RC 22307 (W0201-078) January 15, 2002
Computer Science

IBM Research Report

An Approach for Managing Service Dependencies with XML and the Resource Description Framework

Alexander Keller
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Christian Ensel
Munich Network Management Team,
University of Munich,
Oettingenstr. 67, 80538 Munich, Germany,

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

An Approach for Managing Service Dependencies with XML and the Resource Description Framework

Christian Ensel * Alexander Keller †

Abstract

We describe a novel approach for applying XML, XPath and RDF to the problem of describing, querying and computing the dependencies among services in a distributed computing system. This becomes increasingly important in today's networked environments where applications and services rely on both local and outsourced sub–services. However, service dependencies are not made explicit in today's systems, thus making the task of problem determination particularly difficult.

A key contribution of the paper is a web–based architecture for retrieving and handling dependency information from various managed resources. Its core component is a dependency query facility allowing the application of queries and filters to dependency models; its output is a consolidated dependency graph that can then be used by fault management applications to perform additional problem determination tasks or event correlation. The definition of an XML based notation for specifying dependencies facilitates information sharing between the components involved in the process.

Keywords

Web–based Application Management; Dependency Analysis; RDF; XML; XPath

1 Introduction

The identification and tracking of dependencies between the components of distributed systems is becoming increasingly important for integrated fault management. Applications and services rely on a variety of supporting services that might be outsourced to a service provider; moreover, emerging dynamic e-business architectures [18] allow the composition of web–based e–business applications at runtime: The concept of *Web Services* [20, 27] consists in the dynamic advertisement, discovery and access of business functionality among multiple cooperating partners. Consequently, failures occurring in one service affect other services being offered to a customer, i.e., services have **dependencies** on other services.

For our discussion, we call services that depend on other services **dependents**, while services on which other services depend are termed **antecedents**. It is important to note that a service often plays both roles (e.g., a name service is required by many applications and services but is dependent on the proper functioning of other services, such as operating system and network infrastructure), thus leading to a **dependency hierarchy** that can be modeled as a directed graph. Figure 1 depicts a simplified application dependency graph for various components of an e–business system that we have used for designing, implementing and testing our approach. It represents an experimental Internet storefront application that involves a Web Server for serving the static content of the site, a Web Application Server for hosting the business logic (implemented as storefront servlets), and a back–end database system that stores the dynamic content of the application (such as product descriptions, user and manufacturer data, carts, payment information etc.).

Both service providers and customers require management tools that allow navigation through the dependency hierarchy, in order to analyse and track down the root cause of a service failure. In addition, service providers are interested in tools to determine *in advance* the impact of a service outage on other

*Munich Network Management Team, University of Munich, Oettingenstr. 67, 80538 Munich, Germany, E-Mail: *ensel@lmu.de*

†IBM Research Division, T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, E-Mail: *alexk@us.ibm.com*

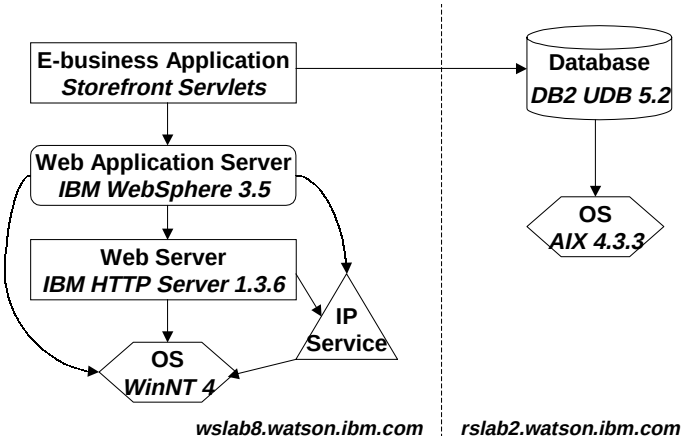

Figure 1: Simplified application dependency graph of an e–business system

services and users for scheduling server maintenance intervals (e.g., deploying backup systems when a production server has to be brought down for performing a software upgrade).

However, the main problem today lies in the fact that dependencies between services and applications are not made explicit, thus making root cause and impact analysis particularly difficult [10]. Solving this problem requires the determination and computation of dependencies between services and applications across different systems and domains, i.e., establishing a 'global' service dependency model and enabling system administrators to navigate through the resulting directed graph from the top to the bottom and in reverse order.

What is needed is a dynamic model reflecting the dependency relationships between different services; in addition, a management system should be capable of providing various mechanisms to select parts of a dependency model according to user–defined criteria. The latter capabilities are similar to the CMISE scoping and filtering mechanism of the OSI/TMN management framework. The difference is that scoping and filtering assumes a tree–like representation of management information while dependencies form more complex, directed graphs, as mentioned above.

The solution described in this paper is based on XML, *XML Path Language (XPath)* [31] and the *Resource Description Framework (RDF)* [24], an emerging specification of the W3 Consortium. It provides a uniform interface to query service and dependency information across the systems of a distributed environment and can be used by various fault and topology management applications, and event correlation systems.

The paper is structured as follows: Section 2 states the requirements for determining application and service dependencies, presents related work and gives an overview of the proposed architecture and its components. Section 3 introduces the core technologies that we have used for designing our solution, namely XML, RDF and the XPath language. Further, it analyses how these can be used to represent and process dependency information and gives a concrete example that applies our methodology to an e–business scenario. The proof–of–concept prototype implementation is described in section 4. Section 5 concludes the paper and presents issues for further research.

2 Service and Application Dependencies

From a conceptual perspective, dependency graphs provide a straightforward means to identify possible root causes of an observed problem: If the dependency graph for a system is known, navigating the graph from an impaired service towards its antecedents—being either co-located on the same host or on different systems—will reveal which entities might have failed. Traversing the graph towards its root yields the

dependents of a service, i.e., the components that might fail if this service experiences an outage. However, there are a couple of roadblocks on the way towards appropriate dependency models:

1. The number of dependencies between many involved systems can be computed, but may become very large. From an engineering viewpoint, it is often undesirable—and sometimes impossible—to store a complete, *instantiated* dependency model at a single place. Traditional mechanisms used in network management platforms such as keeping an instantiated network map in a single platform therefore cannot be applied to dependencies due to the sheer number and the dynamics of the involved dependencies: Dependency instances exist between multiple objects (a single web application server typically supports about 5 applications, 10 Enterprise JavaBeans, hundreds of servlets) and tend to have a fairly short lifetime (sometimes confined to a single request or unit of work). These two facts make it prohibitive to follow a 'network–management–style' approach for the deployment of application, service and middleware dependency models. Instead, we propose to distribute the storage and computation of dependencies across the systems involved in the management process. Section 2.1 describes our architecture that is designed to meet these requirements.

2. As mentioned in the introduction, the acquisition of a service dependency model, even confined to a single host system, is a challenge on its own as today's systems do not provide appropriate management instrumentation. Although a complete discussion of mechanisms for generating dependency models is beyond the scope of this paper, section 2.2.2 gives a brief overview of some promising approaches aiming at establishing such 'local' dependency graphs.

3. Further, facilities for combining local dependency graphs, stored on every system, into a uniform dependency model are required. In addition, these facilities need to provide an API allowing management applications to issue queries against the dependency model. These queries will allow the retrieval of the direct antecedents of a specific service, or recursively determine the whole set of their sub–nodes, etc. The list of nodes received by the management application enables it to perform specific problem determination routines to check whether these services are operational. Section 3 describes our approach of coping with this problem.

4. As a subproblem of the previous issue, it should be kept in mind that dependency models are directed graphs. While, e.g., the OSI scoping and filtering capabilities (having a similar functionality to what we are striving for) are designed to operate on tree–like data structures, dependency analysis faces the problem that a very similar set of operations has to be provided for directed graphs. This raises the question which notation and which data format allows the efficient representation of graphs so that fine–grained query mechanisms can be applied to graphs. Section 3.3.3 describes our solution to this problem.

5. Finally, the notion of dependencies is very coarse and needs to be refined in order to be useful. Examples for this are the *strength* of a dependency (indicating the likelihood that a component is affected if its antecedent fails), the *criticality* (how important this dependency is w.r.t. the goals and policies of an enterprise), the *degree of formalization* (i.e., how difficult it is to obtain the dependency) and many more. While it is outside the scope of this paper to establish a taxonomy for dependencies, there is a need to add attributes to dependencies that allow their qualification and, accordingly, a need to reflect these attributes in the dependency representation. This is addressed by section 3.3.3.

2.1 An Architecture for Service Dependencies

Our distributed three–tier architecture, depicted in figure 2, addresses the issue of dealing with potentially very dynamic dependency relationships among a very large number of components. It follows a 'divide and conquer' approach, which is usually the way of choice for dealing with scalability problems in distributed systems.

We assume that the managed resources (depicted in the right part of the figure) are able to provide XML descriptions of their system inventory and their various dependencies. The details on how this information can be acquired and what the descriptions look like are described in sections 2.2.2 and 3.3.2, resp. 3.3.3.

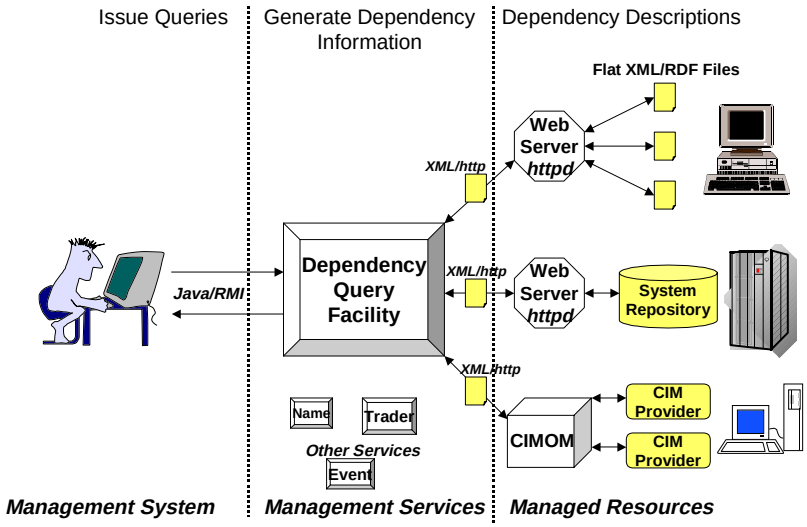

Figure 2: Architecture of our Dependency System

In the center of the figure is the core component of our architecture: The **Dependency Query Facility**, triggered by queries of the management system using Java *Remote Method Invocation (RMI)*, processes them and sends the results back to the manager. Its main tasks are as follows:

- Interacting with the management system. The management system issues queries to the API of the Dependency Query Facility. The API exposes a flexible 'drill–down' method that, upon receiving the identifier of a service, returns:
  - either descriptions of its *direct antecedents*, i.e., the first level below the node representing the service, or
  - the *whole subgraph* below the node representing the service,
  - an *arbitrary subset* of the dependency graph (levels *m* to *n* below a given node).

  A 'drill–up' method with the same facilities, targeting the dependents of the service, is also present. These two methods are equivalent to the aforementioned scoping capabilities. In addition, methods for gathering and filtering information for classes and properties of managed objects are present.

- Obtaining the dependency information from the managed resources (by issuing queries over http) and applying filtering rules (as specified by the manager) to it. If a managed resource does not support http, a web server at another managed system may act as a proxy for this resource.

- Combining the information into a data structure that is sent back to the manager as XML document according to the format specified in 3.3.2 and 3.3.3.

Details of our implementation are given in section 4. It should be noted that due to its fully distributed nature, the architecture aims at keeping the load on every involved system as low as possible. It completely decouples the management system from the managed resources and encapsulates the time consuming *filter* and *join* operations in the dependency query facility, which can be replicated on various systems. Every replica will then act independently and be registered with the management system as responsible for a specific part of the overall distributed system. We are therefore able to achieve a maximum level of parallelism for query operations, since the selection of an instance of the dependency query facility can be done flexibly by the management system.

Another important advantage of our architecture is that the (very large and highly dynamic) overall dependency model is not stored at a specific place but computed on demand in a stepwise manner. The

different parts of the model are stored at the managed resources. The management system therefore always receives the most recent information and is free to store it according to elaborate caching policies.

2.2 State of the Art in Dependency Management

Although the area of dependency management has recently received a lot of attention in the context of new complex outsourced applications, it should be noted there has been a considerable amount of work in the area of tracking and gathering dependency information in the past. We will first describe prior art in modeling and representing dependency information in section 2.2.1, and then provide an overview of the most common approaches to collect dependency information from managed resources in section 2.2.2.

2.2.1 Creating Dependency Models

Early work on identifying dependencies between services and components of distributed systems has been carried out especially within the scope of event correlation (see e.g., [11] and [17]) because an event correlator needs to have an understanding of the structure of a distributed system to aggregate and consolidate multiple events stemming from various resources into meaningful problem determination information. However, since the event correlator has been regarded as the sole consumer of dependency data, the description of service dependencies was often in a proprietary format and thus could not be exchanged between different entities of the fault management process. As motivated in section 1, it is unlikely that different parties involved in the fault management process of outsourced applications use the same toolset for tracking dependencies; therefore it is of fundamental importance to specify dependency information in an open format so that this data can be shared among multiple parties.

In the service management area, [25] presents an approach based on service templates for referencing the subcomponents of which a service is composed. Information provided by the service template is then used as a basis for dynamic service provisioning. In the area of systems management, [26] describes a practical solution to the problem of upgrading dynamic libraries in large-scale shared program repositories without breaking the applications and network services that rely on them. The approach is based on global analysis of library dependencies and is used to perform impact analysis, i.e., to determine in advance, which programs are likely to be affected by a library upgrade when they attempt to load a specific dynamic library.

Work on identifying and tracking dependencies has also been undertaken in component-based software engineering, where one of the challenging problems is to track components and their interrelationships throughout the various stages of the software lifecycle. Existing work in this area makes clear that detailed dependency information, in particular at the runtime stage, is only available in a platform-dependent way, i.e., by taking advantage of the specific features the underlying middleware or operating system have to offer. [21] describes a tool for browsing the dependencies between applications and their underlying libraries on Windows-based systems, while [12] describes how Jini service proxies may be used to track the dependencies between services of a Jini federation at runtime. However, it should be noted that the potential to generalize these concepts in a way that they can be applied to arbitrary distributed environments is limited.

2.2.2 Acquiring Dependency Information from Managed Resources

The question where the dependency information on the managed resources comes from is another crucial issue, although this is not fully within the scope of this paper. For the sake of completeness, we will briefly mention some of the more common approaches:

- The most straightforward way is to provide appropriate instrumentation within the applications and services themselves; the problem is that none of today's applications is able to provide this kind of information at an acceptable granularity.
- Another approach consists in instrumenting the communication protocol stack and/or some shared libraries of the host system to intercept the communication between different parties in order to infer potential dependencies. The resulting information could be either provided by a specific 'dependency

agent' or given out as flat files. [28] describes a CORBA-based approach for message reflection based on portable interceptors to gather information regarding the messages exchanged between the objects of a distributed application.

- [15] describes an approach that makes use of information stored in system configuration repositories for generating appropriate service dependency information.
- A commonly used technique in system and protocol design, which has only recently been applied to service and application management [2] is the active perturbation of components within a system (i.e., injecting faults in a controlled manner and observing the behavior of the components) while running synthetic transactions against it. This technique can be used to obtain the required dependency information; however, great care has to be taken if it is used on production systems.
- Other approaches come from the area of Artificial Intelligence. [7] uses Neural Networks to interpret low-level information like the activity of pairs of services over time. This allows to derive dependencies even in heterogeneous environments.
- The OSI *General Relationship Model (GRM)* [14] defines a powerful generic model for defining relationships between managed objects and provides a mechanism for qualifying these relationships by means of attributes. In addition, the GRM specifies extensible operations that can be invoked on the managed relationships. While its functionality is needed in any distributed system, the GRM is tightly coupled with the OSI Structure of Management Information and CMISE and, thus, has not been used outside of TMN environments. The work described in [6] is a precursor to the GRM and addresses similar issues; the architecture and implementation of a GRM based management system is the subject of [23].
- Finally, a *CIM Object Manager (CIMOM)*, as proposed by the Distributed Management Task Force (DMTF) could be used to expose the necessary information. The CIM Core Model [3] provides an association class *CIM_Dependency*, from which many subclasses are derived.

Every one of the aforementioned approaches for generating dependency models has its specific advantages and drawbacks. Given the fact that dependencies cross system and organizational boundaries, it is likely that a combination of some of these approaches is needed to yield the most comprehensive amount of dependency information.

3 Applying XML Technologies to Dependencies

A key issue in successfully providing information about services and their dependencies to management applications is the introduction of a common description format to represent the dependencies in a uniform way. This does not aim at defining a new information model for service management, but rather at finding a representation format optimized for our purposes and, additionally, embracing available management information exposed by, e.g., CIM (Common Information Model) [3] object managers and repositories. Furthermore, the representation must be easily understood by management applications and able to abstract from the heterogeneity of the described systems, viz. the various ways to obtain their dependency information (as described in 2.2.2). In addition, it should be possible to extend the dependency information determined at the resources without requiring changes to other parts of the infrastructure; e.g., it should be possible to add information that maps business processes onto system resources by means of a different tool than the one used to create the descriptions of system resource dependencies.

In order to meet these goals, our approach is based on several key features of XML, viz. the Resource Description Framework (RDF). These will be briefly explained in the following subsections; we also analyse how they help to fulfill the requirements.

3.1 XML Parsers

Our main motivation for using XML is the fact that it provides flexible and extensible mechanisms to define a notation for the description of dependency information; in addition, it is easy to generate and can be parsed with powerful parsers that are freely available.

There are basically two techniques for parsing XML documents. The first one is used by **DOM parsers** (DOM: Document Object Model); they generate an object model (Java objects instantiated from pre–defined DOM classes) with hierarchically linked objects reflecting the exact structure of the document. These structures can then be traversed to select the required information. The second (more lightweight) method are **SAX parsers** (SAX: Simple API for XML); they sequentially read the document, calling certain user defined functions whenever a new start–tag or end–tag is encountered. DOM parsers are more powerful, but their drawback is that they consume more resources than SAX parsers, especially for large documents. However, the main disadvantage of DOM parsers (at least in their original form that is still used in many applications), namely the need to keep the whole document in memory, has been significantly reduced by parsers of the latest generation, which provide lazy node evaluation and other optimization features. Common XPath implementations (see below) are usually based on such DOM parsers.

3.2 XPath: Querying XML Documents

The aforementioned parsers provide the basic means to access information in the document. However, for many purposes a more powerful way to retrieve information is needed. XPath [31] provides an extensive query language to extract parts of an XML document. Each query describes a 'path' through the virtual tree structure of the XML document that is generated by a DOM parser. Each step on the path consists of:

- an axis—the 'search direction', e.g., towards the `child` or `ancestor` nodes,
- a node test—the name of the nodes (i.e., the tag–name) to be chosen, and
- one or more predicates that apply filters to the result. The predicate itself may consist of further XPath–expressions.

The simple XPath query `/descendant::ds:Service[@rdf:about=`*`ID`*`]` selects a certain element description from an XML document: The axis `descendant` directs the search to anyplace in the document below the current node (in this case the root node). After '`::`' follows the name of the desired node (`ds:Service`) and a filter predicate (in square brackets), which specifies the selection of nodes with an attribute `rdf:about` that has a certain value (*`ID`*).

The use of current XPath implementations also brings the use of the more resource–consuming DOM parsers with it. In our work, this problem is addressed by introducing the following convention which helps to keep the size of the files containing the dependency descriptions within acceptable limits: Every managed object is described in a separate file on the web–server. In order to be able to locate the file, its filename is derived from the name of the managed object.

3.3 Dependency Representation based on the Resource Description Framework

Basically, there are three aspects of dependency modeling:

- knowledge about managed objects,
- the overall dependency structure between them and
- information about each of the dependencies.

Current management information models are designed with a focus on pure managed object representation. Thus, they easily cover the first of the three points above. Modern approaches like CIM (but also already the OSI GRM) provide additional capabilities for representing information about individual dependencies (third point of the above list), e.g., by means of association classes.

However, our investigation of CIM and its XML mapping [4, 5] has led us to the conclusion that navigating dependency graphs in such a model is not supported adequately, although this is an important requirement of management applications based on dependency models, like event correlation mentioned earlier.

The main reason is that the representation of dependencies as individually instantiated associations does not provide an overall graph-like structure, but only references to related objects on a per-association basis. Searching for a managed object's dependents or antecedents can only be carried out by enumerating all the references in every association. In addition, information on these associations may even be stored at places remote from the CIM objects for which the association is relevant. In terms of the three aspects above, this means that the second requirement can not be met in CIM environments. The report [9] draws a similar conclusion for GRM based modeling.

The following subsections describe how our work applies the concepts and means of the Resource Description Framework to address all three aspects of dependency modeling.

3.3.1 RDF Principles

RDF is actually not part of XML, but comes from an independent working group (also within the W3C) and specifies a common representation format for resource description in the form of directed graphs. However, RDF documents are valid XML documents.

The purpose of RDF is to provide a means for defining additional semantics for XML tags in a formal way. Originally focused on document enrichment, it now allows the description of any resource by defining **RDF properties** and provides an extensible type system. Note that, according to RDF terminology, anything that has (or can be represented by) an *Universal Resource Identifier (URI)* is regarded as a resource. It is then called an **RDF resource**, which can be described by one or more **RDF descriptions**, each listing properties (attributes) of the resource. The value of each RDF property can either be a `Literal` (a string) or a pointer to another resource. One or more descriptions form an RDF graph. The described resources plus the `Literals` are the nodes of the graph. Edges are formed by the RDF properties. The type of resource an RDF property can be applied to is called its 'domain'; the types it may point to are called its 'range'.

An interesting question deals with the relationship of RDF and the emerging XML Schema standard [29, 30], as XML Schema could also be used to describe the XML syntax of RDF. However, the semantic mapping of dependency graphs onto RDF can be done more easily than using XML Schema, which is used to define a syntax for the tree-like structures of XML nodes. The reason for RDF's suitability lies in its inherent graph structure and its orientation towards the meta-data layer, rather than the data layer, like XML. The consequences are, e.g., that the order of entries in a document becomes significant in XML, while this is irrelevant from the knowledge description perspective taken by RDF.

Thus, the main purpose of using RDF in our project stems from the fact that RDF provides a very convenient and efficient way for representing directed graphs as an XML document. The fact that RDF provides a mechanism for allowing one node to reference other nodes (that can be either part of the same or a different XML document, perhaps located on another host) circumvents a typical problem of simpler XML mappings, where nodes with multiple antecedents would be described at multiple places, thus leading to redundancy.

3.3.2 RDF based Managed Object Representation

Every described resource (here, managed object) can be embedded into a type system, thus, enabling the RDF parser to check whether the attributes, methods, etc. are used correctly. This allows a clean object description, without the need to use tags on a meta level (e.g., `<ds:Service>` instead of `<ms:Class classname="ds:Service">`; see [5] for detailed discussions). Furthermore—and this makes it superior to purely XML based solutions—it does not lead to the otherwise extremely complex mechanisms for manually checking the syntactical correctness of inherited elements because this is provided by the RDF parsers in a ready–to–use way (but, in contrast, not definable in XML DTDs or XML Schema).

The code extract in figure 3 defines the RDF class `GenericNode` that will be used as the superclass of any node in any dependency graph. Derived from this is the subclass `Service`, which is the type of any service description. The last element demonstrates the definition of attributes as RDF properties.

In RDF terminology, such meta information is called an **RDF schema**. It is referenced by all RDF documents actually describing the services via XML namespaces. For our purposes, an appropriate schema is stored at web servers reachable by all involved systems. Its URLs are contained in the namespace definition

```
<rdfs:Class rdf:ID="GenericNode" >
</rdfs:Class>
<rdfs:Property rdf:ID="NodeDescription">
   <rdfs:range rdf:resource="rdfs:Literal"/>
   <rdfs:domain rdf:resource="#GenericNode"/>
</rdfs:Property>
<rdfs:Class rdf:ID="Service">
   <rdfs:subClassOf rdf:resource="#GenericNode"/>
</rdfs:Class>
<rdfs:Property rdf:ID="ServiceIdentifier">
   <rdfs:range rdf:resource="rdfs:Literal"/>
   <rdfs:domain rdf:resource="#Service"/>
</rdfs:Property>
```

Figure 3: RDF schema extract for nodes of the dependency graph

of each RDF document. As these do not change frequently, simple caching mechanisms can reduce the traffic to a minimum.

The naming problem is solved by introducing a new namespace for each class. This automatically binds each RDF element of the class (attributes, methods, etc.) to the same namespaces, which reflects common principles of object oriented languages.

While this shows that RDF is suitable for describing managed objects, one should also recognize that it explicitly allows a hybrid approach of RDF and pure XML in the same document. An RDF parser would only look at those parts of the document that are embraced by the `RDF`–tag, while the other parts are read by an ordinary XML parser.

3.3.3 RDF based Dependency Representation

In finding an appropriate representation of dependencies, the emphasis has to lie on choosing a structure that allows ease of navigation through the overall dependency graph. While RDF graphs are (by construction) well suited for this purpose, one still has to avoid the problem that a straightforward mapping of the service dependency graph onto an RDF graph would introduce: In a direct mapping, managed objects would be reflected by RDF resources, dependencies by RDF properties. However, RDF does not allow the addition of attributes to properties. This would therefore preclude the presence of attributes for instantiated dependencies. Note that although RDF allows the definition of properties for properties, this does not solve the problem. These would be analogous to CIM qualifiers for associations, but not to the required association attributes.

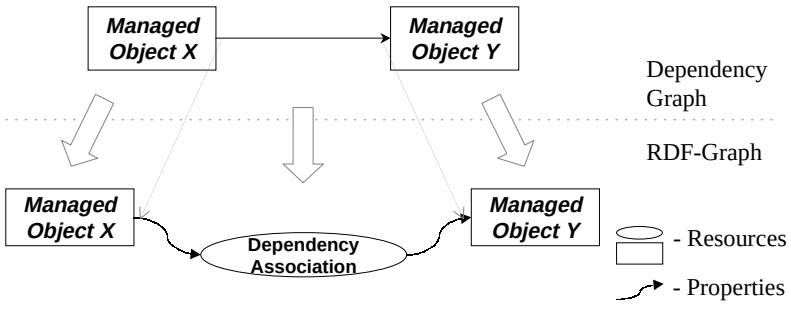

Figure 4: Mapping a dependency to RDF

The solution to the problem that RDF properties may not have further attributes is as follows: Dependencies are mapped to a second type of RDF resources, rather than to RDF properties. As shown in figure

4, the properties are only used to bind the matching managed object resources with the associations, thus spanning a bipartite graph. This maintains the advantage of simple dependency graph traversal and permits not only every object to have a well–defined set of attributes (like caption, identifier etc.), but also the annotation of dependencies (e.g., with the attributes explained in item 5 of section 2). This meets the requirement that a dependency needs to be annotated with attributes providing information about the dependency itself. It is therefore possible to target the dependency attributes for queries by asking, e.g., for all services in the distributed system on which other services depend with a 'high' dependency strength.

```
<rdfs:Class rdf:ID="DependencyAssociation" >
</rdfs:Class>
<rdfs:Property rdf:ID="dependency">
   <rdfs:range rdf:resource="#DependencyAssociation"/>
   <rdfs:domain rdf:resource="#GenericNode"/>
</rdfs:Property>
<rdfs:Property rdf:ID="antecedent">
   <rdfs:range rdf:resource="#GenericNode"/>
   <rdfs:domain rdf:resource="#DependencyAssociation"/>
</rdfs:Property>
<rdfs:Property rdf:ID="DependencyStrength">
   <rdfs:range rdf:resource="rdfs:Literal"/>
   <rdfs:domain rdf:resource="#DependencyAssociation"/>
</rdfs:Property>
```

Figure 5: RDF schema extract for the edges in the graph

The code fragment in figure 5 shows the basic RDF schema for generic dependencies, which we called `DependencyAssociation` (to stay close to CIM terminology), together with the properties needed for the binding to and from the managed object description, as explained above. The lower part of the code further shows an example of an association attribute.

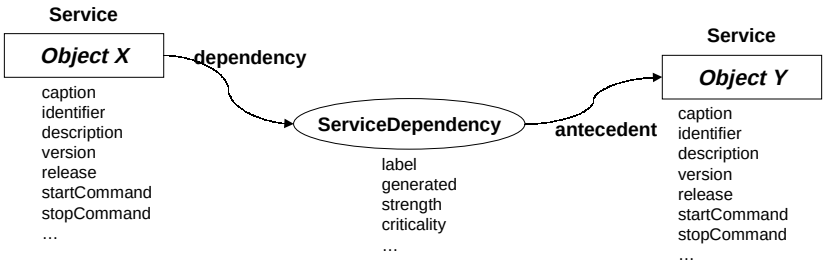

Figure 6: Elements of a dependency description in RDF

Figure 6 gives a graphical representation of the RDF schema we use for representing dependencies. It also shows further attributes we defined for objects and dependencies.

3.4 Discussion

It is fair to say that RDF is ideally suited for the representation of information about managed objects *and* their dependencies. For the developer of a management tool, RDF allows a significantly simpler way to perform document validation, while keeping all the benefits of a hierarchical type system, like in object oriented languages.

An additional aspect worth mentioning is the ability to easily retrieve required information from RDF documents. While XPath is the means of choice for the purely XML based approach, no special query mechanism (beyond parsing) exists that is fully 'aware' of RDF concepts. The obstacle that RDF puts up against

a straightforward use of XPath—although its representation finally is nothing but an XML document—is that it allows various (full and abbreviated) syntaxes for the same RDF concepts. Our solution consists in restricting the use of RDF to only one syntax (the abbreviated). This brings no disadvantages when the documents are processed by RDF parsers, but allows the use of XPath in a way that is as simple as it would be for pure XML documents.

Extending CIM to address dependency management is possible from a conceptual point of view by subclassing from *CIM_Dependency* and adding appropriate properties, such as the ones depicted in figure 6 (strength, criticality, etc.). However, existing CIM implementations take a "CIMOM-centric" view where a CIM Object Manager (CIMOM), playing essentially the same role as a web server in our approach, resides on every managed system and surfaces the dependencies within its scope, i.e., the scope of a *single* CIMOM. Associations (and therefore dependencies) spanning several CIMOMs are not supported yet, thus making a CIM based implementation of a "dependency-centric" management system difficult.

In addition, not every dependency relationship is suitable for being described by properties: A typical example is the use of (a subclass of) *CIM_Dependency* for modeling start/stop/failover dependency relationships. This would allow the specifiation of relationships indicating, e.g., "Application X must be running/terminated before Application Y can be started/stopped" or "Application X acts as a failover (i.e., backup) for Application Y". However, the issue one faces when trying to add properties to a subclass of *CIM_Dependency* is that start/stop/failover relationships are in fact different dependency *types*. Typing in CIM, however, is done by subclassing and not by means of enumerated properties. Adopting the latter would be contradictory to the CIM modeling philosophy. For a detailed discussion of this subject, the reader is referred to [19].

3.5 Example: RDF Representation of Services and Dependencies

We will now present by means of an example how the approach described above can be applied to our e–business scenario of section 1. Figure 8 depicts an example of an RDF/XML document that specifically represents the dependency of `Storefront Servlets` on `IBM WebSphere 3.5` on the one side, and on `DB2 UDB 5.2` on the other. These dependencies are marked as dashed arrows in figure 7.

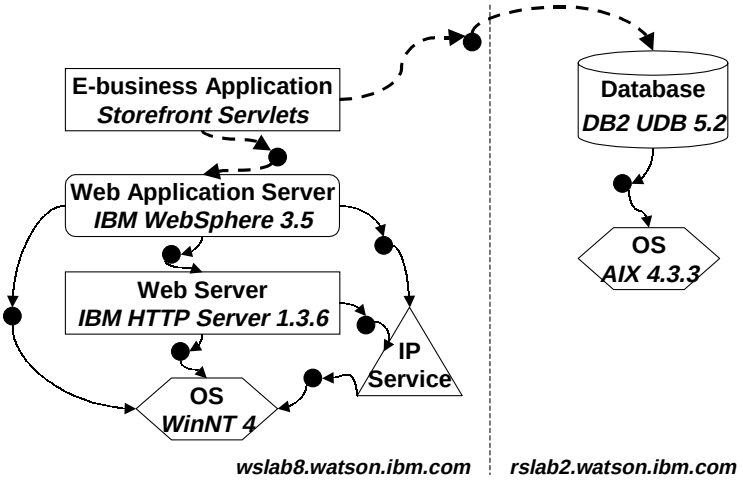

Figure 7: Sample RDF dependency graph

By definition, the header of every document starts with the XML tag (line 1 of the document in figure 8), followed by links to our dependency schema (line 2) as well as the RDF syntax and schema definitions (lines 3 and 4). The body of the document contains the service definition start and end tags (line 5, resp. 29), its attributes (lines 6 to 12) and two dependencies (lines 13 to 20, resp. 21 to 28). The document closes with the RDF end tag (line 30). Note that all pointers to descriptions of antecedents are URIs, thus making

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rdf:RDF xmlns:ds="http://wslab4.watson.ibm.com/DependencySchema#"
3    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
5   <ds:Service>
6    <ds:name>E-business Application</ds:name>
7    <ds:caption>Storefront Servlets</ds:caption>
8    <ds:identifier>my.catalogServlets</ds:identifier>
9    <ds:description>business logic of catalog app.</ds:description>
10   <ds:version>3</ds:version>
11   <ds:release>1</ds:release>
12   <ds:processName></ds:processName>
13   <ds:dependency>
14     <ds:ServiceDependency>
15       <ds:antecedent rdf:resource=
16         "http://rslab2.watson.ibm.com/xmlrepository/db2.xml"/>
17       <ds:generated>automatic</ds:generated>
18       <ds:label>ebusinessApp_DependsOn_database</ds:label>
19     </ds:ServiceDependency>
20   </ds:dependency>
21   <ds:dependency>
22     <ds:ServiceDependency>
23       <ds:antecedent rdf:resource=
24         "http://wslab8.watson.ibm.com/xmlrepository/websph35.xml"/>
25       <ds:generated>automatic</ds:generated>
26       <ds:label>ebusinessApp_DependsOn_webApplServer</ds:label>
27     </ds:ServiceDependency>
28   </ds:dependency>
29  </ds:Service>
30 </rdf:RDF>
```

Figure 8: Sample document with RDF/XML dependency description

their location (local or remote) completely transparent to the dependency query facility. The string `ds:` in the expression is the namespace–prefix we use for the dependency schema.

4 Proof-of-Concept Implementation

4.1 Components of the Prototype

The prototype implementation of our service dependency architecture (as explained in section 2.1) is composed of two main parts. One part is responsible for the description of the managed resources' dependencies in RDF/XML documents that are stored at management web servers close to those resources. This part of the prototype—together with the web servers—form the third tier of the aforementioned architecture. As acquisition of dependencies is not the goal of this paper and an overview of the matter has been presented in section 2.2.2, the following explanations will focus on the second part of the prototype, which implements the architecture's middle tier.

The implementation has to deal with two main interfaces. Towards the lower layer it has to handle the access to the web servers. This access is realized in a rather simple way by the `ResourceProxies` (see figure 9) via `http`. As all dependencies are described according to the same RDF/XML schema (introduced in the previous section), the whole complexity of the dependency acquisition becomes completely transparent and need not be considered in the upper layers. In order to keep the number of http queries low if a service description is requested many times, the proxies also implement caching functionality. Their additional capability of being able to evaluate XPath expressions on the retrieved documents will be explained further down.

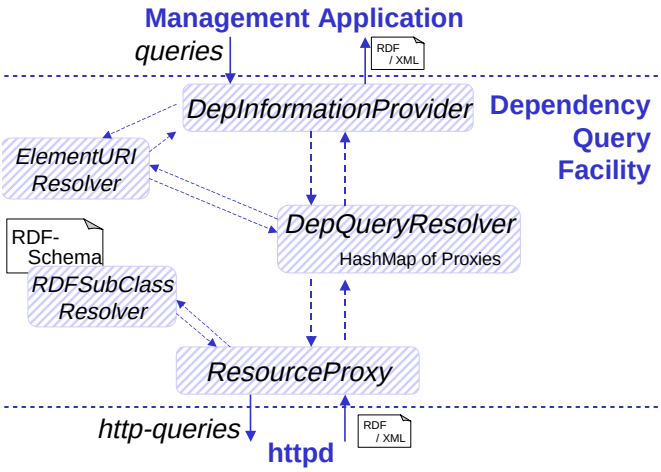

Figure 9: Components and information flows

The main component of the architecture is the so called `DepInformationProvider`. It realizes the interface towards the management application and responds to their queries for element descriptions and performs drill–ups or drill–downs through the dependency graph. For each query it constructs a result document (with the help of further classes, as will be explained in section 4.2 along with examples of XPath–queries). Depending on the query type, it extracts the required parts of dependency information from the web servers with the help of the `DepQueryResolver` that manages the above mentioned `ResourceProxies`. Once the right element descriptions are found, they are merged into the result document by appending them one by one under a new RDF/XML document header.

4.2 Implementation of XPath Queries

The key part of the `DepInformationProvider`'s implementation is the ablity to extract the 'right' information from the obtained managed object descriptions.

As mentioned in section 3, one of the motivations for the use of XML is the power of its XPath language. It will be demonstrated in the following sections by means of two sample queries: The first retrieves all the immediate antecedents of a given service (used, e.g., in root cause analysis) while the second query asks for all immediate dependents of a node (useful for impact analysis).

4.2.1 Drill–down for immediate Antecedents

The most basic operation is graph traversal, one step at a time, along the edges of a dependency graph. In the case of service dependencies this yields all (sub-)services the dependent service is based on. The result is constructed in two phases:

1. getting the dependency information of Service X, and
2. obtaining the description of all antecedents (i.e., the nodes the dependencies point to).

The first query has to be applied to the dependency description of Service X that comes from the web server on X's system. The actual evaluation of the XPath expression may be carried out in two places: If the web server is capable of resolving XPath expressions as part of the URL (and its host has enough free resources) this can be used to relieve the Dependency Query Facility. In other cases, the `ResourceProxy` will retrieve and parse the whole document and apply the XPath query to it.

To be able to find the web server of Service X, the name of the system hosting the service has to be part of the query to the dependency query facility. In our prototype, the hostname of the service is by definition

part of the (hierarchically structured) service name. The full URL is resolved by a class in the prototype called `ElementURIResolver`, which uses an element-to-URI mapping, obtained from a configuration file, or, alternatively, simply applies a default path. The appropriate XPath query is as follows:

```
/descendant::[(self::ds:NodeType)]/child::ds:dependency/*[(
self::ds:DependencyType)]/child::ds:antecedent/@rdf:resource
```

The result is a list of resource identifiers, namely the IDs of the antecedents taken from the `rdf:resource` attribute. There is no need for the ID of Service X to appear in the expression, because each RDF/XML file on the web servers only contains the description of one service. If this requirement is not met, e.g., if a different convention for the dependency description is used, an additional XPath predicate that searches for the service ID needs to be specified.

The example also shows a problem of using XPath, which is not aware of certain RDF features: The above query assumes that both the exact type of the resource (the node) as well as the type of the dependency (the association) have to be known before the query is executed. It would not return the required antecedent if, e.g., the type of the association was replaced by a supertype (i.e., `DependencyAssociation` instead of `ServiceDependency`). We solve this issue by making a little extension to the XPath expression in the `ResourceProxy` part of the prototype that finally applies the XPath to the document. This enables the statement of the types (the classes of the dependency and service objects) to be as generic as possible and as specific as needed. In other words, the XPath expression must be able to match a given class to any of its subclasses. This is implemented by replacing class names in the XPath queries with an or'ed list of all known subclasses, thus enhancing the XPath expression to match any of them. The list of subclasses is obtained from the `RDFSubClassResolver`, which reads the class hierarchy from the same RDF schema that all dependency documents refer to in their namespace statement. Thus, the correctness of the expanded XPath–query is always guaranteed. As mentioned before, the string `ds:` in the expression is the namespace–prefix we use for our "dependency schema".

In the second phase, the descriptions of all IDs from the first step (of Service Y,Z,...) are obtained from their web server by a simple XPath expression, which is almost identical to the example at the end of section 3.2. The only difference lies in the specification of the node type, which must be expanded in the same way as above.

4.2.2 Drill–up for immediate Dependents

The main difference of graph traversal in the opposite (upward) direction is that the IDs of the dependents are not kept in a single description document. This lies in the nature of dependencies, since only the dependents know on which antecedents they depend, while the antecedents are unaware of their dependents. Consequently, the required IDs of the dependents are distributed over a possibly large domain.

To avoid the need to query all possible web servers, a search domain has to be specified for the drill–up. Then—in analogy to phase one of the example above—the following XPath expression is applied to each document obtained from the web servers in a selected search domain:

```
/descendant::*[(self::ds:NodeType)][descendant::ds:antecedent[
rdf:resource=ID]]/@rdf:about
```

In natural language, the query could be expressed as: "Get all elements of type *NodeType* that have a dependency on the element with the specified *ID*"! The expression is much more complex than the previous example, because it contains a predicate (within the second pair of square brackets), which again is an XPath expression. This sub–expression makes sure that the outer expression only matches those elements that have a sub–description called `ds:antecedent` and an attribute `rdf:resource` with the value of the required ID—or, in other words, "that depend on *ID*".

Each query to the various web pages results in either zero or exactly one ID. In the second step, the descriptions of the retrieved elements are merged into one document and returned to the management application.

This example further shows that although XPath expressions always process 'downwards' in the XML document tree, there is no need to insert 'upwards–' pointers in the documents (e.g., for drill–ups) since they can easily be circumvented as demonstrated above.

5 Conclusions and Outlook

We have presented a novel approach for managing service dependencies with XML, XPath and RDF. The need for applying these general–purpose technologies to the area of service and application management stems from the fact that, despite related work in the area of event correlation, no previous work has dealt with describing dependency information in a uniform way so that it does not only meet all the requirements stated in this paper, but enables management systems in general to make use of it. This is necessary in contemporary e–business environments where the outsourcing of services results in a vast amount of dependencies among services that are also highly dynamic.

We have combined several XML related base technologies and are therefore able to represent dependency graphs in a way such that they can not only be parsed by common off the shelf XML parsers, but be also queried with the powerful XPath facility. This allows us to implement an efficient mechanism for querying a potentially very high number of managed objects in parallel for their attributes and dependencies. Our prototype implementation has shown that queries for (recursive) drill–up or drill–down operations are surprisingly compact and relatively easy to write. The problems we experienced during our work are mainly related to XML and, especially, RDF parsers, which were partly still in early stages of development.

In our current work, we are investigating the integration of our approach with a CIM Object Manager that generates the dependency instances and qualifies them with attributes. In the area of multi-role relationships, we are studying whether it is more efficient to define a single dependency relationship whose attributes indicate its various roles vs. creating separate instances for every type of relationship.

A candidate for future work is the investigation of recent developments on native RDF query languages such as RQL, presented in [16]. However, a precondition is that those new languages become popular in the way XML did. If this is the case, developers of management tools operating on RDF/XML dependency graphs will benefit from the same support of the open source community they already enjoy now for RDF, XML and XPath.

Acknowledgments

The authors wish to thank the members of the Systems Management Department at the IBM T.J. Watson Research Center for helpful discussions and valuable comments on previous versions of the paper. In particular, the authors would like to express their gratitude to Dr. Gautam Kar (IBM) and Prof. Dr. Heinz-Gerd Hegering (Ludwig-Maximilians-Universität München) for their continuous support and guidance. The authors are also grateful to the anonymous referees for their constructive suggestions to improve the quality of the paper.

References

[1] N. Anerousis, G. Pavlou, and A. Liotta, editors. *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM 2001)*, Seattle, WA, USA, May 2001. IEEE Publishing.

[2] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Application Environment. In Anerousis et al. [1], pages 377–390.

[3] Common Information Model (CIM) Version 2.2. Specification, Distributed Management Task Force, June 1999. http://www.dmtf.org/standards/cim_spec_v22/.

[4] Specification for the Representation of CIM in XML Version 2.0. Specification, Distributed Management Task Force, July 1999. http://www.dmtf.org/download/spec/xmls/CIM_XML_Mapping20.php.

[5] XML as a Representation for Management Information - A White Paper Version 1.0. Technical report, Distributed Management Task Force, September 1998. http://www.dmtf.org/standards/xmlw.php.

[6] A. Clemm. *Modellierung und Handhabung von Beziehungen zwischen Managementobjekten im OSI-Netzmanagement*. PhD thesis, Ludwig-Maximilians-Universität München, June 1994.

[7] C. Ensel. A Scalable Approach to Automated Service Dependency Modeling in Heterogeneous Environments. In *5th International Enterprise Distributed Object Computing Conference (EDOC '01)*, Seattle, WA, USA, September 2001. IEEE.

[8] O. Festor and A. Pras, editors. *Proceedings of the 12th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'2001)*, Nancy, France, October 2001. INRIA Press.

[9] A. Franzke. Recommendations for an Improvement of GDMO. Final Report of the GDMO Project, University of Koblenz-Landau, March 1997. http://www.uni-koblenz.de/∼ist/retrieve/RR-20-97.pdf.

[10] R. Gopal. Layered Model for Supporting Fault Isolation and Recovery. In Hong and Weihmayer [13], pages 729–742.

[11] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. In *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM '98)*, October 1998.

[12] P. Hasselmeyer. Managing Dynamic Service Dependencies. In Festor and Pras [8], pages 141–150.

[13] J.W. Hong and R. Weihmayer, editors. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Honolulu, HI, USA, April 2000. IEEE Press.

[14] Information Technology – Open Systems Interconnection – Structure of Management Information – Part 7: General Relationship Model. IS 10165-7, International Organization for Standardization and International Electrotechnical Committee, 1997.

[15] G. Kar, A. Keller, and S.B. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. In Hong and Weihmayer [13], pages 61–75.

[16] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF Descriptions for Community Web Portals. In *17ièmes Journées Bases de Données Avancées (BDA'01)*, pages 133–144, Agadir, Morocco, November 2001.

[17] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. In Lazar et al. [22], pages 583–596.

[18] A. Keller, G. Kar, H. Ludwig, A. Dan, and J.L. Hellerstein. Managing Dynamic Services: A Contract based Approach to a Conceptual Architecture. In R. Stadler and M. Ulema, editors, *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS'2002)*, Florence, Italy, April 2002. IEEE Press.

[19] A. Keller, H. Kreger, and K. Schopmeyer. Towards a CIM Schema for RunTime Application Management. In Festor and Pras [8], pages 217–230.

[20] H. Kreger. *Web Services Conceptual Architecture 1.0*. IBM Software Group, May 2001.

[21] M. Larsson and I. Crnkovic. Configuration Management for Component-based Systems. In *Proceedings of the 23th International Conference on Software Engineering (ICSE)*, Toronto, Canada, May 2001.

[22] Aurel A. Lazar, Roberto Saracco, and Rolf Stadler, editors. *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management*, San Diego, CA, USA, May 1997. Chapman and Hall.

[23] E. Nataf, O. Festor, and L. Andrey. RelMan: A GRM-based Relationship Manager. In Lazar et al. [22], pages 661–672.

[24] Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, W3 Consortium, March 2000.

[25] G. Dreo Rodosek and L. Lewis. Dynamic Service Provisioning: A User–Centric Approach. In Festor and Pras [8], pages 37–48.

[26] Y. Sun and A.L. Couch. Global Impact Analysis of Dynamic Library Dependencies. In M. Burgess, editor, *Proceedings of the 15th Systems Administration Conference (LISA '01)*, San Diego, CA, USA, December 2001. USENIX/SAGE.

[27] UDDI Version 2.0 API Specification. Universal Description, Discovery and Integration, uddi.org, June 2001.

[28] M. Wegdam, D.J. Plas, A. van Halteren, and B. Nieuwenhuis. Using Message Reflection in a Management Architecture for CORBA. In A. Ambler, S.B. Calo, and G. Kar, editors, *Proceedings of 11th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM '2000)*, Austin, TX, USA, December 2000.

[29] XML Schema Part 1: Structures. W3C Recommendation, W3 Consortium, May 2001.

[30] XML Schema Part 2: Datatypes. W3C Recommendation, W3 Consortium, May 2001.

[31] XML Path Language (XPath) Version 1.0. W3C Recommendation, W3 Consortium, November 1999.

Biography

Christian Ensel studied informatics at the Technische Universität München, Germany and received his M.Sc. in 1998. He is currently a research member of the MNM Team and a Ph.D. candidate at the Ludwig-Maximilians-Universität München where he is also working as a teaching assistant. His research interests center around integrated network, systems and service management, with a focus on information modeling. He is a member of IEEE.

Alexander Keller is a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, NY, USA. He received his M.Sc. and a Ph.D. in Computer Science from Technische Universität München, Germany, in 1994 and 1998, respectively and has published more than 30 papers in the area of distributed systems management. He does research on service and application management, information modeling for e-business systems, and service level agreements. He is a member of GI, IEEE and the DMTF CIM Applications Working Group.