n is an integer number that indicates its minimum or maximum cardinality. The supported operations include negation, union and intersection, existential and universal quantification, number restrictions and named individuals. The roles as relationships can be also transitive. Known facts or "axioms" are grouped in the knowledge base, divided in the Terminological Box (TBox) which hierarchically groups axioms about concepts and roles and their mutual inclusion, and the Assertional Box (ABox) (see right part of Fig. 9.1), which contains knowledge about individuals (and their inclusion in a class). A reasoner can perform several tasks with that information, such as classifying instances in classes, restructuring the class hierarchy and detecting inconsistencies. We have used the open-source reasoner Pellet [13], which has support for both OWL and SWRL.

9.4.2. CIM Transformation to OWL

Works like the ones by Heimbigner [43], Majewska et al. [55] and García et al. [35] have already implemented mappings of CIM to OWL-DL, and they remark the lack of equivalences for some CIM constructs in OWL. Our mapping is closer to the second and third works, and was implemented with an XSLT stylesheet. Table 9.2 shows some of the mappings from the CIM schema to OWL TBox of our transformation.

We have adapted some of the ideas of previous works and rejected others; for example, mapping CIM key properties was problematic and it was not implemented, so a unique name was assigned to each instance based on the class name and a unique ID generated by the XSLT stylesheet.

Our associations mapping tries to simplify their closure and preserves the semantics of cardinality. Since OWL properties are binary, separate instances for associations are required to house possible association attributes.

Once the pertinent classes from the CIM schema are translated into TBox definitions, the configuration file is translated entirely into ABox assertions. Nevertheless the separation is not perfect, because of the world closure conditions, described next.

A great part of the mapping process is concerned with providing axioms to obtain world closure. OWL reasoners operate following by default the Open World Assumption, so unstated facts are not false, merely unknown. This is mainly to ensure interoperability and that the addition of new knowledge does not invalidate previous conclusions. Unfortunately, it has the side effect of making membership by negation very hard to verify. This is because it requires negation as failure, and only works if there is no possible additional knowledge that invalidates the negation.

In this case, since the configuration file is the universe of discourse we can pre-

CIM entity	OWL TBox mapping
Class	`<owl:Class>`, defined as a closed set of individuals using `<owl:OneOf>`
SubClass	`<rdfs:subClassOf>`, all subclasses declared as disjoint to one another
Properties	`DataProperties` with appropriate types to map CIM types
References	cardinality 1 `ObjectProperty`. Inverse properties are inferred automatically by the reasoner
Association	subclasses of `Association`. Each instance limited to `DataProperty` and two cardinality 1 `ObjectProperties`
Cardinality	Normal in `DataProperty`. `ObjectProperty` in associated classes by limiting the cardinality of inverse object properties
Key	Not implemented, causes undecidability without being needed

Table 9.2: CIM to OWL mapping

vent this by listing the instances of every instantiated class (including associations), restricting the cardinality of associations to exactly one, and defining all individuals pairwise disjoint. This effectively closes the world and allows reasoning with negations. Having to specify these axioms can be viewed as a drawback of OWL, but it also gives flexibility to open certain parts of the world, for example parts modifiable by unknown external imports or concerning unknown extension modules.

9.4.3. OWL Reasoning Implementation in AdCIM

Once the information contained in the miniCIM schema and instances is translated to TBox and ABox axioms, additional axioms are introduced in an included file that verify some conditions. This section shows some examples of applicable restrictions for configuration checking. To better understand this section, refer to Fig. 9.4.

The first example shows a simple case in which unconfigured entities are detected; in this case, ports in a switching service that are not configured. The following set of rules avoid the possibility of the switching service failing due to misconfiguration and causing a malfunction in the node:

$$SwitchPort\_Undefined \equiv SwitchPort \sqcap \tag{9.2a}$$
$$\neg(\exists EndpointIdentitySystemElement^{-}.EndpointIdentity) \tag{9.2b}$$

This restriction declares an undefined port in a switching service as a port not represented with a network endpoint. This is checked by the presence of an inverse property from association `EndpointIdentity`, x^{-} in term 9.2b. The values of these inverse properties are inferred automatically by the reasoner. Since this restriction uses negation as failure, the possible instances of the `EndpointIdentity` association and `SwitchPort` must be closed for it to work.

The effects of a configuration error are made to cascade to other entities defining intermediate classes (OWL-DL does not support composition of properties). The reasoner automatically determines the proper evaluation order, and the cascading can be made arbitrarily deep. The types of errors detected by this process are important in wireless nodes, since obscure high-level errors might have simple motivations solvable on-the-fly. The following set of rules shows how to chain misconfiguration conditions across relationships to detect misconfigured IP addresses:

$$\begin{aligned}
&Not\_current\_IP\_setting \equiv && (9.3a)\\
&ElementSettingData \cap && (9.3b)\\
&(\exists ElementSettingDataIsCurrent = false \mid xsd : string) \cap && (9.3c)\\
&(\exists ElementSettingDataSettingData.IPAssignmentSettingData) && (9.3d)
\end{aligned}$$

$$\begin{aligned}
&Unconfigured\_IP\_Endpoint \equiv IPProtocolEndpoint \cap && (9.3e)\\
&((\forall ElementSettingDataManagedElement^{-}.Not\_current\_IP\_setting) && (9.3f)\\
&\cup (\neg(\exists ElementSettingDataManagedElement^{-}.IP\_setting))) && (9.3g)\\
&BindsTo\_Unconfigured\_IP\_Endpoint \equiv && (9.3h)\\
&BindsTo \cap (\exists BindsToAntecedent.Unconfigured\_IP\_Endpoint) && (9.3i)
\end{aligned}$$

The term 9.3c selects IP setting instances that are not currently used, term 9.3d deselects `ElementSettingData` instances not related to IP settings, and the special intermediate class `Unconfigured_IP_Endpoint` is defined as one Endpoint with no IP settings (9.3e), or an Endpoint in which none are current (9.3f). Finally, term 9.3h declares a subclass of the association `BindsTo` grouping those instances that bind with `Unconfigured_IP_Endpoint` instances. In that way, semantic errors are propagated so they can help diagnose problems in top-level entities.

Policies are also implemented by SWRL rule chaining. SWRL allows both straightforward composition without intermediate classes, and performing inequality comparisons with datatype ranges (instead of only supporting equality comparisons). Some OWL reasoners translate the rules and relevant OWL axioms to another rule engine, sometimes changing the semantics, but the Pellet reasoner integrates them fully with OWL axioms. These rules, for example, detect wireless ports that have illegal frequencies depending on the legislation of the country:

$$\begin{aligned}
&WirelessPortChannel(?x, ?y) \wedge swrlb : multiply(?y5, ?y, 5) && (9.4a)\\
&\wedge\, swrlb : add(?z, ?y5, 2412) \rightarrow WirelessPortFrequency(?x, ?z) && (9.4b)
\end{aligned}$$

$$located(Japan, ?y) \wedge ComputerSystem(?y) \wedge \tag{9.4c}$$
$$\wedge\ SystemDeviceGroupComponent(?z, ?y) \wedge \tag{9.4d}$$
$$\wedge\ SystemDevicePartComponent(?z, ?a) \wedge WirelessPort(?a) \wedge \tag{9.4e}$$
$$\wedge\ WirelessPortChannel(?a, ?b) \wedge swrlb : greaterThanOrEqual(?b, 2500) \tag{9.4f}$$
$$\rightarrow IllegalFrequency\_WirelessPort(?a) \tag{9.4g}$$

$$located(USA, ?y) \wedge ComputerSystem(?y) \wedge \tag{9.4h}$$
$$\wedge\ SystemDeviceGroupComponent(?z, ?y) \wedge \tag{9.4i}$$
$$\wedge\ SystemDevicePartComponent(?z, ?a) \wedge WirelessPort(?a) \wedge \tag{9.4j}$$
$$\wedge\ WirelessPortFrequency(?a, ?b) \wedge swrlb : greaterThanOrEqual(?b, 2467) \tag{9.4k}$$
$$\rightarrow IllegalFrequency\_WirelessPort(?a) \tag{9.4l}$$

Since the legal spectrum depends on the country, it first determines the location of the system housing the wireless ports in terms (9.4c, 9.4h), and then finds their frequency (9.4d-9.4g, 9.4i-9.4l). Since these frequencies are usually expressed as channels, terms 9.4a and 9.4b calculate the frequency by using SWRL built-in operations. Finally, the frequency of each port is compared with the legal maximum (terms 9.4f, 9.4k) and minimum (not shown). If this value is out of limits (for example, channel 13 in the USA), the ports are classified in the `IllegalFrequency_WirelessPort` subclass.

$$ComputerSystem(?y) \wedge SystemDeviceGroupComponent(?z, ?y) \wedge \tag{9.5a}$$
$$\wedge\ SystemDevicePartComponent(?z, ?a) \wedge \tag{9.5b}$$
$$\wedge\ SystemDeviceGroupComponent(?z, ?y1) \wedge \tag{9.5c}$$
$$\wedge\ SystemDevicePartComponent(?z, ?a1) \wedge WirelessPort(?a) \wedge \tag{9.5d}$$
$$\wedge\ WirelessPort(?a1) \wedge WirelessPortFrequency(?a, ?b) \wedge \tag{9.5e}$$
$$\wedge\ WirelessPortFrequency(?a1, ?b1) \wedge swrlb : subtract(?delta, ?b, ?b1) \wedge \tag{9.5f}$$
$$\wedge\ swrlb : lessThanOrEqual(?delta, 24) \rightarrow interferes(?a, ?a1) \tag{9.5g}$$

More complex policies and taxonomies are defined using SWRL to define new properties. In the following example, `WirelessPort` instances are defined as interfering one another (i.e. a symmetric property) if their frequency difference is less than 25 MHz (term 9.5g). This property can be used in turn in more complex rules. For example, if throughput degradation between two ports is detected, an interference can explain its origin.

9.5. Related Work

Some research has been done in the area of managing wireless mesh networks. An interesting scheme for troubleshooting faults in WMNs based on a simulator is presented by Qiu et al.. [85] The approach is novel and suitable for automated fault detection in WMNs. In Kim and Shin [50] a scheme for accurate measurement of link quality in a wireless mesh network is introduced. Link quality experiences fluctuations and often induces performance degradation. Accurately measuring it is therefore important for multiple reasons: routing, fault diagnosis or identifying high-quality channels. In Zhang and Fang [111] the challenges and fundamentals of security operations are enumerated, and an attack resilient security architecture for WMNs (ARSA) is presented. This schema involves the existence of WMN operators, but does not rule out the possible use of them in the context of community WMNs.

For the configuration and accounting of WMNs, commercial solutions are available from Nortel [71] or LocustWorld [53]. Nortel makes use of the Network Operations Support System (NOSS), a system software component that offers centralized monitoring and managing operations. The NOSS consists of the Enterprise Network Management System (ENMS), FTP, Remote Authentication Dial In User Service (RADIUS), Dynamic Host Configuration Protocol (DHCP) and Simple Network Time Protocol (SNTP) servers. ENMS provides performance, configuration and fault management, and discovers every wireless Access Point (AP). The DHCP server provides dynamic IP addresses for wireless APs and mobile nodes. The RADIUS server performs mobile and wireless AP authentication, authorization and accounting. The FTP server stores configuration files downloaded by the wireless APs when turned on. The SNTP server provides the wireless APs time parameters needed for timestamping events. The solution stands out for a high number of nodes in the network, but is limited by the centralised management approach (i.e. centralised NOSS).

Staub et al. [95] tackle the challenges of defective configurations or errors in mesh routers, and propose a distributed automated reconfiguration architecture. The approach uses Cfengine [10] to distribute configuration and updates among the nodes in the WMN backbone. Each node asks its neighbours for configuration updates, and “pulls” the new configuration to its storage. This allows simple overwriting of the current configuration, which is backed up for fall back in case of an erroneous new configuration, but it does not allow for extraction and analysis of the current state of the network.

There are works that use ontologies or formal models for configuration data: Sinz et al. develop in [93] a CIM-based formal model much alike description logics mentioned in Section 9.4.1 (except for the use of predicates with arity greater than 2), which is used to check for inconsistencies on Apache configuration files, as Glasner and Sreedhar do in [36], but with a custom OWL model and with more emphasis on decidability. García et al. [35] make a transformation of the CIM model into OWL and use SWRL rules, but they do not give details about the implementation and the problems of mixing rules and description logics. The work by Quirolgico et al. [86] is an earlier exploratory effort that used RDF Schema instead of OWL (which was still being standardized), so it lacked cardinality restrictions, used in [93], [36] and in the case study of this chapter for certain structural constraints.

9.6. Conclusions

We have presented the application of AdCIM to a real mesh network router, by analyzing its configuration, first separating its format and underlying entities, and defining a CIM mapping that supports the complete expression of the router configuration, with special care for storing format intricacies without polluting more abstract objects. This mapping and its opposite (from CIM objects to router configuration) was implemented through XSLT stylesheets, finally allowing these data to be stored and used to create web forms by the AdCIM framework (see Fig. 6.7 in Chapter 6).

Our approach allows for the representation and off-line analysis of the current state of the WMN and also presents it in an integrated manner that is used to prevent broken configuration states before their application. This off-line analysis is done using an ontological model that allows detection of semantic errors and conflicts. Policy enforcement is also supported, and the semantical representation widens the range of application by both discovering unforeseen equivalences (e.g., a new property value is inferred and a policy is triggered), and by using the associations in the underlying CIM model, for example by employing both the physical location and vendor of a product to enforce rules.

Chapter 10

Conclusions and Future Work

In this chapter, we wrap up the description of the work done in this Thesis and provide some insights on the future research direction.

10.1. Conclusions

In this work we have outlined AdCIM, a model-driven framework based on the CIM model and aimed to the development of integrated system administration applications for distributed systems. Its model-driven nature means that it is capable of rapid prototyping and adaptation to complex scenarios, such as those described in Chapters 7, 8 and 9. AdCIM provides XSLT stylesheets that produce several artifacts from the schema, so extensions and modifications on it can be rapidly transformed into database schemata, forms or XML Schema documents. These artifacts are also guaranteed to preserve semantics and constraints, since they are directly derived from the schema.

AdCIM also supports the extraction of CIM data from several configuration sources, and specially from flat file configurations, prevalent on Unix systems, but also supports the WMI system in Windows systems, firmware configurations and other sources. In the case of flat files, complex configurations such as the Sendmail configuration file are supported, as covered in Chapter 7.

AdCIM also derives database schemata, as part of the supported artifacts, and using provided XSLT stylesheets supports transparent persistence of CIM data on LDAP repositories. The performance of these XSLT stylesheets is shown to be both scalable and multicore-aware. Part of this good performance is due to the use of our miniCIM format, which minimizes the overhead of using XML-encoded CIM data. This is possible using a derived XML Schema to define an efficient encoding.

XForms forms are generated for the manipulation of CIM instances. They provide a general, user-friendly, model-driven access to management and configuration data. These forms are easily prototypable, but at the same time support CSS customization and can be displayed in any common browser. Their access to the CIM data is mediated using a REST web service interface that exposes these data for web clients and external applications.

Last, but not least, the semiformal CIM specification is complemented with added constraints and definitions to build an OWL formal ontological specification, with support for formal reasoning capabilities and for the enforcement of policies based on Horn rules.

Summarizing, AdCIM provides infrastructure for:

- Efficient representation and extension of the CIM model using a custom XML representation (called miniCIM).
- Grammar-based extraction of configuration and management information as miniCIM instances. As a complex case study, we have adapted this process to the configuration of the Sendmail mail agent.
- Transparent persistence of miniCIM data into a repository (an LDAP directory, although AdCIM is repository-independent), including schema mapping. We have shown how this persistence support scales near linearly in multicore systems.
- Querying and modifying miniCIM information backed by the repository through REST-style web services.
- Pregeneration and styling of XForms web user interfaces for the administration applications that can communicate directly with web services.
- Formal ontological specification of management domains with support for reasoning processes and policy enforcement.

These software components are assembled to abstract, integrate and associate all configuration and management data in a coherent whole. The application development task is expedited by framework-provided XSLT stylesheets and the use of integration technologies, such as REST web services, XForms and XSLT. Although the framework footprint in deployed nodes depends on the implementation chosen for the repository, XSLT processors and XForms, we have achieved a small footprint (less than 1 Mb) for the core XSLT transformation functionality by using a C-based XSLT processor with compressed XML inputs.

The AdCIM project also has a homepage at `http://adcim.des.udc.es`.

10.2. Future Work

Future work includes:

- The study of other distributed repositories. Although LDAP has good replication characteristics it is possible to do a complete mapping with good performance (see Chapter 5), there is still some impedance due to the comparative simplicity of the LDAP data model, and currently it is not possible to realize queries on persisted associations without using the AdCIM framework. It would be useful to conduct queries using only standard tools in a repository that did not require modifying and qualifying attribute names. Another interesting possibility would be a distributed persistence solution that did not require dedicated repository nodes and could survive extreme cases of network separation or disconnection.

- Advanced use of ontologies. Chapter 9 formulates ontological processes which chain error conditions, do domain conversions and detect misconfigurations, but there is still much work to be done in this area, specially on fault inference, complementation of network discovery, service availability prediction, failure simulation, load prediction, and many more diagnosis and prediction processes.

- Development of diagnosis and automatic recovery in complex situations. This objective would use ontological tools to achieve continued operation on networks in which the number and quality of nodes are not known. The processes would start with a definition of the tasks needed and the relationships and hard dependences generated by these tasks and would try to map and enforce these on an ad-hoc or unknown network.

- Extension to other management domains. For example, cloud computing offers new domains, like the modelization of common functionality of several cloud solutions, the modelization of the security and integrity of sensitive documents, provisioning with cost and scalability prediction, computation time estimation, and the extraction of the configuration for the cloning and customization of tailored virtual machines for specific computing tasks. All these domains would benefit greatly from the use of ontologies. Additionally, sensor networks could be another interesting domain for AdCIM to cover.

Bibliography

[1] I. F. Akyildiz, X. Wang, and W. Wang. Wireless mesh networks: a survey. *Computer Networks*, vol. 47, num. 4, pages 445–487, 2005. Cited in p. 111

[2] G. Aloisio, M. Cafaro, I. Epicoco, S. Fiore, D. Lezzi, M. Mirto, and S. Mocavero. iGrid, a novel grid information service. In *Advances in Grid Computing - Proceedings of the European Grid Conference, EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 506–515, Amsterdam, The Netherlands, February 2005. Cited in p. 109

[3] P. Anderson and A. Scobie. LCFG: the next generation. In *Proceedings of the UK Unix & Open Systems User Group Large Installation Systems Administration Winter Conference, UKUUG/LISA 2002*, London, UK, February 2002. Available at: `http://www.lcfg.org/doc/ukuug2002.pdf`. Cited in p. 24

[4] Apache Software Foundation. OpenJPA User's Guide. Available at: `http://openjpa.apache.org/builds/1.2.0/apache-openjpa-1.2.0/docs/manual/` [Last accessed 10 March 2010]. Cited in p. 104

[5] Apache Software Foundation. XSLTC documentation. 2001. Available at: `http://xml.apache.org/xalan-j/xsltc/index.html` [Last accessed 10 March 2010]. Cited in p. 21

[6] M. Bajohr and T. Margaria. MaTRICS: a service-based management tool for remote intelligent configuration of systems. *Innovations in Systems and Software Engineering*, vol. 2, num. 2, pages 99–111, July 2006. Cited in p. 23

[7] K. Birman and F. Schneider. The monoculture risk put into context. *IEEE Security & Privacy*, vol. 7, num. 1, pages 14–17, January-February 2009. Cited in p. v, 1

[8] R. Biswas and E. Ort. The Java Persistence API - a simpler programming model for entity persistence. May 2006. Available at: `http://java.sun.com/developer/technicalArticles/J2EE/jpa/` [Last accessed 10 March 2010]. Cited in p. 104

[9] D. Booth, H. Haas, and F. McCabe. Web services architecture. 2004. Available at: `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/` [Last accessed 10 March 2010]. Cited in p. 42

[10] M. Burgess and R. Ralston. Distributed resource administration using Cfengine. *Software Practice & Experience*, vol. 27, num. 9, pages 1083–1101, September 1997. Cited in p. 24, 126

[11] Chiba Project. 2008. Available at: `http://chiba.sourceforge.net/` [Last accessed 10 March 2010]. Cited in p. 72

[12] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. 1999. Available at: `http://www.w3.org/TR/xpath/` [Last accessed 10 March 2010]. Cited in p. 18

[13] K. Clark and B. Parsia. Pellet: the open source OWL DL reasoner. 2009. Available at: `http://clarkparsia.com/pellet` [Last accessed 10 March 2010]. Cited in p. 121

[14] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). October 2003. Available at: `http://www.ietf.org/rfc/rfc3626.txt` [Last accessed 10 March 2010]. Cited in p. 115

[15] B. Costales, C. Assmann, G. Jansen, and C. Saphiro. *Sendmail, Fourth Edition*. O'Reilly & Associates, Inc., Sebastopol, USA, October 2007. Cited in p. 87

[16] Côdeazur brasil. DENG, the modular XML browser engine. 2006. Available at: `http://deng.com.br` [Last accessed 10 March 2010]. Cited in p. 72

[17] A. Dalke. Martel. 2008. Available at: `http://www.dalkescientific.com/Martel/` [Last accessed 10 March 2010]. Cited in p. 38

[18] DataPower. XML accelerator XA35. 2009. Available at: `http://www-01.ibm.com/software/integration/datapower/xa35/` [Last accessed 10 March 2010]. Cited in p. 21

[19] N. Desai. bcfg2. In *Proceedings of the 20th USENIX Conference on Systems Administration, LISA 2006*, Washington DC, USA, December 2006. Cited in p. 24

[20] I. Díaz, G. Fernández, M. J. Martín, P. González, and J. Touriño. Integrating the Common Information Model with MDS4. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing, Grid 2008*, pages 298–303, Tsukuba, Japan, September 2008. Cited in p. VII, 2

[21] I. Díaz, C. Popi, O. Festor, J. Touriño, and R. Doallo. Ontological configuration management for wireless mesh routers. In *Proceedings of the 9th IEEE International Workshop on IP Operations and Management, IPOM 2009*, volume 4773 of *Lecture Notes in Computer Science*, pages 116–129, Venice, Italy, October 2009. Cited in p. VII, 2

[22] I. Díaz, J. Touriño, J. Salceda, and R. Doallo. A framework focus on configuration modeling and integration with transparent persistence. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005, Workshop on System Management Tools for Large-Scale Parallel Systems*, page 297a, Denver, USA, April 2005. Cited in p. VII, 2

[23] Distributed Management Task Force (DMTF). Common Information Model (CIM) Standards. 2008. Available at: `http://www.dmtf.org/standards/cim` [Last accessed 10 March 2010]. Cited in p. VI, 2, 11

[24] Distributed Management Task Force (DMTF). Specification for the representation of CIM in XML. May 2002. Available at: `http://www.dmtf.org/standards/documents/WBEM/DSP201.html` [Last accessed 10 March 2010]. Cited in p. 27

[25] E. Dolstra and A. Löh. NixOS: a purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008*, pages 367–378, Victoria, Canada, September 2008. Cited in p. 22

[26] J. Dunkel, R. Bruns, and A. Holitschke. Comparison of JavaServer pages and XSLT: a software engineering perspective. *Software Practice & Experience*, vol. 34, num. 1, pages 1–13, January 2004. Cited in p. 18

[27] Enabling Grids for E-sciencE. Available at: `http://www.eu-egee.org/` [Last accessed 10 March 2010]. Cited in p. 108

[28] D. W. Erwin. UNICORE - a grid computing environment. *Concurrency and Computation: Practice and Experience*, vol. 14, num. 13-15, pages 1395–1410, 2002. Cited in p. 109

[29] R. Evard. An analysis of Unix system configuration. In *Proceedings of the 11th USENIX Conference on Systems Administration, LISA 1997*, pages 179–194, San Diego, USA, October 1997. Cited in p. 37

[30] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, vol. 2, num. 2, pages 115–150, May 2002. Cited in p. 11, 80

[31] J. Finke. Generating configuration files: the director's cut. In *Proceedings of the 17th USENIX Conference on Systems Administration, LISA 2003*, pages 195–204, San Diego, USA, October 2003. Cited in p. 59

[32] J. Finke. An improved approach for generating configuration files from a database. In *Proceedings of the 14th USENIX Conference on Systems Administration, LISA 2000*, pages 29–38, New Orleans, USA, December 2000. Cited in p. 59

[33] I. Foster. Globus Toolkit version 4: software for service-oriented systems. *Journal of Computer Science and Technology*, vol. 21, num. 4, pages 513–520, July 2006. Cited in p. 97

[34] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the Grid: an open grid services architecture for distributed systems integration. 2002. Available at: `www.globus.org/alliance/publications/papers/ogsa.pdf` [Last accessed 10 March 2010]. Cited in p. 97

[35] F. J. García Clemente, G. Martínez Pérez, J. A. Botía Blaya, and A. F. Gómez-Skarmeta. On the application of the semantic web rule language in the definition of policies for system security management. In *Proceedings of the 4th On the Move to Meaningful Internet Systems Workshops, OTM 2005*, volume 3762 of *Lecture Notes in Computer Science*, pages 69–78, Agia Napa, Cyprus, October 2005. Cited in p. 121, 127

[36] D. Glasner and V. Sreedhar. Configuration reasoning and ontology for web. In *Proceedings of the 4th IEEE International Conference on Services Computing, SCC 2007*, pages 387–394, Salt Lake City, USA, July 2007. Cited in p. 119, 127

[37] Globus Alliance. Globus Toolkit 4.0 web services monitoring and discovery system: cluster monitoring information and the GLUE resource property. Available at: `http://www.globus.org/toolkit/docs/4.0/info/key/gluerp.html` [Last accessed 10 March 2010]. Cited in p. 102

[38] Globus Alliance. Globus Toolkit 4.2.1 web services monitoring and discovery system: UsefulRP Guides. Available at: `http://www.globus.org/toolkit/docs/4.2/4.2.1/info/usefulrp/usefulrp.pdf` [Last accessed 10 March 2010]. Cited in p. 99

[39] Globus Alliance. Globus Toolkit monitoring and discovery system information providers. Available at: `http://www.globus.org/toolkit/docs/latest-stable/info/providers/` [Last accessed 10 March 2010]. Cited in p. 108

[40] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog configuration management framework. *ACM SIGOPS Operating Systems Review*, vol. 43, num. 1, pages 16–25, January 2009. Cited in p. 24

[41] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, 2004. Cited in p. VI, 2

[42] D. Grisby. Free high performance ORB. 2008. Available at: `http://omniorb.sourceforge.net/` [Last accessed 10 March 2010]. Cited in p. 43

[43] D. Heimbigner. DMTF - CIM to OWL: a case study in ontology conversion. In *Proceedings of the 16th Conference on Software Engineering and Knowledge Engineering, SEKE 2004*, pages 470–473, Banff, Canada, June 2004. Cited in p. 121

[44] International Electrotechnical Commission (IEC). Energy management system application program interface (EMS-API) - part 301: Common Information Model (CIM) base. November 2003. Available at: `http://webstore.iec.ch/webstore/webstore.nsf/artnum/033912!OpenDocument&Click=` [Last accessed 10 March 2010]. Cited in p. 22

[45] International Organization for Standardization and International Electrotechnical Commission Joint Technical Committee (ISO/IEC JTC). Document description and processing languages. 1996. Available at: `http://www.`

y-adagio.com/public/sc34wg2/index.htm [Last accessed 10 March 2010]. Cited in p. 17

[46] International Organization for Standardization (ISO). Standard Generalized Markup Language (SGML). 1986. Available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=16387 [Last accessed 10 March 2010]. Cited in p. 17

[47] Internet Engineering Task Force (IETF). Available at: http://www.ietf.org/ [Last accessed 10 March 2010]. Cited in p. 51

[48] M. Kay. The Saxon XSLT processor. 2009. Available at: http://saxon.sf.net [Last accessed 10 March 2010]. Cited in p. 21, 66

[49] B. W. Kernighan and D. M. Ritchie. The M4 macro processor. Technical report, Bell Laboratories, Murray Hill, 1977. Cited in p. 87

[50] K.-H. Kim and K. G. Shin. On accurate measurement of link quality in multi-hop wireless mesh networks. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking, MOBICOM 2006*, pages 38–49, Los Angeles, USA, September 2006. Cited in p. 126

[51] Laboratoire des Sciences de l'Images, de l'Informatique et de la Télédétection (LSIIT). Project NEMO. 2008. Available at: https://lsiit-cnrs.unistra.fr/rp-fr/index.php/DemoIPv6Nemo [Last accessed 10 March 2010]. Cited in p. 113

[52] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, vol. 29, num. 3, pages 76–87, September 2000. Cited in p. 29

[53] LocustWorld. Bio-diverse networking. 2009. Available at: http://www.locustworld.com/ [Last accessed 10 March 2010]. Cited in p. 126

[54] P. Loshin. *Big book of Lightweight Directory Access Protocol (LDAP) RFCs.* Academic Press, Inc., Orlando, USA, 2000. Cited in p. 50, 54

[55] M. Majewska, B. Kryza, and J. Kitowski. Translation of Common Information Model to Web Ontology Language. In *Proceedings of the 7th International Conference on Computational Science, ICCS 2007*, volume 4487 of *Lecture Notes in Computer Science*, pages 414–417, Beijing, China, May 2007. Cited in p. 121

[56] H. Mao, L. Huang, and M. Li. Web resource monitoring based on Common Information Model. In *Proceedings of the 2006 IEEE Asia-Pacific Conference on Services Computing, APSCC 2006*, pages 520–525, GuangZhou, China, December 2006. Cited in p. 108

[57] M. Massie, B. Chun, and D. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, vol. 30, num. 5-6, pages 817–840, June 2004. Cited in p. 102

[58] O. Mehl, M. Becker, A. Köppel, P. Paul, D. Zimmermann, and S. Abeck. A management-aware software development process using design patterns. In *Proccedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management, IM 2003*, pages 579–592, Colorado Springs, USA, March 2003. Cited in p. 22

[59] A. S. Memon, M. S. Memon, P. Wieder, and B. Schuller. CIS: an information service based on the Common Information Model. In *Proceedings of the 3rd IEEE International Conference on e-Science and Grid Computing*, pages 465–473, Bangalore, India, December 2007. Cited in p. 109

[60] Microsoft Corporation. Windows Services for UNIX 3.5 white paper. 2004. Available at: `http://technet.microsoft.com/en-us/library/bb463214.aspx` [Last accessed 10 March 2010]. Cited in p. VI, 1

[61] Microsoft Corporation. WMI - Windows Management Instrumentation. June 2000. Available at: `http://msdn.microsoft.com/en-us/library/aa394582(VS.85).aspx` [Last accessed 10 March 2010]. Cited in p. 37, 40

[62] J. C. Mogul. Clarifying the fundamentals of HTTP. In *Proceedings of the 11th International Conference on World Wide Web, WWW 2002*, pages 25–36, Honolulu, USA, May 2002. Cited in p. 80

[63] B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. In *Proceedings of the 3rd International Semantic Web Conference, ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 549–563, Hiroshima, Japan, November 2004. Cited in p. 120

[64] Mozilla Corporation. Mozilla XForms project. 2008. Available at: `http://www.mozilla.org/projects/xforms/` [Last accessed 10 March 2010]. Cited in p. 72

[65] H. Nakada, K. Saga, Y. Saeki, H. Sato, M. Hatanaka, and S. Matsuoka. Job invocation interoperability between NAREGI Middleware Beta and gLite. In *Proceedings of the 9th IEEE International Conference on High Performance Computing in Asia Pacific Region*, pages 298–303, Seoul, Korea, October 2008. Cited in p. 109

[66] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. Li. Current approaches to XML management. *IEEE Internet Computing*, vol. 6, num. 4, pages 43–51, July 2002. Cited in p. 59

[67] National Research Grid Initiative. Available at: `http://www.naregi.org/index_e.html` [Last accessed 10 March 2010]. Cited in p. 108

[68] National Science Foundation (NSF). The TeraGrid project. Available at: `http://www.teragrid.org/` [Last accessed 10 March 2010]. Cited in p. 108

[69] A. Nikolaidis, G. Doumenis, G. Stassinopoulos, M.-P. Drakos, M. Anastasopoulos, and S. D'Haeseleer. Management traffic in emerging remote configuration mechanisms for residential gateways and home devices. *IEEE Communications Magazine*, vol. 43, num. 5, pages 154–162, May 2005. Cited in p. 44

[70] G. J. Noer. Cygwin32: a free Win32 porting layer for UNIX. In *Proceedings of the 2nd Conference on USENIX Windows NT, WINSYM 1998*, page 4, Seattle, USA, August 1998. Cited in p. VI, 1

[71] Nortel. WMN Solutions. 2009. Available at: `http://www2.nortel.com/go/solution_content.jsp?segId=0&catId=W&parId=0&prod_id=47160&locale=en-us` [Last accessed 10 March 2010]. Cited in p. 126

[72] Object Management Group (OMG). CORBA/IIOP specifications. 2008. Available at: `http://www.omg.org/technology/documents/corba_spec_catalog.htm` [Last accessed 10 March 2010]. Cited in p. 42

[73] Object Management Group (OMG). XMLDOM: DOM/Value mapping specification. 2001. Available at: `ftp://ftp.omg.org/pub/docs/ptc/01-04-04.pdf` [Last accessed 10 March 2010]. Cited in p. 42

[74] Open Science Grid (OSG). Available at: `http://www.opensciencegrid.org/` [Last accessed 10 March 2010]. Cited in p. 108

[75] OpenLDAP Foundation. OpenLDAP. 2008. Available at: `http://www.openldap.org` [Last accessed 10 March 2010]. Cited in p. 59

[76] Orbeon. Orbeon Presentation Server. 2009. Available at: `http://www.orbeon.com` [Last accessed 10 March 2010]. Cited in p. 72

[77] Organization for the Advancement of Structured Information Standards (OASIS). Available at: `http://www.oasis-open.org` [Last accessed 10 March 2010]. Cited in p. 54, 57

[78] Organization for the Advancement of Structured Information Standards (OASIS). The DSMLv2 standard. 2002. Available at: `http://www.oasis-open.org/specs/index.php` [Last accessed 10 March 2010]. Cited in p. 54

[79] OSGi Alliance. Open Services Gateway initiative. 2007. Available at: `http://www.osgi.org/Main/HomePage` [Last accessed 10 March 2010]. Cited in p. 113

[80] OSGi Alliance. *OSGi Service Platform Service Compendium, release 4*. 2007. Available at: `http://www.osgi.org/Specifications/HomePage`. Cited in p. 113

[81] L. D. Paulson. Building rich web applications with Ajax. *Computer*, vol. 38, num. 10, pages 14–17, October 2005. Cited in p. 79

[82] M. Peltier. MTrans, a DSL for model transformation. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference, EDOC 2002*, pages 190–199, Lausanne, Switzerland, September 2002. Cited in p. 23

[83] H. Post and C. Sinz. Configuration lifting: verification meets software configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008*, pages 347–350, L'Aquila, Italy, September 2008. Cited in p. 22

[84] A. Pras, T. Drevers, R. van de Meent, and D. A. C. Quartel. Comparing the performance of SNMP and web services-based management. *IEEE Transactions on Network and Service Management*, vol. 1, num. 2, pages 72–82, December 2004. Cited in p. 44

[85] L. Qiu, P. Bahl, A. Rao, and L. Zhou. Troubleshooting wireless mesh networks. *ACM SIGCOMM Computer Communication Review*, vol. 36, num. 5, pages 17–28, October 2006. Cited in p. 126

[86] S. Quirolgico, P. Assis, A. Westerinen, M. Baskey, and E. Stokes. Toward a formal Common Information Model ontology. In *Proceedings of the 5th International Conference on Web Information Systems Engineering, WISE 2004*, volume 3307 of *Lecture Notes in Computer Science*, pages 11–21, Brisbane, Australia, November 2004. Cited in p. 119, 127

[87] S. Ravelomanana, S. C. S. Bianchi, C. Joumaa, and M. Sibilla. A contextual Grid monitoring by a model driven approach. In *Proceedings of the 3rd Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services, AICT-ICIW 2006*, pages 37–43, Guadeloupe, French Caribbean, February 2006. Cited in p. 108

[88] A.-I. Rivière and M. Sibilla. Management information models integration: from existing approaches to new unifying guidelines. *Journal of Network and Systems Management*, vol. 6, num. 3, pages 333–356, September 1998. Available at: `http://www.springerlink.com/content/u29k27h30500k55w`. Cited in p. 115

[89] J. Salceda, I. Díaz, J. Touriño, and R. Doallo. A middleware architecture for distributed systems management. *Journal of Parallel and Distributed Computing*, vol. 64, num. 6, pages 759–766, June 2004. Cited in p. VII, 2

[90] R. Salz. ZSI: The Zolera SOAP infrastructure. 2005. Available at: `http://pywebsvcs.sourceforge.net/zsi.html` [Last accessed 10 March 2010]. Cited in p. 43

[91] J. Schopf, I. Raicu, L. Pearlman, N. Miller, C. Kesselman, I. Foster, and M. D'Arcy. Monitoring and discovery in a web services framework: functionality and performance of Globus Toolkit MDS4. MCS Preprint 1315-0106, Argonne National Laboratory, January 2006. Cited in p. 97

[92] J. Schott, A. Westerinen, J.-P. Martin-Flatin, and P. Rivera. Common information vs. information overload. In *Proceedings of the 8th IFIP/IEEE Network Operations and Management Symposium, NOMS 2002*, pages 767–781, Florence, Italy, April 2002. Cited in p. 22

[93] C. Sinz, A. Khosravizadeh, W. Küchlin, and V. Mihajlovski. Verifying CIM models of Apache web-server configurations. In *Proceedings of the 3rd International Conference on Quality Software, QSIC 2003*, pages 290–297, Dallas, USA, November 2003. Cited in p. 22, 119, 127

[94] D. D. Spinellis. Outwit: UNIX tool-based programming meets the Windows world. In *Proceedings of the 6th USENIX Annual Technical Conference, ATEC 2000*, pages 149–158, San Diego, USA, June 2000. Cited in p. VI, 1

[95] T. Staub, D. Balsiger, M. Lustenberger, and T. Braun. Secure remote management and software distribution for wireless mesh networks. In *Proceedings of the 7th International Workshop on Applications and Services in Wireless Networks, ASWN 2007*, Santander, Spain, May 2007. Cited in p. 126

[96] F. Strauss and T. Klie. Towards XML oriented internet management. In *Proccedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management, IM 2003*, pages 505–518, Colorado Springs, USA, March 2003. Cited in p. 23

[97] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007*, pages 237–250, Stevenson, USA, October 2007. Cited in p. 23

[98] Y. Sun, T. Li, Q. Zhang, J. Yang, and S. Liao. Parallel XML transformations on multi-core processors. In *Proceedings of the 3rd IEEE International Conference on e-Business Engineering, ICEBE 2007*, pages 701–708, Hong Kong, China, October 2007. Cited in p. 21, 66

[99] S. Tursunova, T. Son, and Y. Kim. Grid resource management with tightly coupled WBEM/CIM local management. In *Proceedings of the 11th IEEE/IFIP Network Operations and Management Symposium, NOMS 2008*, pages 983–986, Salvador, Brazil, April 2008. Cited in p. 108

[100] L. Wang, M. Kunze, and J. Tao. From CIM to GLUE: translate resource information of virtual machines to computational grids. In *GI/ITG Kommunikation und Verteilte Systeme, Fachgespräch "Virtualisierung"*, pages 55–63, Paderborn, Germany, February 2008. Cited in p. 109

[101] D. Wood. Guidelines for CIM-to-LDAP directory mappings. 2000. Available at: `http://www.dmtf.org/standards/documents/DEN/DSP0100.pdf` [Last accessed 10 March 2010]. Cited in p. 59

[102] World Wide Web Consortium (W3C). Available at: `http://www.w3.org/` [Last accessed 10 March 2010]. Cited in p. 17

[103] World Wide Web Consortium (W3C). Cascading Style Sheets, level 2 CSS2 specification. May 1998. Available at: `http://www.w3.org/TR/REC-CSS2/` [Last accessed 10 March 2010]. Cited in p. 11

[104] World Wide Web Consortium (W3C). Extensible Stylesheet Language Family (XSL). 1997. Available at: `http://www.w3.org/Style/XSL/` [Last accessed 10 March 2010]. Cited in p. 18

[105] World Wide Web Consortium (W3C). OWL - Ontology Language for the Web. 2004. Available at: `http://www.w3.org/TR/2004/REC-owl-features-20040210/` [Last accessed 10 March 2010]. Cited in p. 112, 120

[106] World Wide Web Consortium (W3C). Semantic Web Rule Language (SWRL). 2004. Available at: `http://www.w3.org/Submission/SWRL/` [Last accessed 10 March 2010]. Cited in p. 120

[107] World Wide Web Consortium (W3C). XForms 1.0 (third edition). 2007. Available at: `http://www.w3.org/TR/2007/REC-xforms-20071029/` [Last accessed 10 March 2010]. Cited in p. 11, 69

[108] World Wide Web Consortium (W3C). XML Schema. October 2004. Available at: `http://www.w3.org/TR/xmlschema-0/` [Last accessed 10 March 2010]. Cited in p. 29

[109] S.-M. Yoo, H.-T. Ju, and J. W.-K. Hong. Performance improvement methods for NETCONF-based configuration management. In *Proceedings of the 9th Asia-Pacific Network Operations and Management Symposium, APNOMS 2006*, volume 4238 of *Lecture Notes in Computer Science*, pages 242–252, Busan, Korea, September 2006. Cited in p. 44

[110] J.-H. Yoon, H.-T. Ju, and J. W.-K. Hong. Development of SNMP-XML translator and gateway for XML-based integrated network management. *International Journal of Network Management*, vol. 13, num. 4, pages 259–276, July 2003. Cited in p. 23

[111] Y. Zhang and Y. Fang. ARSA: an attack-resilient security architecture for multihop wireless mesh networks. *IEEE Journal on Selected Areas in Communications*, vol. 24, num. 10, pages 1916–1928, October 2006. Cited in p. 126

Glossary

Assertional Box (ABox)

The part of an ontology which contains knowledge about individuals and their assigned class. 121–123

Asynchronous JavaScript and XML (AJAX)

A group of web technologies that enable web applications to retrieve and update data (usually JSON or XML) in the background without interfering in page display and interaction. 78, 79

Access Point (AP)

A network device that interconnects wireless interfaces between them and optionally to other network. 126

Base64

A binary codification scheme for transmitting and storing arbitrary binary contents as ASCII characters. In its MIME encarnation, it codifies 3 bytes into 4 ASCII characters. 43, 55, 90–92, 94

Blocks Extensible Exchange Protocol (BEEP)

A framework for creating network protocols that abstracts common features and facilitates protocol development. 44

Basic Encoding Rules (BER)

A set of rules defined by the International Telecommunication Union (ITU) as part of Abstract Syntax Notation One (ASN.1) to encode abstract structures of data types into byte streams. 44

Component Object Model (COM)

A Microsoft component interface for interprocess communication and interchange of executable objects among different languages. It includes the DCOM distributed object interface for remote object invocation. 40, 45

Common Object Request Broker (CORBA)

A standard defined by the Object Management Group (OMG) that defines interfaces for the intercommunication of components written in different languages and architectures. These components can also be invoked transparently in distributed contexts. 35, 42, 43, 45, 47

Cascading Style Sheets (CSS)

A language which describes the presentation characteristics of HTML and XML languages, such as layout, colors, fonts and accessibility. A CSS stylesheet supports several display devices at the same time and the selective overriding of rules. 69, 72, 77, 78

Dynamic Host Configuration Protocol (DHCP)

A networking protocol used mainly for the centralized configuration of the IP addresses and other necessary parameters of network connectivity for client machines of a network. 114, 126

Distributed Management Task Force (DMTF)

An organization tasked with the development of system management standards for enterprise environments. 2, 8, 27, 59

Distinguished Name (DN)

In LDAP, a unique entry name consisting of the entry RDN comma-appended to the parent DN. 52–57, 59, 60, 62

Domain Name System (DNS)

A distributed directory system for assigning names to fixed IPs in the Internet or in a private network. It is based on a hierarchy of servers associated with a domain responsible of the naming of the machines under it. 50

Document Object Model (DOM)

A standard IDL-based object representation for XML trees that also defines common tree modification operations. It is the most supported interface to manipulate XML data, but implementations must hold all the tree in memory. 21

Directory Services Markup Language (DSML)

An XML-based LDAP interchange format for the description of LDAP entries and operations. 8, 10, 54, 57, 58, 63, 66

Document Style Semantics and Specification Language (DSSSL)

A Scheme-based stylesheet language for the transformation and displaying of SGML documents. 17, 18

Interface Description Language (IDL)

A language for the definition of software interfaces in a language- and architecture-independent way. 42

International Electrotechnical Commission (IEC)

An international non-profit standards organization that defines standards for electrical technologies. 22

Internet Engineering Task Force (IETF)

An international volunteer-based standards organization which defines new Internet standards. 2, 51

Inetd

A Unix daemon tasked with the management of Internet exposed services. 13, 38

Java Server Pages (JSP)

A Java templating technology that generates dynamic web pages from small scripts embedded on the webpage or special markup instructions that are compiled dynamically as Java bytecode. 18

Lightweight Directory Access Protocol (LDAP)

A protocol developed to simplify access to directory services. 3, 8, 10, 49–51, 53, 54, 56–63, 67, 72, 80, 90, 98, 112, 129

LDAP Interchange Format (LDIF)

A text-based LDAP data interchange format for the description of LDAP entries and operations. 8, 10, 54, 56–59, 62, 66

markup

A system of annotations in text content that gives syntactical and structural meaning not intrinsic to that text. 18, 78

Monitoring and Discovery System (MDS)

A framework included in the Globus grid toolkit that enables the generic publication and consumption of grid information. 97–99

Monoculture

In computing, a situation in which all or almost all of the computing resources in an organization are of the same platform. 1

NETwork CONFiguration (NETCONF)

An IETF standard that provides XML-based mechanisms to install, manipulate and delete the configuration of network devices. 44

Network Information Service (NIS)

A directory service developed by Sun Microsystems for the distribution and synchronization of configuration data of Unix systems like host and user data. 1

Open Database Connectivity (ODBC)

A standard language- and architecture-independent API for using relational databases. It is based on the use of drivers for each database system. 1

Open Grid Forum (OGF)

A community organization for the standardization of Grid computing. 97

Open Grid Services Architecture (OGSA)

An Open Grid Forum (OGF) architectural specification defining a service-oriented Grid computing environment. 97

Object Identifier (OID)

A guaranteed world-unique identifier assigned by the IANA organization. It is hierarchical and consists of an arbitrary length sequence of dot-separated numbers. Organizations are assigned a branch and can define local hierarchies adding new levels of numbers. 64, 65

Optimized Link State Routing Protocol (OLSR)

An IP routing protocol designed for mobile ad-hoc networks. It is based on the dissemination of topology and route information on the discovered network nodes. 115

Open Services Gateway initiative (OSGi)

A Java-based framework defining life-cycle managed componented services that can collaborate transparently and interchangeably across several machines. 113–115, 117, 118

Open Systems Interconnection (OSI)

A past effort to define an open common network protocol stack for large networks. It was replaced by TCP/IP due to implementation and performance problems. 50

Web Ontology Language (OWL)

A knowledge representation language to define ontologies proposed by the W3C based on Description Logics and which comes in three flavors: OWL-Full, OWL/DL and OWL-Lite. 112, 113, 115, 119–124, 127, 130

Portable Operating System Interface for Unix (POSIX)

A set of common APIs supported by Unix-like Operating Systems. 1

Remote Authentication Dial In User Service (RADIUS)

A protocol for the authentication, authorization and accounting of external users on a dial-up or wireless service. 115, 126

Redundant Array of Independent Disks (RAID)

A term encompassing several methods to combine several hard disks to ensure reliability and obtain better performance with some storage costs. 13

Resource Description Framework (RDF)

A W3C standard that defines a triple based metadata representation for web resources. 119

Relative Distinguished Name (RDN)

In LDAP, an entry name consisting of an attribute and value present in the entry and designated as identifier. 52, 53, 55, 59

Registry

A registry is a database specialised in gathering information about a large number of entities. 1, 40

Representational EState Transfer (REST)

A type of web service architecture that promotes a single URL for every resource and the use of the common HTTP operators to manipulate them. 69, 79, 80, 130

Request for Comments (RFC)

An IETF memorandum to promote discussion or adoption of new Internet technologies. 54

Remote Procedure Call (RPC)

A class of inter-process communication technologies that generally abstract calls to procedures of other processes, normally on remote machines, so the call appears to clients as made inside the same process. 80

Simple API for XML (SAX)

An event-based API for XML parsers that allows streaming processing of XML documents. Using SAX documents can be processed before being completely received and without storing them in memory. 38, 91, 94

Scheme

A minimalist LISP dialect. As such, is a declarative language using lists as the main data type. 17

Standard Generalized Markup Language (SGML)

A 1986 ISO standard technology to define generalized markup languages. 17, 18

Simple Network Management Protocol (SNMP)

An IETF management protocol for monitoring and controlling network devices. 23, 44, 114

Simple Network Time Protocol (SNTP)

A simpler form of the Network Time Protocol (NTP) geared for embedded devices with less hardware requirements. 126

Simple Object Access Protocol (SOAP)

An XML-based protocol for the implementation of web services based on Remote Call Procedures (RPC) or document interchange. The meaning of the acronym was abandoned after version 1.2. 43, 44, 80

Structured Query Language (SQL)

A widely used standardized language for access to relational databases based on relational algebra. 18

Secure SHell (SSH)

A protocol for remote shell access to Unix-like systems which offered better security and encryption. 44

Service Set IDentifier (SSID)

A string identifier of up to 32 characters which identifies a wireless network. 115

Semantic Web Rule Language (SWRL)

A W3C proposal for a rule language based on Horn rules. 120, 121, 124, 125, 127

System Log

In Unix systems, a typical denomination of background system processes, and by extension of system services. 88

Terminological Box (TBox)

The part of an ontology that defines concepts, roles and their inclusion hierarchy. 121–123

Unified Modelling Language (UML)

A generalized modelling language for software engineering. 100

UNiform Interface to COmputing REsources (UNICORE)

A open source European Grid computing initiative. 109

Uniform Resource Identifier (URI)

An Internet-wide identifier that names and describes an Internet-accessible resource. 80

Uniform Resource Locator (URL)

The part of a URI that defines the server where a resource is available and the protocol needed to access it. 74, 79, 80

Virtual Local Access Network (VLAN)

A group of host machines that communicate as if they were connected to the same Local Access Network independently of their location. 115

Windows Management Instrumentation (WMI)

A Microsoft interface for defining CIM-based instrumentation for Windows systems. 37, 40, 47

Wireless Mesh Networks (WMN)

A type of wireless network arranged in a mesh topology. 111, 112, 119, 126, 127

Web Services Resource Framework (WSRF)

A set of specifications for stateless web services, used in the Globus toolkit. 98, 99

World Wide Web Consortium (W3C)

An international standards organization for World Wide Web technologies. 2, 17, 18, 29, 69, 119

eXtensible Markup Language (XML)

A simplification of the SGML standard used for encoding a large range of documents and messages and read them with common parsers. 7, 8, 10, 17–19, 23, 25, 27, 29, 31, 35, 37, 38, 40, 42–45, 47, 54, 57, 58, 66, 69, 70, 74, 78–80, 91, 92, 94, 98, 99, 114, 117

eXtensible Sheet Language Transformations (XSLT)

An XML-based declarative language for expressing arbitrary transformations on XML data. 2, 3, 8, 10, 11, 16–19, 21, 23, 27, 29, 31, 33, 35, 40, 58, 59, 62, 63, 66, 69, 72, 73, 80, 91, 112, 115, 117, 118, 121, 122, 127, 129