Making Programming Exercises Interoperable with PExIL

Ricardo Queirós[1] and José Paulo Leal[2]
[1] CRACS & DI-ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt
[2] CRACS & DCC-FCUP, University of Porto, Portugal
zp@dcc.fc.up.pt

ABSTRACT
Several standards have appeared in recent years to formalize the metadata of learning objects, but they are still insufficient to fully describe a specialized domain. In particular, the programming exercise domain requires interdependent resources (e.g. test cases, solution programs, exercise description) usually processed by different services in the programming exercise life-cycle. Moreover, the manual creation of these resources is time-consuming and error-prone leading to an obstacle to the fast development of programming exercises of good quality.
This paper focuses on the definition of an XML dialect called PExIL (Programming Exercises Interoperability Language). The aim of PExIL is to consolidate all the data required in the programming exercise life-cycle, from when it is created to when it is graded, covering also the resolution, the evaluation and the feedback. We introduce the XML Schema used to formalize the relevant data of the programming exercise life-cycle. The validation of this approach is made through the evaluation of the usefulness and expressiveness of the PExIL definition. In the former we present the tools that consume the PExIL definition to automatically generate the specialized resources. In the latter we use the PExIL definition to capture all the constraints of a set of programming exercises stored in a learning objects repository.

INTRODUCTION
The concept of Learning Object (LO) is fundamental for producing, sharing and reusing content in eLearning [1]. In essence a LO is a container with educational material and metadata describing it. Since most LOs just present content to students they contain documents in presentation formats such as HTML and PDF, and metadata describing these documents using mostly Learning Objects Metadata (LOM) or other generic metadata format. When a LO includes exercises to be automatically evaluated by an eLearning system, it must contain a document with a formal description for each exercise. The Question and Tests Interoperability (QTI) [2] is an example of a standard for this kind of definitions that is supported by several eLearning systems. However, QTI was designed for questions with predefined answers and cannot be used for complex evaluation domains such as the programming exercise evaluation [3]. A programming exercise requires a collection of files (e.g. test cases, solution programs, exercise descriptions, feedback) and special data (e.g. compilation and execution lines). These resources are interdependent and processed in different moments in the life-cycle of a programming exercise.

The life cycle comprises several phases: in the creation phase the content author should have the means to automatically create some of the resources (assets) related to the programming exercise such as the exercise description and test cases and the possibility to package and distribute them in a standard format across all the compatible systems such as learning management systems (LMS) and learning objects repositories (LOR); in the selection phase the

teacher must be able to search for a programming exercise based on its metadata from a repository of learning objects and store a reference to it in a learning management system; in the presentation phase the student must be able to choose the exercise description in its native language and a proper format (e.g. HTML, PDF); in the resolution phase the learner should have the possibility to use test cases to test an attempt to solve the exercise and the possibility of automatically generating new ones; in the evaluation phase the evaluation engine should receive specialized metadata to properly evaluate the learner's attempt and return enlightening feedback. All these phases require a set of inter-dependent resources and specialized metadata whose manual creation would be time-consuming and error-prone.
This paper focuses on the definition of an XML dialect called PExIL (Programming Exercises Interoperability Language). The aim of PExIL is to consolidate all the data required in the programming exercise life-cycle, from when it is created to when it is graded, covering also the resolution, the evaluation and the feedback. We introduce the XML Schema used to formalize the relevant data of the programming exercise life-cycle. The validation of this approach is made through the evaluation of the usefulness and expressiveness of the PExIL definition. In the former, we use a PExIL definition to generate several resources related to the programming exercise life-cycle (e.g. exercise descriptions, test cases, feedback files). In the latter, we check if the PExIL definition covers all the constraints of a set of programming exercises stored in a learning objects repository.

The remainder of this paper is organized as follows. Section 2 traces the evolution of standards for LO metadata and packaging. In the following section we present the PExIL schema with emphasis on the definitions for the description, test cases and feedback of the programming exercise. Then, we evaluate the definition of PExIL and conclude with a summary of the main contributions of this work and a perspective on future research.

Learning object standards

The increasing popularity of programming contests worldwide resulted in the creation of several contest management systems. At the same time Computer Science courses use programming exercises to encourage the practice on programming. The interoperability between these types of systems is becoming a topic of interest in the scientific community. In order to address these interoperability issues several problem formats were developed such as CATS[1], FreeProblemSet (FPS)[2], Mooshak Exchange Format (MEF)[3] and Peach Exchange Format (PEF)[4]. However several issues were found regarding their expressiveness and proliferation over the Web. The majority of the formats only describe how the program should be compiled and executed and how the statement is composed. The latter issue is due to the fact that these formats are deeply related with the contest management systems that adopted them. Thus, they do not comply with any generic standard for describing, packaging and deployment of the exercises such as the concept of learning objects.
Current LO standards are quite generic and not adequate to specific domains, such as the definition of programming exercises. The most widely used standard for LO is the IMS Content Packaging (IMS CP) [4]. This content packaging format uses an XML manifest file wrapped with other resources inside a zip file. The manifest includes the IEEE Learning Object Metadata (LOM) standard [5] to describe the learning resources included in the package. However, LOM was not specifically designed to accommodate the requirements of automatic evaluation of programming exercises. For instance, there is no way to assert the role of specific resources, such as test cases or solutions. Fortunately, IMS CP was designed to be straightforward to

[1] http://imcs.dvgu.ru/cats/docs/format.html
[2] http://code.google.com/p/freeproblemset/
[3] http://mooshak.dcc.fc.up.pt/
[4] http://peach.win.tue.nl/

extend, meeting the needs of a target user community through the creation of application profiles. When applied to metadata the term Application Profile generally refers to "the adaptation, constraint, and/or augmentation of a metadata scheme to suit the needs of a particular community" [6]. A well known eLearning application profile is SCORM [7] that extends IMS CP with more sophisticated sequencing and Contents-to-LMS communication. The creation of application profiles aims to meet the needs of the target user community, aid integration and enhance interoperability between tools and services of the community. The creation is based on one or more of the following approaches [8]:

1- Selection of a core sub-set of elements and fields from the source schema;
2- Addition of elements and/or fields (normally termed extensions) to the source schema, thus generating the derived schema;
3- Substitution of a vocabulary with a new, or extended vocabulary to reflect terms in common usage within the target community;
4- Description of the semantics and common usage of the schema as they are to be applied across the community.

Following this extension philosophy, the IMS Global Learning Consortium (GLC) upgraded the Question & Test Interoperability (QTI) specification [2]. QTI describes a data model for questions and test data and, from version 2, extends the LOM with its own metadata vocabulary. QTI was designed for questions with a set of pre-defined answers, such as multiple choice, multiple response, fill-in-the-blanks and short text questions. It supports also long text answers but the specification of their evaluation is outside the scope of the QTI. Although long text answers could be used to write the program's source code, there is no way to specify how it should be compiled and executed, which test data should be used and how it should be graded. For these reasons we consider that QTI is not adequate for automatic evaluation of programming exercises [3], although it may be supported for sake of compatibility with some LMS. Recently, IMS GLC proposed the IMS Common Cartridge (CC) [9] that bundles the previous specifications.

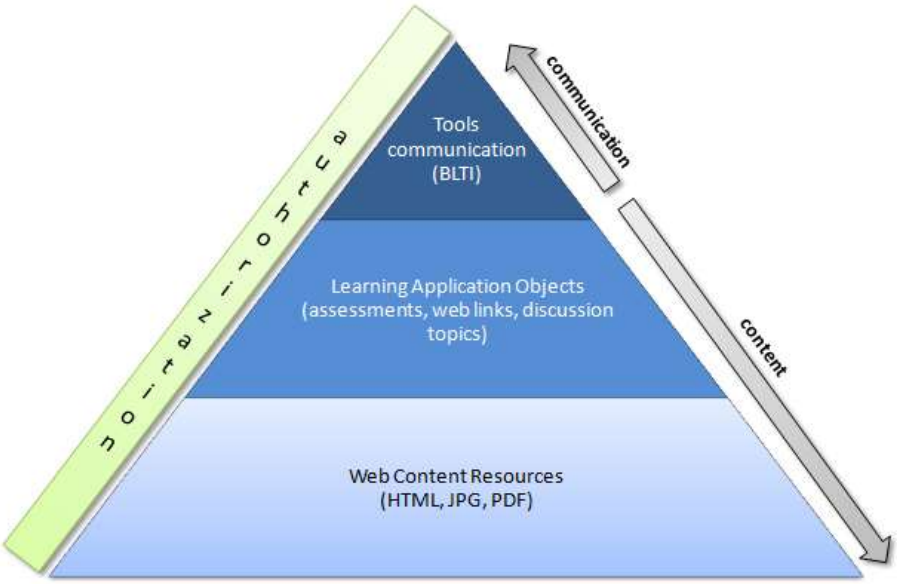

Fig. 1 *Common Cartridge Content Hierarchy.*

The IMS Common Cartridge specification defines an open format for the distribution of rich web-based content. Its main goal is to organize and distribute digital learning content and to ensure the interchange of content across any Common Cartridge conformant tools. The latest revised version (1.1) was released in May 2011. The IMS CC package organizes and describes a

learning object based on two levels of interoperability: content and communication as depicted Figure 1 [10]. In the **content level**, the IMS CC includes two types of resources:

- Web Content Resources (WCR): static web resources that are supported on the Web such as HTML files, GIF/JPEG images, PDF documents, etc.
- Learning Application Objects (LAO): special resource types that require additional processing before they can be imported and represented within the target system. Physically, a LAO consists of a directory in the content package containing a descriptor file and optionally additional files used exclusively by that LAO. Examples of Learning Application Objects include QTI assessments, Discussion Forums, Web links, etc.

In the **communication level** the cartridge describes how the target tool of the cartridge (usually a LMS) should communicate with other remote web applications using the IMS Basic Learning Tools Interoperability (LTI) specification [11]. The LTI is a common interoperability specification that is increasingly supported by major LMS vendors. It provides a uniform standards-based extension point in LMS allowing remote tools and content to be integrated into LMSs. The main goal of the LTI is to standardize the process for building links between learning tools and the LMS. The IMS launched also a subset of the full LTI v1.0 specification called IMS Basic LTI. This subset exposes a unidirectional link between the LMS and the application. However, there is no provision for accessing run-time services in the LMS and only one security policy[5] is supported.

PExIL

In this section we present PExIL - Programming Exercises Interoperability Language - an XML dialect that aims to consolidate all the data required in the programming exercise life-cycle. This covers all the programs that receives data from the standard input and, after data process, send the results for the standard output. This dialect is formalized through the creation of a XML Schema. In the following subsections we present the PExIL XML Schema organized in three groups of elements:

Textual – elements with general information about the exercise to be presented to the learner (e.g. title, date, challenge);
Specification – elements with a set of restrictions that can be used for generating specialized resources (e.g. test cases, feedback);
Programs – elements with references to programs as external resources (e.g. solution program, correctors) and metadata about those resources (e.g. compilation, execution line, hints).

Textual elements

Textual elements contain general information about the exercise to be presented to the learner. This type of elements can be used in several phases of the programming exercise life-cycle: in the selection phase as exercise metadata to aid discoverability and to facilitate the interoperability among other systems such as LMS or even Integrated Development Environments (IDE); in the presentation phase as content to be presented to the learner (e.g. exercise description); in the resolution phase as skeleton code to be included in the student's project solution.

The following table presents the textual elements of the PExIL schema and identifies the phases where they are involved.

[5] OAuth – http://oauth.net

Table 1. *Textual elements.*

Element	Selection	Presentation	Resolution	Evaluation
title	x	x		
creation/author	x	x		
creation/date	x	x		
creation/event	x	x		
creation/institution	x	x		
context		x		
challenge		x		
keywords	x	x		
skeleton		x	x	

The `title` element represents the title of the programming exercise. This mandatory element uses the `xml:lang` attribute to specify the human language of the element's content. The definition of this element in the XML Schema has the `maxOccurs` attribute set to unbound allowing the same information to be recorded in multiple languages. The `creation` element contains data on the authorship of the exercise and includes the following sub-elements: `author` with information about the the name(s) of the author(s); `date` which includes the date of the generation of the exercise, `event` which describes the event for which the exercise was created and `institution` which describes where the exercise will be used. The `context` element is an optional field used to contextualize the student with the exercise. The `challenge` element is the actual description of the exercise. Its content model is defined as mixed content to enable character data to appear between XHTML child-elements. This XML markup language will be used to enrich the formatting of the exercises descriptions. The `keywords` element is used to describe the subject(s) inherent to the exercise. The `skeleton` element refers to a resource containing code to be included in the student's project solution.

Specification elements

The goal of defining programming exercises as learning objects is to use them in systems supporting automatic evaluation. In order to evaluate a programming exercise the learner must submit a program in source code to an Evaluation Engine (EE) that judges it using predefined test cases - a set of input and output data. In short, the EE compiles and runs the program iteratively using the input data (standard input) and checks if the result (standard output) corresponds to the expected output. Based on these correspondences the EE returns an evaluation report with feedback.

In the PExIL schema, the `input` and `output` top-level elements are used to describe respectively the input and the output test data. These elements include three sub-elements: `description`, `example` and `specification`. The `description` element includes a brief description of the input/output data. The `example` element includes a predefined example of the input/output test data file. Both elements comply with the `specification` element that describes the structure and content of the test data.

Table 2. *Specification elements.*

Element	Selection	Presentation	Resolution	Evaluation
input/specification		x	x	x
output/specification		x	x	x

This definition can be used in several phases of the programming exercise life-cycle as depicted in Table 2: by 1) the content author to automatically generate an input and output test example

to be included on the exercise description for presentation purposes and others (private) test cases to be used by the evaluator for evaluation purposes; 2) the learner to automatically validate his attempt against the public test cases generated previously; 3) the Evaluation Engine to evaluate a submission using the test cases.

The `specification` element (Fig. 2) contains two attributes and two top-level elements. The attributes `line_terminator` and `value_separator` define respectively the newline and space characters of the test data. The two top-level elements are: `line` which defines a test data row and `repeat` which defines an iteration on a set of nested elements. The number of iterations is controlled by the value of the `count` attribute.

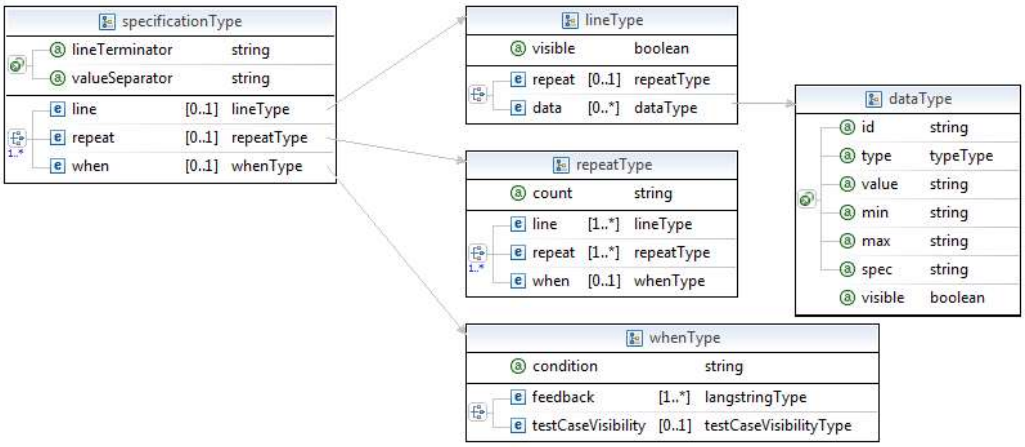

Fig. 2 The `specification` element.

The `line` element defines a data row. Each row contains one or more variables. A variable in the specification model must have a unique name which is used to refer values from one or more places in the `specification` element. A variable is represented in the PExIL schema by the `data` element containing the following attributes:

- `id` - defines the name of the variable. To access a variable one must use the `id` attribute preceded by the character `$` to enable the further resolution and evaluation of XPath expressions while processing the specification model;
- `type` – defines the variable data type (e.g. integer, float, string, enum). In the case of an enumeration the values are presented as a text child node;
- `value` – represents the value to be included in the input/output test file. If filled the variable acts as a constant. Otherwise, the value can be automatically generated based on a set of constraints - the `type`, `min`, `max` or `spec` attributes;
- `min/max` – represents value constraints by defining limits on the values. The semantic of these attributes depends exclusively on the data type: may represent the ranges of a value (integer and float), the minimum/maximum number of characters (string) or a range of values to be selected from an enumeration list;
- `spec` - regular expression for generating/matching strings of text, such as particular characters, words, or patterns of characters.

The following XML excerpt shows the `specification` elements for the input and output test data of an exercise. The exercise challenge is *given three numbers to verify that the last number is between the first two*.

Example of the input test description: “The input begins with a single positive integer on a line by itself indicating the number of the cases following. This line is followed by a blank line, and there is also a blank line between two consecutive inputs. Each line of input contains three float numbers (num1, num2 and num3) ranging values between 0 and 1000. “.

```
<specification line_terminator=”\n” value_separator=” ”>
 <line><data id=”numTestCases” type=”int” value=”3”/></line>
 <line/>
  <repeat count=”$numTestCases”>
  <line>
    <data id=”num1” type=”float” min=”0” max=”1000”/>
    <data id=”num2” type=”float” min=”0” max=”1000”/>
    <data id=”num3” type=”float” min=”0” max=”1000”/>
 </line>
 <line/>
</repeat>
<when condition=”$num1>$num2”>
      <feedback xml:lang=”en-GB”>
           Numbers can be given in descending order
      </feedback>
</when>
 </specification>
```

Example of the output test description: “The output must contain a boolean for each test case separated by a blank line between two consecutive outputs. “

```
<specification line_terminator=”\n” value_separator=” ”>
  <repeat count=”$numTestCases”>
    <line>
    <data id=”result” type=”enum” value=”1”>True False</data>
  </line>
  <line/>
</repeat>
</specification>
```

As said before, the EE is the component responsible for the assessment of an attempt to solve a particular programming exercise posted by the student. The assessment relies on predefined test cases. Whenever a test case fails a **static feedback** message (e.g. "Wrong Answer", "Time Limit Exceed", and “Execution Error") associated with the respective test case is generated. Beyond the static feedback of the evaluator, the PExIL schema includes a `when` element in the `specification` element. This element defines a **dynamic feedback** message to be presented to the student based on the evaluation of an XPath expression included in the `condition` attribute. This expression can include references to input and output variables or even dependencies between both. If the expression is evaluated as true then the element child node (`feedback` element) is used as the feedback message.

The PExIL definition supports the concept of incremental feedback to control the appearance of both types of feedback upon a submission of a student’s attempt. The `feedbackLevels` element is a top-level child element which defines a set of feedback levels that the exercise supports and when it is shown to the student. The following XML excerpt shows an example of a `feedbackLevels` element.

```
<pexil:feedbackLevels
   levels="simple|count_classifications|test_case_feedback_hint"
   incremental="2"
   showAllLevels="false"/>
```

The `levels` attribute may have one or more feedback levels. The existent levels are:

Simple – a feedback message indicating whether the student’s attempt is correct or incorrect (e.g. “Wrong answer!”);

Count_worst_classification – a feedback message indicating the worst classification of all the tests (e.g. "1 test with wrong answer");
Count_classifications – a feedback message indicating the classifications of all tests (e.g. "3 tests accepted and 1 test with wrong answer");
Test_case_feedback_hint – a feedback message to be presented to the student based on the evaluation of a condition defined by the content author. This feedback level is pedagogical relevant since the teacher can cover common errors of his students and warn them with useful and contextual feedback (e.g. "Forgot to divide by the number of input elements");
Test_case_input_result – a feedback message including the input data of an unsuccessful test case (e.g. "Unexpected output for the test with the input data: '5 6' ");
Test_case_input_output – a feedback message with the input and the output data of an unsuccessful test case (e.g. "Unexpected output for the test with the input data: '5 6' and the output data: '5,5' ").

The `incremental` attribute defines a value which is used to control the appearance of the feedback levels. The `showAllLevels` attribute defines if the feedback to be presented to the student should accumulate with previous ones.

In the last example were defined three levels of feedback. Based on the `incremental` attribute value the two first students' unsuccessful attempts will receive a simple feedback, the next two a *count_classification* feedback and so on.

Program elements

Program elements contain references to program source files as external resources (e.g. solution program, correctors) and metadata about those resources (e.g. compilation, execution line, hints). These resources are used mostly in the evaluation phase of the programming exercise life-cycle (Table 3) to allow the EE to produce an evaluation report of a students' attempt to solve a programming exercise.

Table 3. *Program elements.*

Element	Selection	Presentation	Resolution	Evaluation
solution			x	x
corrector				x
hints	x			x

A program element is defined with the `programType` type depicted in

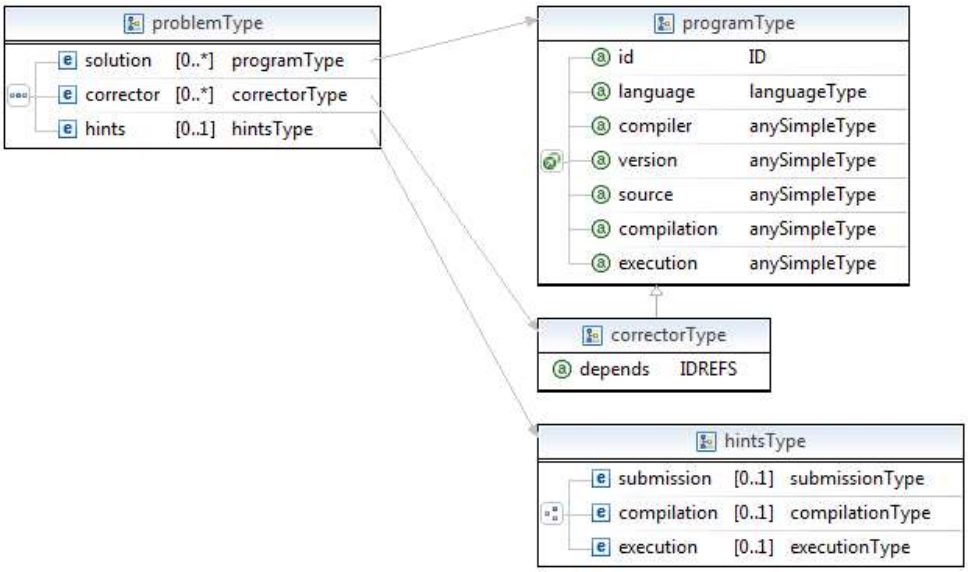

Fig. *3*.

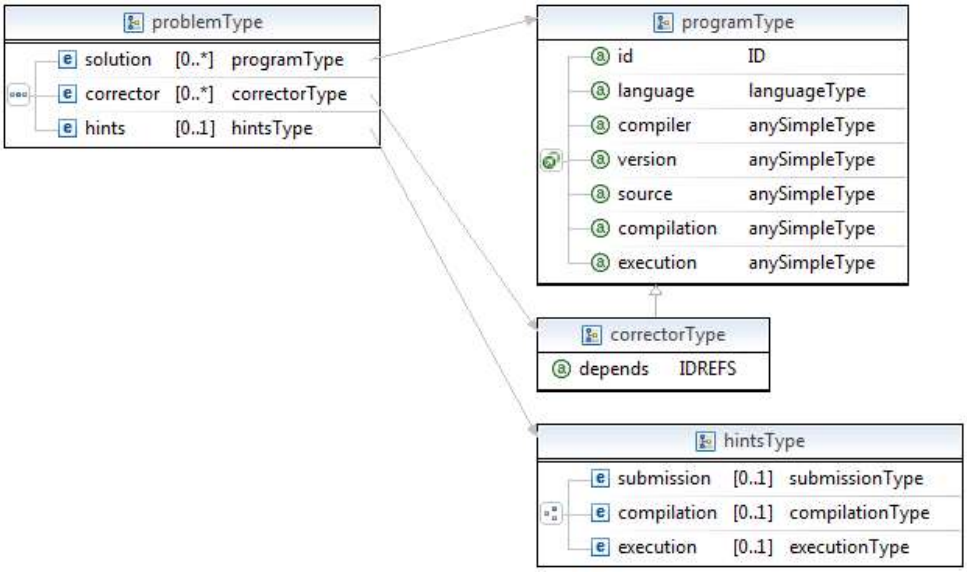

Fig. 3 *Program elements.*

This type is composed by seven attributes: `id` – an unique identifier for the resource; `language` – identifies the programming language used to code the resource (e.g. JAVA, C, C#, C++, PASCAL); `compiler/executer` – defines the name of the compiler/executer; `version` – identifies the version of the compiler; `source/object` - defines the name of the program source/object file; `compilation` – defines a command line to compile the source code; and `execution`– defines a command line to execute the compiled code;

There are two program elements in the PExIL schema: the `solution` and the `corrector` elements. The `solution` element contains a reference to the program solution file. The following XML excerpt shows an example of a `solution` element.

```
<solution
   id="solution" language="JAVA"
   compiler="javac" executer="java"
   version="1.6" source="solution.java" object="solution"
   compilation="$compiler $source"
   execution="$executer $object">
```

The `corrector` element is optional and refers to custom programs that change the general evaluation pattern for a given exercise. The `corrector` element is optional and refers to custom programs that change the general evaluation pattern for a given problem. There are two types of correctors: static and dynamic correctors. The **static corrector** is invoked immediately after compilation, before any execution. The corrector can be used to compute software metrics on the source code, judging the quality of source code; perform unit testing on the program; check the structure of the program's source code. The **dynamic corrector** is invoked after each execution with a test case. Deals with non-determinism (e.g. the solution is a set of unordered values, in this case the corrector normalizes the outputs before comparing them). A single programming exercise may use an arbitrary number of correctors. The order in which they are executed is defined by the `depends` attribute extending the `programType` type

The metadata about the program type resources is consolidated in the `hints` element aggregating a set of recommendations for the submission, compilation and execution of exercises. These recommendations can be used by the EE to improve the evaluation and feedback process. The hints element is composed by three sub-elements: `submission`, `compilation` and `execution` elements.

The `submission` element defines guidelines to follow in submission process. It is composed by the following attributes: `time-solve` – time limit for solving the exercise; `time-submit` – time limit for submitting the exercise; `attempts` – maximum number of attempts for submitting the problem; `code-lines` – maximum number of code lines in the user's code; `length` - maximum length in the user's code.

The `compilation` element defines guidelines to follow in compilation process. It is composed by the following attributes: `time-` time limit to compile the exercise; `size` - maximum size of the execution code.

The `execution` element defines guidelines to follow in execution process. It is composed by the following attributes: `time-` time limit for executing the exercise.

USING PExIL

In this section we validate the PExIL definition according to: its **usefulness** while using the PExIL definition as input of a set of tools related to the programming exercise life-cycle (e.g. generation of a IMS CC learning object package); and its **expressiveness** while using the PExIL definition to capture all the constraints of a set of programming exercises in a repository (e.g. description of crimsonHex [12] programming exercises).

Generating a IMS CC learning object package

In this subsection we validate the **usefulness** of the PExIL definition by detailing the generation of an IMS CC LO package based on a valid PExIL instance. An IMS CC object is a package standard that assembles educational resources and publishes them as reusable packages in any system that implements this specification (e.g. Moodle LMS).

A Generator tool (e.g. PexilUtils) uses the PExIL definition to produce a set of resources related with a programming exercise such as exercise descriptions in multiple languages or input and output test files. The LO generation is depicted in Fig. 4. The generation of a LO package is straightforward. The Generator tool uses as input a valid PExIL instance and a program solution file and generates 1) an exercise description in a given format and language, 2) a set of test cases and feedback files and 3) a valid IMS CC manifest file. Then, a validation step is performed to verify that the generated tests cases meet the specification presented on the PExIL instance and the manifest complies with the IMS CC schema. Finally, all these files are wrapped up in a ZIP file and deployed in a Learning Objects Repository. In the following sub-subsections we present with more detail these three generations.

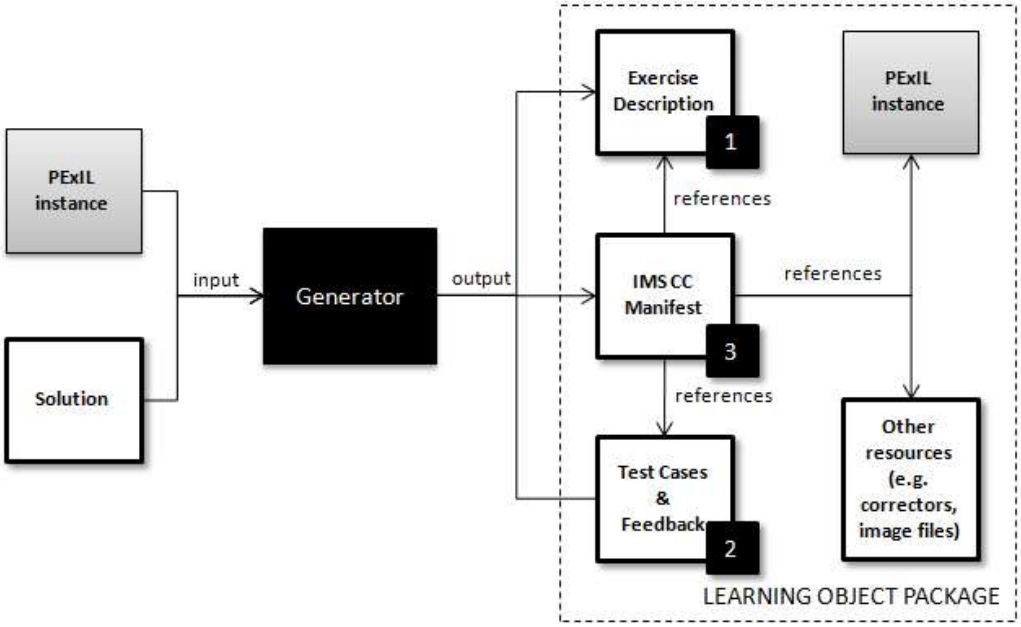

Fig. 4 *Learning Object package generation.*

Exercise description generation

For the generation of an exercise description (Fig. 5) it is important to acquire the format and the human language of the exercise description. The former is given by the Generator tool and the latter is obtained from the total number of occurrences of the `xml:lang` attribute in the `title` element of the PExIL instance.
The Generator tool receives as input a valid PExIL instance and a respective XSLT 2.0 file and uses the Saxon XSLT 2.0 processor combined with the `xsl:result-document` element to generate a set of .FO files corresponding to the human languages values founded in the `xml:lang` attribute. The following code shows an excerpt of the `Pdf.xsl` file. This stylesheet generates the .FO files based on the textual elements of a PExIL instance:

```
<xsl:template match="pexil:title">
 <xsl:variable name="uri"
       select="concat('desc',@xml:lang,'.fo')"/>
   <xsl:result-document href="resources/{$uri}">
       <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
         <!—apply templates over the textual elements --> ...
       </fo:root>
   </xsl:result-document>
</xsl:template>
```

In the next step, the .FO files are used as input to the Apache FOP formatter – an open-source and partial implementation of the W3C XSL-FO 1.0 standard - generating for each .FO file the corresponding PDF file.

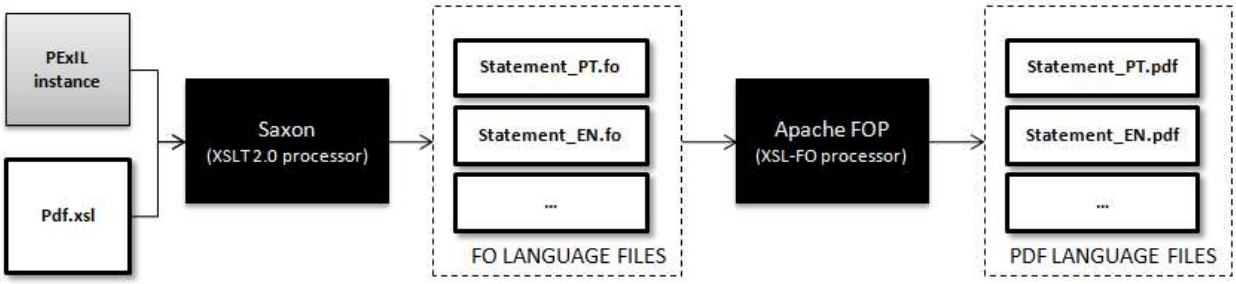

Fig. 5 *Generation of the exercise descriptions.*

The use of the PExIL definition to generate exercise descriptions does not end here since the PExIL definition is included in the LO itself making it possible, at any time of the LO life-cycle, to regenerate the exercise description in other different formats. The following figure shows a typical exercise in PDF format.

Problem C: Selfdescribing Sequence

Solomon Golomb's *selfdescribing sequence* $\langle f(1), f(2), f(3), \ldots \rangle$ is the only nondecreasing sequence of positive integers with the property that it contains exactly $f(k)$ occurrences of k for each k. A few moments thought reveals that the sequence must begin as follows:

n	1	2	3	4	5	6	7	8	9	10	11	12
$f(n)$	1	2	2	3	3	4	4	4	5	5	5	6

In this problem you are expected to write a program that calculates the value of $f(n)$ given the value of n.

Input

The input may contain multiple test cases. Each test case occupies a separate line and contains an integer n ($1 \leq n \leq 2,000,000,000$). The input terminates with a test case containing a value 0 for n and this case must not be processed.

Output

For each test case in the input output the value of $f(n)$ on a separate line.

Sample Input

```
100
9999
123456
1000000000
0
```

Sample Output

```
21
356
1684
438744
```

Fig. 6 *A typical exercise statement.*

The description also includes a description and an example of a test case. In the case of the absence of the `input/description` and `input/example` the Generator relies on the `specification` element to generate the test data and include it in the exercise description.

Test cases and feedback generation

The generation of test cases and feedback relies on the `specification` element of the PExIL definition. The Generator tool can be parameterized with a specific number of test files to generate. Regardless of this parameter, the tool calculates the number of test cases based on the total number of variables and the number of feedback messages. In the former, the number of test cases is given by the formula 2^n where the base represents the number of range limits of a variable and the exponent the total number of variables. Testing the range limits of a variable is justified since their values are usually not tested by students, thus with a high risk of failure. In the latter, the tool generates a test case for each feedback message found. The generation will depend on the successful evaluation of the XPath expression included in the `condition` attribute of the `when` element. The following example helps to understand how the Generator calculates the test cases.

```
 <line>
     <data id="n1" type="float" min="0" max="1000"/>
     <data id="n2" type="float" min="0" max="1000"/>
     <data id="n3" type="float" min="0" max="1000"/>
  </line>
  <line/>
</repeat>
<when condition="$n1>$n2">
       <feedback xml:lang="en-GB">
            Numbers can be given in descending order
       </feedback>
</when>
```

Suppose that the Generator tool is parameterized to generate 10 test cases. Using the previous example we can estimate the number of test cases and its respective input values as demonstrated in the Table 4.

Table 4. *Specification elements.*

Var.	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
n1	0	0	0	0	1000	1000	1000	1000	Min=n2+1	R
n2	0	0	1000	1000	0	0	1000	1000	N2	R
n3	0	1000	0	1000	0	1000	0	1000	R	R

The test values are: eight tests to cover the range limits of all variables ($2^3 = 8$); one test to represent the constraint included in the feedback message. Note that this test case will be executed only if the expression included in the `condition` attribute was not covered in the previous eight test cases; the remaining tests are generated randomly. Also note that whoever is creating the programming exercise can statically define new test cases and use the PExIL definition for validation purposes.

Manifest generation

An IMS CC learning object assembles resources and metadata into a distribution medium, typically a file archive in ZIP format, with its content described by a manifest file named `imsmanifest.xml` in the root level. The main sections of the manifest are: **metadata** which includes a description of the package, and **resources** which contains a list of references to other resources in the archive and dependency among them.

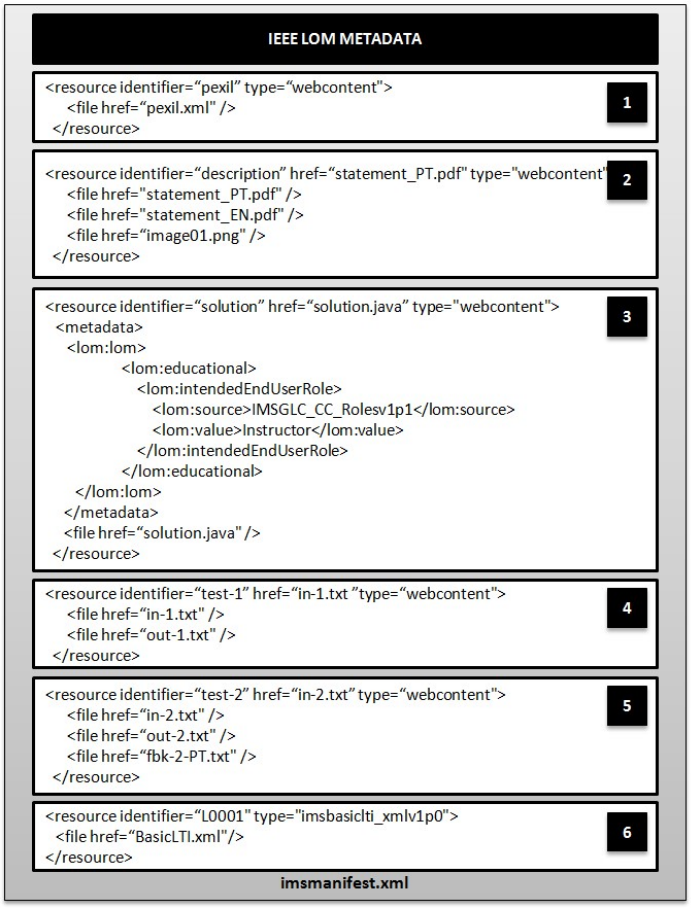

***Fig.** 7 Structure of the IMS CC manifest file.*

The **metadata section** of the IMS CC manifest comprises a hierarchy of several IEEE LOM elements organized in several categories (e.g. general, lifecycle, technical, educational). In order to achieve interoperability have defined a binding of the textual elements of the PExIL definition and the corresponding IEEE LOM elements. The Generator tool uses this binding to generate the LOM elements through a template pattern. The following table presents a binding of the PExIL textual elements and the corresponding LOM elements which will be used by the Generator tool to feed the IMS CC manifest.

***Table 5.** Binding PExIL to IEEE LOM.*

Data Type	Schema	Element path
Title	LOM	lomcc:general/lomcc:title
	PExIL	exercise/title
Date	LOM	lomcc:lifecycle/lomcc:contribute[lom:role='Author']/lom:date
	PExIL	exercise/creation/date
Author	LOM	lomcc:lifecycle/lomcc:contribute[lom:role='Author']/lom:entity
	PExIL	exercise/creation/authors/author/v:VCard/v:fn
Event	LOM	lomcc:general/lomcc:coverage
	PExIL	exercise/creation/event

By defining this set of metadata at the LOM side, eLearning systems continue to use the metadata included in the IMS CC manifest to search for programming exercises, rather than using a specialized XML dialect such as PExIL.

The **resources section** of the IMS CC manifest contains a list of references to other files in the archive (resources) and dependency among them. The `resources` element identifies a collection of `resource` elements. A `resource` is not necessarily a single file. It may be a collection of files internally referenced (within the package) or externally referenced through a URL. Internal files used by the resource are either directly enumerated by `file` elements or indirectly enumerated by using the `dependency` element to reference another resource. The `file` element may contain a `metadata` sub-element allowing content authors to describe additional metadata meaningful for searching or indexing in a repository (e.g. the file statements could have a LOM language element identifying the human language of the statement). The `dependency` element identifies a single resource (based on the `identifierref` attribute) which can be used as a container for several files that this resource depends upon. Rather than having to list all resources each time they are needed, the `dependency` element allows content authors to define a container of resources and to simply refer to that `dependency` element instead of individual resources.

In this example (Fig. 7) the resources section starts with a LAO resource (1) pointing to the PEXIL descriptor. This file is responsible for the automatic generation of all the other files included in the package (with the exception of the solution program and images). The description of the exercise is included on the manifest as a WCR resource (2). This type of resource can be automatically rendered by the browser without any additional processing. The program solution (3) is associated with metadata since this resource should not be made visible in player mode to the students and will be used only to regenerate test cases and in the evaluation phase of the programming life-cycle. The test cases are defined with a pair of input and output files (and feedback files) as resource objects (4 and 5). Finally, the BLTI link is included as a LAO resource (6). This link points to a XML file that includes all the data needed to integrate the cartridge in a LMS-web application communication. This XML file contains information to create a link in a Tool Consumer (e.g. LMS). Upon the user's click on the LMS, the execution flow passes to a Tool Provider along with contextual information about the user and Consumer. The Basic LTI link is defined in the resource section of an IMS Common Cartridge as follows:

```
<resource identifier="MyBLTILink" type="imsbasiclti_xmlv1p0">
   <file href="BasicLTI.xml"/>
</resource>
```

The `href` attribute in the resource entry refers to a file path in the cartridge that contains an XML description of the Basic LTI link. A BLTI link contains several elements. The most important are: the `title` and `description` elements contain generic information about the link; the `custom` and `extensions` elements allow the Tool Consumer to extend the basic communication data; the `launch_url` element contains the URL to which the LTI invocation is sent; the `secure_launch_url` element is the URL to use if secure HTTP is required.

The LTI message signing is performed by a security mechanism designed to protect POST and GET requests called OAuth. OAuth 1.0 specifies how to construct a base message string and then sign that string using a secret. The signature is then sent as part of the POST request and is validated by the Tool Provider using OAuth. The signing process produces a set of values added to the launch request:

```
oauth_consumer_key=b289378-f88d-2929-lmsng.school.edu
oauth_signature_method=HMAC-SHA1
oauth_timestamp=1244834250
oauth_nonce=1244834250435893000
oauth_version=1.0
oauth_signature=Xddn2A%2BjzwjgBIVYkvigaKxCdcc%3D
```

The value of the `oauth_consumer_key` depends on which credentials are being used. The `oauth_consumer_key` is passed in the message as plain text and identifies which Tool Consumer is sending the message allowing the Tool Provider to look up the appropriate secret for validation. The `oauth_consumer_secret` is used to sign the message. Both systems (TP and TC) should support and use the HMAC-SHA1 signing method with OAuth fields coming from POST parameters. Upon receipt of the POST, the TP will perform the OAuth validation using the shared secret it has stored for the `oauth_consumer_key`. The timestamp should also be validated to be within a specific time interval [11]. In order to validate the IMS CC cartridges previously generated we use the IMS validator[6]. This service validates cartridges for conformance with the IMS Common Cartridge v1.0 and/or v1.1 specification. In the validation process the IMS CC Validator tests the whole cartridge (or just the XML manifest) verifying the following type of constraints:

Static - the parameters (e.g. file names) are fixed in the profile (e.g. `imsmanifest.xml` must exist at the root of the package)
Dynamic - the parameters are taken from an instance document in the package (e.g. `href` attribute of a `resource` element must point to a QTI file)
Conditional - the constraint depends on a condition (e.g. if parameter 'contenttype' is 'question' then the `href` attribute must point to a QTI file).

The cartridges generated from PExIL instances using the methodology presented in the previous sub-section passed all tests performed by the validator.

Describing crimsonHex programming exercises

In this subsection we validate PExIL expressiveness by using the PExIL definition to cover the requirements (e.g. the input/output constraints of the exercise) of a subset of programming exercises from a learning objects repository.

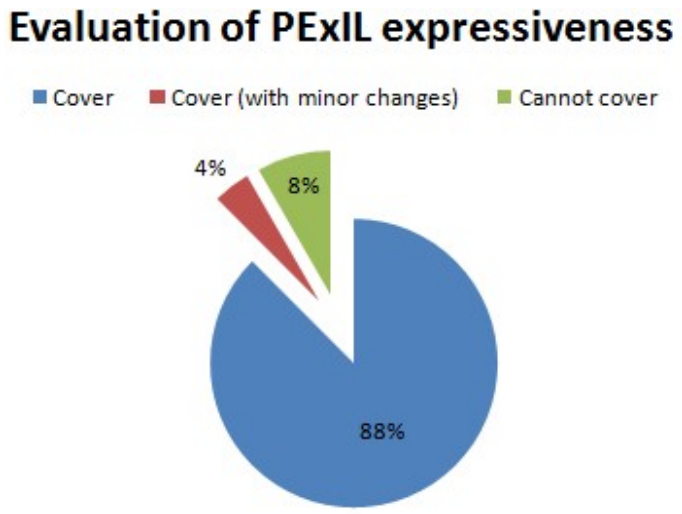

Fig. 8 *Evaluation of PExIL expressiveness.*

For the evaluation process we randomly selected 24 programming exercises (1% of a total of 2393 exercises) from a specialized repository called crimsonHex [12]. We checked manually if the PExIL definition covers all the constraints of the input/output data. The evaluation results, depicted in the Fig. 8, shows that in most cases (21 – 88%), PExIL was expressive enough to

[6] http://validator.imsglobal.org

cover the constraints of the exercise test data. In just one case, we had to make a minor change in the PExIL definition to capture alternative content models.
Finally, two exercises were not completely covered by the PExIL definition. This means that using only the standard data types of PExIL we were able to define the input and output files, and these definitions can be used to validate them. However, these definitions cannot be used to generate a meaningful set of test data. In these cases the programming exercise author would have to produce test files by some other means (either by hand or using a custom made generator). In our opinion, the data types required be these exercises are comparatively rare and do not justify their inclusion in the standard library. However, PExIL does not restrict data types and PexilUtils can be extended with generators for other data types, if this proves necessary.

CONCLUSIONS

In this paper we present PEXIL – a XML dialect for authoring LOs containing programming exercises. Nevertheless, the impact of PExIL is not confined to authoring since these documents are included in the LO itself and they contain data that can be used in its life-cycle, to present the exercise description in different formats, to regenerate test cases or to produce feedback to the student.

For evaluation purposes we validate the PExIL definition by using it as input for the generation of an IMS CC learning object package through a set of tools and by using it to capture all the constraints of a set of programming exercises stored in a learning objects repository called crimsonHex.

In its current status the PExIL schema[7] is available for test and download. Our plans are to support in a near future this definition in the crimsonHex repository. We are currently finishing the development of the generator engine to produce a LO compliant with the IMS CC specification. This tool could be used as an IDE plug-in or through command line based on a valid PExIL instance and integrated in several learning scenarios where a programming exercise may fit from curricular to competitive learning.

REFERENCES

- Friesen, N.: Interoperability & Learning Objects: Overview of eLearning Standardization". Interdisciplinary Journal of Knowledge and Learning Objects. 2005.
- IMS-QTI - IMS Question and Test Interoperability. Information Model, Version 1.2.1 Final Specification IMS GLC Inc., URL: http://www.imsglobal.org/question/index.html.
- Queirós, R. and Leal, J.P.: Defining Programming Problems as Learning Objects. In ICCEIT, October, Venice, Italy, 2009.
- IMS-CP – IMS Content Packaging, Information Model, Best Practice and Implementation Guide, Version 1.1.3 Final Specification IMS Global Learning Consortium Inc., URL: http://www.imsglobal.org/content/packaging.
- IEEE LTSC LOM Learning Technology Standards Committee. Draft Standards for Learning Object Metadata, 2002. Final 1484.12.1 LOM Draft Standard Document - http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf
- IMS Application Profile Guidelines Overview, Part 1 – Management Overview, Version 1.0. URL: http://www.imsglobal.org/ap/apv1p0/imsap_oviewv1p0.html
- ADL SCORM Overview. URL: http://www.adlnet.gov/Technologies/scorm
- Friesen, N.: Semantic and Syntactic Interoperability for Learning Object Metadata. In: Hillman, D. (ed.) Metadata in Practice. Chicago, ALA Editions, (2004)
- IMS Common Cartridge Profile, Version 1.1 Final Specification. URL: http://www.imsglobal.org/cc/index.html
- Queirós, R. and Leal, J.P.: Using the Common Cartridge profile to enhance learning content interoperability. In ECEL - 10th European Conference on e-Learning, November, Brighton, UK (to appear)
- IMS BLTI (2010) "IMS Basic Learning Tools Interoperability Specification" – v.1.0 Final Specification, URL: http://www.imsglobal.org/lti/blti/bltiv1p0/ltiBLTIimgv1p0.html
- Leal, J.P., Queirós, R.: CrimsonHex: a Service Oriented Repository of Specialised Learning Objects. In: ICEIS 2009: 11th International Conference on Enterprise Information Systems, Milan (2009).

[7] Available at http://www.dcc.fc.up.pt/~rqueiros/projects/schemaDoc/examples/pexil/pexil.html