DICOM Structured Reporting: An object model as an implementation boundary

David A. Clunie
ComView Corporation
mailto:dclunie@dclunie.com

ABSTRACT

DICOM Structured Reporting (SR) provides for encoding and interchanging structured information that may reference images, waveforms or other composite objects, in traditional reporting applications as well as for logs, measurements and CAD results. DICOM SR differs from generic content encoding approaches like XML, in that it supports coded entries, values that are strongly typed, and explicit relationships. DICOM structured reports (like images and waveforms) are composite objects that can be stored, transmitted and queried. The traditional DICOM binary encoding is used to encode structured reports.

The structure and content of the SR tree should be accessible regardless of the internal or external representation. XML parsers and XSL-T tree transformation engines that can be used for data entry, presentation (display and printing) and trans-coding (to HL7 2.x and HL7 Clinical Document Architecture (CDA)) need to be interfaced with DICOM tools that support encoding, transmission, storage and retrieval. Issues associated with establishing the appropriate boundaries between tools are discussed, as are how and when to internalize a DICOM SR in an actual or virtual XML representation, the characteristics of such a representation, and the use of SAX events or the Document Object Model (DOM) to drive style-sheet driven tree transformation engines.

Keywords: DICOM, Structured Reporting, XML, XSL, Document Object Model

1. INTRODUCTION

Structured Reporting (SR) is a recent addition to the DICOM standard.[1] It provides a mechanism for encoding and interchanging structured information that may reference images, waveforms or other composite objects. Though applicable to traditional reporting applications, SR can be used for any type of content. Examples include procedure logs, measurements generated by devices and applications, and computer assisted detection (CAD) results.

Unlike XML, DICOM SR has domain specific mechanisms for describing coded entries. Nodes are name-value pairs and values are strongly typed. Explicit relationships other than containment are defined, such as has properties and inferred from . The content can be organized as a directed acyclic graph rather than as a strict tree, by using by-reference relationships. In common with other DICOM objects such as images and waveforms, DICOM structured reports are composite objects. They can be stored, transmitted and queried just like any other object.

The use of traditional DICOM binary encoding to encode structured reports allows for re-use of the existing installed base of DICOM applications and toolkits. It does create a barrier to entry by implementers with no previous DICOM experience. Furthermore, the skills required to build high quality report generation, rendering and analysis applications differ from those required to build DICOM encoders. In the past, DICOM tool kits have been created to ease the burden on application designers. These have been made available either as free libraries or as commercial products. However, each has a different architecture and API.

DICOM structured reports are layered on top of existing DICOM encoding mechanisms and services. There exists a need for structured reporting toolkits which allow access to the structure and content of the SR tree independent of the internal or external representation. Using the W3C recommendation for an XML Document Object Model (DOM)[2] as a basis, a Structured Reporting Object Model is proposed. The DOM cannot be used directly. It contains XML specific features that are not applicable. It also lacks support for

specific DICOM SR features. However, the basic idea of a language-independent interface definition can be re-used, as can the specific classes used to describe nodes, and the accessor and iterator methods for those classes. For example, a SROM Node class may contain methods such as getNodeName() and getFirstChild() that are identical in function to those of DOM.

By deriving DOM from an existing object model used outside the medical imaging domain, opportunities are created to use designs and tools that have broader application. By reaching consensus on an API to access the content of a DICOM SR, applications become independent of the actual encoding. Opportunities exist to transcode the object model into other representations, such as the proposed HL7 Clinical Document Architecture (CDA), which is in XML. In addition, tree re-writing and transformation pattern languages like eXtensible Style Sheet Transformations (XSLT) can more readily be applied to DICOM structure reports. Though definition of an SR Object Model is probably beyond the scope of the DICOM organization, there is no reason consensus amongst implementers cannot be reached.

2. OBJECT MODEL REQUIREMENTS

From an application s point of view, requirements may be defined for access to an internal model of an SR content tree. For the creator of a tree, facilities must be provided to

- create a root node
- create and add sibling and child nodes
- assign names, values and relationships to nodes

The user of a tree must be able to:

- find the root node
- find the siblings and children of a node
- access names, values and relationships of nodes

The editor of a tree must also be able to:

- detach a node and its descendants (a sub-tree) from a tree
- insert a node as a sibling of a specified node, or between nodes
- insert a node as a child of a specified node

and so on.

The requirement for these facilities is generally applicable to any tree, not just an SR tree. Furthermore, even those facilities that are specific to SR, such as the names and values, are independent of the internal representation. By defining classes that represent the facilities in an object-oriented manner, one can take advantage of

- encapsulation, by hiding the internal representation of the tree (such as whether it is in memory or on disk, and whether it uses pointers to dynamically allocated nodes, pre-allocated tables, a database of nodes or whatever)
- inheritance, by defining base or abstract classes that represent a tree in a generic sense, specializing the classes as necessary (for example a base tree class, an SR tree class, an SR tree class that can be serialized to disk, etc.)
- polymorphism, by defining common operations that apply to any tree node regardless of its actual class (such as getNextSibling(), getFirstChild() and so on)

This approach is obviously more desirable than having every function in an application that needs to create a node allocate memory, fill in pointers, and so on.

For example, were one creating such classes from scratch in C++ one might write:

```
class node {
        class node *getNextSibling(void);
        class node *getFirstChild(void);
};
```

```
class srnode : public node {
        coded_entry getName(void) const;
        coded_entry getValue(void) const;
};
class srnode_mutable : public srnode {
        void setName(coded_entry name);
        void setValue(coded_entry value);
};
```

and so on. Ideally, one would not write this all from scratch, but would reuse some suitable existing library of tree-handling code.

Regardless, the application that is creating, using or manipulating the SR tree need have no knowledge of the internal representation of the tree. The object-oriented approach is more desirable than the data-structural approach, but one can do better still in terms of reuse and portability, by taking a language independent approach.

3. DOCUMENT OBJECT MODEL

As has been mentioned before, a DICOM SR tree is not that different from an XML tree. Both DICOM SR and XML have as their primary structure recursively-nested content. Though the details differ, perhaps XML tools can be reused in the context of DICOM SR. In particular, the Document Object Model (DOM) may be of use.

DOM is a W3C recommendation that is very popular amongst XML developers. The goal of DOM is to provide language and representation independent access to the content of an XML document. This is achieved by:

- defining an object that is a document that may be read from and be written to an external representation
- defining methods that may be used to traverse the document, and examine and insert nodes that are its content
- hiding any details of the internal representation of the document
- defining the objects and methods in a language-independent Interface Definition Language (IDL)
- providing multiple language bindings so that the interface can be implemented in widely-used languages

Though the details of DOM are not directly relevant, a few simple examples of its use will illustrate the power of the concept. An XML document may be read, parsed and validated as follows:

```
Parser parser = new Parser( dummy );
Document document = parser.readStream(xmlInputStream);
if (parser.getNumberOfErrors() > 0) return;
```

Having de-serialized the XML document from the input stream into an opaque internal representation, one can traverse the document tree from the top downwards, extracting and examining nodes, moving on to their siblings and descendants. For example:

```
for (Node node=document.getDocumentElement().getFirstChild();
        node != null;
        node=node.getNextSibling()) { /* ... do stuff ... /* }
```

and so on. The example reflects some details of XML that affect the design of DOM. For instance, an XML document doesn t have a single root node as a DICOM SR document does, hence the iteration on siblings. The model of the XML document distinguishes between elements and nodes, there being many different types of nodes that are not elements, such as attributes, character data, and so on.

How is one to make best use of DOM in the context of SR, given that there are many DOM features specific to the peculiarities of XML, and many SR peculiarities not supported directly by DOM? Possibilities include:

- transcoding a DICOM SR document into XML, then using DOM
- parsing a DICOM SR document directly into an internal representation and implementing a DOM interface, according to a defined DICOM to XML transformation, but without ever actually instantiating the XML representation
- use the ideas from DOM to define a DICOM SR specific object model, reusing the same objects and methods where appropriate, adding additional objects and methods to support SR specific constructs

The first approach, transcoding into XML, is feasible though not necessarily desirable.

The second approach, simulating an XML document but directly parsing the DICOM encoding is initially attractive. However, existing XML to DOM parsing tools cannot be used, since a DICOM specific parser is necessary, and the fit of DOM constructs to represent SR is not perfect.

Another approach that is described here is to learn from DOM, but implement a specific object model that applies to DICOM SR. This approach avoids slavishly following XML conventions that are not directly applicable, and allows the full fidelity of the SR document to be represented cleanly.

4. SR OBJECT MODEL

As DOM and SAX have proven in the XML world, a clearly defined boundary can be defined between tools that encode and parse documents, and tools that read or manipulate information content. The value of defining such a boundary cannot be underestimated. If parsers are implemented but accessed through a common API, then the users of the information need never concern themselves with the external (serialized) representation of the information.

The same argument applies to rendering information for human consumption. The expertise in creating rendering engines can be made independent of the serialized encoding of the source document through the use of a common object model internally. Equally, transformation tools that rewrite the tree can also be made independent.

The requirements for the object model in general were described earlier. The SR object model differs from the XML DOM in that:

- SR nodes are name-value pairs
- DOM nodes may be elements, attributes or plain character data
- SR nodes have explicit relationships between parent and child, that may be modeled as attributes of the child node
- DOM has no equivalent concept to relationships; the only relationship is containment and it is implicit
- SR node names are coded entries and values may be of various types
- DOM has no coded entry mechanism for element names, nor does it have a repertoire of value types

Taking these differences into account, and using the DOM (Core) Level 1 interfaces as a guide, one can define an SR-specific API that reuses as many DOM interfaces and methods as possible. In particular, the Node interface can be adopted. The generic Node interface can be extended from DOM to include:

- a coded entry concept name
- a value type
- a relationship type

Specific sub-classes of Node can be created to correspond to the SR-specific value types.

The generic traversal and editing methods can be reused. These include such methods as Node.getFirstChild(), Node.getNextSibling(), Node.insertBefore() and so on. In addition, since nodes are named (by their concept name), it is also appropriate to add additional methods that operate not as iterators but as accessors to specific named nodes. For example, a method such as Node.getFirstChildNamed(name) is potentially extremely useful.

Table 1 summarizes the interfaces and Table 2 summarizes the methods that might be useful for an SR Object Model, which are described in more detail in the references.[3]

An existing tree of nodes can be modified by using the insertBefore(), removeChild(), replaceChild() and appendChild() methods. When using these methods, note that:

- insertBefore(new,ref) adds the new child before the existing ref child; if ref is null then the new child is added as the last child, i.e. is equivalent to appendChild(new); if ref is not null, it must be an existing child
- replaceChild(new,old) replaces the old child with the new child; neither may be null and old must be an existing child
- removeChild(old) removes the old child; old must be an existing child

A special case that needs to be considered is how to intermingle methods that iterate and modify the tree of nodes. In the case of removing nodes while iterating through a list of children, one must keep removing from the head of the list rather than following the successors of a node that has already been removed (and hence whose successors will have been set to null). In other words, this works:

```
while (node.hasChildNodes())
        node.removeChild(node.getFirstChild());
```

and this does not (it only removes the first node):

```
for (Node child=node.getFirstChild(); child != null;
        child=child.getNextSibling()) node.removeChild(child);
```

The SR object model extends the iterators described in DOM to include versions that make use of the name of a node (or more specifically, the coded entry that represents the concept name). The CodedEntry class has an equals() method that compares the code value and coding scheme designator of two nodes to determine equality, ignoring the contents of code meaning. This equals() method is used by the getFirstNamedChild() and getNextNamedSibling() iterators. The following example creates an SR document with a root node and several children, and then iterates through it to find a named child:

```
Node root = nf.createContainerNode(
        cf.createCodedEntry(  209068 , 99PMP , Report ),
        null, SEPARATE );
root.appendChild(nf.createTextNode(
        cf.createCodedEntry( 209007 , 99PMP , Procedure ),
         CONTAINS , PA and lateral ));
root.appendChild(nf.createTextNode(
        cf.createCodedEntry( 209001 , 99PMP , Finding ),
         CONTAINS , Multiple masses ));
root.appendChild(nf.createTextNode(
        cf.createCodedEntry( 209005 , 99PMP , Conclusion ),
         CONTAINS , Metastases ));

Node finding=root.getFirstNamedChild(
        cf.createCodedEntry( 209001 , 99PMP ,  ));
```

In this example, getFirstNameChild() will return the node that is the finding of multiple masses. Notice that the coded entry for finding will be matched, even though the instance of the CodedEntry used is not the same instance, and the code meaning is different.

As a final example, consider a routine that searches an entire SR tree starting from the root, locating all occurrences of findings:

```
void search(Node node,CodedEntry wanted) {
        if (node.hasChildNodes()) {
                for (Node child=node.getFirstChild();
                        child != null; child=child.getNextSibling()) {
                                search(child,wanted);
                }
        }
        if (node.getNodeName().equals(wanted)) {
                /* found - do something deep and meaningful */
        }
}

CodedEntry finding = cf.createCodedEntry( 209001 , 99PMP ,  ));

search(root,finding);
```

Notice in this example that since one has to visit each node anyway, in order to recursively search its descendants, the iterators-by-name are not used. Also notice that a depth-first traversal is defined, since the check for a match and subsequent processing are performed *after* recursively visiting all of a node s descendants. The matching and processing code can be placed differently to achieve other traversal orders.

Since the test for equality is based solely on the code value and coding scheme designator, the code meaning may be localized or internationalized, or even absent, without invalidating the search. In this example, it was left as an empty string in the search key.

5. AN ALTERNATIVE APPROACH - EVENT STREAMS

Both DOM and SROM are aimed at hiding (encapsulating) the internal representation of a tree by providing methods for traversing and manipulating the structure. Though not strictly necessary, there is a presumption that the tree lives in memory. That is, an external document has been parsed into an internal representation, and perhaps validated during the process of parsing, or as a separate step. Alternatively, the tree may have been created de novo. After traversal or manipulation, the internal representation may be used directly, say to drive a display application, or it may be serialized into an external representation, such as an XML or DICOM SR document.

An alternative approach is to consider the top-down traversal of an XML or SR document tree as being a stream of events. In both forms of document, content is recursively nested. It is easy to detect the beginning and the end of a particular structure of content.

For example, during the process of a parsing a fragment of an SR document of the form:

```
<contains CONTAINER:(,,  Description of procedure )>
        <contains TEXT:(,, Description )= PA, lateral views >
```

events of the following form might be generated:

```
startContainer:
relationship: contains
conceptname: (,, Description of procedure )
startText:
relationship: contains
conceptname: (,, Description )
textvalue:  PA, lateral views
endText:
endContainer:
```

This approach is used by another popular interface for handling XML documents, the Simple API for XML (SAX).[4] Suppose the SR fragment shown earlier was transcoded into XML in the following manner, using attributes for coded entries:

```
<container>
        <relationship type= contains />
        <conceptname>
                <codedentry DN= Description of procedure />
        </conceptname>
        <text>
                <relationship type= contains />
                <conceptname>
                        <codedentry DN= Descript ion />
                </conceptname>
                PA, lateral views
        </text>
</container>
```

The SAX events that would be generated during the parsing of such a fragment might look something like this:

```
startElement: container
startElement: relationship
attribute: type= contains
endElement: relationship
startElement: conceptname
startElement: codedentry
attribute: DN= Description of procedure
endElement: conceptname
startElement: text
startElement: relationship
attribute: type= contains
endElement: relationship
startElement: conceptname
startElement: codedentry
attribute: DN= Description
endElement: conceptname
characters:  PA, lateral views
endElement: text
endElement: container
```

In some respects, whether or not to use an internal representation or a stream of events is a matter of taste and style. However, the event-based approach potentially has the advantage that the size of the document that can be handled is not limited by available memory. Furthermore, it is in many cases faster to use events rather

than traversing an object model, and many readily available stylesheet-driven translation tools depend upon the use of events rather than an object model (such as the popular XT and SAXON XSLT engines).

6. CONCLUSIONS

To make use of existing software, it is preferable to use an object model or event based approach to separating applications that generate, manipulate or render DICOM Structured Reports from the toolkits that are used to parse and serialize binary DICOM objects. Approaches have been described that make use of XML oriented tools and APIs such as DOM and SAX. The need to specialize the general purpose Document Object Model to handle specific DICOM SR constructs has been described and example interfaces and methods demonstrated. This has been compared with an alternative approach based on construction of a virtual XML translation into a stream of SAX events.

It remains to be seen which approach will be more popular in implementations for products. The ability to reuse of well understood and readily available consumer oriented software together with minimization of the level of understanding of DICOM required to implement useful reporting applications will likely be critical factors.

7. REFERENCES

1. DICOM (Digital Imaging and Communications in Medicine), *Supplement 23: Structured Reporting*, NEMA, Rossyln VA, 2000. ftp://medical.nema.org/medical/dicom/final/sup23_ft.pdf
2. World Wide Web Consortium. *Document Object Model (DOM) Level 1 Recommendation*. 1998. http://www.w3.org/DOM/
3. Clunie, DA. *DICOM Structured Reporting*, PixelMed Publishing, Bangor PA, 2000. http://www.dclunie.com/pixelmed/DICOMSR.book.zip
4. Megginson D. *Simple API for XML (SAX)*. http://www.megginson.com/SAX/index.html

Table 1. SR Object Model - Interfaces

Interface	Extends	Contains	SR Entity
CodedEntry			Code Sequence macro
Node		CodedEntry concept name relationship type value type	Content item
ContainerNode	Node	continuity of content	CONTAINER
CodeNode	Node	CodedEntry value	CODE
TextNode	Node	string value	TEXT
NumericNode	Node	string value, CodedEntry units	NUM
PNameNode	Node	string value	PNAME
DateNode	Node	string value	DATE
TimeNode	Node	string value	TIME
DateTimeNode	Node	string value	DATETIME
UIDNode	Node	string value	UIDREF
CompositeNode	Node	object reference	COMPOSITE
ImageNode	CompositeNode	object reference	IMAGE
WaveformNode	CompositeNode	object reference	WAVEFORM
SCoordNode	Node	coordinates and type	SCOORD
TCoordNode	Node	coordinates and type	TCOORD
ReferenceNode	Node	reference to another node	By-reference relationship

Table 2 - Object Model - Methods for Interface Node

Returns	Method	Parameters	Exceptions	DOM equivalent
CodedEntry	getNodeName()			yes
String	getNodeType()			yes
String	getRelationshipType()			no
Node	getParentNode()			yes
Node	getFirstChild()			yes
Node	getFirstNamedChild()	CodedEntry name		no
Node	getLastChild()			yes
Node	getPreviousSibling()			yes
Node	getNextSibling()			yes
Node	getNextNamedSibling()	CodedEntry name		no
Node	insertBefore()	Node new, Node ref	SROMException	yes
Node	replaceChild()	Node new, Node old	SROMException	yes
Node	removeChild()	Node old	SROMException	yes
Node	appendChild()	Node new	SROMException	yes
boolean	hasChildNodes()			yes
Node	cloneNode()	boolean deep		yes