eleed
e-learning
& education

Issue 11
2015

ProFormA: An XML-based exchange format for programming tasks

Sven Strickroth
sven.strickroth@hu-berlin.de

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin

Michael Striewe
michael.striewe@paluno.uni-due.de

Paluno – The Ruhr Institute for Software Technology
Universität Duisburg-Essen
Gerlingstraße 16
45127 Essen

Oliver Müller
oliver.mueller@tu-clausthal.de

Institut für Informatik
Technische Universität Clausthal
Julius-Albert-Str. 4
38678 Clausthal-Zellerfeld

Uta Priss
u.priss@ostfalia.de

ZeLL – Zentrum für erfolgreiches Lehren und Lernen
Ostfalia Hochschule für angewandte Wissenschaften
Am Exer 2
38302 Wolfenbüttel

Sebastian Becker
sebastian.becker@hs-hannover.de

ZSW – E-Learning Center
Hochschule Hannover
Expo Plaza 12
30539 Hannover

Oliver Rod
ol.rod@ostfalia.de

ZeLL – Zentrum für erfolgreiches Lehren und Lernen
Ostfalia Hochschule für angewandte Wissenschaften
Am Exer 2
38302 Wolfenbüttel

Robert Garmann
robert.garmann@hs-hannover.de

Fakultät IV – Wirtschaft und Informatik
Hochschule Hannover
Ricklinger Stadtweg 120
30459 Hannover

Oliver J. Bott
oliver.bott@hs-hannover.de

ZSW – E-Learning Center
Hochschule Hannover
Expo Plaza 12
30539 Hannover

Niels Pinkwart
niels.pinkwart@hu-berlin.de

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin

urn:nbn:de:0009-5-41389

Zusammenfassung

Unterstützungssysteme für die Programmierausbildung sind weit verbreitet, doch gängige Standards für den Austausch von allgemeinen (Lern-)Inhalten und Tests erfüllen nicht die speziellen Anforderungen von Programmieraufgaben wie z. B. den Umgang mit komplexen Einreichungen aus mehreren Dateien oder die Kombination verschiedener (automatischer) Bewertungsverfahren. Dadurch können Aufgaben nicht zwischen Systemen ausgetauscht werden, was aufgrund des hohen Aufwands für die Entwicklung guter Aufgaben jedoch wünschenswert wäre. In diesem Beitrag wird ein erweiterbares XML-basiertes Format zum Austausch von Programmieraufgaben vorgestellt, das bereits von mehreren Systemen prototypisch genutzt wird. Die Spezifikation des Austauschformats ist online verfügbar [PFMA].

Stichwörter: Programmierausbildung, Programmieraufgaben, Automatische Bewertung, Grader, Austauschformat, Datenformat, Lernobjekt, e-learning

Abstract

Support systems for programming education are in widespread use. However common standards for the exchange of general (learning) content and tests do not meet the special requirements of programming tasks, e.g. dealing with complex submissions consisting of multiple files or the combination of different (automatic) evaluation and assessment procedures. Thus, due to missing interoperability, programming tasks cannot be exchanged between systems easily despite the fact that this would be desirable due to the high cost of the development of good tasks. In this paper, an extensible XML-based format for the exchange of programming tasks is presented, which is already used in multiple systems. The XML-format is available on-line [PFMA].

Keywords: programming education, programming task, task package, grading support software, grader, exchange format, data format, learning object, e-learning

This article is an updated, extended, and translated version of a conference paper presented at DeLFI'14 [St14].

1 Introduction and motivation

Introductory programming courses are part of computer science and many engineering, mathematics as well as science curricula. These courses usually employ programming tasks as practical exercises where students are frequently asked to write (small) programs from scratch, to implement interfaces, or to expand existing programs for training purposes. Programming tasks are often somewhat open-ended because it is important for students to develop their own solutions [Ro07]. The construction of high quality (i.e., meaningful, engaging, and challenging) programming tasks is complex, requires creativity and takes time [SW06]. There are many aspects to consider such as learning goals, knowledge of the students and possible solution strategies. Also assessment and timely feedback play an important role as the students' submitted solutions have to be evaluated or graded. Due to the usually high number of enrolments in introductory programming classes, this results in a high workload for the teachers if done manually.

Therefore, the idea of (semi-)automatic assessment and feedback generation to programming tasks was born with early systems dating back to at least 1989 [Re89]. This reduced the manual grading effort at the expense of the development of sophisticated test procedures for a specific programming task: an automatic assessment of programming tasks is considerably complex than evaluating multiple choice questions [AR08]. In addition to the written description of each programming task, instructors usually need to create automated review, feedback, test data, or sample solutions depending on the automated grading system. Examples for automatically generated feedback are black box tests, Java JUnit tests [JUNIT] with check for various normal and "corner" cases, or comparisons with sample solutions e.g. for Prolog [Hü05].

The very high costs for creating sophisticated and creative programming tasks could be significantly reduced if tasks as well as tests were re-used, ideally independently of specific systems, and exchanged between teachers, e.g. by having a shared pool of programming tasks for a variety of learning scenarios. Although the development of support systems for computer education and automated graders in various programming languages has progressed for well over ten years (cf. [Ih10] for an overview and [SMB11, SOP11, PJR12, RRH12, Stö13] for examples of recent progress), there is no common exchange format yet. Also, only a few systems, such as DUESIE, eAIXESSOR, JACK, Mooshak and UVa Online Judge have or plan to have an easily accessible pool of re-usable tasks.

In particular, in systems with multiple installations, an import/export function could at least reduce the workload by re-using tasks within these systems. Nevertheless, import or export functionality is currently rare. It is only mentioned in publications about JACK, Praktomat, Mooshak and ELP. In other disciplines, e.g. math or engineering, there are established learning management systems (LMS) such as LON-CAPA [LONC] in which the worldwide

exchange of tasks and learning material is a central component. In this paper we propose an interoperable XML-based exchange format (called ProFormA; based on a project name “Programming tasks for Formative Assessment”) for programming tasks to fill this gap.

The remainder of this paper is organized as follows: In section 2 three different systems for programming education used by the authors are described. Section 3 presents requirements for a generic exchange format based on a literature review on published systems, existing exchange specifications and their support for the derived requirements. Based on these requirements the ProFormA exchange format is introduced in section 4 and discussed in the subsequent section. The last two sections suggest some advanced usage scenarios, future work, and draw a conclusion.

2 Three examples of programming assessment systems

This section describes three different support systems for programming education which have been in use by the authors for several years. Each system has an isolated pool of programming assignments. The purpose of this section is to point out the different usage scenarios and goals of these systems which motivated us to work on a common exchange format for programming tasks.

2.1 GATE

The Generic Assessment & Testing Environment (GATE) [SOP11, GATE] is a web-based, open-source (GPLv3) stand-alone system which was developed at Clausthal University of Technology in 2009. The goal for the development of this system was to improve the programming training in Java and to support human tutors by reducing the manual correction and semi-manual scoring effort in large lectures with several hundred students. Additionally, the system allows for managing exercise groups. GATE is in regular use for introductory programming classes for students studying computer science and economics since 2009 and also for related computer science classes. In the latter case it is used as a submission and grading system only, i.e. without employing all programming related features.

With GATE, students can submit their solutions in digital form which can then be corrected and scored by human tutors within the system. In order to simplify the correction process for the tutors, it is possible to define several automatically runnable tests for programming tasks inspecting the solutions of the students which are executed automatically after the submission deadline. Thus, tutors do not only see the submitted program code, but also the test results. Here, compile tests and function tests (JUnit and black-box tests) for checking the correctness of the submitted Java programs are available. It is also possible to allow students to request specific approved tests: Students can get automatically generated feedback in order to improve their solutions before the submission deadline. GATE also includes plagiarism detection algorithms (e.g. Levenshtein distance for short programs and Plaggie [ASR06] for more advanced Java programs) to help tutors find instances of plagiarism within all submitted solutions especially across exercise groups. Additional feedback by tutors to students can be given through a free-text comment function. The

same comment function can be used by tutors to send comments about student submissions to other tutors. Thus, it is possible to inform other tutors about a suspicion of plagiarism or tutors can ask other tutors for help on correcting and scoring a submission.

2.2 Praktomat

Praktomat [KSZ02, PRAK] is a stand-alone, open-source (GPLv2), web-based grading system for programming exercises. Its development started in 1998 at Passau University, Germany. The source code was freely available since their first publication. Starting from 2011 Praktomat was redesigned and consecutively developed at the Karlsruhe Institute of Technology (KIT), Germany.

Due to its modular test architecture, Praktomat supports numerous programming languages such as Java, C++, Haskell, and Isabelle. Tests usually consist of a compile test and black-box tests using the DejaGnu testing framework [DGNU]. Especially for the Java programing language there are JUnit tests and also Checkstyle tests for testing the adoption of style conventions available.

At Ostfalia University the Praktomat was extended by adding support for Python and databases with SQL [KJ13]. Furthermore, the Praktomat has been embedded into an LMS (LON-CAPA) so that it only serves as a hidden backend grading engine which is fully configured and used via its LMS front-end [PJR12a]. It is currently employed in courses for teaching linear algebra (using Python), databases and introductory Java where it provides formative assessment to students.

2.3 JACK

JACK is a web-based tutoring and assessment system that can be used for programming exercises in Java [SBG09, JACK]. It is able to provide automated grading as well as generation of individual textual and graphical feedback for each solution submitted to an exercise. It has been in use both for formative self-training and summative assignments for first-year-students at the University of Duisburg-Essen since 2006 [SG13]. Students can access JACK to download exercises and submit solution files either via a web-browser or via a customized version of the Eclipse IDE. Feedback provided by JACK can be viewed only via a web-browser.

With JACK, teachers define exercises via a web-browser by providing exercise meta-data and three different types of files: Files to be edited by the students in order to solve the exercise, files to be used but not changed by the students, and files with internal data not visible to the students. Moreover, JACK allows teachers to define different analyses on programming exercises that may or may not make a weighted contribution to the overall grade for a solution and produce textual or graphical feedback: (1) Static checks include compiler checks and rule based analysis of source code to look for desired or forbidden code structures. Typically, solutions receive full points in static tests unless errors are found. In case of errors, appropriate messages are provided as textual feedback. (2) Dynamic checks run test cases, collect tracing data from each executed program step, and compare program outputs to expected outputs. Typically, solutions gain points for each

passed test case. In case of errors, appropriate feedback messages can be provided for each test case as well as a trace table for this test case [SG11]. (3) Metric checks calculate some software metrics for solutions. Typically, they do not contribute to the grades, but provide additional information on the solution that is presented as textual feedback. (4) Visualizations can be created for object structures created during the execution of test cases [SG10]. Typically, they also do not contribute to the grades, but help to illustrate problems in form of graphical feedback.

3 Requirements and existing formats for the exchange of programming tasks

In this section we first present requirements for a generic exchange format for programming tasks which were derived from functionalities of existing systems. Three systems and their features were already presented in the previous section in more detail. These three systems already provide some insight into the spectrum of programming assessment and learning systems. For a generic exchange format, however, we need to systematically analyze a much broader spectrum of systems in order to derive their common requirements. Many systems have been published in a scientific context and there are even more systems in practical use which have not yet been extensively documented. Needless to say that we can only systematically review the published systems and, thus, we conducted a review on published programming assessment and learning systems which deal with source code written by students. The main criterion was that a system had to have been published since 2000 at scientific conferences or in journals related to learning science and technology. We also considered technical reports that have passed some kind of review process and have been published by their universities.

In the second part of this section, we then compare the derived requirements to existing import/export and exchange formats.

3.1 Requirements for an exchange format for programming tasks

Programming tasks can be of very different size and complexity, and they can also be designed in many different ways: In simple cases, some lines of code have to be filled into a predefined code skeleton (often called “fill-in-the-gap”). In complex cases, complete functions or classes have to be created that adhere to interface specifications given in a task description. Depending on a programming language and programming environment, file and folder structures have to be managed by the students as well.

With the exception of the simplest cases, programming tasks are not closed questions, in which the correct answer is known beforehand and in which any solution can easily be graded by comparison to a known sample solution. In fact, many programming tasks are open questions for which no single solution or solution strategy exist [LLP13]. For open questions, however, each submission can still be analyzed and graded with respect to different criteria. For example, a static analysis can be applied to the submitted source code in order to check syntax, use of specific constructs, or programming style. A dynamic analysis can be applied as well by using unit or black-box tests if the interface is properly

described in the assignment description [Al05]. In order to execute such analyses, different tools or environments as well as configuration files that are not visible to the students may be necessary.

The functionality that is offered by existing tools often depends on the priorities and didactic aims of the specific tool. Consequently, many different ways of defining tasks can be found among existing systems. A useful exchange format, thus, has to cover a superset of all the requirements reflected by different tools. Table 1 presents an overview of 33 tools which sufficiently span the requirements space, and the requirements derived from the main tool features. A similar, but smaller, analysis on a more technical level with similar results was performed by [QL13]. Based on our analysis, a common exchange format needs task designers to be able to specify

(R1) the description of the assignment/problem and the title;

(R2) which code skeletons are provided to the student for downloading or in a web-based editor, and expected to be included in a submission;

(R3) which files are provided to the student for reference without expecting them in a submission (e.g. code libraries);

(R4) in which structure and form (e.g. single files, zip archives, etc.) files are provided and requested;

(R5) which additional files not visible to the student are attached to the task (e.g. test drivers);

(R6) which checks and analyses are defined for this task and which conditions have to be fulfilled to apply them;

(R7) which files contained in the task and of the particular solution are relevant for which check or analysis;

(R8) which additional non-technical information (e.g. grading scheme, deadlines) are attached to the task;

(R9) what a sample solution to the task looks like.

Requirements R2 to R4 are all related to the file handling between system and students. Only few systems make explicit requirements with respect to this category, while most systems accept arbitrary file attachments in submissions. Despite the fact that only some systems explicitly model code skeletons (e.g. ELP and WPAS; R2) and reference files (R3), some systems indeed implicitly make use of them: eClaus, for instance, does not have a special field for code skeletons, however, it allows to link external files and embed images in task descriptions (seen in [WW07] Figure 1). Also BOSS(2) allows to upload assignment resources, however, it is not clear if those relate to R2 and/or R3.

Covering the core functionality such as task description, grading schemes, and analysis configuration, the requirements R1, R5, and R8 are common to many systems. Unsurprisingly, there is no system which does not provide a task description (R1). ELP and Web-CAT are remarkable exceptions with respect to R5, as they use tests based on comparisons to sample solutions or tests submitted by the students, respectively. Non-

technical information (R8) is handled very differently across the systems: Some systems allow constructing very specific grading schemes (e.g. JACK) while others just rely on the number of failed tests (e.g. Praktomat).

Requirements R6, R7, and R9 are explicitly stated by one fifth to nearly half of all systems in our review. From the point of view of a particular system, these requirements can be considered optional. Many, but not all systems offer different analysis techniques, so configuration of individual checks may not be necessary if only a fixed set of analyses is offered which are always performed. In particular, R7 is concerned with performance of checks and, thus, irrelevant for systems that do not handle large additional files that are only relevant for particular checks at all. In turn, sample solutions tackled by R9 are used only in a few cases to generate actual tests as discussed above, but used as supplementary and, thus, optional feedback in other systems. For example, they can be requested by students after submitting their solutions in AutoGrader, JOP, and ELP. In CourseMarker/CourseMaster it is possible for students to execute the teacher's solution with their own test/input data. CourseMarker/CourseMaster, eduComponents, Ludwig, and ViPS use the output of a sample solution in order to compare it to the output of the students' solutions based on pre-generated or random input data. In Web-CAT and ProgTest the students' JUnit tests are tested against a sample solution as well as the students' own solutions.

Table 1 classifies the tools with respect to the nine requirements. For the purpose of completing the table we considered the case of a programming task author who is aiming at getting the optimal functionality out of the used tool. A cell containing an "x" in column (Rn) means that the task author using the respective tool must be able to specify information (Rn) in order to fully utilize that tool. An "x" does not define whether the information (Rn) is mandatory for using the tool because sometimes tools replace missing information with defaults. As a result, Table 1 provides an overview of which requirements should be covered by a common task exchange format.

System	R1	R2	R3	R4	R5	R6	R7	R8	R9
Apogee [Fu08]	X		X		X			X	
ASB [Mo07]	X			X	X	X		X	
AutoGrader [No07]	X							X	X
BOSS(2) [JGB05]	X	T	T	X	X	X	X	X	
CourseMarker/ CourseMaster [Hi03]	X	X			X			X	X
DUESIE [HQW08]	X				X	X		X	
eAixessor [AS-S08]	X				X	X	X	X	
eClaus [WW07]	X	T	T		X	X		X	X
eduComponents [APR06]	X				X	X		X	X
ELP [Tr07]	X	X						X	X
GATE [SOP11]	X	X	X	X	X	X	X	X	(X)
Graja [Stö13]	X		X	X	X			X	
JACK [SBG09]	X	X	X	X	X	X	X	X	
JOP [Ei03]	X	X			X	X	X		X
Ludwig [Sh05]	X				X	X			X
Marmoset [Sp06]	X				X			X	
MOE [Ma09]	X				X	X		X	
Mooshak [LS03]	X				X			X	X
Oto [Tr08]	X				X	X		X	
Praktomat [KSZ02]	X			(X)	X	X	X	X	(X)
ProgTest [SMB11]	X				X	X		X	X
Quiver [EFK04]	X	X			X	X		X	
RoboProf [DH04]	X				X				

SAC [Au08]	X				X				
UVa OnlineJudge [RML08]	X		X		X			X	
ViPS [Hü05]	X				X			X	X
VPL [RRH12]	X	X		X	X	X		X	
Web-CAT [Ed04]	X				(X)		(X)	X	X
WebTasks [RH08]	X				X				
WeBWorK-JAG [GSW07]	X	X			X			X	
WPAS [HW08]	X	X							
xLx [SVW06]	X		X		X			X	X

Table 1: Different systems for automated assessment of programming tasks and their requirements with respect to task specifications. Empty cells indicate that no information is available on whether the system states the respective requirement. Markers in parentheses indicate requirements that depend on the tool version. "T" marks special aspects which are described in the text.

3.2 Existing format specifications

Proprietary import and export functionality for tasks and tests is provided by some systems. However, these are based solely on the requirements of "their" systems and, thus, can only be re-imported into instances of the same system. For example, in the ELP system, fill-in-the-gap tasks for Java and C++ including hints and solution can be specified, but an XML DTD or XML schema definition is not available. Mooshak provides its own exchange format which is based on an XML manifest referring file resources for problem statement, images, input/output data, and sample solutions. JACK and Praktomat allow the export and import of tasks in dedicated XML formats. No export functionality, but an XML specification for Java black-box testing is provided by eClaus. Here, tests are not programmed as JUnit tests, but have to be represented in XML.

As a response to the various system-specific task specifications and the lack of a widely accepted standard, Queirós and Leal [QL13] proposed an extensible converter of different (system-specific) programming exercises formats. However, it is limited to exercises defined in the Mooshak and Peach Exchange Format [QL13].

Rather than dealing with system-specific task formats and specifications, there are also approaches for system-independent formats: The most widely used standard for learning objects is the IMS Content Packaging (IMS CP, [IMSCP]) of the IMC Global Learning Consortiums [LQ09]. This specification describes how learning contents can be combined and packaged (in a ZIP-file with a special manifest). Based on this generic specification,

standards such as “Sharable Content Object Reference Model” (SCORM, [SCORM]) and “Question and Test Interoperability Specification” (QTI, [IMSQT]), a specification by Cesarini et al. [CMM04] as well as two specifications developed by Leal and Queirós [LQ09, QL11] especially for programming tasks have been proposed.

The SCORM reference model is restricted to exchangeable electronic learning content and specifies a runtime environment (RTE) for LMS, content aggregations and navigation as well as orders for the presentation of the content. The RTE only provides get and set methods for basic variables for the communication of the learning resources with the LMS (e.g. in order to store the learning progress). Therefore, it does not fulfill the requirements presented in section 3.1. QTI, however, allows the exchange and automatic evaluation of closed question types. Half-open question types (i.e., essays) can be modeled, however, no automatic assessment is possible. As this standard was not designed for programming tasks it only partly fulfills requirements R1, R2 and R8.

Cesarini et al. [CMM04] developed a specification for learning objects which also includes programming tasks and can be automatically assessed by so-called “Test-Engines”. This partially fulfills R1, R6 and R8. However, this specification allows only one test per learning object and no automatic assessment of programming tasks.

The first approach of Leal and Queirós [LQ09] was developed in the context of a project funded by the European Commission and allows to specify and to save programming tasks including tests as learning objects as an extension to the IMS CP metadata standard called “EduJudge Metadata”. The specification, however, was designed for only a single, fixed assessment engine (EduJudge/Mooshak, [Ve11]) and fulfills our requirements R3, R5, R6 and R8 just partially. The website of the project is not accessible anymore and there is no current information about the specification and usage available.

The second approach of Queirós and Leal [QL11] specifies the “Programming Exercises Interoperability Language” (PExIL). The goal of their specification was to model the whole life-cycle of programming tasks – starting with the problem formulation up to assessment and feedback. By using PExIL, tasks and tests (including commands for compilation and execution) for multiple, fixed, predetermined, (imperative) programming languages can be modeled. However, only the requirements R1, R2, R5, R6, R8 and R9 are satisfied. Tests as well as input data are encoded directly in XML and are, therefore, dependent on a specific evaluation-engine which seems to support black-box tests only. Moreover, PExIL seems to support tasks which only consist of a single solution-file. As a special feature to be noted, this format also allows to model feedback and conditions where specific feedback should be displayed, e.g. starting from a third attempt. An XML schema definition is available, but insufficiently documented. The only system supporting this specification is PETCHA [QL12] which was developed by the very same authors as the specification.

Independently of IMS CP the Peach Exchange Format [Ve08] for “Programming Contest” systems was developed. This format has a strong dependency on the Peach system. It supports different programming languages for tasks, however, only pure black-box tests (input/output tests, script-controlled) are supported. The requirements R2, R3, R4, R7 and R9 are not or only partially fulfilled.

Consequently, up to now there is no universal exchange format for programming tasks that satisfies the requirements derived in section 3.1 and that can be used to exchange programming tasks including test descriptions between systems.

4 An XML exchange format for programming tasks

For each task the ProFormA format, described in this section of the paper, specifies a description (including its language for the purpose of internationalization), its programming language, supplied files, technical details of the submission, sample/model solutions and optional hints for marking and meta-data. A task can contain several tests where each has a test type (e.g. “unittest”) and a test configuration. Tests are often executed by standard software engineering tools such as compilers, unit tests, style checkers and debugging tools, and supplied in the format required by such tools. Therefore, the core ProFormA format does not need to specify the details of the content of such tests but only the parameters of their execution, for example tool names and version numbers. A system which imports files in the ProFormA format can determine whether or not the files are compatible with the system’s requirements by checking the test types.

The ProFormA format is XML-based and specified using an XML schema definition (cf. [PFMA]). The main reasons for using XML are that it is a widely accepted and established format for interoperability, that there are parsers available for various if not all programming languages, and that it is human as well as machine-readable which is helpful for debugging and adaptability. Furthermore, an XML schema definition allows to precisely specify all aspects of possible XML documents (e.g. fine grained data types, unique key constraints and logical structures), to validate XML documents if they comply to a specification and supports the usage of different XML namespaces.

Figure 1 depicts the structure of the ProFormA format and the nesting of elements and attributes. For referencing elements, XPath expressions are used. Elements which have three lines in the upper left corner correspond to XML simple elements, usually containing plain text, e.g. */task/description*. Optional elements are surrounded by dashed lines (e.g. */task/grading-hints*). Octagonal boxes symbolize XML sequences. Their cardinalities are provided unless they equal 1. The boxes with “any ##other” refer to possible extensions of the format. The ProFormA format only formalizes the common denominator of programming tasks. However, it can be extended to include tests from any other tool or programming language by providing “hot spots” in the sense of software frameworks which can be filled using format extensions defined in another XML namespace. The ProFormA format is therefore sufficiently flexible to incorporate local, system-specific configurations and novel testing tools. All specifications are versioned via their XML namespace URI in order to support future changes and extensions. For example “urn:proforma:task:v0.9.4” shows that this is the official ProFormA “task” specification in version “0.9.4”.

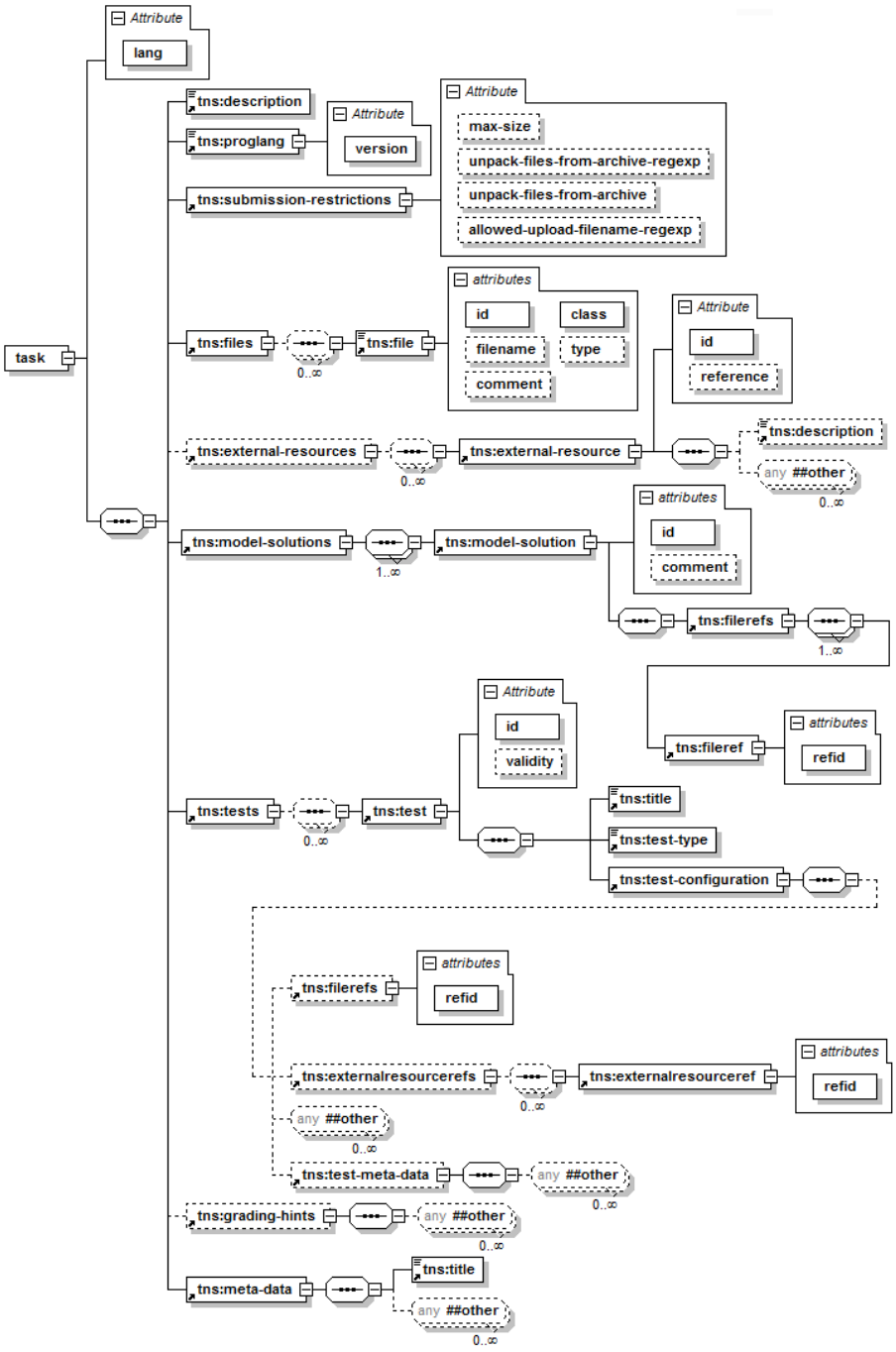

Figure 1: Structure tree of the ProFormA exchange format XML schema definition (version 0.9.4)

A task specification using the ProFormA format can either be a standalone XML document or a special layouted ZIP-archive whereby the XML document describing a programming task has to be named “task.xml” and placed in the root directory of the ZIP-archive (similar to the manifest file in IMS CP).

Some selected elements are discussed below in more detail, in particular with respect to the requirements from section 3.1. Listing 1 shows a minimal example of a task described in the ProFormA format. The task in this example consists of calculating the n-th Fibonacci number via loops or recursion, as often used in introductory Java classes.

Being required by most systems, file attachments (R2, R3, R5 and R9) and their handling in the ProFormA format are discussed first: For every file attached to a task there is a *file* element in */task/files*. Thus, all attached files are defined at a central place within the XML document. A *file* element either holds the plain text contents of a file (cf. Listing 1 line 11ff) or, if the task is packed in a ZIP-archive as described above, a filename reference to a file in the ZIP-archive (cf. Figure 2). Both cases are distinguished by the *type* attribute which either contains the value “embedded” as a default or “file”. Especially for the embedded file case there is an optional *filename* attribute in order to specify how the file should be named when used. This is for example required for Java where the class name “defines” the filename and vice versa. Additionally there is a *class* attribute which specifies the intented access rights and visibility for the file (R2, R3, R5). The ProFormA format provides the following file classes: (Code-)templates/skeletons (*template*), libraries (*library*), input data (*inputdata*), additional information or instructions to process a task (*instruction*), internal libraries (e.g. libraries required for tests, *internal-library*) and internal files (e.g. test drivers or model solutions, *internal*). Especially the latter two are classes of files which should not be accessible to learners by default.

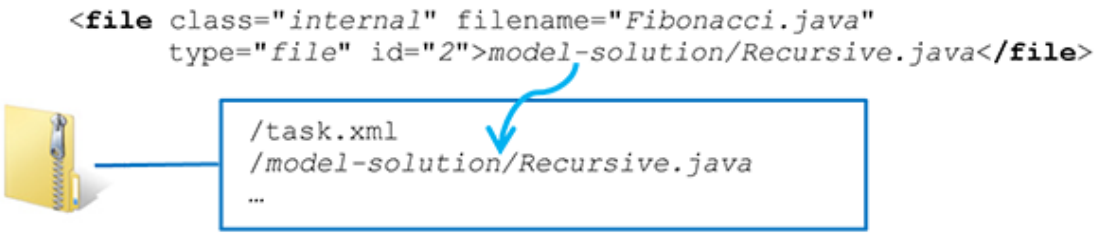

Figure 2: Example of files embedded into a ZIP-archive

Every file has a unique ID for referencing purposes (*id* attribute). In this manner a file can be re-used in multiple contexts without the need to define multiple times (R7). Referencing is done using a filerefs element containing one or more *fileref* elements – one for every file reference. Eventually, a *fileref* element has a *refid* attribute which references the file ID of a specific file (*/task/files/file*).

```
1  <p:task lang="en" xmlns:p="urn:proforma:task:v0.9.4">
2    <p:description>Calculate the n-th Fibonacci number. As a reminder:
3    The first two Fibonacci numbers are 0 and 1. The following numbers
4    are the sum of the previous two numbers (0, 1, 1, 2, 3, 5, 8, 13,
5    21...). The class should be named Fibonacci and the method to be
6    written fibonacci has an int as an input parameter.</p:description>
7    <p:proglang version="1.4">java</p:proglang>
8    <p:submission-restrictions allowed-upload-filename-
9    regexp="Fibonacci\.java" />
10   <p:files>
11     <p:file class="internal" filename="FibonacciTest.java" id="f1">
12       import junit.framework.TestCase;
13       public class FibonacciTest extends TestCase {
14         public void testPos() {
15           assertEquals(13, Fibonacci.fibonacci(7));
16         }
17         public void testNull() {
18           assertEquals(0, Fibonacci.fibonacci(0));
19         }
20       }
21     </p:file>
22     <p:file class="internal" filename="Fibonacci.java" id="f2">
23       public class Fibonacci {
24         public int fibonacci(int i) {
25           if (i &lt;= 0) return 0;
26           else if (i == 1) return 1;
27           return fibonacci(i - 2) + fibonacci(i - 1);
28         }
29       }
30     </p:file>
31   </p:files>
32   <p:model-solutions>
33     <p:model-solution id="m1">
34       <p:filerefs>
35         <p:fileref refid="f2" />
36       </p:filerefs>
37     </p:model-solution>
38   </p:model-solutions>
39   <p:tests>
40     <p:test id="t1">
41       <p:title>Compilation test</p:title>
42       <p:test-type>java-compilation</p:test-type>
43       <p:test-configuration />
44     </p:test>
45     <p:test id="t2">
46       <p:title>Calculation test</p:title>
47       <p:test-type>unittest</p:test-type>
48       <p:test-configuration xmlns:u="urn:proforma:tests:unittest:v1">
49         <p:filerefs>
50           <p:fileref refid="f1" />
51         </p:filerefs>
52         <u:unittest framework="JUnit" version="3">
53           <u:main-class>FibonacciTest</u:main-class>
54         </u:unittest>
55       </p:test-configuration>
56     </p:test>
57   </p:tests>
58   <p:meta-data>
59     <p:title>Calculation of n-th Fibonacci number</p:title>
60   </p:meta-data>
61 </p:task>
```

Listing 1: Example of a programming task specified in the ProFormA format version 0.9.4

Being semantically different from “normal” single files, model solutions (R9) are defined within the */task/model-solutions* element. For each model solution there is one *model-solution* element which must have an unique ID (*id* attribute). It can also contain a human-readable comment describing the model-solution (*comment* attribute). This is useful if several model solutions are provided in order to point out their differences (e.g. an iterative and a recursive implementation). A model solution refers to at least one file. This is achieved using the already described *filerefs/fileref* elements where the *filerefs* element is nested directly under the *model-solution* element (see lines 34ff in Listing 1). Furthermore,

there must be at least one model solution defined for each task. There are several reasons for this decision. First, on the technical level, there are systems which must have a model solution in order to compile JUnit tests provided as source code (e.g. GATE). In addition, there are practical reasons, because a sample solution together with the task description allows teachers to rate the difficulty and appropriateness of a task for the usage in their own classes.

The next important aspect of the ProFormA format is the definition and configuration of tests and evaluation methods (R5, R6 and R7): Tests can be defined in */task/tests/test* elements (R6, see Listing 1 lines 39ff). In addition to specifying which kinds of tests are available for a task (for example compile/syntax or JUnit tests, *test/test-type*) it is possible to specify the configuration of a test (*test/test-configuration*). The definition of test types is also part of the ProFormA format in order to provide a shared set of possible types which are consistently interpreted. However, it is still possible to extend the format by defining personal test types. Extra files needed for a test such as test drivers are defined within */ task/files/file* elements. They are referenced using their unique IDs within *filerefs/fileref* elements (R5). In this manner it is known in advance which files are required for a test and which are not (R7). In order to be able to exchange test type specific parameters such as the name of the main class for a JUnit test, separate XML namespaces for every test type within the *test-configuration* element can be used. The *test-configuration/test-meta-data* element allows to use special namespaces to define system-specific meta-data which belongs to a test (e.g. restrictions for tests).

As already stated above, test types use their personal XML namespace for configuration. The test types “java-compilation” and “unittest” have been specified so far. The first official type “java-compilation” (Listing 1 lines 42f) does not need any special configuration for a minimal instance because the libraries to be used can be detected by referenced files and *class* of the files. JUnit tests, however, need a special configuration at least for the canonical name of the main class which should be executed and should ideally describe the version of the JUnit framework. Therefore, the *unittest/main-class* element in the XML namespace “urn:proforma:tests:unittest:v1” was defined (Listing 1 lines 47ff). There is nothing else needed for a minimal configuration as the system knows how to run the JUnit tests using the main class file and the referenced files containing the JUnit test cases.

Tasks should usually be self-contained as only this condition guarantees that a task is usable in the long-term without losing or missing components. In rare cases the use of external resources is unavoidable. For example, an automatically graded task may need a huge database dump or a large library that should not be bundled reasonably with the task itself (R2, R3, R5). For these cases the */task/external-resources* element was added so that a task may reference the required external resources by a unique name (attribute *reference*). The semantics and format of external references are currently not standardized in the ProFormA format. An example for an external resource is an URL of a widely known database dump (e.g. ftp://ftp.fu-berlin.de/pub/misc/movies/database/). More complex references can be included in a personal XML namespace. As an example consider a Maven coordinate tuple identifying the name and version range of a library (a so called “GAV”) consisting of a small XML snippet, e.g. “<groupId>org.jmock </groupId><artifactId>jmock-junit4</artifactId><version>[2.4.0, 2.5.0]</version>”. The main reason for the inclusion of external references in the current generic form is the usage scenario described in section 6.

The */task/submission-restrictions* element provides the option to define restrictions for submissions of task solutions. For example, it is possible to specify the maximally allowed file size (in byte) for a solution file which should be uploaded into the system (attribute *max-size*). Of course, while this limit represents an expected maximum by the author of a specific task, a system limitation can still be stricter (e.g. for security purposes). Constraints for filenames can be specified via regular expressions using the attribute *allowed-upload-filename-regexp* (line 8 in Listing 1). This is important for Java tasks where a class name determines the filename when using the default class loader. If learners want to upload a solution file for a task into a system, the system first verifies if all constraints of the specified regular expression are satisfied by the filename. If this is not the case the system can prevent learners from uploading their solution files and/or provide a warning. This is intended to ensure that tests will not fail because of an invalid filename (R4).

An instance of the */task/description* element includes the task's description (R1, Listing 1 lines 2ff). Furthermore, it is possible to provide a grading scheme (*/task/grading-hints*) and other meta-data (*/task/meta-data*). As existing systems use very different ways for defining grading schemes (cf. JACK, ViPS, Web-CAT) and grading schemes are quite personal, there is no common specification for this and the element was called "grading-hints". In analogy to the *test-meta-data* element, a personal XML namespace in the */task/grading-hints* element can be used to define system-specific details about grading schemes. The ProFormA format also offers the opportunity to specify attributes belonging to tasks or tests which are not relevant for all but for certain systems, to make sure that system specific information does not get lost on a task export. In this manner, a complete task export into the ProFormA format with subsequent lossless re-import of the task into the same system can be guaranteed. Furthermore, this ensures that only data is imported into a system which this system can handle (R8).

5 Compatibility and discussion

When discussing an interchange standard such as ProFormA, interesting questions are which aspects to include and, finally, how to model them. The first question has two extreme cases: On the one side there are specific aspects such as title, description, and test configurations which are required by all or most systems and on the other side there are specific aspects which are only used by one or very few systems such as sample solutions or reference files (cf. Table 1). The first case is quite clear; those aspects are important and, thus, have to be included. Elements of the latter extreme case are unlikely to be included as required features in the core format specification. All aspects between the two extremes are subject to discussion. The second question is always subject to discussion in order to achieve justified design decisions.

Our proposed specification is supported by the three systems, GATE, JACK, and an enhanced Praktomat version, described in section 2, for exporting and importing programming tasks. The initial focus on Java tasks is due to the fact that the three systems used by the authors mainly deal with Java tasks and also most systems of our review support Java (cf. Table 2 in the appendix for the concrete systems). Furthermore, half of the systems explicitly state that JUnit is used (cf. Table 2). Only 11 out of the 33 reviewed systems state explicitly or indirectly that JUnit is not used. Oftentimes these systems do not deal with Java or use other testing methods (cf. Table 2). The ProFormA exchange format

has been implemented and tested by the tree systems for more than one year. However, the design of the format is not solely based on these three systems but also on requirements of other systems, especially ViPS using Prolog and LON-CAPA external response [LONCM] as well as others mentioned in section 3.1. This ensures that the specification does not only implement requirements of a single system or only supports some programming languages as it is the case for most existing specifications (cf. section 3.2). In addition, the ProFormA format is not designed as a fixed “one-size-fits-all” specification. It does not ignore system-specific aspects, but those can be embedded with freely definable XML namespaces as extensions. Extensions for individual systems, new programming languages as well as novel test types are supported. Extensibility using XML namespaces, however, harbors the risk that fundamental aspects are defined multiple times or in different ways. In order to mitigate this risk, on the one hand, we believe that all fundamental aspects have already been defined in the ProFormA format itself and, on the other hand, the proposed format is versioned. Therefore, changes, additions and moving of “wide-spread” aspects into the core format are possible without losing compatibility for existing tasks.

Metadata for categorization was deliberately not included in the first version of the specification. For metadata, there are, on the one hand, established standards such as IEEE Learning Objects Metadata (LOM) and Dublin Core and, on the other hand, these are often not fully used by authors of learning material [Go04], possibly due to motivational aspects because the author who creates the meta-data is usually not the person who benefits from it. Instead, at least one sample solution is required in the ProFormA format. This should allow potential users of tasks to rate them according to difficulty and appropriateness, thereby judging whether or not a task fits into their course.

Restrictions for filenames of requested files (R4) are a requirement of just some systems. However, we decided to include this into the format specification (*/task/submission-restrictions element*) with the optional *allowed-upload-filename-regexp* attribute. The rationale is that this does not cause harm if ignored or unsupported by systems, but can help to prevent failures of tests (see section 4). An alternative way which explicitly specifies all required filenames ensuring that no file is missing (as used in e.g. JACK) is under discussion. The explicit statement of filenames can easily be converted using the existing *allowed-upload-filename-regexp* attribute, however, the opposite direction is not possible.

A special editor is, in principle, not needed for the ProFormA format, since each system already provides its own authoring functionality. However, the development of a system-independent editor is planned which can be extended by plugins to support system-specific aspects – especially with respect to the usage scenario described in section 6. Also, a dedicated editor would allow for creating tasks system-independently.

In addition to the export functionality, support for importing tasks has been implemented for the three systems described in section 2. In this context, cross-system imports, which export from one system and import into another system, and same-system imports/exports were carried out for testing purposes for these tree systems. In principle an export is possible from all three systems (GATE, JACK and the enhanced Praktomat). However, in the JACK system there are no model solutions available and, thus, the necessary model solutions are not yet exported up to now. GATE and Praktomat generate fully valid XML

documents complying with version 0.9.1 of the specification. Support for the new version 0.9.4 is currently being implemented for these systems and is already available in Praktomat.

A same-system and inter-system import is possible for each of the three systems in principle. However, it is not always guaranteed that all features of an imported task which was previously exported from another system are fully re-usable. On the one hand, this is due to the supported assessment procedures of the importing vs. exporting system and, on the other hand, because of the special system features which separate different systems. For example, all three systems are used for Java programming tasks. GATE currently only supports JUnit 3 tests and Praktomat supports several JUnit versions (hence the use of JUnit3 in Listing 1). Therefore, syntax and JUnit3 test procedures can be shared between these two systems without losing functionality. The JACK system, however, does not support JUnit tests. It relies on other methods which are currently exclusive to the JACK system. Nevertheless, system-specific elements can be exported in the form of metadata using the elements */task/meta-data* and */task/tests/test/test-configuration/test-meta-data*. As already mentioned above, the essential elements of programming tasks are shareable across systems, and also same-system imports are possible without losing information.

6 Integration of learning management systems and programming support systems

Learning management systems such as LON-CAPA [LONC], Stud.IP [STDIP], OLAT [OLAT], or Moodle [MOOD] are widely used in educational contexts and have their own user management, tools for communication, such as e-mail or forums, and sophisticated course management features. These tools can be seen as general purpose support tools in this context. Hence, they usually cannot assess computer science exercises or programming tasks. Many of the systems mentioned in section 3.1 are well-suited for this purpose (depending on their particular goals). Nevertheless, nearly all of the systems seem to be stand-alone (cf. Table 2 in the appendix), having their own user management but usually without sophisticated course management features. Thus, these programming assessment systems are called “grading systems” (GS) in this section as the grading functionality is the focus. Additionally, GSs tend to be specialized for the assessment of programming tasks in specific programming languages using a fixed set of test types. As a consequence LMSs and programming assessment systems are often either used side by side in various LMS and GS combinations or only a single system with a limited feature set.

An obvious idea is to combine the advantages of both system types, LMSs and GSs, in order to get the “best of both worlds” using the course management and community features from an LMS and adding the special features of grading systems. A first simple connection would be to bridge the gap between these distinct tools by employing some kind of secure link to the other system combined with an auto-login feature. This approach is used by the JACK system in combination with Moodle, or in eClaus with OLAT/OPAL. The IMS Learning Tool Interoperability (LTI, [IMSLT]) standard provides such a feature and also allows an external tool to pass learning or assessment results back to the calling LMS. However, when a student clicks on the secure link, full control is handed to the external tool which means that the student is directly interacting with the external tool. This often has a layout and handling that is completely different from the LMS. Furthermore, just providing a

login is not sufficient for exchanging complex assessment data between the systems. A slightly more complex connection would be to integrate an assessment plugin directly into an LMS or to write a GS specific wrapper (ViPS, VPL, WeBWorK-JAG, or planned by SAC [Au08]). This, however, might have security implications or limits the usage of different GS at the same time.

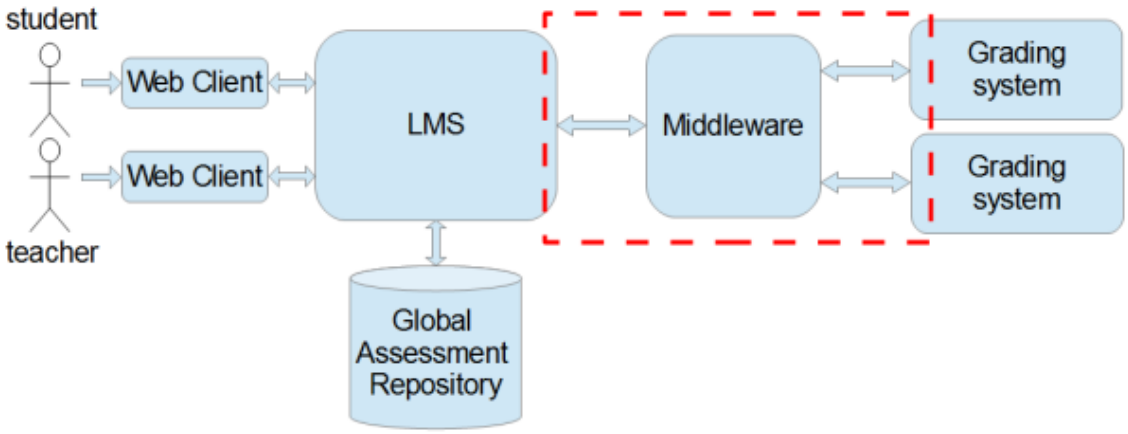

Figure 3: Proposed middleware architecture for LMS-grader integration

A more flexible solution could be an architecture incorporating a special middleware which supports a detailed and secure data exchange between the systems based on the ProFormA format and a yet to be determined exchange protocol (cf. Figure 3): Lecturers as well as students utilize their preferred web clients for connecting to their usual LMS. Within this familiar learning environment, students can work on programming tasks which are stored within the LMS. Assessment and feedback generation is dealt with transparently by one or multiple external GSs which can even be of different types. The GSs are connected to the LMS using a middleware which only needs to send the task specification utilizing the ProFormA format and the student solution to the grader.

Such a middleware architecture has multiple advantages: It allows an LMS to use different GSs, it can implement load balancing for multiple GSs of the same type, and at the same time it hides the complexity of the connection details and of the data exchange. Thus it simplifies the interfaces at both ends – for the LMS and for the GS. A properly designed middleware can easily be re-used by different LMS and, therefore, lower the requirements for changes/extensions necessary to a specific LMS.

Special attention needs to be given to the results and feedback generated by the GS. In the proposed architecture, graders are concerned with generating feedback about a submission received from the middleware. Feedback usually consists of a grade, or a score, a percentage or just a boolean value and some documentation about the GS's results. Documentation can range from a simple text (e.g., in form of a console dump) over XML documents with markup for message categories and message content to complete documents that are ready for immediate presentation to the user such as HTML, PDF, images or other media types. Some GSs can generate separate feedback for students and teachers at different levels of detail. Just as the ProFormA format of this paper aims at covering any programming exercise configuration and description, a unifying feedback format is needed that covers grading results from any grader. Ideally, an LMS should allow students and teachers to inspect the results working with the user interface of the familiar

LMS. Many LMSs already support the concept of manually or automatically graded assignments. Extending an LMS so that it will integrate grader results into its LMS-specific assignment concept is less costly under the assumption of a unifying grading feedback format. Currently we are eliciting the respective requirements from several GSs and learning management systems for such a unifying format.

Two prototypes have been developed to serve as a connector component between LMS and GS. In [PJR12a] an LMS (LON-CAPA) is connected to a grader (Praktomat) with the help of a dedicated middleware server. Another implementation describes the “Grappa” server, which connects the LMS Moodle to the grader aSQLg [Stö14]. Both prototypes are either capable of or currently being developed towards supporting the ProFormA format described in this paper. But none of the systems already uses a shared feedback format.

7 Summary and future work

In this article, specific requirements for a special exchange format for programming tasks were presented. Based on these requirements, an XML-based exchange format called ProFormA was derived. This format can be used for the exchange of programming tasks across systems. This schema is not a "one-size-fits-all" specification for fixed languages and test types, but a format which allows the use of different XML namespaces for extensibility. Consequently, this specification provides a good basis for increasing the interoperability between different systems. Export and import functionality using the ProFormA format is already available in several systems: Essential elements of programming tasks that were exported in the ProFormA format can be imported into these systems. Thus, a cross-system sharing of tasks using the ProFormA format has proven to be feasible. In principle, all 33 systems reviewed in section 3.1 could make use of it.

The next steps in our work include the refinement of the elements of the described specification and the integration with other systems: in stand-alone systems for import and export as well as usage for LMS-GS communication. The current version of the ProFormA format builds the basis for sharing the integral elements of programming tasks. Future refinements might include parametric exercises, exercise bundling and further aspects regarding security and runtime limitations of tests. Also, a formal way for suggesting new test type specifications for inclusion in the “official” ProFormA format will need to be developed.

For ease of sharing, a repository of programming tasks described in the ProFormA specification shall be set up in the future, so that tasks can be searched and imported directly into a specific system. A repository provides an opportunity to increase the quality of teaching by repeated usage of good tasks, especially across different universities. This does not mean that a separate repository needs to be built, the utilization of existing repositories is explicitly not ruled out. However, there are important requirements on a shared repository of programming tasks: Especially, it is of particular significance that access to the complete task specifications is restricted in a way so that learners cannot “cheat” by using the model-solutions given in the tasks (cf. LON-CAPA, [LONC]).

References

[Al05] Ala-Mutka, K.: A Survey of Automated Assessment Approaches for Programming Assignments. In: Computer science education, 15(2), 2005, pp. 93-102.

[APR06] Amelung, M.; Piotrowski, M.; Rösner, D.: eduComponents: experiences in e-assessment in computer science education. In: Proc. ITiCSE'06, 2006, pp. 88-92.

[AR08] Amelung, M.; Rösner, D.: Experiences in Hybrid Learning with eduComponents. In: Hybrid Learning and Education. Springer, Berlin Heidelberg, 2008, pp. 259-270.

[ASR06] Ahtiainen, A.; Surakka, S.; Rahikainen, M.: Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In: Proc. Baltic Sea conference on Computing education research. ACM, New York, NY, USA, 2006, pp. 141–142.

[ASS08] Altenbernd-Giani, E.; Schroeder, U.; Stalljohann, P.: eAixessor - A Modular Framework for Automatic Assessment of Weekly Assignments in Higher Education. In: Proc. IASTED'08, 2008, p. 99.

[Au08] Auffarth, B.; López-Sánchez, M.; Miralles, J. C.; Puig, A.: System for Automated Assistance in Correction of Programming Exercises (SAC). In Proc. CIDUI'08, 2008, pp. 104-113

[CMM04] Cesarini, M.; Mazzoni, P.; Monga, M.: Learning Objects and Tests. In: The IASTED International Conference on Web-Based Education, 2004, pp. 520-524.

[DGNU] DejaGNU, GNU Project. https://www.gnu.org/software/dejagnu/ (last check 2015-02-26)

[DH04] Daly, C.; Horgan, J. M.: An automated learning system for Java programming. In: Education, IEEE Transactions on Education, 47(1), 2004, pp. 10-17.

[Ed04] Edwards, S. H.: Using software testing to move students from trial-and-error to reflection-in-action. In ACM SIGCSE Bulletin 36(1), 2004, pp. 26-30.

[EFK04] Ellsworth, C. C.; Fenwick Jr, J. B.; Kurtz, B. L.: The quiver system. In: ACM SIGCSE Bulletin, 36(1), 2004, pp. 205-209.

[Ei03] Eichelberger, H.; Fischer, G.; Grupp, F.; Von Gudenberg, J. W.: Programmierausbildung Online. In: Proc. DeLFI'03, 2003, pp. 134-143.

[Fu08] Fu, X.; Peltsverger, B.; Qian, K.; Tao, L.; Liu, J.: APOGEE: automated project grading and instant feedback system for web based computing. In: ACM SIGCSE Bulletin, 40(1), 2008, pp. 77-81.

[GATE] https://cses.informatik.hu-berlin.de/research/details/gate/ , https://github.com/csware/si/ , (last check 2015-02-26)

[Go04] Godby, C. J.: What Do Application Profiles Reveal about the Learning Object Metadata Standard? In: Adriane Article in eLearning Standards, 2004.

[GSW07] Gotel, O.; Scharff, C.; Wildenberg, A.: Extending and contributing to an open source web-based system for the assessment of programming problems. In: Proc. international symposium on Principles and practice of programming in Java, 2007, pp. 3-12.

[HQW08] Hoffmann, A.; Quast, A.; Wismüller, R.: Online-Übungssystem für die Programmierausbildung zur Einführung in die Informatik. In: Proc. DeLFI‘08, 2008, pp. 173-184.

[Hü05] Hügelmeyer, P.; Mertens, R.; Schröder, M.; Gust, M.: Integration des Virtuellen Prüfungssystems ViPS in die Lehr-/Lernplattform Stud.IP. In: Proc. Workshop on e-Learning 2005, HTWK Leipzig, pp. 187-196.

[HW08] Hwang, W. Y.; Wang, C. Y.; Hwang, G. J.; Huang, Y. M.; Huang, S.: A web-based programming learning environment to support cognitive development. In: Interacting with Computers, 20(6), 2008, pp. 524-534.

[Ih10] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: Proc. Koli Calling‘10, pp. 86-93.

[IMSCP] IMS Global Learning Consortium. Content Packaging Specification. http://www.imsglobal.org/content/packaging/ (last check 2015-02-26)

[IMSLT] IMS Global Learning Consortium. IMS Learning Tools Interoperability (LTI) Implementation Guide. http://www.imsglobal.org/lti/ltiv2p0/ltiIMGv2p0.html (last check 2015-02-26)

[IMSQT] IMS Global Learning Consortium. IMS Question & Test Interoperability™ Specification. http://www.imsglobal.org/question/ (last check 2015-02-26)

[JACK] http://www.s3.uni-duisburg-essen.de/research/jack.html (last check 2015-02-26)

[JGB05] Joy, M.; Griffiths, N.; Boyatt, R.: The boss online submission and assessment system. In: JERIC, 5(3), 2005, p. 2.

[JUNIT] https://junit.org (last check 2015-02-26)

[KJ13] Kruse, M.; Jensen, N.: Automatische Bewertung von Datenbankaufgaben unter Verwendung von LON-CAPA und Praktomat. Proc. Workshop Automatische Bewertung von Programmieraufgaben, ABP’13, 2013.

[KSZ02] Krinke, J.; Störzer, M.; Zeller, A.: Web-basierte Programmierpraktika mit Praktomat. In: Softwaretechnik-Trends, 22(3), 2002, pp. 51-53.

[LLP13] Le, N. T.; Loll, F.; Pinkwart, N.: Operationalizing the Continuum between Well-defined and Ill-defined Problems for Educational Technology. In: IEEE Journal Transactions on Learning Technologies, 2013, 6(3), pp. 258-270.

[LONC] The LearningOnline Network with CAPA. http://loncapa.org (last check 2015-02-26)

[LONCM] Learning Online Network with CAPA. Author’s Tutorial And Manual. https://loncapa.msu.edu/adm/help/author.manual.pdf (last check 2015-02-26)

[LQ09] Leal, J. P.; Queirós, R.: Defining Programming Problems as Learning Objects. In: Proc. ICCEIT’09, 2009.

[LS03] Leal, J. P.; Silva, F.: Mooshak: A Web-based multi-site programming contest system. In: Software: Practice and Experience, 33(6), 2003, pp. 567-581.

[Ma09] Mareš, M.: MOE – Design of a modular grading system. In: Olympiads in Informatics, 3, 2009, pp. 60-66.

[Mo07] Morth, T.; Oechsle, R.; Schloß, H.; Schwinn, M.: Automatische Bewertung studentischer Software. In: Proc. Pre-Conference Workshops der DeLFI'07, 2007.

[MOOD] Moodle - Open-source learning platform. https://moodle.org (last check 2015-02-26)

[No07] Nordquist, P.: Providing accurate and timely feedback by automatically grading student programming labs. In: Journal of Computing Sciences in Colleges, 23(2), 2007, pp. 16-23.

[OLAT] OLAT Lernmanagement. https://www.bps-system.de/cms/produkte/olat-lernmanagement/ (last check 2015-02-26)

[PFMA] ProFormA specification and whitepaper. https://github.com/ProFormA/taskxml (last check 2015-02-26)

[PJR12] Priss, U.; Jensen, N.; Rod, O.: Software for E-Assessment of Programming Exercises. In: Informatik 2012, GI LNI, P-208, pp. 1786-1791.

[PJR12a] Priss, U.; Jensen, N.; Rod, O.: Software for Formative Assessment of Programming Exercises. In: elearning Baltics 2012, Proc. International eLBa Science Conference, 2012, pp. 63-72.

[PRAK] https://github.com/KITPraktomatTeam/Praktomat (last check 2015-02-26)

[QL11] Queirós, R.; Leal, J. P.: Pexil: Programming exercises interoperability language. In: Proc. Conferência - XML: Aplicações e Tecnologias Associadas (XATA), 2011.

[QL12] Queirós, R.; Leal, J. P.: PETCHA: a programming exercises teaching assistant. In: Proc. ACM ITiCSE'12, 2012, pp. 192-197.

[QL13] Queirós, R.; Leal, J. P.: BabeLO – An Extensible Converter of Programming Exercises Formats. In: IEEE Transactions on Learning Technologies, 6(1), 2013, pp. 38-45.

[Re89] Reek, K. A.: The TRY system-or-how to avoid testing student programs. In: ACM SIGCSE Bulletin, 21(1), 1989, pp. 112-116.

[RH08] Rößling, G.; Hartte, S.: WebTasks: online programming exercises made easy. In: ACM SIGCSE Bulletin, 40(3), 2008, pp. 363-363.

[RML08] Revilla, M. A.; Manzoor, S.; Liu, R.: Competitive learning in informatics: The UVa Online Judge Experience. In: Olympiads in Informatics, 2, 2008, pp. 131-148.

[Ro07] Romeike, R.: Three Drivers for Creativity in Computer Science Education. In: Proc. of Informatics, Mathematics and ICT: a 'golden triangle'. Boston, US, 2007.

[RRH12] Rodríguez-del-Pino, J.; Rubio-Royo, E.; Hernández-Figueroa, Z.: A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In: Proc. CSREA EEE'12, 2012.

[SBG09] Striewe, M.; Balz, M.; Goedicke, M.: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In: Proc. CSEDU'09, 2009, pp. 54-61.

[SCORM] Rustici Software. Technical SCORM. http://scorm.com/scorm-explained/technical-scorm/ (last check 2015-02-26)

[SG10] Striewe, M.; Goedicke, M.: Visualizing Data Structures in an E-Learning System. In: Proc. CSEDU'10, 2010, pp. 172-179.

[SG11] Striewe, M.; Goedicke, M.: Using Run Time Traces in Automated Programming Tutoring. In: Proc. ACM SIGCSE ITiCSE'11, 2011, pp. 303-307.

[SG13] Striewe, M.; Goedicke, M.: JACK revisited: Scaling up in multiple dimensions. In: Proc. EC-TEL 2013, 2013, pp. 635-636.

[Sh05] Shaffer, S. C.: Ludwig: an online programming tutoring and assessment system. In: ACM SIGCSE Bulletin, 37(2), 2005, pp. 56-60.

[SMB11] de Souza, D. M.; Maldonado, J. C.; Barbosa, E. F.: ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In: IEEE Software Engineering Education and Training (CSEE&T), 2011, pp. 1-10.

[SOP11] Strickroth, S.; Olivier, H.; Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: Proc. DeLFI'11, 2011, pp. 115-126.

[St14] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Bott, O. J.; Pinkwart, N.: Wiederverwendbarkeit von Programmieraufgaben durch Interoperabilität von Programmierlernsystemen. In: Proc. DeLFI'14, 2014, pp. 97-108.

[STDIP] Stud.IP-Portal: Stud.IP 3: Das Portal. Projekt, Informationen und Ressourcen. http://studip.de (last check 2015-02-26)

[Stö13] Stöcker, A.; Becker, S.; Garmann, R.; Heine, F.; Kleiner, C.; Bott, O. J.: Evaluation automatisierter Programmbewertung bei der Vermittlung des Sprachen Java und SQL mit den Gradern aSQLg und Graja aus studentischer Perspektive. In: Proc. DeLFI'13, 2013, pp. 233-238.

[Stö14] Stöcker, A.; Becker, S.; Garmann, R.; Heine, F.; Kleiner, C.; Werner, P.; Grzanna, S.; Bott, O. J.: Die Evaluation generischer Einbettung automatisierter Programmbewertung in Moodle. In: Proc. DeLFI'14, 2014, pp. 301-304.

[Sp06] Spacco, J.; Hovemeyer, D.; Pugh, W.; Emad, F.; Hollingsworth, J. K.; Padua-Perez, N.: Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. In: ACM SIGCSE Bulletin 38(3), 2006, pp. 13-17.

[SVW06] Schwieren, J.; Vossen, G.; Westerkamp, P.: Using software testing techniques for efficient handling of programming exercises in an e-learning platform. In: The Electronic Journal of e-Learning, 4(1), 2006, pp. 87-94.

[SW06] Stevenson, D. E.; Wagner, P. J.: Developing real-world programming assignments for CS1. In: ACM SIGCSE Bulletin, 38(3), 2006, pp. 158-162.

[Tr07] Truong, N.: A Web-Based Programming Environment for Novice Programmers. PhD Thesis, 2007: http://eprints.qut.edu.au/16471/1/Nghi_Truong_Thesis.pdf (last check 2015-02-26)

[Tr08] Tremblay, G.; Guérin, F., Pons, A.; Salah, A.: Oto, a generic and extensible tool for marking programming assignments. In: Software: Practice and Experience, 38(3), 2008, pp. 307-333.

[Ve08] Verhoeff, T.: Programming Task Packages: Peach Exchange Format. In: Olympiads in Informatics, 2, 2008, pp. 192–207.

[Ve11] Verdú, E.; Regueras, L. M.; Verdú, M. J.; Leal, J. P.; de Castro, J. P.; Queirós, R.: A distributed system for learning programming on-line. In: Computers & Education 58(1), 2011, pp. 1-10.

[WW07] Weicker, N.; Weicker, K.: Automatische Korrektur von Programmieraufgaben – Ein Erfahrungsbericht. In: Flexibel integrierbares e-Learning - Nahe Zukunft oder Utopie, Proc. Workshop on e-Learning 2007, pp. 159-173.

Acknowledgements

The work of the authors Becker, Müller, Priss and Rod has been partially funded by the German Federal Ministry of Education and Research (BMBF) under grant numbers 01PL11066H and 01PL11066L.

Appendix

System	Dealing with Java tasks	Using JUnit	Standalone
Apogee [Fu08]	?	no	X
ASB [Mo07]	X	X	X
AutoGrader [No07]	X	?	?
BOSS(2) [JGB05]	X	X	X
CourseMarker/ CourseMaster [Hi03]	no	no	X
DUESIE [HQW08]	X	X	X
eAixessor [ASS08]	X	no	X
eClaus [WW07]	X	no	X
eduComponents [APR06]	X	no	no
ELP [Tr07]	X	no	X
GATE [SOP11]	X	X	X
Graja [Stö13]	X	X	no
JACK [SBG09]	X	no	X
JOP [Ei03]	X	X	X
Ludwig [Sh05]	no	no	X
Marmoset [Sp06]	X	X	X
MOE [Ma09]	no	no	X
Mooshak [LS03]	X	no	X
Oto [Tr08]	X	X	X
Praktomat [KSZ02]	X	X	X
ProgTest [SMB11]	X	X	X
Quiver [EFK04]	X	no	X
RoboProf [DH04]	X	no	X
SAC [Au08]	X	X	X
UVa OnlineJudge [RML08]	X, in newer version	?	X

ViPS [Hü05]	no	no	no
VPL [RRH12]	X	X	no
Web-CAT [Ed04]	X	X	X
WebTasks [RH08]	X	X	X
WeBWorK-JAG [GSW07]	X	X	no
WPAS [HW08]	no	no	?
xLx [SVW06]	X	X	X

Table 2: Reviewed systems and their properties regarding support of Java tasks, support of JUnit, and system architecture.

Additional Material

- Whitepaper (Version 0.9.4): whitepaper_094-md.html
- Official XSD: taskxml.xsd.zip
- Example of a programming task specified in the ProFormA format version 0.9.4: listing1.xml.zip