DISARM — Document Information Set Articulated Reference Model

Rick Jelliffe

Discussion Draft

This note proposes an ISO standard "Document Information Set Articulated Reference Model" be developed, to provide the basis for ISO DSDL and for renewing ISO 8879 SGML.

Motivation

Since 1986, there have been four notable streams in markup languages:

- ISO 8879 SGML, extended by the General Facilities, Architectural Forms Definitions Requirements (ADFR), Lexical Types Definition Requirements (LTDR), Formal System Identifiers (FSI), Annexes J to L, augmented with OASIS Catalogs. A parser implementation of mature SGML in Open Source is James Clark's SP.
- W3C HTML, in various versions, with dialects including ASP, JSP, PHP, and Blogger. A parser implementation for mature HTML in Open Source is Dave Ragget's Tidy.
- W3C XML, extended by Namespaces, XBase, XInclude. Widespread implementations of parsers use the mature SAX API.

- The current ISO DSDL project, informed by RELAX Namespaces, RELAX NG, W3C XML Schemas, Schematron. The Xerces XNI API is a recent attempt to cope with post-processing XML, for uses such as validation and creating typed information sets.

In all these cases, the natural increase in complexity of evolving standards has made it difficult to understand the processing order and operation. ISO 8879 has been widely criticized for not being amenable to simple grammatical analysis ("not using 'computer science concepts'"), yet the same problems are experienced even with overtly layered specifications such as the XML family, due to this entropy.

These problems would be reduced by introducing a reference model which was neutral with regard to each of the four main streams, but allowed clear and diagrammatic exposition of the stages of parsing and processing a marked-up document incrementally from bits to a terminal information set.

This discussion paper suggests such a model: DISARM - Document Information Set Articulated Reference Model.

The utility of DISARM might include that it can provide an attractive way to allow a top-down re-specification of SGML in a future ISO 8879. It would might also provide some help for DSDL.

Model

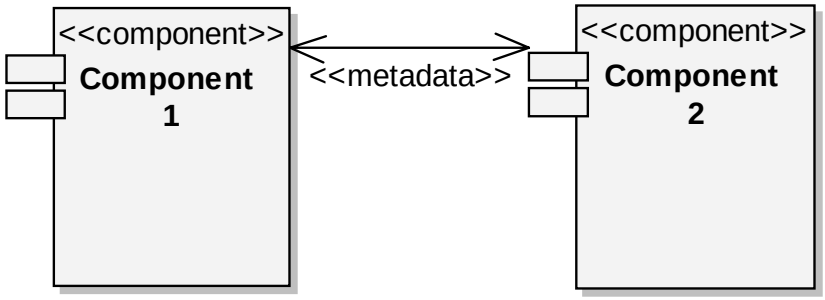
The reference model uses UML terminology and diagrams at the top-level only. If desired, specific graphical stereotypes could be created, as allowed by UML.

It models the kinds of markup processing of interest as a chain of components, one connected to the next, each of which implements a common event-passing interface. Different markup languages and SGML features can be modeled using particular chains of components.

NOTE: Though the components could be used for a design, the intention of the reference model is to facilitate clear and rigorous exposition. Components in the reference model could be implemented, for example, as objects, as function or procedure calls, callbacks, or folded together for efficiency.

COMPONENTS

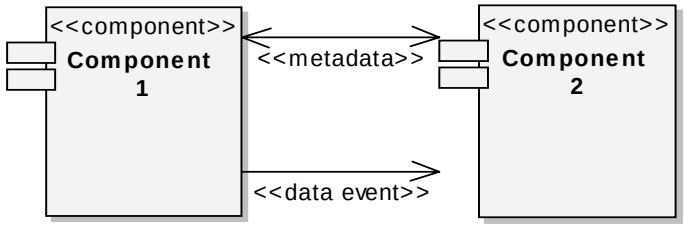
FIGURE 1. Component and metadata events



Each component can send and receive *metatdata events* from its predecessor and successor. A component decides whether to use the *metadata event* based on its own criteria. Each component propagates each *metatdata event* down the chain. For example, a component that parses markup declarations would send the entity declarations as a *metadata event*, and a component that was the entity manager would use this event to load its internal tables.

NOTE: Though this could be used for a design, the intention of the reference model is to facilitate clear and rigorous exposition. An actual design could, for example, broadcast events to all components, use a listener system, or use a global metadata store.

FIGURE 2. A component with metadata and forward data events



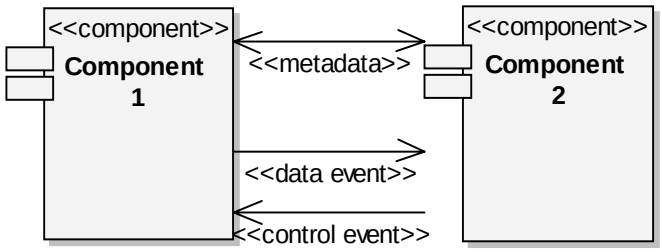
Each component can send forward *data events* to its successor. A component acts on the incoming *data events* and passes them on or generates some other *events* in response.

A component may propagate forward *data events* it does not recognize, to allow robust filters. For example, a component that performs newline normalization could receive characters for its incoming *data events* and replace Macintosh newlines with XML newlines in the outgoing *data events*. *Data events* can include what ISO 8879 terms *signals*.

NOTE: Though this could be used for a design, in a streaming context, the intention of the reference model is to facilitate clear and rigorous exposition. An actual design could, for example, send references to some external text file or GROVE.

The peek abstraction is available on *data events*, as a shorthand for a component reading some *data events* and them pushing them back in a *control event*. Components that are merely filters should propagate *control events* to their predecessor, to allow pushback *control events* with subsequent mode-changing *control events*.

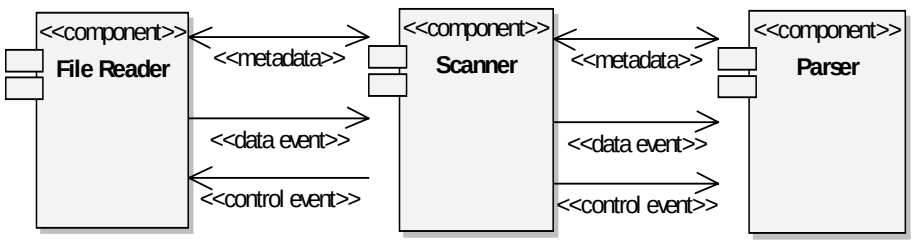
FIGURE 3. A component with metadata and data events, and control events



Each component can send back *control events* to its predecessor. A component acts on the incoming *control events* and passes them back or generates some other *events* in response. A component may propagate back *control events* it does not recognize, to allow robust filters.

NOTE: Though this could be used for a design, the intention of the reference model is to facilitate clear and rigorous exposition. An actual design could, for example, use function calls, non-blocking IO, etc.

FIGURE 4. A chain of three components



Above is a simple example. This markup language is modeled using three

components:

1. A *file reader* converts bits to characters, and passes the characters as *data events* to the *scanner*, terminating with an *end-of-file* data event.
2. The *scanner* reads the characters, and passes the tokens as *data events* to the *parser*.
3. The *parser* reads the tokens, and parses them against some internal grammar.

Within this framework both push-style data flows (where the *data event* comes first, with the *control event* acknowledging it*)* or pull-style data flows (where the *control event* requests the next *data event*) are possible. Indeed, mixtures are possible: for example where the scanner requests characters from the file reader, then pushes tokens to the parser.

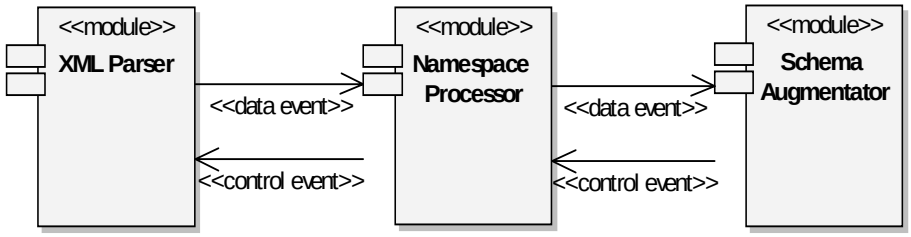
NOTE: Though the particular event protocol could be used for a design, the intention of the reference model is to facilitate clear and rigourous exposition.

MODULES

Components can be grouped in modules, if there is no metadata dependencies and only simple pull-request/responses or push/acknowledgment pairs of control events.

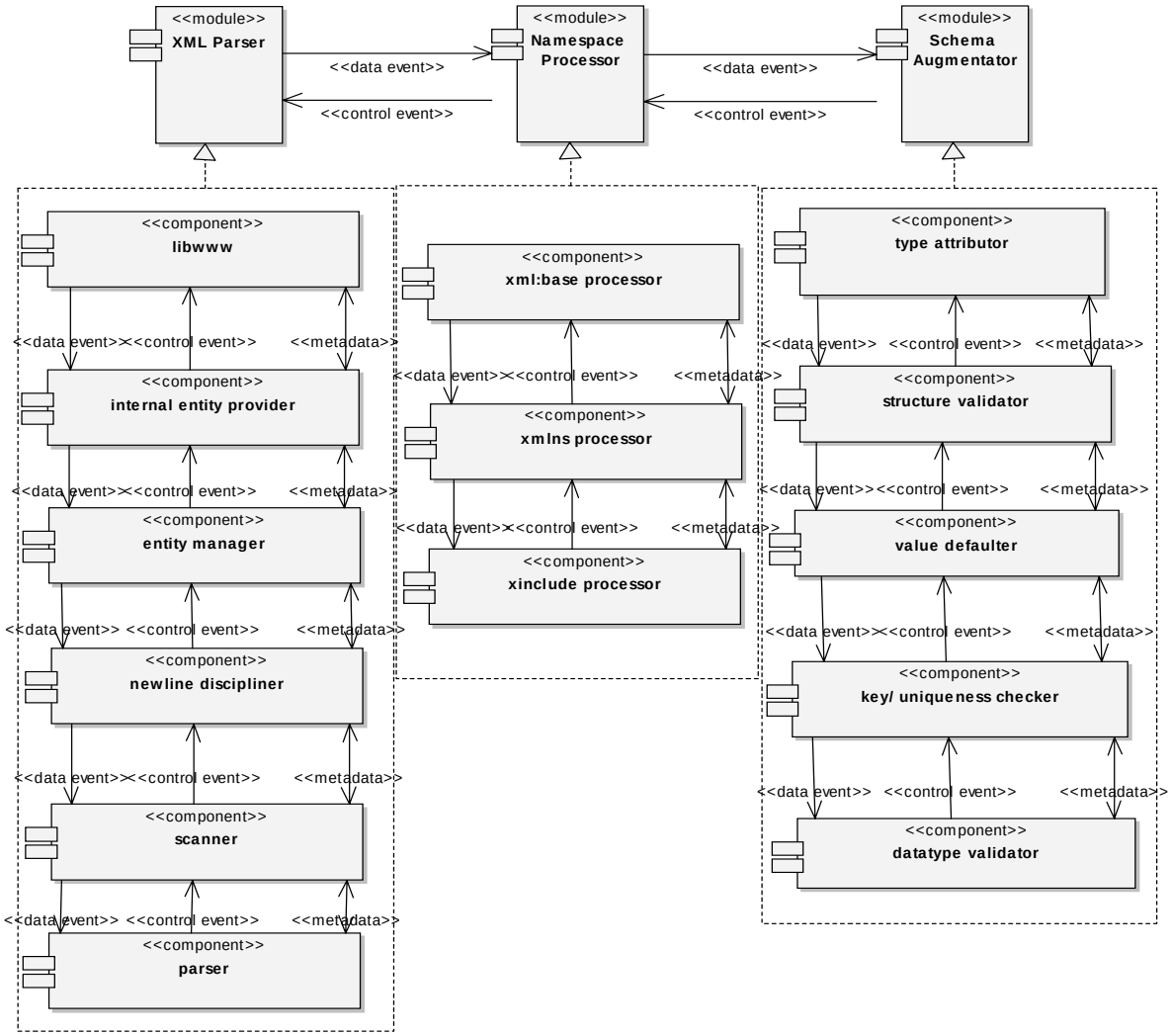
The following example is of a parser for a simplified XML with no markup declaration support, but with namespaces, and then XML Schema augmentation.

FIGURE 5. Modules for an XML system



Notionally, a module contains the components. But the modules can components can be shown in the same diagram, as in the following diagram. This diagram is becoming quite complex; but it is the complexity of repeating patterns not the complexity of differences. From the point of view of standards-writing, it provides many clear positions which the rest of the text can refer to.

FIGURE 6. Modules and components for an XML system



Xinclude processing: In this case, inclusions can be handled merely as special entity references: the URL being sent first to entity manager as a *metadata event*, immediately followed by *control event* acting as an entity-push request.

Using the Diagrams

For DSDL, the diagrams are useful for explaining processing flow.

For explaining SGML/XML/HTML, the diagrams are useful because many features can be expressed as components that can be inserted into the stream.

For example, a component that performs OMITTAG YES normalization can be placed before (or after?) the parser. Some features, such as SHORTREF YES, are probably better left embedded into the scanner, but they could be placed after the scanner. The LINK processor can go before (or is it after? these diagrams makes that kind of question clear) the attribute value defaulting module. A CONCUR YES processor can go after the scanner, to filter out tags.

Architectural processors would similarly be easier to explain, as just more modules.

Furthermore, the diagrams can be used (in conjunction with UML activity diagrams, perhaps) to show more complex interactions, such as white-space movement or peeking for NAMECHARs. It seems likely to me that complex control/data event interactions between modules represent points where implementers are likely to experience difficulty. XML can be seen, in part, as an attempt to reduce the need for any control events (apart from simple control events of the get() or ack() kind) setting modes in previous components.

Examples

(I have not created drawings for these yet. The reader is invited to mentally draw the diagrams.)

FIGURE 7. (Diagram. Components are labelled: external entity provider, internal entity provider, entity manager, newline discipline, scanner, de-minimizer, parser, default value decorator, ID/IDREF checker.)

This first example is for RCS SGML (eliding declaration processing).

FIGURE 8. (Diagram. Components are labelled: libwww, internal entity provider, entity manager, newline discipline, scanner, parser, namespace, type attribution, structure validation, value defaulting, key/uniqueness-checking, datatype validation)

The example is of a parser for XML with no markup declarations, but with namespaces, Xinclude processing and then XML Schema augmentation.

FIGURE 9. (Diagram: Components are labelled: storage manager, entity manager, newline discipline, declaration scanner, declaration parser, scanner, parser, default value decorator, ID/IDREF checker.)

This example is for XML 1.0. The declaration parser would provide *metadata events* to the other components to process the XML instance. The declaration scanner and parser would pass all *data* and *control events* through after the internal subset had ended. A real implementation might be able to remove the components the from the processing chain.

FIGURE 10. (Diagram. Components are labelled: external entity provide, CATALOG entity resolver, internal entity provider, entity manager, newline discipline, SGML declaration scanner, SGML declaration parser, markup declaration scanner, link parser, declaration parser, scanner, concur-stripper, de-minimizer, parser, link processor, default value decorator, AF processor, LT processor.)

This example is for full-blown SGML, this time with OASIS CATALOGS, short references, omittag minimization, CONCUR, links, architectural forms, and data typing (eliding declaration processing.).

Short-references are explained in the following way. This scanner component has a table of delimiter maps, loaded from *metadata events* from parsing the declarations, and a stack of open maps. *Control events* from the parser push and pop delimiter maps (i.e., SGML's USEMAP). When a short reference delimiter is found, the scanner sends a *control event* for the entity manager to push the entity declared for that reference (in a short reference table.)

How other Features are Supported

For XIncluded documents in other configurations, it may be better to think of the XInclude component acting as a liaison between separate invocations of the DISARM chain.

In the case of SUBDOC, components must have all their fields on a stack. The declaration scanner and parser must be active during the document's life.

A control event might build the next stage or push components based on parsed

declarations. For example, discovering a document starts with <?xml or <!SGML could cause the construction of different processing chains.

The entity providers (storage managers?, data sources?) can provide *data events* with character data, as CDATA, PCDATA.

There is scope for discussion on whether general entity references should handled by the scanner, by the parser, or by a subsequent stage. If by a subsequent stage, then marked section and declared content types need to cause relabelling of incoming data as CDATA
and RCDATA.

Also, there is scope for discuttion on the need to support outside world at each end? E.g. a validator may have a Boolean data event as its final output.

Comments

The complexity of a system can be judged, to a certain extent, by the level of coupling which remains after grouping functionality into intellectually cohesive components: in particular by the number of different control events required. However, the length of the chain and the internal complexity of each component are also important metrics.

To Do:

Explain how self-describing versus externally labelled is expressed

Explain handling of Inline markup vs. declarations v external markup.
Explain how SUBDOC and XInclude fits in