Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 163864, 6 pages
doi:10.1155/2008/163864

Research Article

Embedded XML DOM Parser: An Approach for XML Data Processing on Networked Embedded Systems with Real-Time Requirements

Esther Mínguez Collado,[1] M. Angeles Cavia Soto,[2] José A. Pérez García,[3] Iván M. Delamer,[1] and Jose L. Martínez Lastra[1]

[1] *Institute of Production Engineering, Tampere University of Technology, 33101 Tampere, Finland*
[2] *Departamento de Ingenieria Electrica y Energetica, Universidad de Cantabria, 39005 Santander, Spain*
[3] *E.T.S. de Ingenieria Industrial, Univerisdad de Vigo, 36310 Vigo, Spain*

Correspondence should be addressed to Jose L. Martínez Lastra, lastra@ieee.org

Received 5 February 2007; Revised 18 June 2007; Accepted 8 October 2007

Recommended by Valeriy Vyatkin

Trends in control and automation show an increase in data processing and communication in embedded automation controllers. The eXtensible Markup Language (XML) is emerging as a dominant data syntax, fostering interoperability, yet little is still known about how to provide predictable real-time performance in XML processing, as required in the domain of industrial automation. This paper presents an XML processor that is designed with such real-time performance in mind. The publication attempts to disclose insight gained in applying techniques such as object pooling and reuse, and other methods targeted at avoiding dynamic memory allocation and its consequent memory fragmentation. Benchmarking tests are reported in order to illustrate the benefits of the approach.

1. INTRODUCTION

Current trends in the industrial automation domain are pushing the adoption of information and communication technologies (ICT) at the device level, increasing the amount of information processing local to where process control occurs. Networked embedded systems (NES) are being equipped with increasing computation power and communication resources that allow direct integration with enterprise and supervisory control systems. Among the technologies that are used to represent and communicate information, the eXtensible Markup Language (XML) is emerging as a prevailing syntax in the embedded domain, answering to requirements of interoperability and integration with desktop and server-type systems. A few example technologies that illustrate this trend are the OPC Unified Architecture (OPC-UA) [1], the Computer Aided Manufacturing using XML (CAMX) framework [2], and the Devices Profile for Web Services (DPWS) [3].

One of the challenges faced by the processing of XML data in embedded automation controllers is the fulfillment of real-time requirements. The majority of currently available off-the-shelf XML processors do not provide features that address deterministic behavior, and background research presented in Section 2 has found no available embedded XML processor or research activities pointing in this direction. In addition, the volume of data exchange and data processing within industrial controllers is increasing, taking more resources which were previously just granted to control tasks without need for arbitration. The challenge is therefore to provide a predictable behavior for XML processing activities so that the performance of real-time control tasks is not affected.

Among the major sources of problems introduced by XML processing is the dynamic allocation of memory during parsing operations. This process is not time-deterministic, and may lead to memory fragmentation and eventual failure to allocate sufficient memory for the operation. In environments such as Java, dynamic allocation leads to indeterminist garbage collection.

This paper presents a design and experimentation of a prototype XML processor designed to avoid dynamic memory

allocation. The processor is denominated EXDOM (Embedded XML DOM Parser), and was developed using the Java 2 Micro Edition (J2ME) platform. EXDOM is specifically designed for data analysis on NES and it is focused in optimized memory use. EXDOM offers in addition a new approach for processing data in environments where the structure of exchanged XML messages is known in advance.

The rest of this publication is structured as follows. Chapter 2 surveys related work and summarizes the state-of-the-art on XML parsers, with special focus on the embedded domain. Section 3 presents the set of methodologies and algorithms used on the EXDOM design. Section 4 describes benchmarking tests and results. The final section presents conclusions and proposes future research lines.

2. XML PROCESSORS

According to Maruyama et al. [4], the fundamental functionality of XML processors is called *parsing*. Parsing is the process of analyzing input data (XML documents), and generating an internal and structured data representation which can be accessed by application programs. An XML processor performs both parsing and its inverse operation: generation, or *serialization*, of XML documents. The functions of an XML processor are illustrated in Figure 1 [4].

A first classification of XML processors can be made between "heavy" and "light" processors. "Heavy" processors are called those which have been designed for advanced computation environments, and are not suitable for working in NES because of their size and memory requirements. However, relevant design concepts are often suitable for being applied on "light" processors, which are designed for environments with limited processing power and memory availability.

A basic classification of XML processor includes

(i) tree-based APIs,

(ii) event-based APIs.

2.1. Tree-based API

The document object model (DOM) [5, 6] is the predominant tree-based API for accessing XML documents. The XML document is represented in a tree structure where every XML tag is a node. Data is stored on memory as a tree that represents the complete XML document, allowing the possibility to navigate the tree, modify it, and/or serialize back the information. Thus, a drawback is the need of enough memory to store the entire document, which produces excessive consumption of system resources when only a portion of the document is to be processed [7].

In addition, tree-based APIs can use XPath (XML Path Language) for finding information. XPath is a language designed for addressing parts of an XML document [8]. Using a query language, it is possible to point to a desired tree node, and the API will walk through the tree until finding the data requested, minimizing the required programming effort for tree navigation.

2.2. Event-based API

The Simple API for XML (SAX) is another API for accessing XML documents, and it is the predominant API for event-based processing. This type of API reports parsing events directly to the application through callback methods. SAX is renowned for being less resource intensive than DOM, however, it is not convenient when storage or modification of data is needed [7, 9].

2.3. Available XML processors

A wide set of APIs like JDOM, JAXP, Xerces, or Xalan are available [4, 10–12]. Most of them offer support for both DOM and SAX. However, these kinds of processors are typically resource intensive and require megabytes of memory. Thus, small J2ME parsers fit better for limited devices such as NES, in which size and memory restrictions make unfeasible the use of "heavy" processors.

NanoXML, kXML, Xparse-J, ASXMLP, WoodStox, and TinyXML among others are examples of "light" XML-processors [13, 14]. Size varies between six to sixteen kilobytes, which makes them appropriate for small devices.

2.4. Limitations analysis

Both, "heavy and light" processors, still have the problem of memory fragmentation that results in garbage collections in environments such as Java or C#, which produces runtime overhead. No available parser could be found that has been designed considering efficient memory use in terms of predictable real-time performance. For that reason, an alternative solution is needed.

In the design of an XML processor that overcomes this limitation, focus is made on tree-based parsers, because despite of the fact that it requires more resources, the structured data representation provides a simple interface and offers possibility to modify information. For such kind of model representation, the use of XPath proves a powerful tool for easily processing a specific node, but current implementations introduce a further performance limitation. Typically, first a processor parses the XML document to build the data structure, and then starting from the root, XPath walks through the tree trying to find the data requested. With this mechanism the tree is passed several times in the worst case to reach each node. A solution using Xpath approach for finding information with just one pass is therefore desirable.

3. EXDOM DESIGN

J2ME platform provides the necessary tools to build a portable and light solution for handling XML data. The object oriented paradigm has been used in the design and development of EXDOM in order to provide modularity.

According to Cheng [15], a set of optimization practices such as class merging, elimination of variables, or method *Inlining* reduce either the code size or Heap usage. Reduction of code size decreases the total amount of bytes that the program occupies on memory, while reduction of heap usage

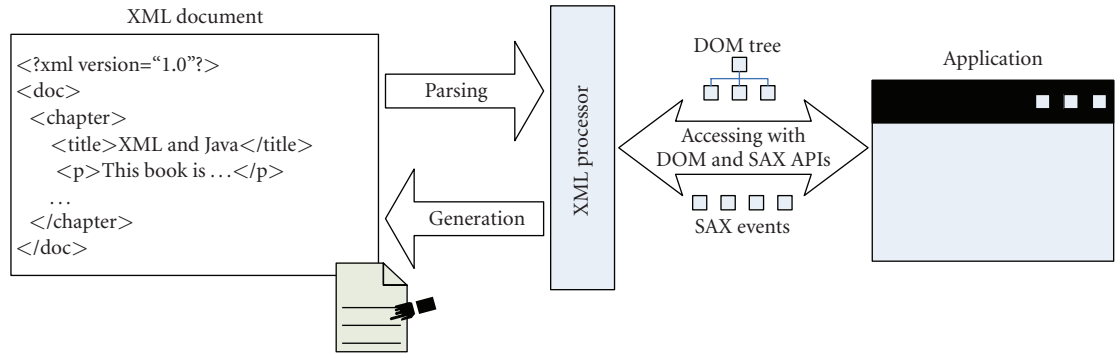

Figure 1: Overview of an XML processor.

implies more availability for (dynamic) memory allocation for other tasks. In order to adopt the advantage of class merging and method *Inlining* benefits, the lexical and syntactic analysis processes of XML parsers have been merged in the design of EXDOM.

When scanning the input document, a deterministic state machine defines the state in which the parser is. Then, valid inputs produce changes on current parser state. If there is a scenario in which current parser state does not support the current input token, an error is reported. This finite automaton ensures that all the elements, attributes, and other XML tokens are correctly nested in the document. Thus, XML well-formedness is verified. However, validation is not provided, that is, no semantic analysis against a Schema is performed. Validation is time-consuming and mostly not necessary for NES applications working in a well-defined environment: messages can be validated during design and application commissioning in order to guarantee proper structure in order to avoid repeated and time-consuming validation at run time.

The design approach is based on memory reuse instead of dynamic allocation and deallocation of objects. Thus, a one-instance policy is applied. Objects are allocated in constructor methods and treated as private variables that will be reused during the program lifecycle. Appropriate reinitialization is needed every time the parser is launched. Therefore, this approach suggests the use of a fixed amount of memory being used while data is processed. Under this theory, garbage collections may be avoided.

3.1. Design constraints

The reuse of objects introduces additional programming effort when compared to programming with dynamic de-/allocation of objects. One of the main problems in a Java environment is that Strings, Vectors, Stacks, and other classes, as well as the majority of standard libraries, use dynamic memory allocation at runtime. Thus, reimplementation of basic data structures is needed. One of the characteristics of Strings is that they are immutable while by contrast, it is possible to change the content of arrays [16]. That is to say, Strings cannot change their content once created, but their reference can be assigned to a new value and the old one will be left as garbage to be collected by garbage collector. However, arrays, once created can change their content without allocating new memory. Thus, a String equivalent is implemented using an array of bytes, and is called ByteString on EXDOM.

A Stack of ByteStrings is also needed for use in parser content and navigation functionalities. It provides basic functionalities like push(), pop(), isEmpty(). The stack must allocate memory and be initialized also from the very beginning in constructors. As a result, the stack is also able to provide other functionalities, like the possibility to change the content of a particular position on the stack.

3.2. Memory pools

A key point on reuse of memory is the use of *object pools*. An object pool contains a set of preallocated objects which are used and reused at runtime without need for dynamic allocation. The approach avoids the use of the operator *new* and therefore no new memory is allocated, but instead objects are reused after they are returned to the pool. Preallocation of a number of XML tree nodes in a memory pool, which is passed as a parameter to the parser, allows each application to determine the maximum amount of nodes needed. On real-time systems working on a well-defined context with prespecified messages, a memory pool behaves more efficiently than the dynamic allocation counterpart.

3.3. Iterative tree navigation

A common solution when walking a tree is to use recursion. Although elegant, the recursive mechanism can be substituted for a more efficient iterative solution. As a consequence, the EXDOM approach is based on a reference pointer moving through the tree. Despite increasing the code sophistication, it avoids recursion drawbacks by saving processor time, and memory and stack space.

3.4. Node structure

EXDOM is a DOM-like parser since it does not conform entirely to the DOM specifications. One of the main differences is the structure of the nodes. DOM specifications

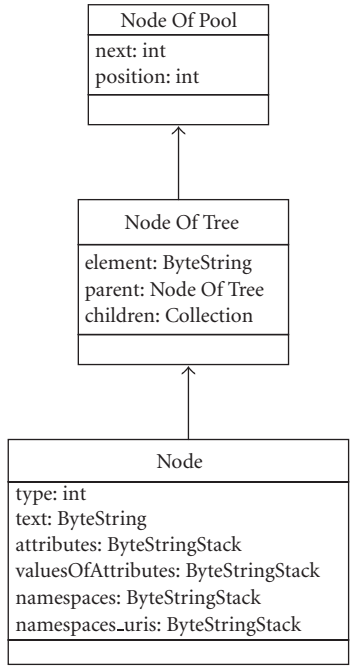

Figure 2: EXDOM Node structure

stipulate twelve node types [17]. However, in order to increment parsing performance, EXDOM provides two different node types: *Document* and *Element*. Attributes, text, entity references, and CDATA sections are contained within a single node, instead of being treated as multiple nodes. Also Namespaces defined under a particular element are stored on their correspondent node.

The design of EXDOM nodes has been done using the object-oriented *inheritance* concept, also known as the "is-a" concept. Every derived class (or inherited class) is a clone of its base class (or parent class), but the inherited class adds more functionality and can modify the clone [18]. The main advantage in this design decision is that inheritance permits to adapt the parser for application of specific requirements with minimum reimplementation.

Therefore, and according to Figure 2, a "Node" is-a "Node Of Tree" which is-a "Node Of Pool." The class "Node Of Pool" implements the concept of object pool explained before. It is implemented as a linked list for optimizing fast access to the first available node. This optimization takes in account that deletion of nodes is not needed when parsing.

The class "Node Of Tree" implements a generic node in a tree which has a link to its parent and to a collection of children. It also includes basic data as the name of the node stored in the *element* variable. Its purpose is to contain the name of the tag element on XML.

The class "Node" extends "Node Of Tree" and implements our concept of node:

(i) *type* that can be either "Document" if it is the root node of the tree, or "Element" if it is every other node,

(ii) *text* that stores text associated to the node.

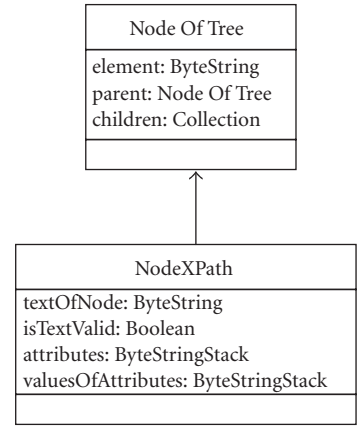

Figure 3: XPath node structure.

Two symmetric stacks store attributes associated to the node and their attribute value, respectively:

(i) *attributes*,
(ii) *valuesOfAttributes*.

Finally, two other symmetric stacks store the list of namespace prefixes declared on the node and their unique resource identifier (URI) value, respectively:

(i) *namespaces*,
(ii) *namespaces_uris*.

3.5. XPath solution

The second possibility that EXDOM offers for parsing is a novel approach: the guidance of the parsing process. That is to say, to offer the possibility to add expected/known paths before parsing and retrieve directly the expected data at the same time of parsing, instead of parsing and then searching for the data. The scope of this approach is applicable in well-defined contexts in which a user already knows in advance the path for finding an attribute value or a text under a node.

A first step is to create a parallel tree with all paths provided. This tree is intended to mirror the tree that will be constructed during parsing, but with special marks in those nodes where values will be retrieved. Then, as the EXDOM tree is being created, the marked nodes are filled when a match between the trees is detected. If data was not found after parsing, marks will remain empty.

This solution offers a significant optimization since there is no need to walk from the root through the tree for each path after the document is parsed. A reference marking the last position in the XPath tree in which it is still possible to find a marked attribute or text under it avoids starting from the root each time.

The second tree incurs in a low memory cost, since nodes do not store values of attributes, but references to the ones on the main tree. The same applies for text. This can be explained by the XPath nodes design shown in Figure 3.

Again inheritance has been used. Structure of XPath node includes

(i) IsTextValid: mark for text in the node;

```
(A) When E_i is opened
        . . .
    Insert E_i on T
    . . .
    If (∃ child of R = E_i), then
            R ⟵ child of R = E_i
     If (∃ m_i on R), then
read values of attributes and set references to them
     Else
        read values of attributes
        . . .
(B) When E_i is closed
        . . .
        E_i ⟵ E_i-1
    . . .
        If (E_i ≠ R)
    R ⟵ parent of R
        . . .
```

Algorithm 1

(ii) textOfNode: a reference to the text on the main tree, or null if IsTextValid is false;
(iii) attributes: a set of attributes expected to find marks for attributes;
(iv) valuesOfAttributes: references to attribute values or null if they were not found.

As mentioned before, while parsing, a reference on XPath tree is moving in order to create references to expected data. Naming the XPath tree XT, marked positions m_i, the node to which XPath reference is pointing R, the main tree T, and an elementtag E_i, then Algorithm 1 holds.

3.6. Other current restrictions

EXDOM does not comply with the document type declaration as well as its related features. It recognizes references to XML Schemas but not to DTDs.

4. EXPERIMENTAL TEST AND RESULTS

EXDOM has been benchmarked against Xerces and Xparse-J 1.1 [19, 20].

XParse-J 1.1 is a small DOM-like parser developed in J2ME technology, being a very light parser with only 6 KB [19]. It provides similar functionality than EXDOM and also offers possibility to find information using XPath notation, therefore it appears as the most similar alternative. Xerces is a DOM parser, widely used in desktop and server environments, and is chosen to illustrate the difference in memory use scale.

Tests have been made for parsing XML files which generate symmetric trees with node counts of 27, 53, 105, 209, 417, and so on. Comparative results between parsers are made in a PC Intel Celeron 2.97 GHz and 760 MB of RAM running the operating system Microsoft Windows XP Professional version 2002 with Service Pack 2.

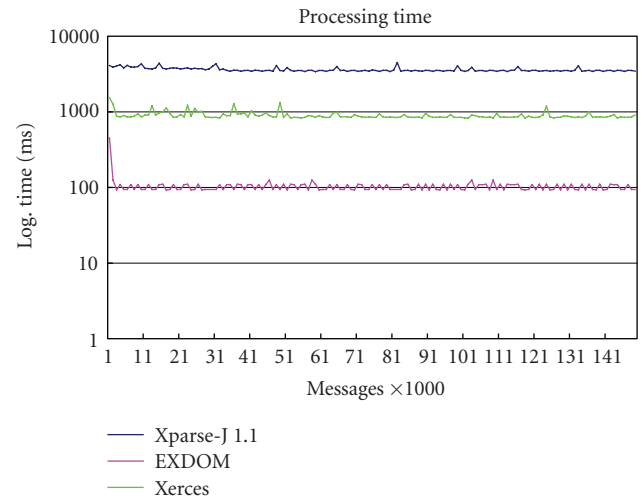

Figure 4: Execution time results.

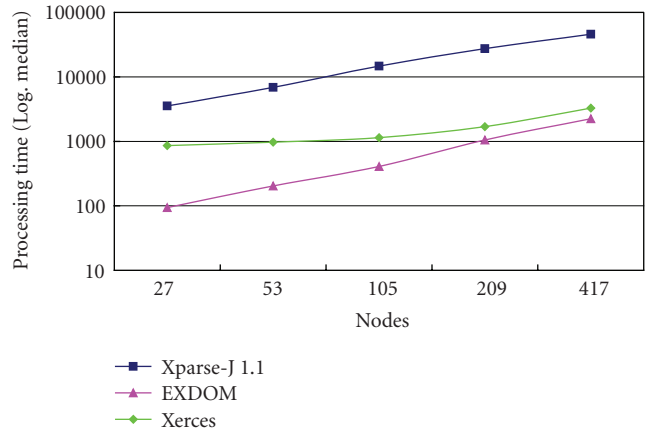

Figure 5: Execution time versus file size results.

4.1. Processing time

In order to minimize the impact of measurements in obtained values, statistics are performed for 1000 parsing operation of messages with a size of 2 KB each. The results are illustrated in Figure 4.

The observations indicate that EXDOM shows a significantly better performance on execution time for small XML documents, which are typically found in NES environments.

4.2. Execution time versus file size

Figure 5 illustrates that the results obtained alter measuring of the execution time of 1000 parsing operations for messages of different size. The message size is quantified according to the number of XML elements.

In this case, the results show that Xparse is growing exponentially with the increase of nodes and therefore file size, Xerces, and EXDOM show better performance. Even though EXDOM performs significantly better than Xerces for small files (less than 50 nodes), it can be seen that the performance becomes similar for larger XML documents

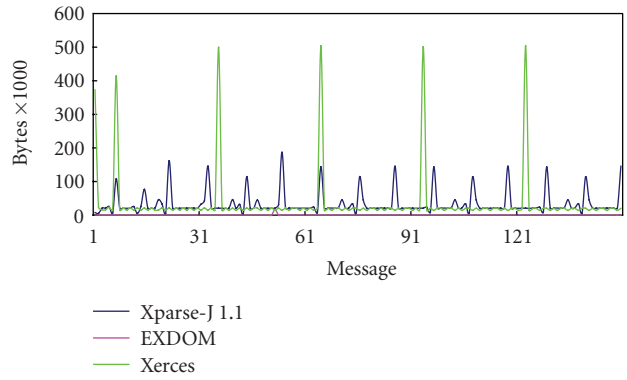

Figure 6: Observations on the variation of free memory in the JVM.

Table 1: Statistics from Figure 6.

	EXDOM	Xerces	Xparse-J
Maximum	16 876	50 3992	18 6872
Minimum	0	14 192	4068
Range	16 876	48 9800	18 2804
Average	111.76	35787.7	33134.8
Median	0	20 200	20 764
Standard deviation	1373.35	88 450	35 297
Variance	1.88e6	7.82e9	1.24e9

(100 to 500 nodes). Analyzing the trend and due to the exponential complexity of EXDOM, it can be expected that Xerces performs better than EXDOM when the document size is large (more than 1000 nodes), indicating that Xerces is optimized for manipulation of large data quantities. However, EXDOM is still an order of magnitude faster for small messages (less than 27 nodes), which are typical of real-time factory automation applications such as CAMX [2].

4.3. Memory usage

Figure 6 illustrates the variation in amount of free memory in the Java Virtual Machine (JVM). A sudden increment on free memory implies the action of the garbage collector.

While EXDOM maintains a constant amount of memory used, avoiding garbage collections, the other parsers cause variations in memory availability. Such variations cause indeterminism in process timing.

Table 1 shows various statistics obtained from Figure 6 which clarifies differences among parsers.

4.4. Analysis

The better performance of EXDOM for processing time can be attributed only partially to an optimized implementation of the parsing mechanism. The timing metric includes not only the parsing time but the time needed for memory allocation and for garbage collection, which significantly increases the averages over 1000 messages. Thus, careful use of memory improves both response time and determinism.

5. CONCLUSIONS AND FUTURE WORK

EXDOM has been introduced as a solution to XML processing in environments that provide limited memory and computing power, and have the added challenge of requiring predictable real-time response. Among the design highlights of the approach are the pooling and reuse of objects, node value retrieval with a single tree navigation operation, and programming optimization with method *Inlining*.

Among future research directions are the application of similar or new methods in order to improve the performance and predictability of XML document serialization. In addition, further research is needed in order to determine the appropriate methods to achieve full XML compliance whilst maintaining the performance characteristics obtained thus far.

REFERENCES

[1] OPC Foundation, "OPC UA Part 1—Concepts 1.00 Specification," July 28, 2006.
[2] A. Dugenske, A. Fraser, T. Nguyen, and R. Voitus, "The national electronics manufacturing initiative (NEMI) plug and play factory project," *International Journal of Computer Integrated Manufacturing*, vol. 13, no. 3, pp. 225–244, 2000.
[3] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1, pp. 62–70, 2005.
[4] H. Maruyama, K. Tamura, and N. Uramoto, *XML and Java: Developing Web Applications*, Addison Wesley, Upper Saddle River, NJ, USA, 2002.
[5] "Document Object Model (DOM) specifications," http://www.w3.org/DOM/.
[6] "Document Object model Core," 2004, http://www.w3.org/TR/DOM-Level-3-Core/core.html.
[7] XML tutorial, "Introduction to XML and XML With Java," http://totheriver.com/learn/xml/xmltutorial.html.
[8] "XML path language (XPath)," http://www.w3.org/TR/xpath.
[9] Simple API for XML (SAX), http://www.saxproject.org/.
[10] Java Web Services, "Java API for XML Processing (JAXP)," http://java.sun.com/webservices/jaxp/.
[11] "The Apache XML Project," http://xml.apache.org/.
[12] "JDOM," http://www.jdom.org/.
[13] J. Knudsen, "Parsing XML in J2ME," 2002, http://developers.com/techtopics/mobility/midp/articles/parsingxml/.
[14] "Java ME Open Source Software," http://ngphone.com/j2me/opensource/xml.htm.
[15] S. Cheng, "Squeezing the last byte and Last Ounce of Performance on your MIDLETS," http://developers.sun.com/learning/javaoneonline/2006/mobility/TS-3418.pdf.
[16] Open University course M254, "Java everywhere," The Open University, 2005.
[17] XML DOM Node Types. http://www.w3schools.com/dom/dom_nodetype.asp.
[18] B. Eckel, *Thinking in Java*, Prentice-Hall, Santa Barbara, Calif, USA, 3rd edition, 2003.
[19] M. Claben, "Xparse-J 1.0 User documentation," http://www.webreference.com/xml/tools/xparse-j.html.
[20] The Apache XML project, "Xerces2 Java Parser 2.9.0 Release," http://xerces.apache.org/xerces2-j/.