Specification in a Large Industry Consortium – The MOST Approach –

Thilo Gaul

Welf Löwe

Markus L. Noga

University of Karlsruhe
Program Structures Group
Adenauerring 20a, D-76131 Karlsruhe, Germany
{gaul | loewe | noga}@ipd.info.uni-karlsruhe.de
<http://i44w3.info.uni-karlsruhe.de>

Abstract—The MOST Cooperation (*Media-Oriented Systems Transport*) is an industry group of automotive and supplier companies that specifies interfaces of multimedia components. Many members are competitors: they collaborate on standards while preserving their competitive advantages. Moreover, development processes differ for each member, but the MOST standard must be concise and consistent. We develop an XML based architecture to support individual workflows and the specification process as a whole.

I. INTRODUCTION

Some industry standards emerge by themselves when a particular technology is widely adopted as a *de facto* standard. Others are the result of an explicit specification process designed to carefully balance special interests in a multi-vendor environment.

These environments usually take the form of an industry consortium, an alliance of organizations united by a common goal. In contrast to a merger, all constituent companies guard their identity and independence. Large numbers of organizations and even larger numbers of employees are involved. Interactions between consortium members are scattered over space and time. Despite this complex setting, a consortium has to project a consistent and reliable exterior image to promote its results with outsiders.

The complexities of consortium dynamics call for computer aided specification. In this paper, we first examine the problem domain using a real-world case study in section II. Requirements for workflow and the supporting software architecture are derived there. In section III, we develop such an architecture to support the specification process. As we will see, simple solutions simply fail. Finally, we discuss experiences with the working system and lessons learned in section V.

II. THE CASE STUDY

Automobiles contain ever increasing amounts of electronics. This development originated with anti-lock brakes and computer-controlled injection systems. Electronic Control Units (ECU) now play a crucial part in all the functional areas of a vehicle such as body control (e.g. instrument panel, window lift, automatic door lock), vehicle control (e.g. engine management, transmission, brakes), or infotainment/multimedia (e.g. radio system, road guidance, cellular phone). This was made possible through the advent of

networking technology such as CAN. Typical vehicles consist of several interconnected networks that more or less reflect distinct functional areas. For instance, current cars like the Volvo S80 include 18 ECUs connected via six networks: a low-speed body electronics CAN bus (125kbit/s), a high-speed powertrain CAN bus (250kbit/s) and four other networks. It has been estimated by Dataquest that the *average* semiconductor content of a vehicle will reach \$240 by 2001, with consumption of DSPs, microcontrollers and microprocessors reaching \$4.9 billion, not counting the costs for software development.

Tomorrow, an automotive company may be seen as an electronic systems integrator, with sidesticks and electronic actuators replacing steering wheels and transmission trains. Fuel cells may well replace the internal combustion machine as the industry's main workhouse [1].

Automotive companies are responding to the challenge. As demonstrated, the importance of universal bus systems to interconnect on-board subsystems as well as to integrate external diagnostic and maintenance systems is widely acknowledged.

Historically, interoperability between vendors has been patchy, limiting a company's choice of suppliers. Older bus systems specified only a communication standard, that is, they provided a space for devices to talk. To enable vendor interoperability, their devices must also know what they are talking about. This is what MOST was founded to accomplish.

The industry approach to this problem is application standards. By defining an abstract telephone and its interfaces, complying telephones of different vendors can be exchanged transparently. A different approach, the semantic web [2], describes individual semantics per device instance. Clearly, this strategy does not translate well to deeply embedded systems.

The MOST Consortium [3] is an industry group of automotive and supplier companies. It was founded in 1998 to specify communication and application standards for the car industry, which will be available free of royalties. In alphabetical order, its founding members are BMW, Daimler-Chrysler, Harman/Becker and Oasis SiliconSystems. To date, the MOST Cooperation has embraced over 50 associated members. Since June 2000 the Program Structures Group has been advising MOST on specifica-

tion processes for application standards and implemented tools to support specification workflow.

The following subsections observe the specification process in MOST, which lead to a requirements list. Any software architecture supporting the process must comply with these requirements.

A. Consistence and Conformance to Rules

The specification process should lead to a unique, concise document. The document has to be internally consistent, that is, it must conform to strong structural rules. These rules should be defined in advance, and it is desirable to check for rule violations automatically. Hence, the specification document should be strongly structured and formal.

B. Hierarchical Organization and Hierarchical Documents

The decision process is hierarchical: most work is done in groups dealing with parts of the problem. Their solutions are revisited and composed by higher instances, which in turn contribute parts to the next higher level. Each working group contribution must be specified in a concise formal document. The composition of these documents at the next level should work without changing their content. Either the document is accepted as is, or it is rejected for further processing by the authors. This approach leads to hierarchical documents with strict responsibilities.

C. Document and Document Views

A formal document is hard to read. Many details agreed on in a working group are negligible at higher levels of the decision process. Additionally, some content should be hidden from the public, while other parts are to be published and discussed as early as possible.

Furthermore, different views on the document are required: to filter information, to restructure it for human readability or simply to generate different formats. To guarantee document consistence, the formal document must be the sole authoritative reference and the unique source for view generation.

D. Editing the Document

Editing should be done on a human-readable view of the document. As mentioned above, these views are generated from the document source, but editing implies the existence of an inverse transformation. The editor should also support strong structural and rules consistency checks on the document.

The document may also change as the result of proprietary processes transparent to the consortium (e.g., parts may be regenerated from a member's database or case tool). Hence, consistency checkers should operate independent from an editor. As rule sets may evolve, both editor and checker must be maintained. Independent redesigns are highly time consumptive and error prone, so it is desirable to formalize rules and automate their application in both editor and checker.

E. Versioning the Document

Versioning must refer to the documents produced by the individual working groups, and their compositions by higher levels of the decision hierarchy. It must be mapped to version numbers in diverse systems of the contributing companies, as they are unlikely to discontinue internal versioning.

F. Communicating the Document

The working group members are distributed over the participating companies. Communication via the web is standard and should be supported.

III. THE ARCHITECTURE

Fig. 1 shows the overall MOST Architecture. The internet, represented by a cloud, separates client processes and the MOST Document Server, which stores specifications and manages client communications.

More specifically, the MOST Document Server acts as a version repository for specification documents and fragments thereof. It hosts document consistency checkers and tools to generate views. The former guarantee that only consistent documents are stored and served. The latter generate consistent, high-quality printable versions of the specification.

Arbitrarily many client processes may communicate with the server. A canonical client, the MOST editor, provides smart context-sensitive editing functionality and integrates local copies of the server-side checker and transformation tools. While this allows fast local checks, the server remains the final authority on consistency.

The server may also be accessed with proprietary clients, provided they support MOST document formats (see III-A below). E.g., clients may manage documents or fragments thereof in RDBMS or arbitrary document management systems. The server optionally provides them with stand-alone versions of the consistency checker and print version generator.

Client-server communication is mediated by change managers. On commit, a change manager determines which

Fig. 1. The MOST Architecture

changes were made by the client. It displays these deltas and requests confirmation prior to transmitting them to the server.

Until now, we pictured the MOST Document Server as a unique central server. This prevents fragmentation of the specification into mutually incompatible versions. However, there is a need for proprietary extensions to the specification. Consortium members need to integrate them seamlessly with the basic specifications, so they must conform to the same consistency rules.

Duplicate servers with modified access policies enable this mode of operation. They act as clients of the central instance. Like normal clients, they may check out documents and check in updates. They may partake of the existing client infrastructure for editing, checking and print transformations.

A. Document formats

The specification is represented as an Extensible Markup Language (XML, see [4]) document. This is a formal notation for which consistency checking and automatic processing are easily tractable.

Many restrictions on the specification document can be formalized in a Document Type Definition (DTD). However, DTDs are limited to basic structural properties and its types are limited to string and don't-care. Consistency rules not expressible with DTDs were previously specified in a precise but informal document.

We formalized those consistency rules with checker scripts. These scripts are implemented as Extensible Stylesheet Language Transformation (XSLT, see [5]) scripts. Further formalizations replace the DTD with XML Schema descriptions, whose superior type and constraint system can express closer consistency rules. Altogether, this approach lead to a formal notation for the strong consistency of documents.

Moreover, checking strong document consistency is performed automatically by standard validating XML parsers and XSLT processors.

The structure of the documents is hierarchical. The whole specification is split in so called function blocks. Function blocks in turn contain a sequence of functions. Working groups deliver function block specifications. Usually, a function block contains 10 to 100 functions. Each function is *per se* a meaningful unit. Its specification can be printed on one or two pages. XSLT filter can hide function specifications or function block specifications completely or skip the body of those units. The former is required to hide proprietary developments, the latter to abstract from details.

Views on the documents are also generated by such scripts. They usually target HTML. PDF and other formats are covered by standard transformation tools. Other views, like IDL views (defining interfaces), code views (defining interfaces in some programming language) etc. will follow the same principle.

B. The Editor Architecture

The editor architecture has two layers: a basic XML editor serves as the foundation for a special MOST editor.

A general, validating XML editor facilitates basic editing. It parses general XML documents and validates them against a given DTD. The editor captures an internal representation of the documents and visualizes it as a tree or as a plain text document. It provides basic editing functionality, e.g., insert, delete, copy and paste. Each operation can be validated against document standards.

The MOST editor defines certain special views on XML documents conforming to the MOST DTD. These views are similar to the print version of the document. To obtain them, the MOST editor filters the tree data structure provided by the general XML editor. Some of its elements are not displayed in a tree anymore. We therefore distinguish tree nodes (displayed in the tree structure) and non-tree nodes (not displayed in the tree structure):

- The root element type is a tree node.
- If the content model of an element type¹ only contains an alternative choice of or finite sequences over element types, the child elements are non-tree nodes, i.e., they are filtered from the tree structure.
- If the content model of an element type contains an iteration over child element types, those child elements are also tree nodes. The parent node in the tree structure is the parent node in the document if it is also a tree node. Alternatively, the parent of a tree node is the first transitive predecessor of the node in the document that is a tree node.

Non-tree nodes are displayed in special views: alternatives in choice boxes and sequences in sequences of text fields. Each non-tree element is assigned to a tree element, namely to the next transitive parent that is a tree element. The view of a non-tree element is displayed whenever the tree element it is assigned to is selected in the tree.

The editor functionality of the MOST maps to the basic editor functions of the validating XML editor.

These operations pertain not only to the MOST DTD, but rather to DTDs in general. They can be used profitably in other ventures, if two problems are solved. First, there are structural mismatches when comparing the XML document and its print view. Second, many consistency rules cannot be expressed with DTDs (but rather with XSLT scripts).

In the MOST context, the first problem takes this shape. A parameter points to the operations it occurs in (and to the exact position in these operations) in the XML source document. In the print version it is the other way around: each operation refers to the parameters it requires. It is easy to see that such a transformation cannot be performed with the simple filtering operations described above.

This problem is solved by defining a second DTD (editing DTD) and transformations between the original document and the view on the document conforming to this second

¹In a DTD, the content model of an element type defines legal sequences of child elements.

DTD. The editing DTD defines the structure of the print and editor version of the original document (conforming to the source DTD). The required transformations are defined by an XSLT script. It is similar to the above transformations to the HTML view (print version) of the document but do not contain any layout or font information. The inverse transformation also exists as the print version must capture all relevant details of the specification. It is also implemented with XSLT.

The second problem, the check for consistency, is now easy to solve: it is performed whenever either document, the source or the edit document, changes and a transformation to the other is required. The checkers are also defined by XSLT scripts and integrated into the transformation scripts. Optimizations that perform only incremental checks are possible if the checkers only require local information.

With this pair of DTDs (source and editing DTD) and transformations between the corresponding documents, editing works as follows:

1. The XML source document (source XML) is converted to the editing view XML document (edit XML). Its structure is close to the print view structure. This conversion validates the source document against the source DTD and performs consistency checks.
2. The XML editor loads the edit XML document and validates it against its DTD (edit DTD). It captures the data structures of this document but does not display them.
3. The MOST editor displays these structures. It filters the tree as described and displays the special views. Editing operations on this level are mapped to the editing operations of the XML editor.
4. If the user confirms changes, the relevant parts are converted back to the source XML. Again, this triggers transformation scripts and the integrated checking operations.

We omit a discussion how the local changes in the edit XML document can map to local changes of the source XML document and how the XSLT checkers can work locally too.

We conclude this section with the remark that print documents should be generated in a two stage transformation: first the source XML is transformed into the edit XML document, then the edit XML document is transformed into HTML including stylesheet information. The former transformation is shared between the editor and the print viewer.

C. The Versioning Architecture

A major achievement for the consortium wide specification management is a centralized and automated document processing. The usual way to process the specification(s) used to be to annotate and discuss modifications on printed paper. Document management meant to spread a printed version of the document to argue about. Modification proposals used to consist of fragments of printed document. Every partner of the consortium had its own in-house technology to process and store the documents, applying document changes on technical level among the different storage

techniques has been far too complex. Also the access to former revisions or even to proposals from other partners immanently stuck to printed versions.

The approach chosen to attack these problems is to establish a centralized document management instance, that also realizes a version management.

C.1 The Versioning Mechanism

The granularity of the versioning for arbitrary documents must be adapted to the concrete application context and the document workflow. For the MOST consortium, we chose an approach with tree levels (see fig. 2): Top level is a whole catalog, which is split into blocks of functions on level two. The nodes in this hierarchy tree are single functions of which function blocks are constituted. Any single atom in this setting is version controlled, parallel visible proposals are maintained for functions.

The concept of configurations realizes the notion of a snapshot or the formerly notion of a printed version. Versioned atoms of the document tree are bundled together and can be accessed further on as one configuration version. This notion is comparable to a release in software development terms.

Fig. 2. Hierarchical Versioning

C.2 The Version Control Server

The three architectural layers depicted in fig. 3 are reflected in the server implementation as follows. The user interface layer abstracts the management functions from the user:

- a WWW server provides the user interface HTML pages
- this server maintains user accounting and access control on HTML level
- a (maybe different) server processes dynamic content, which initiates catalog management functions and delivers results of the management layer

The catalog processing management layer performs actual check-in and check-out operations on the version:

- administers release configurations of catalogs and sub-components
- compiles catalogs and partial documents of a distinct version and configuration to single documents

Fig. 3. Three Layer Server Architecture

- manages users and access rights on repository level

The version control base system:

- forms the real repository of XML fragments
- controls versions and branches, stores version updates as compacted formats
- stores release configurations

C.3 Server Implementation Issues / Architecture Instances

A main focus for the implementation of the described architecture was put on the realizability with standard and possibly freely available software and hardware components.

The current instantiation of our versioning architecture comprises all the described features and implements them with industrial standard components:

- The web interface is implemented with dynamically generated but standard HTML4 (see [6]) pages for management and data transfer. The internal interface was designed that way that it is possible to access the main catalog management functions remotely by non-web interface clients.
- The web server itself is a basic Apache 1.2 web server (see [7]), which is freely available. It provides upload and download features as well as a fine-grained access control on the web interface level.
- The catalog management software is the main part of the server application logic. In principal this part could be written in any server side application language, i.e. any scripting language or Java Servlets etc. The current implementation is written in Perl5, Windows and Unix implementations of Perl5 are freely available and wide-spread.
- The version control software is the central application for the repository and version control logic. The current implementation is done with the Concurrent Versions System (CVS, see [8]), a freely available version control system. Version 1.0 of CVS is sufficient for the current approach, implementations of CVS are freely available for Windows (NT/9x) and most UNIX platforms (Linux, SunOS, HP/UX, AIX, IRIX). A CVS repository can be accessed and (widely) managed remotely without terminal access to the server.

Fig. 4. Parsing times for specific and generic parsers

- The current server machine itself is a standard, mid-performance Linux 386PC (RedHat6).

IV. PERFORMANCE ISSUES

Standard XML processing tools and techniques include SAX parsers, DOM structure builders and XSLT transformation interpreters. They are highly flexible and perfectly suitable for small and medium data sets. However, processing large data sets at interactive or near-interactive speeds, as required by the MOST architecture, requires a different set of tools, the aXMLerate toolbox [9]. Its domain and components are described in the subsections below.

A. Parsing

When filtering small amounts of information from large documents, such as outlines or specific excerpts, query time usually dominates transformation time by far. As version control systems integrate quite poorly with databases, the MOST architecture is forced to store the majority of data in flat files. Thus, parsing time accounts for most of overall query time.

We applied compiler construction techniques to this problem. By transforming document type descriptions into context-free grammars, we can generate highly efficient parsers for conforming documents with standard tools. This approach trades full generality for faster processing.

Fig. 4 compares a parser generated with our Xsdc tool with a fully general implementation from the Apache Xerces project (see [10]). Both implementations run in the Java Virtual Machine.

B. Structure building

Object creation is the single most expensive operation in the majority of object-oriented languages. In Java, the number of object creations may dominate all other terms of execution time for a wide spectrum of input sizes. Standard DOM implementations require ten and more times the original file size for a representation in main memory.

Savings can be reaped in two areas here: by reducing the size of the tree to build, and by optimizing the representation of individual nodes. The first takes the shape of tree generators optimized for specific queries, which e.g. discard low-level nodes. This approach yields significant speedups for sparse views on MOST data, such as outlines.

The second area is covered by specific data structures. By exploiting type information from the document type, nodes may be stored more compactly and more suitable for subsequent processing, e.g., by storing enumeration values as numbers, not as character strings.

We are working in both directions. Fig. 5 shows how parsing and structure generation time for structures generated with our aXMLerate toolkit compares against standard DOM implementations, with standard SAX and well-formedness curves displayed for comparison. These are all native C implementations.

Fig. 5. Specific and generic structure building times

C. Transformation

XSLT supports conditionals, named functions and recursion, making it a Turing-complete programming language. But completeness does not imply tight bounds on execution time. Indeed, several linear time algorithms become superlinear when implemented in XSLT.

One simple example is matrix transposition, which is commonly required for HTML table output. A transposition algorithm reads the n elements of a matrix in transposed order. Unfortunately, selecting an element (i, j) is $O(\sqrt{n})$ in XSLT because the specification mandates the construction of intermediate processing results, so overall time for transposition becomes $O(n\sqrt{n})$. Another example is unique sorting. Future optimizing implementations of XSLT may alleviate these problems, but the semantics of XSLT notation will remain counterproductive to performance in any case.

The MOST view transformation is an example of a complex transformation over a large document. Among other algorithms, it massively employs transposition and unique sorting. Initial implementations in XSLT took over one and

a half minutes to generate a view of the complete specification. By addressing the weaknesses of XSLT in our programming idiom and limiting scopes as far as possible, we were subsequently able to reduce the time for this transformation to eight seconds on an identical machine.

Optimizing XSLT compilers developed by our research unit further accelerate this transformation by a factor of three. We again traded the flexibility of interpreters for the swiftness of pre-compiled special transformers, enabling view updates in near interactive time.

Another approach pursued by our research group is replacing XSLT entirely with more powerful transformation languages. Exploratory studies reported a speedup of a factor of three over equivalent interpreted XSLT, with much potential still remaining.

V. LESSONS LEARNED

The process that we developed in this paper and the architecture that supports it are relatively independently of the actual documents that are produced. It nicely generalizes to similar processes. For the distributed development of other documents we need:

1. A concept of atomic working units and their hierarchical composition to a document.
2. An XML DTD or an XML Schema specification defining structurally correct documents.
3. XSLT scripts defining additional consistency checks if necessary.
4. XSLT scripts to generate views on the document. Especially, a script to generate the editor view on the document and the inverse transformation.

We strive to parameterize the MOST architecture with these variables, thus generalizing it to a universal framework for distributed document processing. Realizing this vision is a task for future work.

REFERENCES

- [1] A dream car. *The Economist*, March 8th 2001.
- [2] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.
- [3] *Media Oriented Systems Transport*. MOST Cooperation, <http://www.mostnet.de/>.
- [4] *Extensible Markup Language (XML) 1.0*. W3C Recommendation, <http://www.w3.org/TR/1998/REC-xml-19980210>, 1998.
- [5] *XSL Transformations (XSLT)*. W3C Recommendation, <http://www.w3.org/TR/xslt>, 1999.
- [6] *HTML 4.01 Specification*. W3C, <http://www.w3.org/TR/html4/>.
- [7] *The Apache Web Server*. Apache, <http://www.apache.org/>.
- [8] *CVS: Concurrent Versions System*. CVS, <http://www.cvshome.org/>.
- [9] *aXMLerate Project*. B2B Group, University of Karlsruhe, <http://i44pc29.info.uni-karlsruhe.de/B2Bweb/>.
- [10] *Xerces Java Parser*. Apache XML Project, <http://xml.apache.org/xerces-j/>.