Behavioural Specification Using XML

P Mckee and I Marshall
BT Advanced Communications Technology Centre
Adastral Park, Martlesham Heath, Ipswich, IP5 3RE
(paul.mckee@bt.com, ian.w.marshall@bt.com)

Abstract

Active networks are an example of a wide area distributed system. The objects in this system require platform independent machine-readable, flexible behaviour specifications. Current mechanisms in CORBA, DCOM and similar technologies are not adequate. We demonstrate for the first time how specifications with the required properties can be written in XML, the new language of the WWW.

1.Introduction

Network users always want more services, better services or different services. As a result, network operators typically have to support a large array of services and features. The administrative overhead for the management of the services is a significant component of the cost. This management cost and complexity is increasing rapidly as the range of available services increases, and following current trends will ultimately dominate service provisioning overheads. Future management systems must therefore be simplified as much as possible, whilst avoiding the risk of rendering them somewhat inflexible, making introduction of new services time consuming and expensive. Active Networks [1] is a world wide programme of research that aims to address this problem. The idea is to enable customers to add new services by sending programmes to networked devices along with the information they are sending across the network. In other words the network is treated as a distributed programmable system. This system must have the property of dynamically loading and running programmes (objects) safely, without operator intervention. This means that objects must have a machine readable behaviour specification, which is guaranteed to be correctly interpreted at any active node in the network, regardless of ownership or software build.

In existing distributed systems , such as those based on CORBA or DCOM the capabilities of programmes are described in IDL. The IDL describes an interface in syntactic terms, naming operations it supports, their input and output parameters and any exceptions that may occur. However, the IDL does not provide a behavioural specification, i.e. it does not describe the semantics of the interface (there may be some comments that do but these are not machine readable), nor does it specify the non-functional properties of the component (e.g. management, dependencies, performance, security). The IDL signature of an operation carries no information about the ranges of parameters, how the operation affects system state and the effect of this change on any returned values. It is entirely possible for a client to communicate with a software component through an interface described in IDL and yet fail to interoperate successfully with it. Some of these issues are addressed by CORBA facilities such as the meta-object facility, but not all. In any case this approach requires the target node to be running the version of all the necessary object facilities that the code author has assumed, a condition that cannot be guaranteed in an active network.

A number of research projects in the area of high performance wide area computing are described in a recent article[2] . Systems such as Legion, Condor and Globus all contain mechanisms for describing resources, however these mechanisms are not portable between systems, and the competing systems use different resource description languages. Behavioural specification is also discussed in [3], and a new Java framework is introduced, this frame work extends Java to include some of the behavioural specification constructs found in Eiffel such as invariants, preconditions and postconditions. But this framework is built to leverage the power of Java and in particular the use of reflection and this limits this approach to interworking between systems that are running Java programs.

2. XML as a solution

We propose the use of XML [4] to describe system resources and application resource requirements in large scale distributed systems, such as active networks. XML offers a number of significant advantages:

- it is a standard designed for a totally heterogeneous platform.
- it is independent of operating system or programming language,
- it has a fixed syntax but an unlimited vocabulary, this extensibility (via the definition of new tags) enables the straightforward addition of new capabilities, for example two components may specify communication in a previously unknown protocol.

- The structure of the XML specification can be described using a schema , either a document type definition (DTD) or as standards mature one of the more rigorous proposals such as XML-Schema. This mechanism allows code authors to instruct target nodes how to interpret any proprietary constructs in their specification, and avoids the assumptions implicit in more conventional object services.

In our proposed architecture every object is contained in an XML structure and the associated behavioural specification forms part of the XML metadata. This specification will thus be strongly associated with the instantiation and will additionally be automatically stored in any local repositories which cache the object.

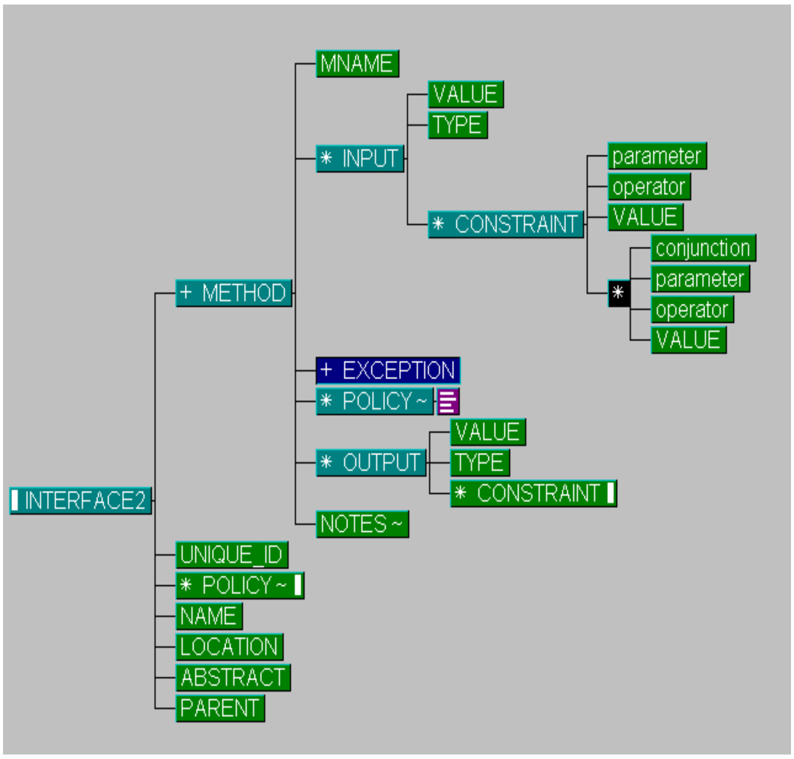
In our initial implementation the XML resource descriptions have a logical structure composed of a root, and nested elements defined by start and end tags. The tags define the syntax of the description, and the user is free to supply the content that describes the resource. The structure of the description is defined in a DTD or XML-Schema. In order to ease development the larger DTD's contain smaller modular DTD's. This structuring makes it easier to mix and match DTD's to generate larger more elaborate descriptions, as well as aiding the understanding of the more complex structures. Using the DTD and a validating editor such as IBM's XEENA the resource specification can be created in a reproducible way. Similarly given the structure defined by the DTD information can be easily extracted with a parser such as IBM's XML4j; parsing having a low overhead. Transferring the information in a standard format allows users and managers of resources to define their own processes for handling requests and their own policies on which resources they are willing to share, allowing them to retain as much autonomy as desired.

3.Requirements for Metadata

Our metadata is designed to allow the user or service provider to build collaborative applications with predictable performance characteristics. It should be possible to decide at an early stage in the collaboration whether the application will complete successfully or should be promptly terminated. The metadata may contain details of the expected performance of the collaborating components and must include a description of the management policies that may affect the collaboration. We can therefore describe a number of requirements that our proposal should fulfil. If in our model system component A wishes to interact with component B the first requirement of our metadata is a description of the interface syntax of B in a form that A can use to assess compatibility. The next question posed is where component B will run. Our metadata must describe the resource requirements of B, and any system management policies that affect its execution. Given adequate resources an indication of the performance of B will be desirable. The metadata may describe the expected performance. Obviously B may require the co-operation of other components to achieve the desired result, and these software dependencies must be available in the metadata. Finally in order to complete our metadata description the semantics of the interaction between the two components must be defined, this is achieved in our proposed system by the use of explicit tag definitions and inheritance relationships between tags. Having an initial set of requirements for our metadata the next sections will discuss the specification of behaviour in our framework, how the framework may be extended and a description of an early trial implementation.

4. Behavioural Specification

The behavioural specification of a software component may be described as a formal description of what is supposed to happen when the software executes. There are a number of tools that can be used to verify, statically or at run time that the software meets these specifications. Formal methods using languages such as Z and VDM may be used to create such behavioural specifications, our approach using XML is not intended to compete with these established formal methods, but rather to provide a uniform way of communicating this information between potential users of the system. In our proposed structure all objects such as software components, hardware resources and users have associated resource and behavioural specifications. The flexibility and relative informality of XML allows us to specify the behaviour of any system component at a number of levels. If we start at the software interface level, existing IDL's describe only the names and type signatures of the attributes and operations together with any exceptions that may be raised. It is easy to extend this description using XML to include information about constraints on the ranges of the inputs, any pre and post conditions that apply and any invariants . Each of these may have a separate exception or number of exceptions associated with them. A graphical representation of this DTD fragment is shown in the following diagram, the notations "*" and "+" indicate the number of allowed entries of the annotated tags, zero or more and one or more respectively.



Our overall XML description of a software component is a much larger modular document including additional information about the supplier of a component, the components usage and the runtime environment required by the component. The usage section includes a performance section, it is proposed that this might include information about the scalability of the implementation and details of response times and other quality of service metrics that allow any user to make an informed choice.

The third element of behavioural specification that our system embodies is that of policies. Every entity in the system has an associated set of policies that govern what it is able to do and how it responds to external events. Our initial work uses the policy structure proposed by Sloman [6] which elaborates two classes of policy, authorisation policies that define what the object is allowed to do and obligation policies that define what an object must do. Policies define a relationship between subjects and targets and include attributes specifying the action to be performed and any constraints limiting the applicability of the policy. Authorisation policies define what a subject is allowed to do in terms of operations it is allowed to perform on a target object , authorisation policies may be positive in that they permit an action to take place or negative in that they prohibit an action. The simplest policies of this type can be expressed in terms of subject, target and activity:

User is permitted to read file (positive)
User is prohibited from reading file (negative)

It is also possible to include a predicate based on object state to the authorisation policy:

User is permitted to read files where creation date is before 1998

Obligation policies define what a subject must or must not do, based on the assumption that an object will attempt to discharge it's obligations if at all possible. This may be a safe assumption in the case of automated systems but may be less useful if human intervention occurs. Again the simplest policies may be expressed in terms of subject, target and activity, but may also specify an event which triggers the activity:

When error(X) occurs send warning to system engineer (Positive)
User must not modify any policies(Negative)

Obligations may also include a predicate based on the state of the object;
Object must maintain response time below 2 seconds

The constraints that may be optionally defined as part of the policy description include temporal constraints that specify time limits after, before or between which the policy applies or may be used to specify a validity time or expiry time of the policy. Constraints based on parameter values define permitted values for the policy based management operation as in setting a minimum length for a user password. It is now possible to list the components of a policy specification:

1. Subjects - objects which perform activities specified in the policy

2. Targets - objects which are affected by the policy's activities

3. Modality - mode of the policy either authorisation or obligation (positive or negative in both cases)

4. Activities - the actions of the policy

5. Constraints - limits on the action of the policy

It is also useful if the policy has a name, so given these requirements a suitable DTD for policy transmission may be proposed:

```
<!ELEMENT policy  ( source , target , action+ , constraint+ )>
<!ATTLIST policy   name CDATA  #REQUIRED
mode  (PositiveAuth | NegativeAuth | PositiveOb | NegativeOb )  #REQUIRED >
<!ELEMENT source (#PCDATA)>
<!ELEMENT target (#PCDATA)>
<!ELEMENT action (#PCDATA)>
<!ELEMENT constraint (#PCDATA)>
```

This simple DTD requires that the policy has a name and that a mode has to be chosen from the supplied list, the elements source and target occur only once each but there may be multiple actions and constraints. The one drawback of this simple DTD is that it allows no validation of the type of source and target, they are purely text strings expressed as PCDATA . As standards mature these entities may be described as types with a predefined structure, of course any action event will need to be caught and processed by the operating system, or in the case of active networks, the node manager.

5. Extensibility

One of the possible reasons for the slow take up of distributed operating systems such as CORBA is the fact that it may be considered hard to use and that although it is capable of running on many operating systems communicating with non-CORBA systems is not easy. Our metadata proposal start from a lightweight basis, and is extensible to allow the developer to include the appropriate level of complexity. At the most basic level for two objects to communicate the only metadata information required is the language used to describe the interface. This may be one of the many standard interface description languages, or even in the most lightweight context a CGI script. Developers writing code for in house use that may never be accessible to a wider audience need only to supply the minimum of data, the metadata may be as simple as

```
<interface_description> CGI </interface_description>
```

For larger scale applications additional data fields if supplied will aid in the construction and management of large scale applications with known performance characteristics and limitations. The stucture and relationships between elements in the more detailed metadata is described using a DTD or XML-Schema, but this description must be extensible to accommodate changing requirements.

The DTD is a closed model, so the easiest way to add extensibility is the provision of optional tags that any user may choose to use for their own additional data requirements. As an example of this approach in our initial definition of an interface DTD the method tag is qualified by the "+" qualifier, indicating that one or more methods must be defined to complete the specification of an interface. There are problems with this approach in the naming of tags, the author of the original DTD will have no idea of the information requirements of other developers and an arbitrary tag name chosen to provide extensibility may ultimately confuse the intent of the data. A generic tag will give no clue as to the nature of the content.

It is possible to build large DTD's from collections of smaller DTD's, that is one DTD may link to another and pull in the elements and entities declared, in this structure cycles are prohibited, but the nested DTD's may still become large and complex. Structuring the DTD into small chunks however may confer a number of important advantages: the smaller DTD's will be easier to analyse, and the smaller DTD's may be reused in a number of different documents, or DTD's from other application domains may be imported for use. In this way proprietary information may be added to a more general DTD. As an example let us consider the DTD fragment used to describe a policy described earlier. Such a DTD doesn't even permit the creation of a document because it doesn't declare a root element, although that can easily be supplied to create a valid file:

```
<?xml version ="1.0"?>
<!DOCTYPE policy SYSTEM "F:\dtds\policy.dtd">
<policy name = "acess" mode = "PositiveAuth"><!-- (source , target , action+ , constraint+ )-->
        <source>operator</source>
        <target>backup system</target>
        <action>log in</action>
        <constraint>between 5:00pm and 8:00am</constraint>
</policy>
```

Although a trivial example there are a number of common structures that might be reused in a number of documents and could therefore be imported as DTD's. The importation of a DTD would also allow user specific customisation of an existing document. DTD's are connected using external parameter entity references as follows:

```
<!ENTITY % policy SYSTEM " policy.dtd">
%policy
```

Or if the DTD is being loaded from a remote web site:

```
<!ENTITY % policy SYSTEM " http://dtdsareus.co.uk/xml/metadata/policy.dtd">
% policy
```

There are therefore a number of possibilities of extending a DTD , although the text nature of any content so described will limit the applicability of the DTD approach. The more recently proposed XML Schema [5] is much more flexible, the most important innovation for content models in XML Schema is that content models are "open" by default. An open content model enables additional tags to be present within an element without having to declare each and every element in the XML Schema. This provides an extensibility mechanism not present when using a Document Type Definitions (DTD). This extensibility mechanism could be used to add additional elements and attributes to a schema. for example if our metadata contained a requirement for a diskspace with the datatype "int" extended tags from the namespace "myTags" may be used to add additional limitations.

```
<ElementType name="diskspace" xlmns:myTags="urn:performanceschema-extensions">
<datatype dt:type = "int"/>
<myTags:max>500</myTags:max>
<myTags:min>50</myTags:min>
</ElementType
```

Validation will only check that the value of a particular "diskspace " element is an integer, but the additional information is available to the users application for additional validation. It is necessary that the additional information is namespace qualified. In instances where an open content model is not desired, the **model** attribute can be used on the ElementType as in:

```
<ElementType name="diskspace" model="closed"/>
```

effectively preventing any user defined extensions.

5.Storage and Querying of Metadata

Our initial experiments focussed on developing the DTD's and XML-Schema to store resource requirements of the system components. Example description documents have been created and the information has been extracted using a parser. In order to complete the system we require efficient storage and querying of the XML documents to ensure that the metadata is always available, correct and comprehensible. We propose a linked network of high performance stores.. Some experiments have been undertaken to store the XML documents in an LDAP directory but this proved too limiting and recent work uses XSet [7] being developed at Berkley. The primary goals of XSet are to support the storage and query of XML whilst maintaining fast and scalable performance, simplicity and ease of use. An experimental release of XSet was obtained and evaluated for use as a high speed metadata store. Due to the rapid advances in XML technology XSet is not frozen but sufficient functionality was available to demonstrate the desired actions.

6.Experimental work

In order to aid the development of our ideas, a simple policy distribution demonstrator has been implemented, using the policy DTD previously described. When started the policy management program reads the DTD and generates a DTD specific input screen that will only allow generation of a valid XML document. The XML policy document is fed into a lightweight group communications system and distributed to nodes that have registered an interest. Upon arrival at each node the policy file is indexed in XSet and is then available for system use. If necessary the incoming policy may be merged with an existing metadata description, and the new enhanced description re indexed. Whenever an appropriate event is detected in the system the XSet store may be queried for a policy that details the actions to be taken upon receipt of such an event. One immediate observation that arose from the implementation of this small demonstrator was the that of different DTD versions, it is possible to reference two different DTD's in different locations to create XML documents with the same names, future work will address this problem. The demonstrator also served to indicate the need for DTD or Schema extensions, the originator of the policy should be identified either by name or by role, to allow resolution of policy conflicts, and the policies may also require a time to live indicator to facilitate garbage collection within the policy store.

7.Future Work

In addition to resource matching, the implemented description files also include constraint descriptions of both input and output parameters, and a policy structure to allow the transmission of management policies throughout the system. They could also be easily extended to include information on billing, software dependencies, timings, and any other attribute that users may consider important. One area of considerable interest is that of distributed scheduling, where the resource requirements may be associated with quality of service requirements. The problem here is the sharing of dynamic information between resources. We propose to handle this with an event driven model. The behavioural specification will therefore need to include event ordering and the linkage between events and state. Our initial thoughts in this area have concentrated on the use of a specialised language with appropriate temporal primitives, and we are currently evaluating a range of candidates through a model based approach.

7. Summary

In summary, we supply the facility to describe the behavioural properties of an object using a standardised notation, and widely available web-based mechanisms. Users of the object are, of course, free to ignore the metadata, just as web browsers are free to ignore non-standard tags. The additional information will simply improve the performance of those parts of the system which use it, and encourage component reuse. We believe the proposal overcomes many of the barriers to introduction of more complex middleware solutions.

8. References

1. Special Issue: Active and programmable networks IEEE Network 12, 3, May 1998

2. A Grimshaw, A Ferrari, F Knabe and M Humphrey, "Wide-Area Computing: Resource Sharing on a Large Scale" IEEE Computer May 1999 29-37

3. C Della Torre Cicalese and S Rotenstreich , "Behavioural Specification of Distributed Software Component Interfaces" IEEE Computer July 1999 46-53

4. Extensible Markup Language, W3C recommendation Feb 1998 http://www.XML.com/aXML/testaXML.htm

5. XML Schema Part 1: Structures W3C Working Draft 6-May 1999 http://www.w3.org/TR/xmlschema-1/

6. Morris Sloman "Policy driven management for distributed systems " Journal of Network and Systems Management Vol.2 No.4 1994

7. Ben Y. Zhao and Anthony d. Joseph "XSet: A High Performance XML Search Engine", University of California, Berkley [submitted to Usenix ITS]