# Managing Application Service Dependencies with XML and the Resource Description Framework

*Christian Ensel*
*Munich Network Management Team*
*University of Munich*
*Oettingenstr. 67*
*80538 Munich, Germany*
*ensel@informatik.uni-muenchen.de*

*Alexander Keller*
*IBM Research Division*
*T.J. Watson Research Center*
*P.O. Box 704*
*Yorktown Heights, NY 10598, USA*
*alexk@us.ibm.com*

**Abstract**

This paper describes 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 web–based business architectures allow the composition of web–based e–business applications at runtime: The concept of *Web Services* [10] 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 in a testbed for designing, implementing and testing our approach. It represents a fictitious 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 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).

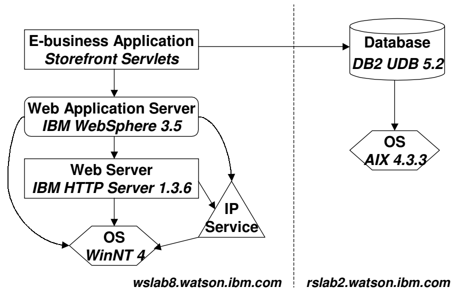


**Figure 1:** Simplified application dependency graph of an e–business system

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 [3]. 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.

While previous work (often within the scope of event correlation, see e.g. [4] and [8]) has focused on identifying and describing service dependencies in a proprietary format, it remains unclear how dependency information can actually be exchanged between different entities of the fault management process. Since it is unlikely that the different parties involved in the fault management process of outsourced applications use the same toolset for tracking dependencies, it is of fundamental importance to define an open format for specifying and exchanging dependency information. This is the topic addressed by this paper. The proposed solution is based on XML, *XML Path Language (XPath)* [12] and the *Resource Description Framework (RDF)* [9], 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 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 the platform database therefore cannot be applied to dependencies due to the sheer number and the dynamics of the involved dependencies. 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 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 out of 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.

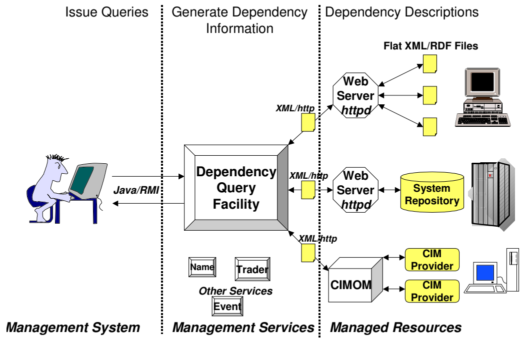


**Figure 2:** Architecture of our Dependency System

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 and 3.3.2, resp. 3.3.3.

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.
- 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.

The 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. 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 but is still free to store it according to elaborate caching policies.

## 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.
- [7] describes an approach that makes use of information stored in system configuration repositories for generating appropriate service dependency information.
- A technique used in system and protocol design which could be applied to service and application management is the active perturbation of components within a system (i.e., injecting faults in a controller manner and observing the behavior of the components) while running synthetic transactions against it. This technique could be used to obtain the required dependency information; however, great care has to be taken if this technique is used on production systems.

- Other approaches come from the area of Artificial Intelligence. [2] uses Neural Networks to automatically derive dependency information by looking at pairs of systems' behavior over time.

- The OSI *General Relationship Model (GRM)* [6] 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.

- 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 [1] provides an association class *CIM_Dependency*, from which eight subclasses are derived (in the Core Model alone).

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 to successfully provide information about dependencies to management applications is the introduction of a common description format to represent the dependencies in a uniform way. This is especially necessary to hide the heterogeneity of the described systems, resp. the various ways to obtain their dependency information (as described in 2.2). Furthermore, the representation must be easily understood by management applications. 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., 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, resp. the Resource Description Framework (RDF). These will be briefly explained in the following subsections; we also analyse how they aid 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 if a document is large because they have to keep the whole document in memory. Common XPath implementations (see below) are usually based on 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 [12] 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` specifies to search anywhere in the document below the current node (in this case the root node). After '`::`' follows the name of the desired node (`ds:Service`) and the filter predicate (in square brackets), which specifies to select only 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 description 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 Resource Description Framework

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.

### 3.3.1 RDF Principles

The goal of RDF is to provide a means for defining additional semantics for XML tags in a formal way, mainly focusing on document enrichment. RDF at its current stage allows the description of any resource by defining **RDF properties** and making use of the extensible type system. Note that in the terminology of RDF, anything is regarded as a resource that has (or can be represented by) an *Universal Resource Identifier (URI)*. It is then called an **RDF resource** and 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 on is called its ‘domain’. The type it may point to is called its ‘range’.

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, eventually located on another host) circumvents a typical problem of simpler XML mappings, where nodes with multiple antecedents would be described at multiple places.

### 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 [11] for detailed discussions). Furthermore—and this makes it superior to purely XML based solutions—it does not lead to the otherwise extremely complex mechanisms needed to check inherited elements because this is provided by the RDF parsers in a ready–to–use way.

The following code extract 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 for any service description. The last element demonstrates the definition of attributes as RDF properties.

```
<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>
```

In RDF terminology, such meta information is called an **RDF schema**. It will be 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 each RDF document’s namespace definition. 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 an 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

A straightforward approach to describe service dependencies with RDF is to directly map the service dependency graph onto an RDF graph. However, this precludes the definition of attributes for instantiated dependencies, because RDF properties may not have further attributes.

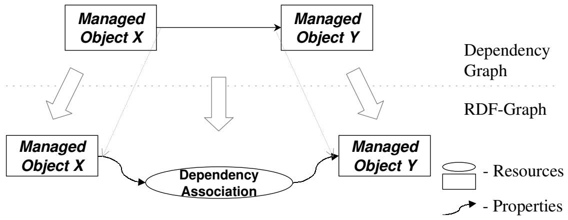


**Figure 3:** Mapping a dependency to RDF

The solution to this problem is to map dependencies to a second type of RDF resource, rather than to an RDF property. As shown in figure 3, the properties are used to bind the matching managed object resources with the associations, thus spanning a bipartite graph. The advantage of simple dependency graph traversal is not restricted by this approach. It permits not only every object to have a well–defined set of attributes (caption, identifier etc.), but also the annotation of dependencies (e.g., strength, criticality, generatedby etc.). This fulfills the fifth requirement of section 2, which states that a dependency needs to be annotated with attributes that provide information about the dependency itself. It is therefore possible to target the dependency attributes for queries by asking, e.g., for all the services in the distributed system on which other services depend with a ‘high’ dependency strength.

The code extract below shows the basic RDF schema for the generic dependencies, which we called `DependencyAssociation`, 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.

```
<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 4 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.

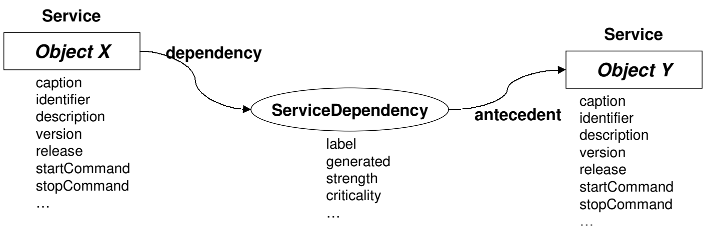


**Figure 4:** Elements of a dependency description in RDF

There remain only very few issues that cannot be checked by an RDF parser. E.g., if further constraints are imposed on ranges of attributes (RDF properties), this cannot yet be specified in an RDF schema (but neither in an XML–DTD).

An additional aspect that has to be mentioned is the ability to easily query 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.

## 3.5 Example: RDF Representation of Services and Dependencies

We will now present by means of an example how the approach described in section 3 can be applied to our e–business scenario of section 1. More precisely, we show the content of the 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 5.

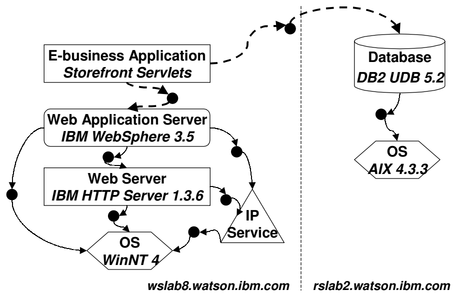


**Figure 5:** Visualized RDF graph

By definition, the header of every document starts with the XML tag (line 1 of the following listing), 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 their location (local or remote) completely transparent to the dependency query facility.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:ds="http://wslab4.watson.ibm.com/DependencySchema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
 <ds:Service>
  <ds:name>E-business Application</ds:name>
  <ds:caption>Storefront Servlets</ds:caption>
  <ds:identifier>my.catalogServlets</ds:identifier>
  <ds:description>business logic of catalog app.</ds:description>
  <ds:version>3</ds:version>
  <ds:release>1</ds:release>
  <ds:processName></ds:processName>
  <ds:dependency>
   <ds:ServiceDependency>
    <ds:antecedent rdf:resource=
     "http://rslab2.watson.ibm.com/xmlrepository/db2.xml"/>
    <ds:generated>automatic</ds:generated>
    <ds:label>ebusinessApp_DependsOn_database</ds:label>
   </ds:ServiceDependency>
  </ds:dependency>
  <ds:dependency>
   <ds:ServiceDependency>
    <ds:antecedent rdf:resource=
     "http://wslab8.watson.ibm.com/xmlrepository/websph35.xml"/>
    <ds:generated>automatic</ds:generated>
    <ds:label>ebusinessApp_DependsOn_webApplServer</ds:label>
   </ds:ServiceDependency>
  </ds:dependency>
 </ds:Service>
</rdf:RDF>
```

# 4 Proof-of-Concept Implementation

## 4.1 Components of the Prototype

Our prototype consists of two parts. The first part is responsible for the generation of the managed resources' RDF/XML descriptions that are obtained from the web servers (lowest section of figure 6). The central element of the architecture, the `DepInformationProvider` responds to the queries from the management application for dependency or element descriptions. It constructs the result document (with the help of further classes) by collecting the appropriate document parts from the web servers, resp. from the `ResourceProxies`. The latter implements a cache to store already retrieved documents in order to keep the number of http queries low if a service description is requested many times. Once the right element descriptions are found, it is easy to combine them into a complete document by appending them under one RDF/XML document header.

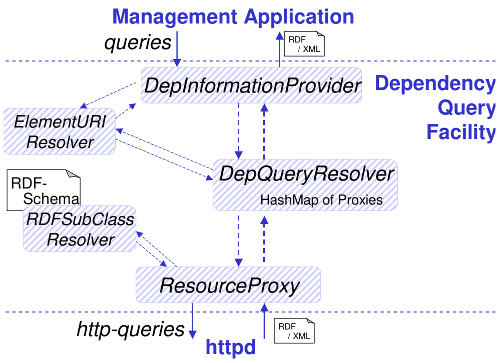


**Figure 6:** Components and information flows

## 4.2 Implementation of XPath Queries

The more interesting and complex part is the extraction of 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–query language. This will be demonstrated in the following by means of two queries: The first fetches all the immediate antecedents of a given service (used, e.g., in root cause analysis) while the second query gets all the immediate dependents (useful for impact analysis).

### 4.2.1 Drill–down for immediate Antecedents

The most basic operation is the 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. get the dependency information of Service X
2. get the description of all antecedents.

The first query is applied to the description of a service that comes from the web server on the same machine. 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 fetch and parse the whole document and apply the XPath query to it.

The name of the machine hosting the service has to be part of the query to the dependency query facility. In our prototype, the hostname of the service is part of the (hierarchically structured) service name. The full URL is resolved by a class in the prototype called `ElementURIResolver`, which reads the mapping either from a configuration file or uses a default path. The exact XPath query is:

```
/descendent::[(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. The ID of service X must not appear in the expression as the file only contains descriptions of the queried service. If this requirement is not met, an additional XPath predicate has to be specified.

The example also shows the 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. Otherwise, it would not return a required antecedent where, e.g., the type of the association is replaced by its supertype (i.e., `DependencyAssociation` instead of `ServiceDependency`). We solve this issue by making a little extension to the XPath expression in the part of the prototype that finally applies the XPath on the document `ResourceProxy`. It allows to state the type as generic as possible, by accepting any superclass (for both cases). These are replaced by an or'ed list of all known subclasses, thus enhancing the XPath expression to match all of them. The list of subclasses is obtained from the `RDFSubClassResolver`, which reads the class hierarchy from the RDF schema. 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 treated 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 is the nature of dependencies, because only the dependents know on which antecedents they depend, but the antecedents cannot know all their possible dependents. Thus, 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—analogous to phase one above—the following XPath expression is applied to the documents obtained from the restricted number of web servers:

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

It consists of only two steps. The first selects the node of the service description, while the second extracts its ID. However, the expression for the first step is much more complex than in the previous examples: It contains a predicate which again is an XPath expression. This one states that there has to be a descendent XML node in the description that is called `ds:antecedent` and has an attribute `rdf:resource` with the value of the ID for which the drill–up has to be performed.

Each query to the various web pages either results in zero or one ID. The second phase is equivalent to the one in the first example and collects the descriptions of the IDs.

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 application 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 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 are 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.

# References


[1] Common Information Model (CIM) Version 2.2. Specification, Distributed Management Task Force, June 1999.

[2] C. Ensel. Automated generation of Dependency Models for Service Management. In *Workshop of the OpenView University Association (OVUA'99)*, Bologna, Italy, June 1999.

[3] R. Gopal. Layered Model for Supporting Fault Isolation and Recovery. In J.W. Hong and R. Weihmayer, editors, *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS'2000)*, pages 729–742. IEEE Press, April 2000.

[4] 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.

[5] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems — Concepts, Architectures and their Operational Application*. Morgan Kaufmann, 1999.

[6] 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.

[7] G. Kar, A. Keller, and S.B. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. In J.W. Hong and R. Weihmayer, editors, *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS'2000)*, pages 61–75. IEEE Press, April 2000.

[8] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM 97)*, pages 583–596, May 1997.

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

[10] Universal Description, Discovery and Integration. Programmer's API Specification, Ariba, Inc., IBM Corp., Microsoft Corp., September 2000.

[11] 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/spec/xmlw.html.

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