An Automated Approach for Verification of Software Requirements[⋆]

Amador Durán, Antonio Ruiz, and Miguel Toro

Departamento de Lenguajes y Sistemas Informáticos, Facultad de Informática y Estadística, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012 Sevilla, España
`{amador,aruiz,mtoro}@lsi.us.es`

Abstract *In this paper, we present an automated approach for the verification of software requirements. This approach is based on the representation of software requirements in XML and the usage of the XSLT language to automatically verify some desired quality properties. These ideas have been implemented in* REM*, an experimental requirements management tool that is also described in this paper.*

1 Introduction

Paraphrasing Boehm [5], requirements validation and verification can be informally defined by the questions "*Am I building the right requirements?*" (validation) and "*Am I building the requirements right?*" (verification).

In other words, the goal of requirements validation is to ensure that requirements documents contain *actual* requirements and that these requirements are *all* the known requirements by the time the requirements documents are baselined.

On the other hand, the goal of requirements verification is to ensure the quality of requirements according to desired quality properties. Some of these quality properties have to do with requirements semantics but others have to do with syntactic, structural or pragmatic aspects of requirements (see [12] for a complete classification of quality properties of requirements).

Verification of semantic properties of requirements is closely related to requirements validation (distinction between requirements verification of semantic properties and requirements validation is sometimes subtle and many authors use both terms interchangeably) and requires human participation, whereas verification of non–semantic properties should be as automated as possible.

In this article, we present an automated approach for the verification of some quality properties of requirements. Most of these properties can be classified as non–semantic, but we have also developed some heuristics to check potential problems with some semantic properties. Our approach is based on the emergent technology built around XML [4] and its companion language XSLT [3].

The rest of the article is organized as follows. First, we briefly describe the basics of XML and XSLT needed to understand the following sections. Then, we describe REM,

[⋆] This work is partially funded by the Spanish CICYT project *GEOZOCO* TIC 2000–1106–C02–01 and by the international CYTED project *WEST*.

an experimental requirements management tool [8, 9], the XML model of requirements used by REM and how XSLT can be used to verify some quality properties of requirements expressed in XML. Finally, we discuss some related work, present some results and point out future work.

2 XML and XSLT

2.1 XML Basics

There are millions of web pages written in HTML available in Internet. In these web pages, pure information is mixed with formatting elements, making the automatic processing of information very difficult. XML [4] is a language designed for representing pure information in Internet. Information in XML is represented by *elements*. An XML element is made up of a start tag, an end tag, and other tags or data in between. For example, for representing the information about a book, we might have the following XML element named book:

```
<book isbn="1-234-56789-0">
  <author>Miguel de Cervantes</author>
  <title>El Quijote</title>
</book>
```

As you can see, the information about a book is between the <book> and </book> tags and it is easy to parse by a computer program. The author and title elements are considered as children of the book element, thus forming a hierarchy. An XML document must always have one and only one *root element* at the top of its hierarchy.

In order to allow information interchange between two or more parties using XML, they must agree about element grammar and semantics. Element grammar is specified as regular expressions in DTDs (Document Type Definitions) [4]. For example, the DTD fragment for the previous XML data would be the following:

```
<!ELEMENT book (author+,title)>
  <!ATTLIST book isbn ID #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title  (#PCDATA)>
```

where it is stated that a book element can contain one o more author elements and only one title element. An XML element can also have attributes. For example, isbn is defined as a required identification attribute of book, *i.e.* there cannot exist two books with the same value for the isbn attribute in the same XML document. Those elements that contain only text are said to contain #PCDATA, that stands for *parsed character data*.

2.2 Transforming XML

There are many situations in which XML data need to be transformed. For example, for presenting XML data as an HTML page. XSLT [3] is a language based on transformation patterns. An XSLT stylesheet, which is also a an XML document, searches for patterns in the XML data and applies programmed transformations, thus generating some output results. For example, if we wanted to show information about books in a web browser, we could apply the following XSLT transformation rule:

```
<xsl:template match="book">
    <B><xsl:value-of select="title"/></B>
    (ISBN <xsl:value-of select="@isbn"/>)
    was written by
    <EM><xsl:value-of select="author[1]"/></EM>
</xsl:template>
```

The informal semantics of this XSLT rule are "when you find a book element, generate its title in boldface, then its ISBN attribute (notice the @ prefix for attributes), and then its first author in emphasized mode". In the XSLT code, text literals like HTML tags can be mixed with element values, which are obtained by means of the xsl:value-of statement. If we applied this XSLT rule to the previous XML data, the result of the transformation would be something like this when rendered in a web browser:

El Quijote (ISBN 1-234-56789-0) was written by *Miguel de Cervantes*

Although there are many more details about XML and XSLT, we think that this brief introduction should be enough for those readers not familiar with XML technologies in order to understand the rest of this article.

3 REM: An XML–based Requirements Management Tool

REM (*REquirements Manager*) is an experimental requirements management tool developed by one of the authors [8, 9]. In REM, a requirements engineering (RE) project is considered to be composed by three documents:

1. a customer–oriented requirements document (the *requirements document* [13]), usually containing requirements in natural language expressed in terms of customer's vocabulary, also known as *C–requirements* [7].
2. a developer–oriented requirements document (the *specification document* [13]), usually containing requirements models and more technical information, also called *D–requirements* [7].
3. a registry for detected conflicts and negotiation support.

In REM, C–requirements and conflicts are expressed in natural language using predefined requirements templates and some linguistic patterns (see [9] for details). For expressing D–requirements, we have chosen a subset of the UML [6]

3.1 REM Architecture

REM documents, *i.e.* RE projects composed by the three documents previously described, are stored in relational light–weight databases. When the user creates a new REM document, the basic structure is taken from a *REM base document* (see figure 1),

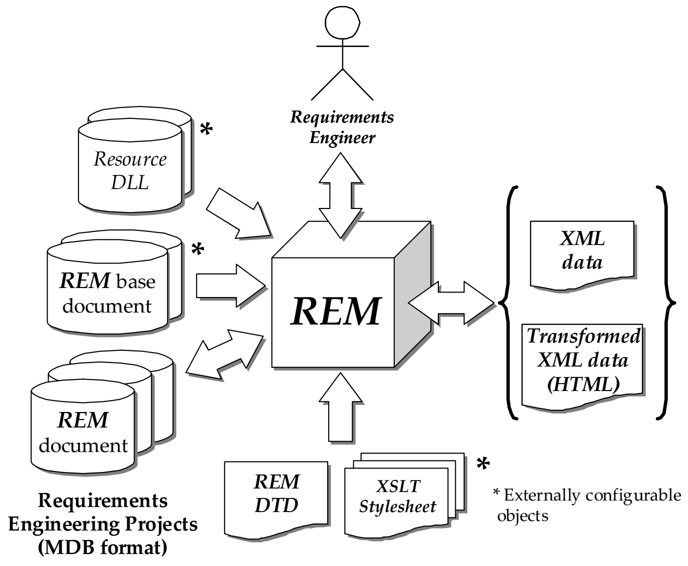

Requirements Engineer
Resource DLL
*
REM base document
*
REM document
Requirements Engineering Projects (MDB format)
REM
XML data
Transformed XML data (HTML)
REM DTD
XSLT Stylesheet
*
* Externally configurable objects

Figure1. REM Architecture

that can be empty or can contain the mandatory sections of software requirements standards like [1] or [15]. Any ordinary REM document can be selected as a base document, so users can create their own base documents or reuse other REM documents.

In order to provide immediate feedback on user actions, REM generates XML data corresponding to the document being edited, applies an external XSLT stylesheet that transforms XML data into HTML and shows the resulting HTML to the user. In this way, whenever the user changes a requirements document, he or she can see the effects immediately.

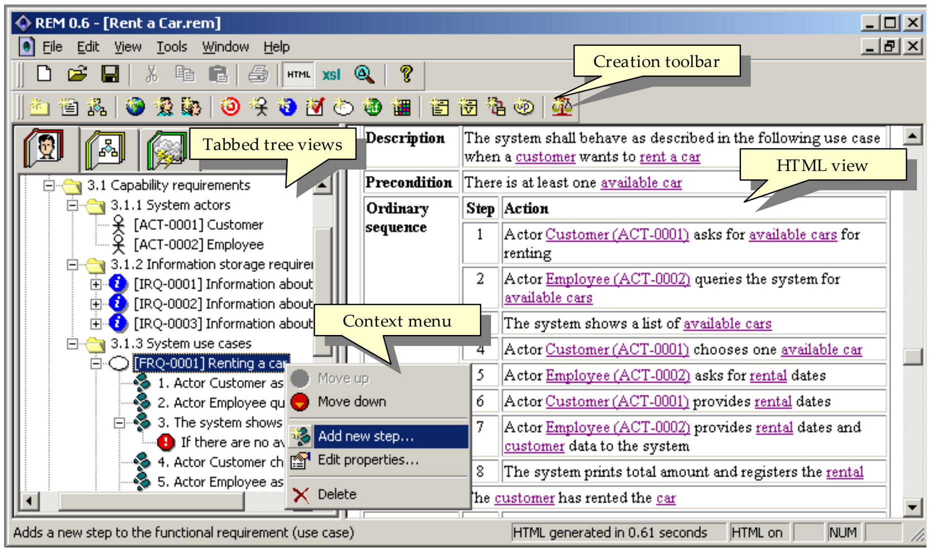
In the same way the REM base document can be customized, the user can also change document appearance by selecting or creating different external XSTL stylesheets. The default XSLT stylesheet generates a highly hyperlinked document, making navigation of requirements documents easier (see right side of figure 2).

Other configurable aspect of REM is the language of the user interface. The user can choose it by selecting an external resource dynamic link library (DLL). At this moments, we have developed two external resource DLLs for REM, one in Spanish and one in English.

3.2 REM User Interface

The user interface of REM presents two different views to the user (see figure 2). On the left, the user can see a tabbed view with three tree views, one for each requirements document in the RE project. On the right hand, the result of the XSLT transformation of the XML data is presented to the user in a embedded web browser.

In any of the three tree views, the user can directly manipulate objects by drag and drop or by context menus. Only actions that have sense can be performed, following



3.1 Capability requirements
3.1.1 System actors
[ACT-0001] Customer
[ACT-0002] Employee
3.1.2 Information storage require
[IRQ-0001] Information about
[IRQ-0002] Information about
[IRQ-0003] Information about
3.1.3 System use cases
[FRQ-0001] Renting a car
1. Actor Customer as
2. Actor Employee qu
3. The system shows
If there are no av
4. Actor Customer ch
5. Actor Employee as

Description	The system shall behave as described in the following use case when a customer wants to rent a car	
Precondition	There is at least one available car	
Ordinary sequence	**Step**	**Action**
	1	Actor Customer (ACT-0001) asks for available cars for renting
	2	Actor Employee (ACT-0002) queries the system for available cars
		The system shows a list of available cars
	4	Actor Customer (ACT-0001) chooses one available car
	5	Actor Employee (ACT-0002) asks for rental dates
	6	Actor Customer (ACT-0001) provides rental dates
	7	Actor Employee (ACT-0002) provides rental dates and customer data to the system
	8	The system prints total amount and registers the rental
		he customer has rented the car

Adds a new step to the functional requirement (use case) HTML generated in 0.61 seconds HTML on NUM

Figure2. REM User Interface

a *correct–by–construction* approach, thus increasing quality and avoiding verification effort.

For example, actions of use case steps can be of three different classes (see figure 4): *actor action*, if the action is performed by an actor; *system action* if the action is performed by the system, or *use case action*, if the action consists of performing other use case, *i.e.* an use case *inclusion* or *extension* [6]. Actor actions and use case actions can be created only if some actor or some use case have been previously created. In general, objects can be created by using context menus on potential parents or by using the creation toolbar.

4 XML Model of Requirements in REM

REM is based on an UML [6] model of requirements (a partial view of this model is shown in figure 4). The main object class of the model is the *Requirements Document*, that is composed by a sequence of REM objects (see figure 3).

We have translated our UML model of requirements into a relational schema and into a DTD. As an example, the UseCase class in figure 4 has been translated into the following DTD element definition:

```
<!ELEMENT rem:useCase (
  rem:name, rem:version, rem:authors?, rem:sources?, rem:comments?,
  rem:importance, rem:urgency, rem:status, rem:stability,
  rem:isAbstract?, rem:triggeringEvent,
  rem:precondition, rem:postcondition,
  rem:frequency, rem:step* )>
<!ATTLIST rem:useCase oid ID #REQUIRED>
```

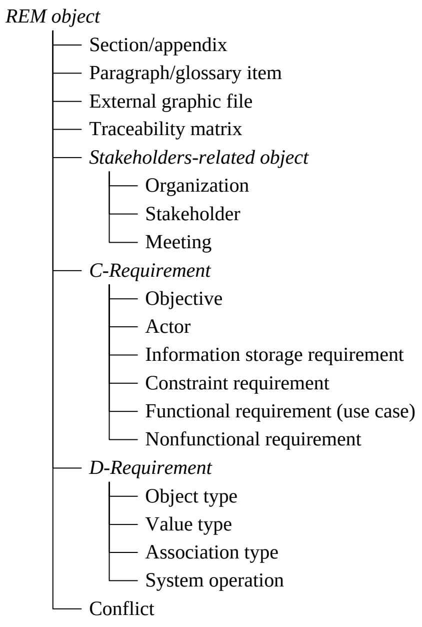


Figure3. Classification of objects in REM

Many of the elements in the previous DTD fragment (comments, triggeringEevent, pre and postcondition), contains only text, *i.e.* natural language. In REM, text can be composed by any combination of free text, references to other objects and TBD (*To Be Determined*) marks, defined as follows:

```
<!ELEMENT rem:text (#PCDATA|rem:ref|rem:tbd)*>
<!ELEMENT rem:ref (#PCDATA)>
    <!ATTLIST rem:ref oid IDREF #REQUIRED>
<!ELEMENT rem:tbd EMPTY>
```

where the rem:ref element must have a required attribute called oid that it is declared as an IDREF, *i.e.* a reference to other element with a matching identification attribute value. An IDREF attribute is very similar to a *foreign key* in relational databases.

The rem:tbd element is declared as an EMPTY element, *i.e.* it cannot have neither subordinate elements nor data. It is simply a mark.

The DTD elements corresponding to use case steps and actions of figure 4 have been described as follows:

```
<!ELEMENT rem:step (
  rem:number, rem:condition?,
  ( rem:systemAction | rem:actorAction | rem:useCaseAction ),
  (rem:stepException*),
  rem:comments )>
<!ATTLIST rem:step oid ID #REQUIRED>
```

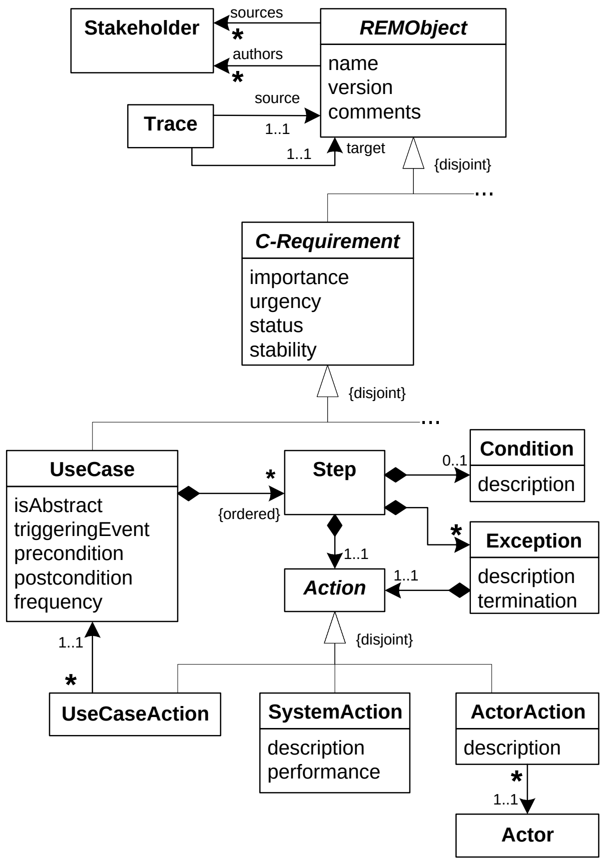


Figure4. UML model of use cases in REM

```
<!ELEMENT rem:systemAction (
  rem:description, rem:performance? )>

<!ELEMENT rem:actorAction (rem:description)>
    <!ATTLIST rem:actorAction actor IDREF #REQUIRED>

<!ELEMENT rem:useCaseAction EMPTY>
    <!ATTLIST rem:useCaseAction useCase IDREF #REQUIRED>

<!ELEMENT rem:stepException (
  rem:condition,
  ( rem:systemAction | rem:actorAction | rem:useCaseAction ),
  rem:termination,
  rem:comments )>
<!ATTLIST rem:stepException oid ID #REQUIRED>
```

Elements not defined in the previous DTD code (condition, description, termination, etc.) are defined as containing only text. For example:

```
<!ELEMENT rem:condition (#PCDATA|rem:ref|rem:tbd)*>
```

5 Using XSLT as a Requirements Verification Language

In the following sections we describe how some of the quality factors described in [10] can be automatically verified using XSLT when requirements are electronically stored in XML format according to the REM DTD.

5.1 Unambiguity

A requirement is unambiguous if and only if has only one possible interpretation [1]. This is obviously a semantic property of a requirement and cannot be verified automatically, but we can give some *hints* about potential ambiguities in a requirements document.

We agree with Leite [11] in the importance of understanding the language of the problem and in the importance of building a glossary (called *Language Extended Lexicon*, LEL, in [11]). Following Leite, the glossary should follow two principles: the *principle of circularity*, (the glossary must be as self–contained as possible) and the *principle of minimal vocabulary* (use as much glossary items as possible in your requirements descriptions). Leite's principles cannot guarantee unambiguity, but they can help to build unambiguous, understandable, verifiable, consistent, concise, and cross–referenced requirements [10].

XSLT can be used to measure glossary circularity (GLC) and minimality of vocabulary (MOV). GLC can be measured as the ratio between glossary items and references to glossary items from other glossary items. The following XSLT code, where we have declared two variables for the sake of readability, can be used for this purpose:

```
<xsl:variable name="GLO"
  select="count(//rem:glossaryItem)"/>
<xsl:variable name="REF"
  select="count(//rem:glossaryItem//rem:ref)"/>
<xsl:value-of
  select="format-number($REF div $GLO, '#0.00')"/>
```

where the expression //rem:glossaryItem is an XPath expression [2] meaning "any rem:glossaryItem element descendant of the root", whereas the expression //rem:glossaryItem//rem:ref means "any rem:ref element descendant of any rem:glossaryItem descendant of the root". In XPath, the language for building navigation expressions over XML trees, an element is considered as descendant of other element if it is its child at any level of depth in the hierarchy.

A similar ratio between the number of references to glossary items in requirements and the number of requirements can be used to measure MOV. From the MOV viewpoint, it is also possible to detect those "suspicious" requirements that do not have any reference to any glossary item in their text. Since those requirements are not using the vocabulary of the customer, they should be checked for potential problems of ambiguity or understandability [10]. For example, if we want to know what use cases are "suspicious", we can use the following XSLT code:

```
<xsl:template match="rem:useCase[not(.//rem:ref)]"/>
  Use case
  <xsl:value-of select="rem:name"/>
  does not use any glossary item
</xsl:template>
```

where the match expression uses brackets to select only those use cases with no descendant references. Another possibility is to determine a threshold value for the number of references per requirement and consider as suspicious all requirements with a number of references under the threshold. In that case, the match expression would be rem:useCase[count(.//rem:ref) < m], with m being the MOV threshold.

5.2 Completeness

A requirements document is complete if it includes [10]:

1. Everything that the software is supposed to do, *i.e. all* the requirements
2. Responses of the software to all classes of input data in all realizable situations
3. Page numbers, figure and table names and references, a glossary, units of measure and referenced material
4. No sections marked as TBD

In our approach, the third completeness condition is partially satisfied by means of the *correct–by–construction* paradigm of REM: figure and table names are automatically generated, references are automatically inserted and updated, and the user can easily create a glossary. If we want to be sure about the existence of a section named Glossary, we can apply the following XSLT code:

```
<xsl:choose>
  <xsl:when test="//rem:section[rem:name='Glossary']"/>
    There is a glossary
  </xsl:when>
  <xsl:otherwise>
    There is no glossary
  </xsl:otherwise>
</xsl:choose>
```

where the structure formed by xsl:choose, xsl:when and xsl:otherwise is basically an if–else–endif statement with multiple else branches. Notice that if we want to check the existence of an element we cannot use an XSLT template. If there is no such an element, the template will never match and we will have no output.

Similar XSLT code can be used to verify if requirements documents are *organized* [10], *i.e.* if they have mandatory sections in the mandatory order with mandatory content.

The fourth condition of completeness, the absence of TBD marks, can be easily verified using XSLT. If we want to know how many TBD marks are in a requirements document we can apply the following XLST code:

```
There are
<xsl:value-of select="count(//rem:tbd)"/>
TBD marks
```

that would generate in the output the number of occurrences of elements of type rem:tbd anywhere in the XML data. If we want to be more precise and we want to know what use cases have TBD marks inside their text and how many TBD marks they have, we could write the following XSLT code:

```
<xsl:template match="rem:useCase[.//rem:tbd]"/>
  Use case <xsl:value-of select="rem:name"/>
  has <xsl:value-of select="count(.//rem:tbd)"/>
  TBD marks
</xsl:template>
```

in which the select expression "rem:useCase[.//rem:tbd]" means "any use case with at least one descendant of type rem:tbd".

5.3 Traceability

In [10], a requirements document is said to be *traceable* if and only if it is written in a manner that facilitates the referencing of each individual requirement. Since REM assigns automatically an unique identifier to every requirement (the required identifier attribute oid, see the DTD for use cases), this quality factor does not have to be verified explicitly.

What it must be checked is if the origin of every requirement is clear, *i.e.* if requirements are *traced* [10]. In our UML model of requirements, any REM object can be traced to and from other REM objects and to their human sources and authors (see figure 4). Checking if a requirement has sources and authors and if it is traced to or from other requirements is easy with XSLT. For example, the following XSLT template will match all use cases with no human sources:

```
<xsl:template match="rem:useCase[not(rem:sources)]">
  Use case
  <xsl:value-of select="rem:name"/>
  has no sources
</xsl:template>
```

And this XSLT template will match all non functional requirements not traced to other REM objects:

```
<xsl:template match="rem:nonFunctionalRequirement">
  <xsl:if test="not(//rem:trace[@source=current()/@oid])">
    Non functional requirement
    <xsl:value-of select="rem:name"/>
    is not traced to any object
  </xsl:if>
</xsl:template>
```

In REM, traces are defined as elements with two required attributes of type IDREF, namely source and target. The user of REM can also use traceability matrices for visual checking of non–traced requirements.

5.4 Other verifiable quality factors

Applying the same ideas, other quality factors defined in [10] can be verified using XSLT, for example:

- What requirements are not annotated with relative importance, relative stability or version.
- What requirements have potentially ambiguous words in their description, like *easy to*, *user–friendly*, etc. by means of XSLT string functions like contains [3].
- If use cases are not well structured, *i.e.* if there are too few or too many *includes* or *extends* relationships.
- What use cases have too few or too many steps, or too much exceptions, *i.e.* too many alternative courses.

6 Related Work

Most work on automated requirements verification is based on Natural Language Processsing (NLP), like [14] or [12]. Those approaches, focused on semantic analysis of requirements, usually make requirements engineers write requirements in a subset of natural language, demand many computer resources and have not been widely adopted in industry.

The Automated Requirement Measurement (ARM) tool [16], is probably the most related work to the approach presented in this article. It is a simple yet powerful tool that scans requirements documents searching for *indicators*, *i.e.* words that have been identified as indicators of good or bad quality properties.

Our approach does not use NLP but an open, simpler and lighter technology like XML/XSLT. We can offer the same functionality of ARM plus all additional verification described in this paper, and the user of REM can defined his or her own XSLT verification stylesheets. From a practical point of view, we think that our results are useful for the average requirements engineer.

7 Conclusions and Future Work

In this article we have briefly presented an automated approach for the verification of software requirements. Our approach is based on a open technology like XML and XSLT. In fact, if requirements are represented in XML using a different DTD, many of the XSLT code presented in this paper should be easily adapted. Our approach does not need hard computer resources and it has proved to be useful when used with our students at the University of Seville.

Our future work is focused in developing quality metrics, so we can detect potential problems with requirements comparing quantitative values. We expect to identify some useful metrics soon by applying data mining techniques to the requirements documents generated by our students.

References

[1] IEEE Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830–1998, Institute of Electrical and Electronics Engineers, 1998.

[2] XML Path Language (XPath) 1.0. W3C Recommendation, November 1999.

[3] XSL Transformations (XSLT) 1.0. W3C Recommendation, November 1999.

[4] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000.

[5] B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, 1984.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, 1999.

[7] J. W. Brackett. Software Requirements. Curriculum Module SEI–CM–19–1.2, Software Engineering Institute, Carnegie Mellon University, 1990.

[8] A. Durán. *A Methodological Framework for Requirements Engineering of Information Systems (in Spanish)*. PhD thesis, University of Seville, 2000.

[9] A. Durán, B. Bernárdez, A. Ruiz, and M. Toro. A Requirements Elicitation Approach Based in Templates and Patterns. In *WER'99 Proceedings*, Buenos Aires, 1999.

[10] A. Davis *et al.* Identifying and Measuring Quality in a Software Requirements Specification. In *Proceedings of the 1st International Software Metrics Symposium*, pages 141–152, 1993.

[11] J. C. S. P. Leite *et al.* Enhancing a Requirements Baseline with Scenarios. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, 1997.

[12] F. Fabbrini, M. Fusani, V. Gervasi, S. Gnesi, and S. Ruggieri. Achieving Quality in Natural Language Requirements. In *Proceedings of the 11 th International Software Quality Week*, 1998.

[13] B. L. Kovitz. *Practical Software Requirements: A Manual of Content & Style*. Manning, 1998.

[14] N. A. Maiden, M. Cisse, H. Perez, and D. Manuel. CREWS Validation Frames: Patterns for Validating Systems Requirements. In *Fourth International Workshop on Requirements Engineering: Foundation for Software Quality (RESFQ)*, 1998.

[15] C. Mazza, J. Fairclough, B. Melton, D. de Pablo, A. Scheffer, and R. Stevens. *Software Engineering Standards*. Prentice–Hall, 1994.

[16] L. Rosenberg, T. Hammer, and J. Shaw. Software Metrics and Reliability. In *9th International Symposium on Software Reliability Engineering*, 1998.