Supporting Requirements Verification Using XSLT[*]

Amador Durán Antonio Ruiz–Cortés Rafael Corchuelo Miguel Toro
Dpto. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
{amador,aruiz,corchu,mtoro}@lsi.us.es

Abstract

In this paper we present a light-weight approach for the automatic verification of requirements. This approach is not based on natural language parsing techniques but on the representation of requirements in XML. In our approach, XSLT stylesheets are used not only to automatically generate requirements documents, but also to provide verification–oriented heuristics as well as to measure the quality of requirements using some verification–oriented metrics. These ideas have been implemented in REM, an experimental XML–based requirements management tool also described in this paper.

1. Introduction

It is widely acknowledged within the software community that requirements quality is one of the most important factors in the quality and overall success of software projects. That is the reason why some requirements engineering (RE) activities have as their main goal increasing the quality of requirements, namely *requirements analysis*, *requirements verification* and *requirements validation* (see the UML activity diagram in figure 1).

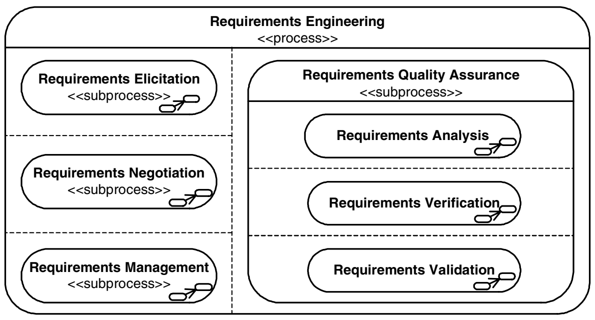


Figure 1. RE process model

[*]This work is partially supported by the CICYT project TIC 2000–1106–C02–01 (GEOZOCO) and by the CYTED project VII.18 (WEST)

These quality–oriented activities act as a *requirements quality filter* in a similar way the *quality gateway* does in the Volere method [16]. The main goals for these activities are summarized in table 1.

Table 1. Quality–oriented RE activities

Activity	Main goal
Analysis	Identify conflicts in requirements
Verification	Detect defects in requirements
Validation	Certify requirements are consistent with the intentions of customers and users

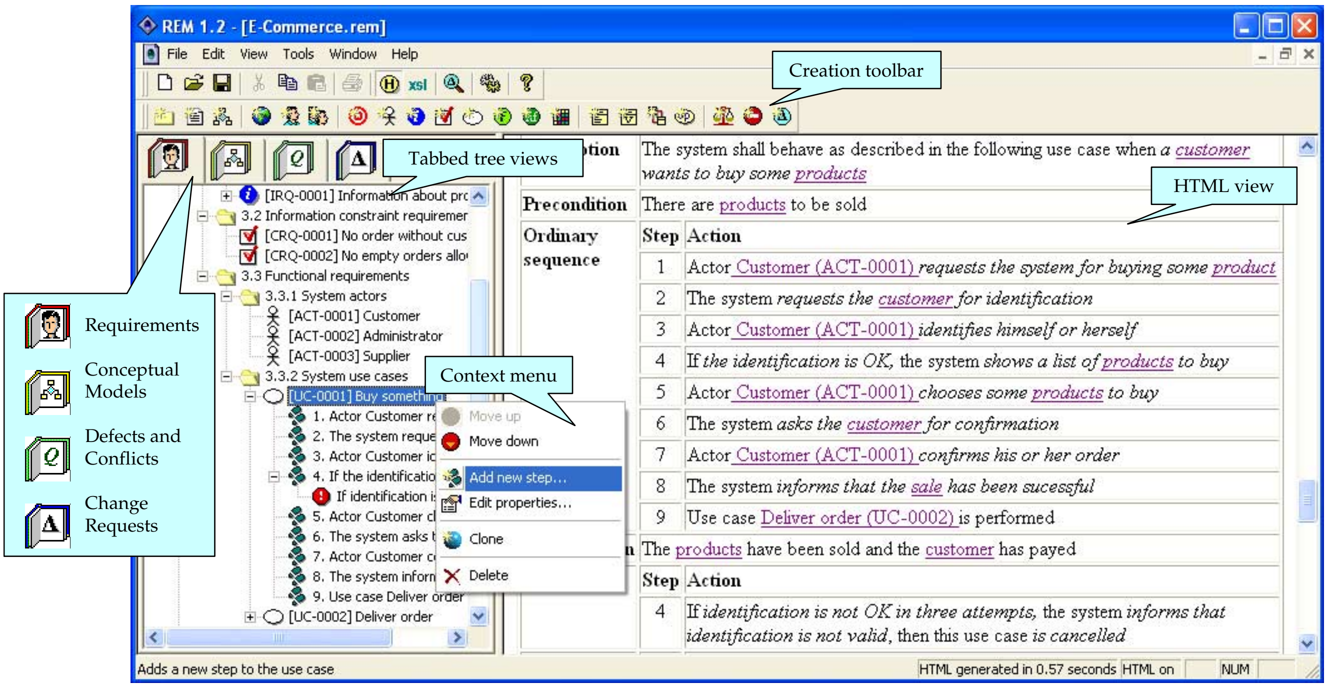
In this paper we focus on automatic verification of so–called *specification errors* [8] and in assisting the requirements verifier in detecting *knowledge errors* [8] by means of some verification–oriented heuristics and requirements metrics that can be easily computed. Our approach is based on the emergent technology built around XML [22] and its companion language XSLT [21], as initially described in [7].

The rest of the paper is organized as follows. First, we describe REM, our RE tool [5, 6]. Then, we present a brief overview of the XML model of requirements used by REM and how XSLT can be used to support the verification of some quality properties of requirements expressed in XML. Finally, we discuss some related work and point out conclusions and future work.

2. REM: an XML–based requirements tool

REM (*REquirements Manager*) is an experimental RE tool developed by one of the authors as part of his PhD. Thesis [5]. A REM project is considered to be composed of the four documents corresponding to the four tabbed tree views in figure 2. In REM, requirements, conflicts and defects are expressed in natural language using predefined templates and some linguistic patterns (see [6] for details). For expressing conceptual models, we have chosen a subset of the UML [3].

IEEE COMPUTER SOCIETY



[IRQ-0001] Information about prc
3.2 Information constraint requiremer
[CRQ-0001] No order without cus
[CRQ-0002] No empty orders allo
3.3 Functional requirements
3.3.1 System actors
[ACT-0001] Customer
[ACT-0002] Administrator
[ACT-0003] Supplier
3.3.2 System use cases
[UC-0001] Buy something
1. Actor Customer re
2. The system reque
3. Actor Customer ic
4. If the identificatio
If identification i
5. Actor Customer cl
6. The system asks t
7. Actor Customer c
8. The system inforn
9. Use case Deliver order
[UC-0002] Deliver order

ption	The system shall behave as described in the following use case when *a customer wants to buy some products*	
Precondition	There are products to be sold	
Ordinary sequence	**Step**	**Action**
	1	Actor Customer (ACT-0001) *requests the system for buying some product*
	2	The system *requests the customer for identification*
	3	Actor Customer (ACT-0001) *identifies himself or herself*
	4	If *the identification is OK,* the system *shows a list of products to buy*
	5	Actor Customer (ACT-0001) *chooses some products to buy*
	6	The system *asks the customer for confirmation*
	7	Actor Customer (ACT-0001) *confirms his or her order*
	8	The system *informs that the sale has been sucessful*
	9	Use case Deliver order (UC-0002) is performed
n	The products have been sold and the customer has payed	
	Step	**Action**
	4	If *identification is not OK in three attempts,* the system *informs that identification is not valid*, then this use case *is cancelled*

Adds a new step to the use case
HTML generated in 0.57 seconds HTML on NUM

Figure 2. REM user interface (default XSLT stylesheet)

2.1. REM architecture

REM projects are stored in relational, light–weight databases. When the user creates a new REM project, the basic structure is taken from a *base project*, that can be empty or can contain the mandatory sections of requirements standards like [9]. Any ordinary REM project can be selected as a base project, so REM users can reuse whole REM projects or only individual requirements by means of the *copy&paste* and *drag&drop* features of the REM user interface.

In order to provide immediate feedback on user actions, REM generates XML data corresponding to the project being edited, applies an external XSLT stylesheet that transforms XML data into HTML and shows the result to the user. The user can change document appearance or content by selecting or creating different XSTL stylesheets. The default XSLT stylesheet generates a highly hyperlinked document, easing navigation of requirements documents.

2.2. 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 four tabbed tree views, one for each document in the project. On the right, the result of the XSLT transformation of the XML data corresponding to the selected document is presented to the user in a embedded web browser.

In any of the four tree views, the user can directly manipulate REM objects by *drag&drop* or by context menus. Only actions that make sense can be performed, following a *correct–by–construction* approach, thus increasing internal consistency [8] and saving verification effort. For example, actions of use case steps can be of three different classes (see figure 3): *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 another use case, *i.e.* a *inclusion* or *extension* [3]. Steps with actor actions or use case actions can be created only if some actor or another use case have been previously created. On the other hand, only actors or use cases not referenced by any step can be deleted.

3. XML model of requirements in REM

REM is based on a UML model of requirements described in [5] which has been translated into a Document Type Definition (DTD) [22]. As an example, the UseCase class in figure 3 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>
```

where the rem:useCase element, as any other REMObject, must have a required identification attribute called oid. Children elements of rem:useCase like rem:triggeringEevent or rem:precondition contain only text, *i.e.* natural language. In REM, text can be composed of 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 another element with a matching identification attribute value. The rem:tbd element is declared as an EMPTY element, *i.e.* it is simply a mark.

4. Using XSLT for requirements verification

Simply by the fact of using REM, some quality properties described in [8] like modifiability, electronically storage, version annotation or being traceable are automatically

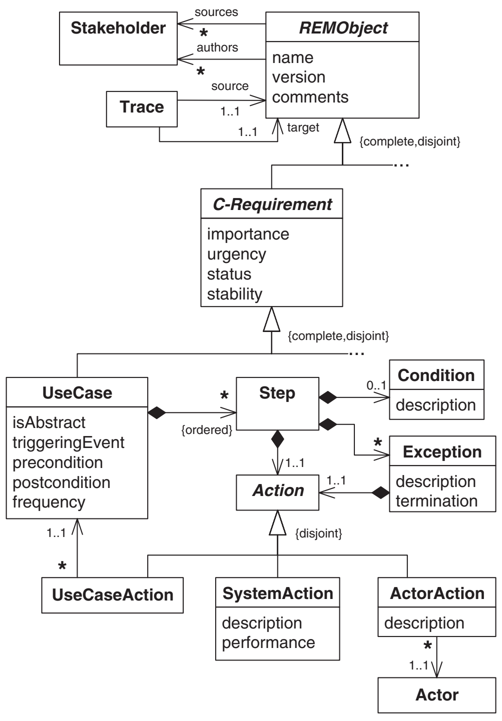


Figure 3. UML model of use cases in REM

fulfilled. In the following sections we describe how others quality properties described in [8] can be automatically verified, or some heuristics provided, using XSLT when requirements are electronically stored in XML format according to the REM DTD.

Although it is also possible to apply this approach to the verification of conceptual models created with REM, in this paper we focus only on requirements. For XML–based conceptual model verification see [13], where the *xlinkit* language and how it is used for the verification of UML models in XMI format [14] are described.

For the sake of readability, all examples in the following subsections use the rem:useCase element previously described, although they can also be applied to other REM objects.

4.1. Unambiguity

A requirement is unambiguous if and only if has only one possible interpretation [9]. Obviously, this is a knowledge property and cannot be automatically verified, but we can provide some heuristics about potential ambiguities in requirements in order to focus verification effort on *potentially ambiguous requirements*.

A simple yet powerful heuristic for detecting ambiguity (also design dependence, for example) is looking for *weak phrases indicators* (WPI) in requirements descriptions, *i.e.* "clauses that are apt to cause uncertainty and leave room for multiple interpretations", as described in [18]. This can be achieved by defining WPI in a external XML file like this (wpi.xml):

```
<?xml version="1.0"?>
<wpis>
  <wpi>easy</wpi>    <!-- potential ambiguity -->
  <wpi>etc</wpi>     <!-- potential ambiguity -->
  <wpi>normal</wpi> <!-- potential ambiguity -->
  . . .
  <wpi>click</wpi>   <!-- potential design dependence -->
  <wpi>button</wpi> <!-- potential design dependence -->
  . . .
</wpis>
```

and using the following XSLT code:

```
<xsl:for-each select="//rem:useCase">
  <xsl:variable name="uc" select="current()"/>
  <xsl:for-each select="document('wpi.xml')//wpi">
    <xsl:if test="contains($uc,.)">
      Use case <xsl:value-of select="$uc/rem:name"/>
      contains WPI <xsl:value-of select="."/>.
      Please, check it.
    </xsl:if>
  </xsl:for-each>
</xsl:for-each>
```

in which every use case is checked against every WPI in wpi.xml. Notice that the uc variable is needed since XSLT does not provide any way of accessing more than one current node at different levels in nested xsl:for-each structures.

Another complementary approach for ambiguity heuristics is measuring how much the customer's vocabulary is used in requirements descriptions. We agree with Leite [12] in the importance of understanding the language of the problem and in the importance of building a glossary at the beginning of the elicitation process. Following Leite, two principles should be followed: the *principle of circularity*, (the glossary must be as self–contained as possible) and the *principle of minimal vocabulary* (requirements descriptions must use as many glossary items as possible). Leite's principles cannot guarantee unambiguity, but they help to build more unambiguous, understandable, consistent, concise, and cross–referenced requirements [8], and can be used to measure the quality of the glossary and to detect potentially ambiguous requirements.

In the REM XML model, glossary items elements are mainly composed of rem:text elements, so they can contain references to other REM objects. In this context, XSLT can be used to measure glossary circularity (GLC) and minimality of vocabulary (MOV). GLC can be measured as the ratio between the number of references to glossary items from other glossary items and the number of glossary items, as shown in the following XSLT code:

```
<xsl:variable name = "GLC"
   select = "count(//rem:glossaryItem//rem:ref
                  [@oid = //rem:glossaryItem/@oid]) div
              count(//rem:glossaryItem)"
/>
```

GLC can be used as an indicator of the glossary quality. GLC values under 1 indicate a low quality glossary, since that implies that there are glossary items not referencing other glossary items. GLC can also be computed for single glossary items. It seems clear that glossary items not referencing other glossary items, or referencing just a few ones (less than 2, for example), should be verified for potential problems. The following XSLT code can be used for that purpose:

```
<xsl:for-each select="//rem:glossaryItem
   [count(.//rem:ref[@oid=//rem:glossaryItem/@oid]) < 2]"
>
   Please, check definition of glossary item
   <xsl:value-of select="rem:name"/>
</xsl:for-each>
```

MOV can be computed, in a similar way to GLC, as the ratio between the number of references to glossary items in requirements and the number of requirements. MOV can be used to pinpoint those requirements that do not have any reference, or just a few (less than 4, for example), 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 [8]. The same schema used for detecting suspicious glossary items can also be used for detecting potentially ambiguous requirements.

4.2. Completeness

A requirements document is complete if it includes [8]:

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.

The first completeness condition must be checked during requirements validation activity (see figure 1) and is therefore out of the scope of our approach.

The second condition is a knowledge property very difficult to verify automatically, but some heuristics can be applied when use cases are used for expressing functional requirements (see section 4.4).

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.

Organization, as described in [8], can also be considered as part of the third completeness condition, so XSLT can be used to verify if requirements documents are *organized*, *i.e.* if they have mandatory sections in the mandatory order with mandatory content. For example, if we want to be sure about the existence of a section named A with a child section named B, we can apply the following XSLT code:

```
<xsl:if test="not(//rem:section[rem:name='A']/
                  rem:section[rem:name='B'])">
   There is no A/B section organization
</xsl:if>
```

The fourth condition of completeness can be also verified using XSLT. If we want to know how many TBD marks are in a requirements document we can simply use the XPath [20] expression count(//rem:tbd). If we want to be more precise and we want to know, for example, what use cases have TBD marks inside their text and how many TBD marks they have, we can use the following XSLT code:

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

REM automatically inserts TBD marks in those requirements not annotated with relative importance or stability [8], so any of these properties can be checked using a XSLT code like this:

```
<xsl:for-each select="//rem:useCase[./rem:importance/rem:tbd]">
  Use case <xsl:value-of select="rem:name"/>
  is not annotated with relative importance
</xsl:for-each>
```

4.3. Traceability

In [8], 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 automatically assigns an unique identifier to every requirement, this quality property does not have to be verified explicitly.

What it must be checked is whether the origin of every requirement is clear or not, *i.e.* if requirements are *traced* in the sense described in [8]. In the REM model of requirements, any REM object can be traced to and from other REM objects and to their human sources and authors (see figure 3). In the REM DTD, traces are defined as elements with two required attributes of type IDREF, namely source and target:

```
<!ELEMENT rem:trace EMPTY>
<!ATTLIST rem:trace source IDREF #REQUIRED
                    target  IDREF #REQUIRED
>
```

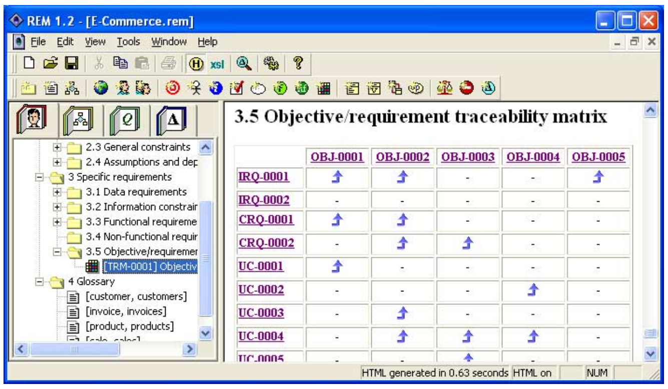
Having said that, we can use the following XSLT code to list all use cases with no human sources:

```
<xsl:for-each select="//rem:useCase[not(rem:sources)]">
 Use case <xsl:value-of select="rem:name"/>
 has not defined sources
</xsl:for-each>
```

and the following one to detect all use cases not traced to other REM objects:

```
<xsl:for-each
  select="//rem:useCase[not(//rem:trace/@source = @oid)]"
>
   Use case <xsl:value-of select="rem:name"/>
   is not traced to any object
</xsl:for-each>
```

REM users can also use traceability matrices for visual verification of traceability (see figure 4). REM users can create as many TraceabilityMatrix objects as they wish, selecting the classes of REM objects they want to be shown in rows in columns.



3.5 Objective/requirement traceability matrix

	OBJ-0001	OBJ-0002	OBJ-0003	OBJ-0004	OBJ-0005
IRQ-0001	↑	↑	-	-	↑
IRQ-0002	-	-	-	-	-
CRQ-0001	↑	↑	-	-	-
CRQ-0002	-	↑	↑	-	-
UC-0001	↑	-	-	-	-
UC-0002	-	-	-	↑	-
UC-0003	-	↑	-	-	-
UC-0004	-	↑	↑	↑	-
UC-0005	-	-	↑	-	-

Figure 4. REM Traceability matrix example

4.4. Verification of use cases

Use cases [15] are a popular form of expressing functional requirements. REM supports both *classic* functional requirements (*i.e.* plain text) and use cases.

For the verification of use cases, we have adopted a metrics–based approach taking the verification heuristics used in the scenario construction process described in [11] as a reference. Some metrics based on the REM model of use cases are defined in table 2.

Table 2. REM use case metrics

Metric	Description
NOS	Number of steps of the use case
NOAS	Number of actor action steps of the use case
NOSS	Number of system action steps of the use case
NOUS	Number of use case action steps of the use case
NOCS	Number of conditional steps
NOE	Number of exceptions of the use case
NIE	Number of times a use case is included or extends other use cases

So, considering the following DTD fragment in which the REM XML model of use case steps is described:

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

heuristics based on metrics defined in table 2 can be applied using the following XLST code as a pattern, where metrics values are stored in auxiliary variables:

```
<xsl:for-each select="//rem:useCase">
  <xsl:variable name="NOS"
    select="count(rem:step)"/>
  <xsl:variable name="NOAS"
    select="count(rem:step[rem:actorAction])"/>
  <xsl:variable name="NOSS"
    select="count(rem:step[rem:systemAction])"/>
  <xsl:variable name="NOUS"
    select="count(rem:step[rem:useCaseAction])"/>
  <xsl:variable name="NOCS"
    select="count(rem:step[rem:condition])"/>
  <xsl:variable name="NOE"
    select="count(rem:step/rem:stepException)"/>
  <xsl:variable name="NIE" select=
   "count(//rem:useCaseAction[@useCase=current()/@oid])"/>
  <!-- apply verification heuristics here -->
</xsl:for-each>
```

The set of verification–oriented, metrics–based heuristics for use cases we have developed is described below, including their corresponding XLST code (that must be inserted into the XSLT pattern). The intervals for usual values of metrics have been taken after analyzing 414 use cases developed by our students using REM.

4.4.1. Checking the number of steps

Since use cases describe interactions between actors and the system, a use case should have at least 2 steps for a minimal interaction *actor request – system response* (other authors like [4] recommend a value of NOS in the interval $[3, 9]$).

Use cases with NOS < 2 should be checked for potential incompleteness. On the other hand, a high value of NOS is an indicator of too much complexity or of the lack of structure (*i.e. includes* or *extends* relationships) in use cases. The XSLT code corresponding to this heuristic is the following:

```
<xsl:if test="($NOS &lt; 2) or ($NOS &gt; 9)">
  Use case <xsl:value-of select="rem:name"/>,
  has an unusual number of steps. Please, check it.
</xsl:if>
```

4.4.2. Checking the rate of different types of steps

The three types of use case steps in REM, actor action, system action and use case action steps, should not appear with the same frequency. Using the same reasoning that in the previous heuristic, since use cases describe interactions between actors and the system, approximately half the steps of a use case should be actor action steps and the other half system action steps. On the other hand, a use case should not have most of its steps being inclusions or extensions, *i.e.* a high value of NOUS is a clear indicator of an abusive use of use cases relationships.

Data from our students' use cases point to the following usual values: NOAS/NOS in $[30\%, 70\%]$, NOSS/NOS in $[40\%, 80\%]$ and NOUS/NOS in $[0\%, 25\%]$.

As an example, the following XSLT code lists all use cases with a NOAS/NOS value out of usual interval:

```
<xsl:if test="(($NOAS div $NOS) &lt; 0.3) or
             (($NOAS div $NOS) &gt; 0.7)">
  Use case <xsl:value-of select="rem:name"/> has
  an unusual ratio of actor steps. Please check it.
</xsl:if>
```

4.4.3. Checking the number of exceptions

One of the completeness conditions enumerated in section 4.2 is that a requirements document must include responses of the software to all classes of input data in all realizable situations. In the REM model of use cases, system behavior in abnormal situations is described as exceptions associated to use case steps. So, if a use case has no exceptions or just a few when compared to the number of steps, it is probably because only the ordinary sequence of steps has been defined. Thus, this use case should be checked for completeness. On the contrary, a use case with most of its steps having associated exceptions is probably too complex to be understood.

Data from our students' point to usual values of NOE/NOS in the interval $[5\%, 45\%]$, so the XSLT code corresponding to this heuristic is the following:

```
<xsl:if test="(($NOE div $NOS) &lt; 0.05) or
             (($NOE div $NOS) &gt; 0.45)">
  Use case <xsl:value-of select="rem:name"/> has
  an unusual ratio of exceptions. Please check it.
</xsl:if>
```

4.4.4. Checking cyclomatic complexity

In the same way a use case with a high NOE/NOS rate is probably too complex to be understood, a use case with many conditional steps is also difficult to understand. Despite of the fact that use cases are very different from source code, the heuristic of keeping cyclomatic complexity (CC) low can also be applied to use cases. Cyclomatic complexity for use cases in REM can be computed as the number of decision points + 1, *i.e.* the number of conditional steps + the number of exceptions + 1.

The following XSLT checks cyclomatic complexity values of use cases, in which we have taken $[1, 4]$ as the usual interval for CC as suggested by our students' use cases:

```
<xsl:variable name="CC" select="$NOCS+$NOE+1"/>
<xsl:if test="$CC &gt; 4">
  Use case <xsl:value-of select="rem:name"/>
  has a unusual value of cyclomatic complexity.
  Please, check it.
</xsl:if>
```

4.4.5. Checking abstract use cases

Abstract use cases are those use cases that can only be performed from other use cases and therefore cannot be triggered by any event. The usual motivation for the creation of abstract use cases is avoiding the repetition of common sequences of steps in other use cases. If an abstract use case is neither included from other use cases nor it extends other use cases is clearly useless and should be deleted from the specification. If it is used for only one use case, it should be merged into the calling use case. The XSLT code corresponding to this heuristic is the following:

```
<xsl:if test="rem:isAbstract and $NIE &lt; 2">
   Abstract use case <xsl:value-of select="rem:name"/>
   is hardly used. Please, check it.
</xsl:if>
```

4.4.6. Checking actor participation

Another verification heuristic included in [11] is checking if every actor participate in at least one use case. All step action elements in the REM DTD have an actor attribute of type IDREF, so the following XSLT code can be used for checking if an actor participates at least in one use case:

```
<xsl:for-each select="//rem:actor
  [not(//rem:actorAction/@actor = @oid)]">
  Actor <xsl:value-of select="rem:name"/>
  does not participate in any use case.
  Please, check it.
</xsl:for-each>
```

5. Related work

Some of the results of the ESPRIT project CREWS, especially style and content guidelines for textual specification of use cases [17] have influenced our work. Their approach, implemented in the *CREWS-SAVRE* and *L'Ecritoire* tools, combines natural language processing (NLP) techniques and linguistic patterns, guiding the construction of use cases and providing ambiguity detection mechanisms which are much more powerful than ours.

We use linguistic patterns extensively [6] and our UML model of requirements is partially based on CREWS results, but we have not adopted an NLP–based approach for verification because one of our goals was to let REM users write their own verification programs (*i.e.* XSLT stylesheets). Using an NLP–based verification approach would have made this goal very difficult to satisfy because of their complexity. We are currently conducting experiments with our students in order to know if our simpler approach yields comparable results to the ones described in [2].

The Automated Requirement Measurement (ARM) tool [18], is a simple yet powerful, not NLP–based requirements verification tool that scans requirements documents for specific words and phrases that are considered *indicators* of the quality of requirements. As shown in section 4.1, REM can perform the same type of analysis using XSLT and apply a wider set of verification–oriented heuristics.

Schematron [10] is an XML–based language for specifying assertions on XPath expressions in XML documents. A Schematron document must be transformed into a XSLT stylesheet using the Schematron XSLT stylesheet and then applying the resulting stylesheet to the XML document being analyzed. Our approach and Schematron relies on XPath expressions, but ours needs only one XSLT transformation and provides greater flexibility for presenting results to users and for computing requirements–oriented metrics.

The *xlinkit* language [13] is an XML–based language for specifying consistency rules between XML documents using a restricted set of first order logic and XPath expressions. Most XSLT verification code presented in this paper can be expressed in *xlinkit*. For example the XSLT code in section 4.3 for detecting use cases not traced to other REM objects can be translated into the following *xlinkit* code, which identifies all use cases no traced to any object as *inconsistent links*:

```
<globalset id="$useCases" xpath="//rem:useCase"/>
<globalset id="$traces" xpath="//rem:trace"/>
<consistencyrule id="beTraced">
  <description>
    Every use case must be traced to some object,
    i.e. must be the source of some trace
  </description>
  <forall var="uc" in="$useCases">
    <exists var="t" in="$traces">
      <equal op1="$uc/@oid" op2="$t/@source"/>
    </exists>
  </forall>
</consistencyrule>
```

Translating other XSLT fragments into *xlinkit* requires the use of the XPath function true (*xlinkit* only allows equality and inequality expressions on XPath expressions as predicates), and the expansion of variable expressions (*xlinkit* does not allow the declaration of local variables inside forall or exists elements). For example, the *xlinkit* code corresponding to the *Checking the number of steps* heuristic is the following:

```
<globalset id="$useCases" xpath="//rem:useCase"/>
<consistencyrule id="checkingNOS">
  <description>
    Every use case should have 2<=NOS<=9
  </description>
  <forall var="uc" in="$useCases">
    <equal op1="(count($uc/rem:step) &gt;= 2) and
                (count($uc/rem:step) &lt;= 9)"
           op2="true()"
    />
  </forall>
</consistencyrule>
```

REM verification stylesheets must be XSLT in order to be processed by the REM user interface and they must be as efficient as possible for fast visual feedback to the user. *xlinkit* rules must be processed by a Java program so that makes it incompatible with REM. On the other hand, *xlinkit* presents results as xlinks, something that is not currently supported by most web browsers.

DOORS, a commercial RE tool with much more features than REM, includes a C++–like scripting language called DXL [1]. Using DXL, it is possible to implement some of the heuristics described in this paper, but the DOORS user must define all necessary attributes and types of objects to have a requirements model similar to ours in DOORS. Apart from the fact that our approach is based on standard languages and not in a proprietary one like DXL, the versatility offered by XML and XSLT (for report generation, for example) is far from the offered by DXL (unless XML is generated from DXL).

6. Conclusions and future work

In this paper we have presented an automated, light–weight, XSLT–based approach for the verification of requirements which is integrated in the REM RE tool. Our approach is based on a open technology like XML and offers all the flexibility of XSLT. In fact, if requirements are represented in XML not using the REM DTD, many of the XSLT–based verification–oriented heuristics presented in this paper should be easily adaptable.

Our future work is focused on identifying accurate interval values for those metrics–based verification heuristics (glossary–based and use case heuristics mainly). We are currently conducting an experiment with our students at the University of Seville with two main goals in mind. The first goal is determining interval values by applying data mining techniques to our students' RE projects, thus identifying correlations between defects and metric values. The second goal is to know whether our approach improves requirements verification or not, *i.e.* if more defects in requirements are detected when requirements verification is supported by our XSLT–based heuristics.

Other lines for future work are the developing of XSLT stylesheets for computing *use case points* [19] and for graphical visualization of verification–oriented metrics in web browsers.

Acknowledgments

We wish to thank Julio Leite for his comments and suggestions on an early version of this paper.

References

[1] I. Alexander. Getting Started with DXL, the DOORS eXtension Language. Technical report, Telelogic Innovate, 2001.

[2] C. Ben Achour, C. Rolland, N. A. M. Maiden, and C. Souveyet. Guiding Use Case Authoring: Results of an Empirical Study. In *ISRE'99 Proceedings*, 1999.

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

[4] A. Cockburn. *Writing Effective Use Cases*. Addison–Wesley, 2001.

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

[6] 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.

[7] A. Durán, A. Ruiz, B. Bernárdez, and M. Toro. Verifying Software Requirements with XSLT. *ACM Software Engineering Notes*, 27(1), 2002.

[8] 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.

[9] IEEE. Recommended Practice for Software Requirements Specifications. IEEE/ANSI Standard 830–1998.

[10] R. Jelliffe. The Schematron Assertion Language 1.5. Technical report, Academia Sinica Computing Centre, 2001.

[11] J. C. S. P. Leite, H. Hadad, J. Doorn, and G. Kaplan. A Scenario Construction Process. *Requirements Engineering Journal*, 5(1), 2000.

[12] J. C. S. P. Leite, G. Rossi, F. Balaguer, V. Maiorana, G. Kaplan, G. Hadad, and A. Oliveros. Enhancing a Requirements Baseline with Scenarios. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, 1997.

[13] C. Nentwich, W. Emmerich, and A. Finkesltein. Static consistency checking for distributed specifications. In *Proceedings of Automated Software Engineering*, 2001.

[14] OMG. XML Metadata Interchange (XMI) Specification 1.1. Technical report, November 2000.

[15] OMG. Unified Modeling Language, v1.4. Technical report, September 2001.

[16] S. Robertson and J. Robertson. *Mastering the Requirement Process*. Addison–Wesley, 1999.

[17] C. Rolland and C. B. Achour. Guiding the Construction of Textual Use Case Specifications. *Data & Knowledge Engineering Journal*, 25(1–2), 1998.

[18] L. Rosenberg, T. F. Hammer, and L. L. Huffman. Requirements, Testing and Metrics. In *15th Annual Pacific Northwest Software Quality Conference*, Utah, 1998.

[19] G. Schneider and J. P. Winters. *Applying Use Cases: a Practical Guide*. Addison–Wesley, 1998.

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

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

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