XML-based Monitoring of Services and Dependencies

Christian Ensel
Munich Network Management Team, University of Munich
Oettingenstr. 67, 80538 Munich, Germany
ensel@nm.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*— We present a novel approach for describing, querying and monitoring the dependencies among services in a distributed system. Until now, dependency information has to be acquired either through custom-built scripts or proprietary tools; the absence of a common description format makes the exchange of service dependency models across multiple platforms and their monitoring particularly difficult. We show how this problem can be addressed by a web based architecture for retrieving and handling dependency information from various heterogeneous managed resources. Specifying dependencies in an XML based notation facilitates the sharing of information among the heterogeneous systems involved in the monitoring process.**

***Keywords*— Web–based Service Monitoring, Dependency Analysis, RDF, XML, XPath**

I. Introduction

The monitoring of services and their dependencies on other components of distributed systems is becoming increasingly important for integrated service management because applications and services rely on a variety of supporting services that are often outsourced to a service provider. 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 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 service 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 involving a Web Server for serving static content, a Web Application Server for hosting the business logic and a back–end database system that stores the dynamic content of the application (such as product descriptions, user and manufacturer data, shopping carts).

Both service providers and customers require management tools that allow to navigate 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 windows.

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 traverse 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.

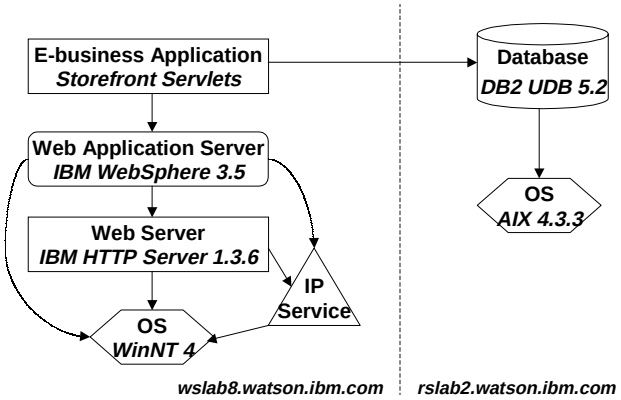

Fig. 1. Simplified service dependency graph of an e–business system

While previous work (often within the scope of event correlation, see e.g. [4] and [6]) has focused on identifying service dependencies and describing them in a proprietary format, it remains unclear how dependency information can actually be exchanged between the different entities involved in the management process. Since it is unlikely that different parties involved in the monitoring of outsourced services 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)* [9] and the *Resource Description Framework (RDF)* [7]. It provides a uniform interface to monitor and query service dependency information across the systems of a distributed environment and can be used by fault and topology management applications or event correlation systems.

The paper is structured as follows: Section 2 states the requirements for determining service dependencies 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 system. The proof–of–concept prototype implementation is described in section 4. Section 5 concludes the paper and presents issues for further research.

II. Service Dependencies

Conceptually, 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 underlying 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.

A. Requirements for Service Dependency Models

1. The number of dependencies between many involved systems is computable, 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. Thus, we propose to distribute the storage and computation of dependencies across the systems involved. Section II-B describes our architecture that is designed to meet these requirements.
2. Further, facilities for combining local dependency graphs, stored on every system, into a uniform dependency model are required. The facilities need to provide an API so that management applications can query 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 problem determination and to check whether these services are operational. Section III describes our approach to cope with this problem.
3. Dependency models are usually directed graphs. This raises the question which data format is able to represent dependency graphs efficiently so that fine–grained queries can be applied to them. Section III-C describes our solution to this issue.
4. 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 business objectives), 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 further qualification. This is addressed by section III-C.

B. 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 large number of components. 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 of these descriptions are presented in section III.

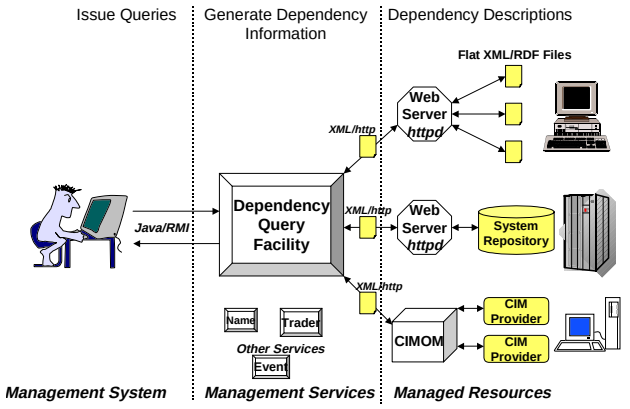

Fig. 2. Architecture of our Dependency System

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 service's node,
  - 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. 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 III-B and III-C.

Details of our implementation are given in section IV. 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. Thus, we are 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 central place but computed on demand from the different parts located at the resources. The management system therefore always receives the most recent information but is still free to store it according to elaborate caching policies.

III. Mapping Resource Data to XML / RDF

A key issue to successfully provide information about services and their dependencies to management applications is the introduction of a common description format. This does not aim at the definition of an new information model for service management, but at an underlying representation optimized for our purposes and additionally to embrace existing management information, e.g., from CIM (Common Information Model [1]) object managers and repositories. Furthermore, the representation must be easily understood by management applications and be able to hide the heterogeneity of the described systems, resp. the various ways to obtain their dependency information.

In order to meet these goals, our approach is based on XML. It provides the basis for defining extensible structures for data representation and comes with publicly available tool implementations for many platforms. Besides XML parsers, which are needed to read XML documents for further processing, we make use of XPath [9], an extensive query language to extract parts of an XML documents' information. Each query describes a 'path' through the virtual tree structure of the XML document. The ease of use of the existing XPath tools is one of the reasons that makes our approach powerful and easy to apply at the same time. An example on how XPath is used in our project is described in section IV-B.

A second basis of our solution is the Resource Description Framework (RDF) from the W3C. The main reason for using RDF stems from the fact that it provides a very convenient and efficient way for representing directed graphs in an XML document. The following sections will explain the core features of RDF and show its advantages over an "XML only" approach—in particular, for object oriented information representation.

A. Resource Description Framework

The goal of RDF is to provide a formal means of defining semantics of XML tags. Originally, it focused on document enrichment, but now allows the description of any resource by defining **RDF properties** and provides an extensible type system. According to the terminology of RDF, anything that has (or can be represented by) a *Universal Resource Identifier (URI)* is a potential **RDF resource** and can be described by one or more **RDF descriptions** that list its properties (attributes). The value of each RDF property can either be a `Literal` (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 types it may point to its 'range'.

B. Mapping Service Descriptions

Every described resource 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 [8] for detailed discussions). Furthermore—and this makes it superior to purely XML based solutions—it does not lead to the otherwise very complex mechanisms for manually checking the syntactical correctness of inherited elements because this is already provided by RDF parsers (but, in contrast, not definable in XML DTDs).

The following code fragment defines the RDF class `GenericNode` that will be used as the superclass for nodes in our 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. It is a common attribute for all sub–typed services definitions.

```
<rdfs:Class rdf:ID="GenericNode"/>
<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>
```

Such meta information is called an **RDF schema**. It is referenced from all RDF documents describing service dependencies by denoting its URL in the XML namespaces definition. For our purposes, an appropriate schema is stored at web servers reachable by all involved systems. As the schemas 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, automatically binding each of its RDF elements (attributes, methods, etc.) to the same namespace. This 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.

C. Mapping Dependencies

Dependency representation covers two aspects: The dependency structure (whether or not dependencies exist between nodes) and information about the dependencies' properties.

In existing approaches, e.g., in CIM models (and therefore also in its XML-mapping [2]) the latter is addressed using association classes, which may—just like any CIM class—define their own attributes to reflect any kind of property. However, this leads to disadvantages in regard to the first aspect, as especially the navigation through the dependency graphs (as stated in section II-A) becomes too complex. This is due to the fact that instantiated associations may be stored at different places than the CIM objects for which the association is relevant. A second reason why we chose not to use the CIM XML-mapping is that CIM objects referenced from within an association are tagged with a CIM object identifier, which cannot be used as a simple 'pointer' to the XML object description.

Both aspects are handled better by RDF. However, one has to avoid the following problem that a straightforward approach (to directly map the service dependency graph onto an RDF graph)

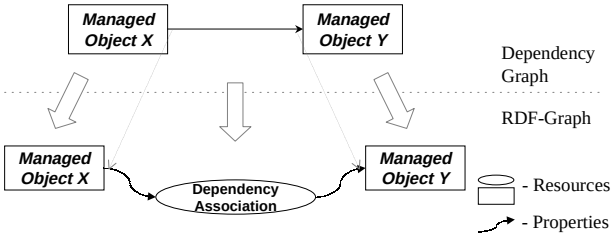

Fig. 3. Mapping a dependency to RDF

would lead to: In a direct mapping, dependencies would be reflected by RDF properties. However, RDF does not allow to add 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 the analogy to CIM association qualifiers, but not to the required association attributes.

The solution is to map dependencies to a second type of RDF resource, as shown in figure 3. The RDF properties are only used to tie the matching managed object resources to 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, but also allows the annotation of dependencies (e.g., strength, criticality, etc.). This meets the requirement of section II-A, stating that a dependency needs to be annotated with attributes that provide information about the dependency itself. It is therefore possible to use values of attributes in queries, e.g., by asking for all the services with a 'high' dependency strength.

The code fragment below shows the basic RDF schema for the 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.

```
<rdfs:Class rdf:ID="DependencyAssociation" />
<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>
```

D. Example: Representation of a Service with Dependencies

We will now present by means of an example how the approach described in sections III-B and III-C can be applied to our e–business scenario of section I. More precisely, we show a fragment of the documents' content specifically representing the dependency of `Storefront Servlets` on `DB2`.

By definition, the header of every document starts with the XML tag, followed by links into the dependency schema and RDF syntax resp. schema definitions (lines 1 and 2). The body of the document contains the service definition start and end tags (line 3, resp. 20), its attributes (lines 4 to 11) and one dependency (lines 12 to 19). Note that all pointers to descriptions of antecedents are URIs, thus making their location (local or remote) transparent to the dependency query facility. The string `ds:` in the expression is the namespace–prefix we use for the dependency schema.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <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#">
3  <ds:Service>
4   <ds:name>E-business Application</ds:name>
5   <ds:caption>Storefront Servlets</ds:caption>
6   <ds:identifier>my.catalogServlets</ds:identifier>
7   <ds:description>business logic of catalog app.
8   </ds:description>
9   <ds:version>3</ds:version>
10  <ds:release>1</ds:release>
11  <ds:processName></ds:processName>
12  <ds:dependency>
13   <ds:ServiceDependency>
14    <ds:antecedent rdf:resource="http://rslab2.
       watson.ibm.com/xmlrepository/db2.xml"/>
15    <ds:generated>automatic</ds:generated>
16    <ds:label>ebusinessApp_DependsOn_database
17     </ds:label>
18   </ds:ServiceDependency>
19  </ds:dependency>
20 </ds:Service>
```

It is fair to say that RDF is ideally suited for representing information about managed objects *and* their dependencies. For a management tool developer, 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. There remain only very few issues that cannot be checked by an RDF parser: E.g., one can not specify in an RDF schema (but neither in an XML–DTD) if further constraints are imposed on ranges of attributes (RDF properties).

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 a 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 is nothing but an XML document—is that it allows various (full and abbreviated) syntaxes for the same RDF concepts. Our solution is to restrict the use of RDF to a single (abbreviated) syntax only. This brings no disadvantages when the documents are processed by RDF parsers, but allows the straightforward use of XPath.

IV. Proof-of-Concept Implementation

A. Components of the Prototype

The main part of the prototype implements the middle tier of the architecture described in II-B. `DepInformationProvider` provides the interface to the manager tier by answering queries for dependency and service descriptions. It constructs the result document by collecting and combining the appropriate document parts via `ResourceProxies`, which access the RDF/XML descriptions at the managed resources' web servers (lower part of figure 4) and implement caches to enable a high overall performance. Special attribute tags help to distinguish 'static' attributes from those with a high change frequency. The prox-

ies are also able to resolve XPath expressions, in cases where queried web servers are not capable of doing so by themselves. 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.

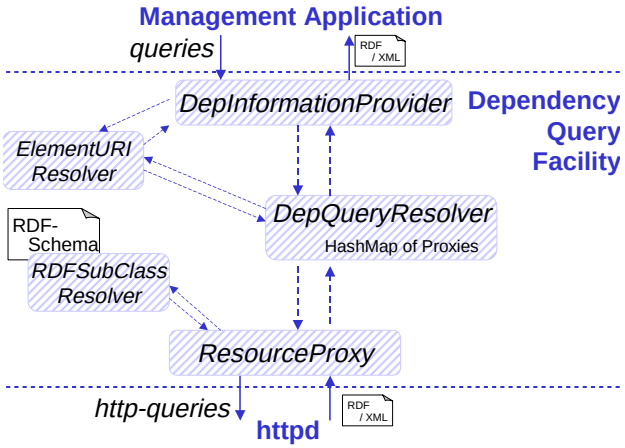

Fig. 4. Components and information flows

B. Implementation of XPath Queries

A key part of the implementation is the extraction of information from various RDF documents. This data extraction is based on the aforementioned XPath query language. It will be demonstrated by means of a drill-down query issued, e.g., by root cause analysis tools (for details on XPath, the reader is referred to [9]).

The procedure consists of two phases:

1. getting the dependency information of Service X
2. getting the descriptions of all antecedents.

The first phase reads the description of Service X from its web server (which is determined by the `ElementURIResolver` from X's ID) and applies the following XPath expression to extract the IDs of its antecedents:

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

The example also shows that XPath 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 the type of the association or service is replaced by its supertype (e.g., `DependencyAssociation` instead of `ServiceDependency`). We solve this issue by allowing slightly extended XPath expressions in the upper parts of the prototype architecture, that allow to specify any supertype. The expressions get translated into a standard XPath in the `ResourceProxy`, which (with help from the `RDFSubClassResolver`) replaces each supertype by an or'ed list of all known subclasses, thus enhancing the XPath expression to match any of them.

In the second phase, the descriptions of all antecedent services are obtained from their web servers by the simple XPath expression `/descendant::ds:Service`. It allows to store additional non–RDF descriptions in the same XML document, without risking any interference.

Usually, XPath expressions do not become much more complex than the one above. In drill–up operations (used by impact analysis tools to recursively navigate the dependency hierarchy towards the root node), e.g., one has to search for all nodes with a certain antecedent. This is mapped to an XPath expression matching all nodes that fulfill the predicate "has the antecedent *ID*", which is expressible by a simple (nested) XPath.

V. 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 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.

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.

References

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

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

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

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

[8] 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.

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