Olympiads in Informatics, 2008, Vol. 2, 192–207
© 2008 *Institute of Mathematics and Informatics, Vilnius*

Programming Task Packages: Peach Exchange Format

Tom VERHOEFF
Department of Mathematics and Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands
e-mail: t.verhoeff@tue.nl

Abstract. Programming education and contests have introduced software to help evaluation by executing submitted taskwork. We present the notion of a *task package* as a unit for collecting, storing, archiving, and exchanging all information concerning a programming task. We also describe a specific format for such task packages as used in our system *Peach*, and illustrate it with an example. Our goal is to stimulate the development of an international standard for packaging of programming tasks.

Key words: programming education, programming contest, programming task, task package, grading support software, data format.

1. Introduction

Programming education and contests have introduced software to help evaluation by executing submitted taskwork. Typically, a programming task is understood to be a short text that describes the requirements on the program to be constructed. Such task descriptions can be found in various problem archives, such as the (UVa Online Judge, 2008). However, it takes more than just its task description to be able to (re)use a programming task. In this article, we present the notion of a *task package* as a unit for collecting, storing, archiving, and exchanging task-related information. Ideally, such task packages can be dropped into your favorite programming education and contest hosting system to configure it for the task.

We have used task packages for many years now in our programming education and contest hosting system, called *Peach* (Peach, 2008). Using these task packages has helped us ensure completeness and correctness of task data.

We will discuss the contents and format of task packages and also the interface and operations for task packages. Particular concerns are the support for

- multiple languages to express human readable texts (including, but not restricted to, the task description);
- multiple programming languages allowed for solving the task;
- multiple platforms to deploy tasks;
- diverse task styles;

- validation of package content;
- handling of relationships between tasks, e.g., where one task is a variant of another task;
- flexibility to allow easy incorporation of changes, such as changing the task's name.

2. Package Contents

In this section, we discuss the various pieces of information that (could) belong in a task package. As a running example, we use the task *Toy Division*. Here is its task description:

TOY DIVISION

PROBLEM

K kids together receive T toys. They wonder whether it is possible to divide these toys *equally* among them, that is, with each kid obtaining the same number of toys and no toys remaining. If this is possible, they also want to know how many toys Q each of them receives.

Write a program to answer the questions of these kids.

INPUT

A single line is offered on standard input. It contains two the numbers K and T, separated by a space.

OUTPUT

The answer must be written to standard output. The first line contains the string `'Yes'` if it is possible to divide the toys equally, and `'No'` otherwise. If equal division is possible, then a second line must be written, containing the number Q of toys each kid obtains. If there are multiple answers, then it does not matter which answer your program writes.

CONSTRAINTS

K, T, and Q are non-negative integers, less than 10^8. The execution time limit is 0.1 s.

EXAMPLE

Standard Input
`3 15`

Standard Output

`Yes`
`5`

SCORE

There are 10 evaluation cases. Each completely solved case gets 10 points. Incomplete outputs get 0 points; there is no partial score.

(END OF TASK DESCRIPTION)

The following kinds of task-related information can be distinguished:

- textual information in natural language, such as the task description;
- data files, such as input data for evaluation runs;
- configuration parameters, such as resource limits;
- tools, such as a task-specific output checker;
- scripts, e.g., to build an executable or coordinate the evaluation process;
- submissions (good and bad), as exemplary solution and to help verify everything;
- metadata, e.g., classification of task type; some metadata could be textual.

It is good to be aware of the various **stakeholders** of task-related information. The terminology often depends on the setting.

Supervisor makes management-level decisions, e.g., about the task name, set of tasks to use together, presentation style, etc.; could also be called **owner**. In an educational setting, this role is played by the *teacher*; in a contest setting, by the *contest director* or *chair of the scientific committee*.

Author creates the task; makes technical decisions about the task; often needs to carry out various experiments to explore alternatives and tune parameters.

Solver attempts to carry out the task, i.e., solve the stated problem, resulting in work to be submitted for evaluation; could also be called *performer*. In an educational setting, this is the *student* participating in a course; in a competition, it is the *contestant*. Keep in mind that the solver is the primary stakeholder.

Grader is involved in evaluating the submitted work for a task. In an educational setting, this is often a teaching assistant; in a competition, this is nowadays often supported by an automated grading system, which is administered by a *grading system administrator*, who configures the system for specific tasks. Note that also *developers of automated grading systems* are to be considered as stakeholders.

Mediator helps interpret evaluation results. In an educational setting, this could be done by an *instructor*, sometimes *parents* play this role; in a competition, this is done by a *coach* or *team leader*.

Trainer helps in preparing the solver, e.g., through directed practicing. In an educational setting, this is often the job of a *teaching assistant*; in a contest setting, there are typically several trainers, each with their own area of expertise. It is also possible that solvers practice by themselves. This stakeholder has a specific interest in the ability to *reuse* tasks easily.

Researcher investigates tasks, submitted work, and evaluation results; this could be for historic reasons, but also to help improve education and competition practices, or with a purely scientific interest. This stakeholder is helped by consistent and complete archiving of tasks, similar to what (Sloane's OEIS, 2008) does for integer sequences.

2.1. *Textual Information*

The **task description** contains all information (or pointers to such information) that the task author wants to offer to the students or contestants confronted with the task. In (Verhoeff, 1988) some guidelines for creating programming problem sets are presented. The task description must specify unambiguously, precisely, consistently and completely what qualifies as an acceptable solution. Typically, each participant submits work on the task in the form of (one or more) files. For files with program code, the task description states the interfaces and the relevant contracts (assumptions and obligations), and provides at least one example.

Besides a task description the following texts are useful to include in a task package.

Hints In an educational setting, we find it useful at times to include a separate text with hints, which can be disclosed to participants under appropriate circumstances. Even for contests, it can be useful to have separate hints.

Background information This includes, for instance, historic and technical information (algorithms, data structures), and motivation for particular features, e.g., their relationship to a syllabus (Verhoeff *et al.*, 2006).

Grading information That is, information that concerns the process of grading submitted work for this task. In an educational setting, this could involve instructions for assistants doing the actual grading. In a contest setting, it could cover information useful in understanding the grading result.

Notes These are short texts about the contents of other items, for instance, summarizing the purpose of the task, evaluation cases, and test submissions, and they can be used to generate various overviews (also see Section 3 about operations on task packages). These notes belong to the category of *metadata*, which can include machine-readable information as well (treated in Section 2.7).

The hints for our example are:

HINTS FOR TOY DIVISION
Consider the integer equation $T = K * Q$ with unknown Q.
What special cases are there?
(END OF HINTS)

Some background information for the example:

BACKGROUND INFORMATION FOR TOY DIVISION
The purpose of this task is to present a simple, nontrivial programming problem. The cases $K = 0$ could be eliminated to simplify it even further, by constraining the input to $K > 0$.
It has proven to be a good exercise in careful reading of specifications, and the use of integer division, including the modulo operator.
(END OF BACKGROUND INFORMATION)

Some grading information for the example:

GRADING INFORMATION FOR TOY DIVISION

The proper case distinction needs to be made. For $K = 0$, the equation $T = K * Q$ degenerates to $T = 0$. That is, equal division is not possible if $T \neq 0$, it is possible if $T = 0$, in which case any Q (in range) will do.

For $K \neq 0$, the equation $T = K * Q$ is a linear equation in Q. It is solvable in integers if and only if K is a divisor of T, or, alternatively, if $T \bmod K = 0$. In that case, Q is uniquely determined by $Q = T/K$.

Manual graders should judge the layout, comments, names, and idiom. Particular points of further attention are:

- avoiding division by zero;
- use of 32-bit arithmetic (or better);
- use of integer division, rather than floating-point division (inaccurate) or repeated subtraction (too slow).

#	K	T	Q	Remarks
1	0	0	*	The only divisible case with $K = 0$
2	0	1	`No`	Smallest T with $K = 0$ that is not divisible
3	0	99999999	`No`	Largest T with $K = 0$ that is not divisible
4	1	0	0	Smallest case with $K > T = 0$
5	1	99999999	99999999	Largest Q that is divisible
6	2	1	`No`	Smallest case with $K > T > 0$
7	2	65536	32768	$T = 2^{16}$, fails with 16-bit arithmetic
8	99999998	99999999	`No`	Largest T, and K, that are not divisible
9	99999999	99999998	`No`	Largest K, and T, that are not divisible
10	99999999	99999999	1	Largest K and T that are divisible

Legend:

# Identifier of evaluation case

K Number of kids (input)

T Number of toys (input)

Q Number of toys per kid (quotient), if equally divisible, else `No` (output)

* indicates that any value Q satisfying $0 \leqslant Q < 10^8$ is correct.

(END OF GRADING INFORMATION)

Each of these texts could be available in several languages. In our educational setting, we often have material both in Dutch and in English. In contests like the International Olympiad in Informatics (IOI, 2008), task descriptions are translated into the native language of the contestants, resulting in dozens of versions of the same information, whereas background and grading information is often presented in English only. Note that the tabular overview of evaluation cases in the grading information would ideally be generated from the actual evaluation data and summary texts (metadata).

A general concern with ‘plain’ text files is their **encoding**. For simple texts, ASCII suffices, but especially for non-English texts, additional characters are desirable. We recommend the use of **UTF-8** (RFC 3629, 2003), one of the *Unicode* standards.

Many texts, however, will not be written in a ‘plain’ text format, but some other format. Some relevant open format standards are:

- LaTeX, TeX, especially suited for texts involving mathematics (CTAN, 2008);
- OpenDocument, used by OpenOffice (OpenOffice, 2008);
- (X)HTML, used in web browsers (W3C, 2008);
- DocBook (DocBook, 2008);
- reStructured Text, used by Docutils (Docutils, 2008);
- various wiki formats.

Each of these open formats may have multiple variants. Note that these formats are aimed at flexible text entry and editing. They can be converted into various (open) presentation formats, such as PDF.

One should also be aware of the need for **version control** on texts. This issue is addressed further in Section 4.

2.2. *Data Files*

Besides human-readable texts, a task can also involve various other files, in both text or binary format. We call them data files, even though they could well be source files with program fragments, such as an interface definition for a library. These files could be part of the material that the solver receives along with the task description, but they could also be related to evaluation. Here is an overview:

- Data files accompanying the task description, possibly including source files. In our educational setting, we sometimes have assignments where we provide a program text 'with holes' to be completed by the students. Such a source file with holes is created by a generic tool on the basis of our own solution with special 'hole markers' in comments:

```
//# BEGIN TODO body of method TPointSet.Count
  ... author's solution, to be suppressed ...
//# END TO DO
```

- Input data for evaluation runs; per run there could be several files;
- Expected output data for evaluation runs of deterministic[1] programs, possibly modulo some simple equivalence relation. The equivalence could concern white space, upper versus lower case characters, the order of numbers in a set, etc.;
- Auxiliary data used in evaluation runs of nondeterministic programs. This could concern parts of the output that are deterministic, or some description of the expected output, e.g., in the form of a regular expression;
- Input data for tools that generate other files, such as large input for evaluation.

It is important that one can motivate the choice of data. A haphazard collection of inputs does not make a good grading suite. Make it a habit to write a *note* for each data file, summarizing its *purpose* (as opposed to its contents; for the latter, generator input or characterizer output is more useful, see Section 2.4 on tools). Such notes can be (partially) included in tabular overviews of data sets. This is especially useful for larger data sets. The overview can be attached to the grading information.

There are some platform-related issues to keep in mind:

[1]By deterministic we mean that the input uniquely determines the output.

End-of-line markers In text files, the end of a line is marked by a special character or combination of characters, depending on the platform. Unix uses a *line feed* (LF), Mac OS uses a *carriage return* (CR), and Windows uses the combination CRLF. This is particularly relevant for files made available to the solver, and files directly used in evaluation (e.g., to compare to the output of an evaluation run).

Byte order In binary files, the ordering of bytes in some multi-byte data items (such as numbers) may vary between the platforms. The two main flavors are *big-endian* and *little-endian*. The concerns are similar to those for end-of-line markers.

2.3. *Configuration Parameters*

The data files discussed in the preceding section play a specific role in grading the functionality requested in the task: input(-related) data and output(-related) data.

A task can specify more than just functionality, It can, for instance, also impose performance requirements. Such requirements are often expressed in terms of **resource limits**. In particular, the following resources have been limited:

- *size of submission* (total size of source files);
- *build time* (total time allowed for compilation and linking);
- *memory* (RAM, hard disk space);
- *execution time* (total run time, or response time to specific events);
- *number of times* that certain resources may be used, for instance, that some function in a library may be called.

Other things that can be treated as configuration parameter are: *compiler options* and *linker options*.

Such configuration parameters are intrinsic to the task, and are sometimes – but not always – communicated explicitly to the solver in the task description. They also need to be taken into account during evaluation runs. For automatic grading and for later analysis and comparison of tasks, it is useful to include configuration parameters in a task package in a *machine readable* way. They should be easy to tune at a late stage.

Note, however, that the meaning of such parameters depends on the actual platform used for evaluation runs. Platform information is discussed in Section 2.7 about metadata. Also, it is imaginable that not all evaluation runs use identical parameter values.

2.4. *Tools*

When solving a task and when evaluating work submitted for a task, various generic software tools are needed. Most notably these include editors, compilers, libraries, linkers, loaders, debuggers, file comparators, etc. Generic tools are discussed in Section 2.7 along with metadata.

There is often also a need for task-specific tools. These are to be developed by the task author (or assistants). One can think of the following kinds of task-specific tools:

Input generator to produce specific input cases, for instance large cases with special properties. Use of an input generator also helps ensure that valid data is created.

Input validator to check that input files satisfy the assumptions stated in the task description. These assumptions often include **format** restrictions: what types of input data appear in what order and in what layout (i.e., distribution over lines); but also concern **value** restrictions: range limits on values, specific relationships between values (e.g., a graph that needs to be connected).

Input data files need to be of high quality, and one should not simply assume that they are valid (unless they are automatically generated maybe, but even then it is useful to have the ability to independently check their validity). The application of an input validator needs to be automated, because otherwise it will not be used when it is most needed, viz. under pressure when last-minute changes are made. Also see Section 3 about package operations.

Output format checker to check that output conforms to the format requirements of the task. This tool can be given to the solver to help achieve the correct output format. Note that this tool will not give information about the actual correctness of the output. It can also be used during evaluation as a filter to ensure that a tool that checks for correctness does not have to deal with badly formatted data.

Input/output characterizer to summarize characteristics of data files, in particular, to generate, from actual input and output data files the tables appearing in the grading information. Such summaries are useful in determining to what extent evaluation runs cover various ‘corners’ of the input space. Doing this by hand is cumbersome and error prone.

Expected-output generator to produce expected output on the basis of given input data. This is useful when a task is (almost) deterministic. Note that in most cases a solution to the task can act as expected-output generator. But it need not satisfy the task’s performance requirements; it can be run in advance (or afterwards) and even on a different platform.

Output checker to check that output[2] generated in an evaluation run corresponds with the input data in accordance with the requirements stated in the task description. An output checker takes as inputs the input data file, the output data file produced in the evaluation run, and sometimes also some preprocessed data (to avoid the need for recalculating certain information, e.g., concerning deterministic parts of the output; that way the checker can be kept smaller and more generic).

This applies especially to nondeterministic tasks. In case of a deterministic task, output checking can be done generically by comparing actual output to expected output, possibly modulo some simple equivalence relation.

Evaluation drivers and/or stubs to be combined with submitted program fragments to build executables used in evaluation runs. In particular, if the task does not require the solver to submit a main program (but, for instance, a module or library), then the task author needs to provide a main program (or more than one) to act as an evaluation driver of the submitted module or library. And, conversely, when the

[2]Occasionally, also the order of *input-output interleaving* needs to be checked.

task requires the solver to submit a main program with one or more holes (e.g., in the form of a pre-defined module or library), then the author may need to provide evaluation stubs to fill these holes.

Not every task will need each of these task-specific tools.

Such tools need to incorporate task-specific knowledge. Often it is a good idea to create a separate library with task-specific facilities (data types and related operations), rather than duplicating such definitions in each tool. Duplication hinders future changes, especially when a task is still under development.

Some tools can be combined, though this is not advisable. It is better to refactor common functionality into a task-specific library. For instance, an input/output characterizer needs to read input and output files, and so could also report on the validity of their format and contents. But combined functionality complicates the interface of the tool, and increases the risk that changing one piece of functionality will also (adversely) affect other pieces.

There is an opportunity to use generic libraries for functionality common to multiple tasks. For instance, **RobIn** (Verhoeff, 2002) was developed to assist in the construction of input validators and output (format) checkers, by providing some simple operations for *robust input*, that is, without making any assumptions about the actual input. RobIn was used by the author at IOI 2000 in China to validate the input files.

2.5. *Scripts*

Besides task-specific tools, there will also be various task-specific scripts. Tools concern task-specific technical operations, whereas scripts are more for management and for coordinating the application of task-specific tools. Scripts can

- coordinate the entire grading process of a submission for the task, involving such steps as
    1) preprocessing of submitted work,
    2) building various executables,
    3) running executables with various inputs, capturing output and interleaving,
    4) evaluation of the outputs of each run according to various criteria,
    5) scoring to obtain numeric results,
    6) reporting to present and summarize all results.

  Such a grading script should be runnable by a daemon in an automated grading system, but also by a task author or human grader in a stand-alone situation; a task author may want to explore how a particular submission is handled, and a human grader (teaching assistant) may want to re-evaluate a submission locally under several what-if scenarios by making manual changes;
- coordinate the generation of all evaluation data;
- generate various overviews;
- generate an archive of material to be presented to solver, especially when this consists of more than just the task description;
- validate the package contents, by evaluating all test submissions and checking the results.

2.6. *Solution and Test Submissions*

A task author not only needs to invent and describe the task, specify how it will be graded, and provide data and (where applicable) task-specific tools, but also needs to write an **exemplary solution** worthy of imitation. This solution is needed for pedagogical reasons, and it also serves as a test for the grading data and tools. However, package testing should not end there. In fact, solutions are needed in *all allowed programming languages*. Of course, the grading tools and data should also be tested with **imperfect solutions**, to check that these are graded in agreement with the intentions.

These test submissions (ranging from good to bad) belong in the task package, and must be used to validate the package contents and in particular, the entire grading chain. They also provide a means to test the installation of a package on a particular grading system. There should be sufficient variety in submissions to ensure a broad coverage.

As with data files, it is recommended to include a separate *note* with each test submission, motivating its purpose. These notes can be summarized in a tabular overview, together with actual and expected grading results.

The work submitted by solvers, when this task is actually used in a course or competition, does not belong inside the task package, but should be stored separately. The relationship between submissions and tasks does need to be recorded.

2.7. *Metadata*

We have come a long way in defining the package contents. What we have described so far would already allow one to run a nice programming course or competition. When one is involved in multiple events, year after year, the need arises to look at things from a somewhat different perspective. For these longer term interests, it is useful to include certain metadata in a task package from the very beginning. One can think of the following items.

Task-instrinsic metadata including

- *Summary*, describing the task in one sentence; this is useful when dealing with *task bundles*;
- *Task type*, for instance, *batch* (through stdio, files, sockets, ...), *reactive* (through stdio, using or providing a library, sockets, ...), *output file only* (for given input files), etc.;
- *Difficulty level*, possibly distinguishing *understanding* (what to do), *designing* (abstract solution), and *implementing* (concrete submission); this is, of course, a somewhat subjective judgment, relative to specific context parameters (skill of solver, amount of time for solving, resource limits, programming language allowed, development tools available, etc.); each of these could be expressed on a scale of *easy*, *medium*, *hard*, possibly extended with *easy-medium*, *medium-hard*. This is usually done in a review meeting;
- *Topic classification*, what topics are involved in the task description, what topics are involved in a (typical) solution; this can be done in terms of a syllabus (Verhoeff *et al.*, 2006);

- *Notes* for data files and test submissions, summarizing their purpose.

Author-related data such as *name*, *contact information*.

Event-related data such as *name* of (or even better, some standardized *identifier* for) the original event (course or competition) at which it has been or will be used; *date* of that event, *amount of time allowed for solving*, *number of solvers* involved, etc.

Solver-related data such as their *background* (educational level, experience), *platform* used by solvers, characterizing the hardware architecture (processor, memory hierarchy), operating system, but also compilers, linkers, standard libraries, possibly also specific other tools allowed in solving the task at hand. This metadata helps in interpreting such things as configuration parameters (time and memory limits), because these are expressed relative to a certain platform.

Grading-related metadata such as *grading scale* (e.g., accepted–rejected, numeric 0–100, numeric 1–5, letter A–F, ...); *amount of time* it typically takes to grade a single submission. If the grading *platform* differs from the solver's platform, then it must also be characterized.

Management-related data such as *status of development* (in preparation, already used; incomplete, complete; draft, ready for review, approved); *version information*, *revision log* of content and status changes, *comments* (by author and reviewers), and a *to-do list*. A supervisor might also be interested in the amount of effort (time) it typically takes to translate the task description (possibly relative to some standard).

What metadata to include will also depend on the *style* of the course or competition. Compare, for instance, the styles of (IOI, 2008) and (ACM ICPC, 2008). Some metadata will be the same for all tasks used together in the same event. Good tasks must be expected to be reused later in other events, for example, on a training camp. It is advisable to copy that common information in each task package, so that a task in isolation is still complete.

2.8. *Miscellaneous Considerations*

The preceding compilation of items that can be included in a task package is not claimed to be complete and final. On some occasions, it may seem overkill; on other occasions, one may wish to include additional information.

There is a trade-off between putting data inside the package or keeping it outside. When data is not directly incorporated in the package, one has the option of incorporating some form of **reference** (like a URL) to that information instead. Our system (Peach, 2008) can be configured with *time intervals* for when a task is available to solvers and when submissions are accepted. We keep this information outside the package, because it will differ for each reuse of the package, e.g., in next year's course.

In an international event like the (IOI, 2008), texts presented to solvers must be translated. This is a major effort because so many languages are involved. It can be useful to provide **translation support**, such as separate figures and a *glossary*.

Another issue to be addressed concerns **task bundles**, that is, sets of tasks used together in a course or competition. In a task bundle, one often strives for a certain level of *uniformity*. This can be achieved by copying common information into each task package. However, this makes it harder to change common information easily and consistently. An alternative is to introduce a kind of *inheritance mechanism* for task packages, and *abstract packages*. In fact, task bundles call for **bundle packages**, that contain (references to) task packages, but in particular also contain common items, such as document templates to be used for all tasks. But his is beyond the scope of the present article.

3. Package Interface and Operations

In the preceding section, we have discussed the contents of a task package. When constructing a package, the author is mainly "working inside it". Once a package is completed, there are several different ways of using it. At that stage, the package users (often not the author) wish to abstract from all internal details, and concentrate on specific package-level operations, such as

viewing (a summary of) (parts of) the package contents;

validating the package contents (for internal consistency and completeness; this is what the test submissions are for);

generating various items from the package, e.g., an archive to be made available to solvers, or information for a mediator (like a team leader);

grading a submission for the task; this could be done locally on the user's platform, or remotely inside an automated grading system; grading can be done *completely*, that is, fully performing all grading steps (preprocess, build, execute, evaluate, score, report), or *partially*, that is, performing only some user-selected steps;

cleaning up a package by removing auxiliary and temporary files;

installing a package in an automated grading system, e.g., by a simple *drag-and-drop*.

Especially for the automated use of packages, it is necessary to have a well-defined, clear, and uniform interface for the package operations. The implementations of these operations are provided by the scripts and tools inside the package, involving various external facilities (like compilers and libraries). The interface is intended to protect (the integrity of) the package contents.

It could be useful to include in the interface some limited ways of modifying a package as well. *Renaming* a task is good candidate, as is *tuning* (some of) the configuration parameters.

4. Package Format

There are many ways in which the package contents can be stored and given an interface. By using appropriate wrappers, one can convert between formats. However, an abundance of different formats is far from convenient. We now briefly describe the format currently used in (Peach, 2008).

4.1. *Peach Exchange Format for Programming Task Packages*

Peach task packages are stored in a directory tree with a predefined structure, naming scheme, and files formats. Fig. 1 shows the main features.

There are separate subdirectories for

- evaluation data subdivided in cases;
- test submissions subdivided by programming language;
- texts subdivided by natural language;
- tools.

The subdirectories for texts are named by their (RFC 4646, 2006) code; this code is based on (ISO 639-1, 2002) alpha-2 language identifiers and (ISO 3166-1, 2006) alpha-2 country codes. Unfortunately, there is no international standard for programming language and dialect identification codes. We use common names in lower case, and currently do not distinguish dialects.

At present, human-readable metadata can be stored in one language only, and is distributed over the tree (e.g., in various `summary.txt` files). Scripts are not in a separate subdirectory but spread out as well. A Python script in the root coordinates the grading steps, including language-dependent builds through a generic module. This script can be run locally or by a daemon inside our grading system (Peach, 2008). Other scripts are in (Unix) makefiles, e.g., for building tools and cleaning up, and in (Unix) shell scripts for viewing evaluation cases, and generating expected output through a known-correct solution. Evaluation drivers and stubs are stored with the test submissions, because in most cases they depend on the programming language. When communication is via sockets, it could be language independent, in which case, they are put in a `generic` subdirectory.

The current format does not standardize storage of additional task-specific information (other than texts), and of scripts to generate an archive for solvers. These things are handled in an ad hoc way.

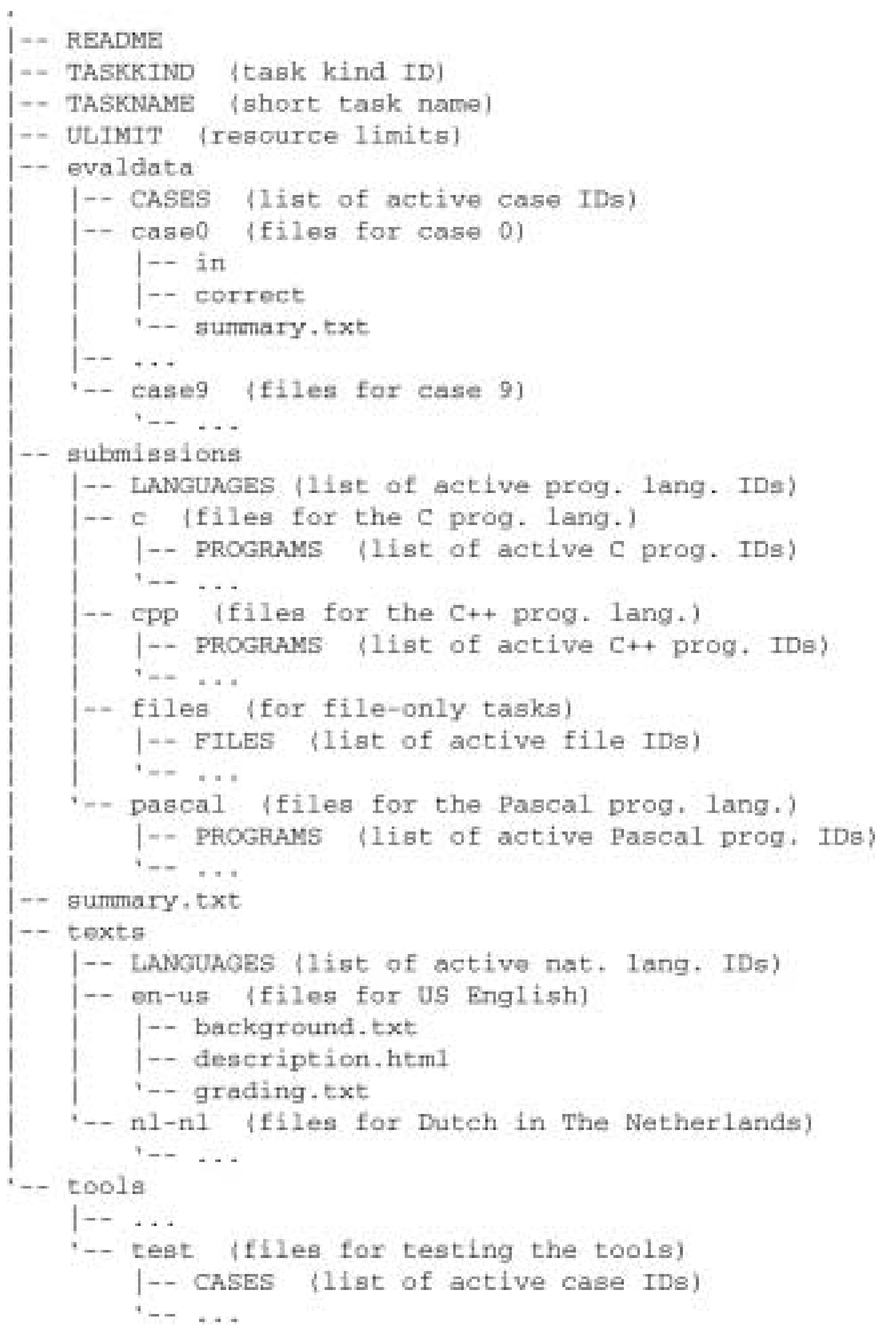
Platform dependencies and tool dependencies (like compilers and libraries) are handled implicitly by references. Submissions are graded on a Linux platform, whereas in our educational setting, most students use Windows. There are minor issues concerning compiler versions.

The current format does not support inheritance or sharing of common information. Related packages are mostly created by branching.

Some temporary files are created inside the package when using it. This limits the possibilities of *concurrent usage*. Evaluation-related files are stored in a working directory outside the package.

Peach^3, the latest version of (Peach, 2008), still uses the format introduced for Peach^2. We are working on an improved Peach Exchange Format to overcome the limitations mentioned above.

Our task packages are stored in a central Subversion repository for configuration management. This works quite well because most persistent data is stored in line-based text files. Each package is treated as a composite configuration item. Tags are used to record status changes, and branches are used for variants.



```
.
|-- README
|-- TASKKIND  (task kind ID)
|-- TASKNAME  (short task name)
|-- ULIMIT  (resource limits)
|-- evaldata
|   |-- CASES  (list of active case IDs)
|   |-- case0  (files for case 0)
|   |   |-- in
|   |   |-- correct
|   |   '-- summary.txt
|   |-- ...
|   '-- case9  (files for case 9)
|       '-- ...
|-- submissions
|   |-- LANGUAGES (list of active prog. lang. IDs)
|   |-- c  (files for the C prog. lang.)
|   |   |-- PROGRAMS  (list of active C prog. IDs)
|   |   '-- ...
|   |-- cpp  (files for the C++ prog. lang.)
|   |   |-- PROGRAMS  (list of active C++ prog. IDs)
|   |   '-- ...
|   |-- files  (for file-only tasks)
|   |   |-- FILES  (list of active file IDs)
|   |   '-- ...
|   '-- pascal  (files for the Pascal prog. lang.)
|       |-- PROGRAMS  (list of active Pascal prog. IDs)
|       '-- ...
|-- summary.txt
|-- texts
|   |-- LANGUAGES (list of active nat. lang. IDs)
|   |-- en-us  (files for US English)
|   |   |-- background.txt
|   |   |-- description.html
|   |   '-- grading.txt
|   '-- nl-nl  (files for Dutch in The Netherlands)
|       '-- ...
'-- tools
    |-- ...
    '-- test  (files for testing the tools)
        |-- CASES  (list of active case IDs)
        '-- ...
```

Fig. 1. Directory structure for information in Peach task package.

5. Conclusion

We have introduced the notion of a programming task package containing all task-related information, and serving as a unit for storage and communication. We have inventoried the stakeholders and contents of such packages, and the package interface and operations. This helps put in perspective the issues that arise when dealing with programming tasks on a larger scale.

Our automated programming education and contest hosting software (Peach, 2008) uses a package exchange format, which we have briefly described. The format is currently under revision to make it more generally usable. The Peach software is available under an open-source license.

At the moment, an application is lacking to handle task packages. Such an application should be supported on multiple platforms, and preferably should (also) provide a graphical user interface, possibly via a web browser.

Having a widely used task package format helps to improve the quality of programming tasks. But using task packages does not automatically lead to good quality. Task authors must still pay attention to many details when formulating a programming task; see for instance (Verhoeff, 2004).

We hope that this article will stimulate the development of an international standard for programming task packages. It would be good to standardize the interfaces of various task-specific tools as well. The *International Standard Task Number* and an *ISTN Registration Authority* (ISTN/RA) will then arise naturally.

Acknowledgments. The author is much indebted to Erik Scheffers, who co-designed Peach and who did most of the implementation work. We also wish to give credit to the more than 1500 users of Peach, who took part in dozens of courses and competitions, causing Peach to grade well over 25 000 submissions to date.

References

ACM ICPC (2008). *ACM International Collegiate Programming Contest*.
 `http://icpc.baylor.edu/` (visited March 2008).
CTAN (2008). *Comprehensive TEX Archive Network*.
 `http://www.ctan.org/` (visited March 2008).
DocBook (2008). `http://www.docbook.org/` (visited March 2008).
Docutils (2008). `http://docutils.sourceforge.net/` (visited March 2008).
IOI (2008). *International Olympiad in Informatics*.
 `http://www.IOInformatics.org/` (visited March 2008).
ISO 639-1 (2002). *Codes for the representation of names of languages – Part 1: Alpha-2 code*.
 `http://www.iso.org/iso/language_codes/` (visited March 2008).
ISO 3166-1 (2006). *Codes for the representation of names of countries and their subdivisions ĐPart 1: Country codes*.
 `http://www.iso.org/iso/country_codes/` (visited March 2008).
OpenOffice (2008). `http://www.openoffice.org/` (visited March 2008).
Peach^3 by E.T.J. Scheffers and T. Verhoeff (2008). Technische Universiteit Eindhoven, The Netherlands.
 `http://peach3.nl/` (visited May 2008).
RFC 3629 (2003). IETF Standard 63 concerning UTF-8, a transformation format of ISO 10646, Nov. 2003.
 `http://www.ietf.org/rfc/rfc3629.txt` (visited March 2008).

RFC 4646 (2006). IETF Best Current Practice concerning *Tags for the Identification of Languages*, Sep. 2006. `http://www.ietf.org/rfc/rfc4646.txt` (visited March 2008).

Sloan, N. (2008). *Online Encyclopedia of Integer Sequences*. `http://www.research.att.com/ njas/sequences/` (visited March 2008).

UVa Online Judge (2008). `http://icpcres.ecs.baylor.edu/onlinejudge/` (visited March 2008).

Verhoeff, T. (1988). *Guidelines for Producing a Programming-Contest Problem Set*. Oct. 1988, expanded July 1990. `http://www.win.tue.nl/ wstomv/publications/guidelines.pdf` (visited March 2008).

Verhoeff, T. (2002). *RobIn for IOI I/O*, version 0.7. July 2002. `http://www.win.tue.nl/ wstomv/software/robin/Doc07.txt` (visited March 2008).

Verhoeff, T. (2004). *Concepts, Terminology, and Notations for IOI Competition Tasks*. Sept. 2004. `http://www.win.tue.nl/ wstomv/publications/terminology.pdf` (visited March 2008).

Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A proposal for an IOI syllabus. *Teaching Mathematics and Computer Science*, **IV**(1), 193–216. `http://www.win.tue.nl/ wstomv/publications/ioi-syllabus-proposal.pdf` (visited March 2008).

W3C (2008). *World Wide Web Consortium*. `http://www.w3c.org/` (visited March 2008).

T. Verhoeff is an assistant professor in computer science at Technische Universiteit Eindhoven, where he works in the Group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI distinguished service award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.