DOMjudge Jury Manual

by the DOMjudge team Sun, 3 Nov 2013 18:22:07 +0100

This document provides information about DOMjudge aimed at a jury member operating the system during the contest: viewing and checking submissions and working with clarification requests. A separate manual is available for teams and administrators. Document version: 9f226f4

Contents

1 DOMjudge Overview

DOMjudge is a system for running a programming contest, like the ACM ICPC regional and world championship programming contests.

This means that teams are on-site and have a fixed time period (mostly 5 hours) and one computer to solve a number of problems (mostly 6-10). Problems are solved by writing a program in one of the allowed languages, that reads input according to the problem input specification and writes the correct, corresponding output.

The judging is done by submitting the source code of the solution to the jury. There the jury system automatically compiles and runs the program and compares the program output with the expected output.

This software can be used to handle the submission and judging during such contests. It also handles feedback to the teams and communication on problems (clarification requests). It has web-interfaces for the jury, the teams (their submissions and clarification requests) and the public (scoreboard).

1.1 Features

A global overview of the features that DOMjudge provides:

- Automatic judging with distributed (scalable) judge hosts
- Web-interface for portability and simplicity
- Modular system for plugging in languages/compilers
- Detailed jury information (submissions, judgings) and options (rejudge, clarifications)
- Designed with security in mind
- Has been used in many live contests
- Open Source, Free Software

1.2 Copyright and licencing

DOMjudge is developed by Jaap Eldering, Thijs Kinkhorst, Peter van de Werken and Tobias Werth. Development is hosted at Study Association A-Eskwadraat , Utrecht University , The Netherlands.

1.3 Contact

The DOMjudge homepage can be found at: http://www.domjudge.org/

We have a low volume mailing list for announcements of new releases.

The authors can be reached at the following address: domjudge-devel@lists.a-eskwadraat.nl . You need to be subscribed before you can post. See the list information page for subscription and more details.

Some developers and users of DOMjudge linger on the IRC channel dedicated to DOMjudge on the Freenode network: server `irc.freenode.net`, channel `#domjudge`. Feel free to drop by with your questions and comments.

2 General

The jury interface is accessed through a web browser. The main page shows a list of various overviews, and the most important of those are also included in the menu bar at the top. The menu bar will refresh occasionally to allow for new information to be presented. It also has the current 'official' contest time in the top-right corner.

Most pieces of information are clickable and bring up a new page with details. Many items also have tooltips that reveal extra information when the mouse is hovered over them. Problem, language and team pages have lists with corresponding submissions for that problem, language or team. Tables can be sorted by clicking on the column headers.

The most important pages are 'Submissions': the list of submitted solutions made by teams, sorted by newest first, and 'Scoreboard': the canonical overview of current standings.

2.1 Judges and Administrators

The DOMjudge system discerns between *judges* and *administrators* (admins). An administrator is responsible for the technical side of DOMjudge: installation and keeping it running. The jury web interface may be used by both.

Depending on configuration, there may either be a separate administrator view or one is shared between judges and administrators. In the first case you will not have access to the admin-specific options. In the latter, you may see options directed at admins, like options to edit or delete data. Only use these options if you're sure that it's correct to do so.

2.2 Scoreboard

The scoreboard is the most important view on the contest.

The scoreboard will display an upcoming contest from the given 'activatetime'; the contest name and a countdown timer is shown. Only at the first second of the real start of the contest it will show the problems to the teams and public, however. The jury always has a full view on the scoreboard.

It is possible to freeze the scoreboard at a given time, commonly one hour before the contest ends, to keep that last hour interesting for all. From that time on, the public and team scoreboard will not be updated anymore (the jury scoreboard will) and indicate that they are frozen. It will be unfrozen at a specified time, or by a button click in the jury interface. Note that the way freezing works, a submission from before the freeze and judged after may still update the scoreboard even when frozen.

The problem headings can display the colours of balloons associated with them, when set.

Nearly everything on the scoreboard can be clicked to reveal more detailed information about the item in question: team names, specific submissions and problem headers.

3 Before the contest

Before the contest starts, a number of things will need to be configured by the administrator. You can check that information, such as the problem set(s), test data and time limits, contest start- and end time, the time at which the scoreboard will be frozen and unfrozen, all from the links from the front page.

Note that multiple contests can be defined, with corresponding problem sets, for example a practice session and the real contest.

3.1 Problems and languages

The problem sets are listed under 'Problems'. It is possible to change whether teams can submit solutions for that problem (using the toggle switch 'allow submit'). If disallowed, submissions for that problem will be rejected, but more importantly, teams will not see that problem on the scoreboard. Disallow judge will make DOMjudge accept submissions, but leave them queued; this is useful in case an unexpected problem shows up with one of the problems. Timelimit is the maximum number of seconds a submission for this problem is allowed to run before a 'TIMELIMIT' response is given (to be multiplied possibly by a language factor). Problems can be imported and exported into and from DOMjudge using zip-files that contain the problem metadata and testdata files. See appendix B (DOMjuge problem format specification). Problems can have special *compare* and *run* scripts associated to them, to deal with problem statements that require non-standard evaluation. For more details see the administrator's manual.

The 'Languages' overview is quite the same. It has a timefactor column; submissions in a language that has time factor 2 will be allowed to run twice the time that has been specified under Problems. This can be used to compensate for the execution speed of a language, e.g. Java.

3.2 Verifying testdata

DOMjudge comes with some small tools to check for mistakes in the testdata. These tools are all located in the `misc-tools` directory in the source tree.

checkinput checkinput.awk fixinput.awk

The 'checkinput' programs are meant to check testdata input (and optionally also output). They check for simple layout issues like leading and trailing whitespace, non-printable characters, etc. There's both a C program and AWK script which do essentially the same thing. See 'checkinput.c' for details. All scripts take a testdata file as argument. The 'fixinput.awk' script corrects some of these problems.

checktestdata

This program can be used as a more advanced replacement of checkinput. It allows you to not only check on simple (spacing) layout errors, but a simple grammar file must be specified for the testdata, according to which the testdata is checked. This allows e.g. for bounds checking. See appendix A () for a grammar specification. Two sample scripts `checktestdata.{hello,fltcmp}` are provided for the sample problems `hello` and `fltcmp`.

This program is built upon the separate library `libchecktestdata.h` (see `checktestdata.cc` as an example for how to use this library) that can be used to write the syntax checking part of special compare scripts: it can easily handle the tedious task of verifying that a team's submission output is syntactically valid, leaving just the task of semantic validation to another program. When you want to

support 'presentation error' as a verdict, also in variable output problems, the option `whitespace-ok` can be useful. This allows any non-empty sequence of whitespace (no newlines though) where the `SPACE` command is used, as well as leading and trailing whitespace on lines (when using the `NEWLINE` command). Please note that with this option enabled, whitespace matching is greedy, so the script code

```
INT(1,2) SPACE SPACE INT(1,2)
```

does *not* match `1__2` (where the two underscores represent spaces), because the first `SPACE` command already matches both, so the second cannot match anything.

3.3 Testing jury solutions

Before a contest, you will want to have tested your reference solutions on the system to see whether those are judged as expected and maybe use their runtimes to set timelimits for the problems. There is no special method to test such solutions; the easiest way is to submit these as a special team before the contest. This requires some special care and coordination with the contest administrator. See the administrator's manual for more details.

3.4 Practice Session

If your contest has a test session or practice contest, use it also as a general rehearsal of the jury system: judge test submissions as you would do during the real contest and answer incoming clarification requests.

4 During the contest

4.1 Monitor teams

Under the Teams menu option, you can get a general impression of the status of each team: a traffic light will show either of the following:

gray
: the team has not (yet) connected to the web interface at all;

red
: the team has connected but not submitted anything yet;

yellow
: one or more submissions have been made, but none correct;

green
: the team has made at least one submission that has been judged as correct.

This is especially useful during the practice session, where it is expected that every team can make at least one correct submission. A team with any other colour than green near the end of the session might be having difficulties.

4.2 Judging Submissions

4.2.1 Flow of submitted solutions

The flow of an incoming submission is as follows.

1. Team submits solution. It will either be rejected after basic checks, or accepted and stored as a *submission*.
2. The first available *judgehost* compiles, runs and checks the submission. The outcome and outputs are stored as a *judging* of this submission.
3. If verification is not required, the result is automatically recorded and the team can view the result and the scoreboard is updated (unless after the scoreboard freeze). A judge can optionally inspect the submission and judging and mark it verified.
4. If verification is required, a judge inspects the judging. Only after it has been approved (marked as *verified*) will the result be visible outside the jury interface. This option can be enabled by setting `verification_required` on the *configuration settings* admin page.

4.2.2 Submission judging status codes

The interface for jury and teams shows the status of a submission with a code.

QUEUED/PENDING
: submission received and awaiting a judgehost to process it *;

JUDGING

a judgehost is currently compiling/running/testing the submission *;

TOO-LATE

submission received but submitted after the contest ended;

CORRECT

submission correct, problem solved;

COMPILER-ERROR

the compiler gave an error while compiling the program;

TIMELIMIT

program execution time exceeded the time defined for the problem;

RUN-ERROR

a kind of problem while running the program occurred, for example segmentation fault, division by zero or exitcode unequal to 0;

NO-OUTPUT

there was no output at all from the program;

WRONG-ANSWER

the output of the program did not exactly match the expected output;

PRESENTATION-ERROR

the submission only had presentation errors; e.g. difference in whitespace with the reference output.

* in the team interface a submission will only show as PENDING to prevent leaking information of problem time limits. The jury can see whether a submission is QUEUED or JUDGING. In case of required verification, a submission will show as PENDING to the team until the judging has been verified.

Under the Submissions menu, you can see submitted solutions, with the newest one at the top. Click on a submission line for more details about the submission (team name, submittime etc), a list of judgings and the details for the most recent judging (runtime, outputs, diff with testdata). There is also a switch available between newest 50, only unverified, only unjudged or all submissions. The default (coloured) diff output shows differences on numbered lines side by side separated by a character indicating how the lines differ: `!` for different contents, `$` for different or missing end-of-line characters, and one of <> when there are no more lines at the end of the other file.

Under the submission details the 'view source code' link can be clicked to inspect the source code. If the team has submitted code in the same language for this problem before, a diff output between the current and previous submission is also available there.

It's possible to edit the source code and resubmit it as the special 'domjudge' user. This does not have any effect for the teams, but allows a judge to perform a 'what if this was changed'-analysis.

A submission can have multiple judgings, but only one valid judging at any time. Multiple judgings occur when rejudging, see 4.2.3 (Rejudging).

4.2.3 Rejudging

In some situations it is necessary to rejudge a submission. This means that the submission will re-enter the flow as if it had not been judged before. The submittime will be the original time, but the program will be compiled, run and tested again.

This can be useful when there was some kind of problem: a compiler that was broken and later fixed, or testdata that was incorrect and later changed. When a submission is rejudged, the old judging data is kept but marked as 'invalid'.

You can rejudge a single submission by pressing the 'Rejudge' button when viewing the submission details. It is also possible to rejudge all submissions for a given language, problem, team or judgehost; to do so, go to the page of the respective language, problem, team or judgehost, press the 'Rejudge all' button and confirm.

Submissions that have been marked as 'CORRECT' will not be rejudged. Only DOMjudge admins can override this restriction for individual submissions.

Teams will not get explicit notifications of rejudgings, other than a potentially changed outcome of their submissions. It might be desirable to combine rejudging with a clarification to the team or all teams explaining what has been rejudged and why.

4.2.4 Ignored submissions

Finally, there is the option to *ignore* specific submissions using the button on the submission page. When a submission is being ignored, it is as if was never submitted: it is not visible to the team that sent it nor on the scoreboard. It will show striked through in the jury submissions list though. This can be used to effectively delete a submission for some reason, e.g. when a team erroneously sent it for the wrong problem. The submission can also be unignored again.

4.3 Clarification Requests

Communication between teams and jury happens through Clarification Requests. Everything related to that is handled under the Clarifications menu item.

Teams can use their web interface to send a clarification request to the jury. The jury can send a response to that team specifically, or send it to all teams. The latter is done to ensure that all teams have the same information about the problem set. The jury can also send a clarification that does not correspond to a specific request. These will be termed 'general clarifications'.

Under Clarifications, three lists are shown: new clarifications, answered clarifications and general clarifications. It lists the team login, the problem concerned, the time and an excerpt. Click the excerpt for details about that clarification request.

Every incoming clarification request will initially be marked as unanswered. The menu bar shows the number of unanswered requests. A request will be marked as answered when a response has been sent. Additionally it's possible to mark a clarification request as answered with the button that can be found when viewing the request. The latter can be used when the request has been dealt with in some other way, for example by sending a general message to all teams.

An answer to a clarification request is made by putting the text in the input box under the request text. The original text is quoted. You can choose to either send it to the team that requested the clarification, or to all teams. In the latter case, make sure you phrase it in such a way that the message is self-contained (e.g. by keeping the quoted text), since the other teams cannot view the original request.

The menu on every page of the jury interface will mention the number of unanswered clarification requests: "(1 new)". This number is automatically updated, even without reloading the page.

5 After the contest

Once the contest is over, the system will still accept submissions but will not judge them anymore. Teams will see this as a 'TOO-LATE' response.

If the scoreboard was frozen, it will remain frozen until the time set as unfreeze time, as seen under Contests. It is possible to publish the final standings at any given moment by pressing the 'unfreeze now' button under contests.

There's not much more to be done after the contest has ended. The administrator will need to take care of backing up all system data and submissions, and the awards ceremony can start.

A Checktestdata language specification

Grammar and command syntax below. A valid checktestdata program consists of a list of commands. All commands are uppercase, while variables are lowercase with non-leading digits. Lines starting with '#' are comments and ignored.

The following grammar sub-elements are defined:

```
integer  := 0|-?[1-9][0-9]*
float    := -?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
string   := ".*"
varname  := [a-z][a-z0-9]*
variable := <varname> | <varname> '[' <expr> [',' <expr> ...] ']'
value    := <integer> | <float> | <string> | <variable>
compare  := '<' | '>' | '<=' | '>=' | '==' | '!='
logical  := '&&' | '||'
expr     := <term> | <expr> [+-] <term>
term     := <term> [*%/] <factor> | <factor>
factor   := <value> | '-' <term> | '(' <expr> ')' | <factor> '^' <factor>
test     := '!' <test> | <test> <logical> <test> | '(' <test> ')' |
            <expr> <compare> <expr> | <testcommand>
```

That is, variables can take integer, floating point as well as string values. No dynamic casting is performed, except that integers can be cast into floats. Integers and floats of arbitrary size and precision are supported, as well as the arithmetic operators `+-*%/^` with the usual rules of precedence. An expression is integer if all its sub-expressions are integer. Integer division is used on integers. The exponentiation operator `^` only allows non-negative integer exponents that fit in an unsigned long. String-valued variables can only be compared (lexicographically), no operators are supported.

Within a string, the backslash acts as escape character for the following expressions:

- `\[0-7]{1,3}` denotes an octal escape for a character
- `\n, \t, \r, \b` denote linefeed, tab, carriage return and backspace
- `\"` and `\\` denote " and \
- an escaped newline is ignored (line continuation)

A backslash preceding any other character is treated as a literal backslash.

Tests can be built from comparison operators, the usual logical operators `!` `&&` || (not, and, or) and a number of test commands that return a boolean value. These are:

`MATCH(<string> str)`

Returns whether the next character matches any of the characters in 'str'.

`ISEOF`

Returns whether end-of-file has been reached.

`UNIQUE(<varname> a [,<varname> b ...])`

Checks for uniqueness of tuples of values in the combined (array) variables a, b, ... That is, it is checked that firstly all arguments have precisely the same set of indices defined, and secondly that the tuples formed by evaluating (a,b,...) at these indices are unique. For example, if x,y are 1D arrays containing coordinates, then `UNIQUE(x,y)` checks that the points (x[i],y[i]) in the plane are unique.

`INARRAY(<value> val, <varname> array)`

Checks if val occurs in one of the indices of array.

The following commands are available:

`SPACE / NEWLINE`

No-argument commands matching a single space (0x20) or newline respectively.

`EOF`

Matches end-of-file. This is implicitly added at the end of each program and must match exactly: no extra data may be present.

`INT(<expr> min, <expr> max [, <variable> name])`

Match an arbitrary sized integer value in the interval [min,max] and optionally assign the value read to variable 'name'.

`FLOAT(<expr> min, <expr> max [, <variable> name [, option]])`

Match a floating point number in the range [min,max] and optionally assign the value read to the variable 'name'. When the option 'FIXED' or 'SCIENTIFIC' is set, only accept floating point numbers in fixed point or scientific notation, respectively.

`STRING(<value> str)`

Match the string (variable) 'str'.

`REGEX(<string> str [, <variable> name])`

Match the extended regular expression 'str'. Matching is performed greedily. Optionally assign the matched string to variable 'name'.

`ASSERT(<test> condition)`

Assert that 'condition' is true, fail otherwise.

`UNSET(<varname> a [,<varname> b ...])`

Unset all values for variables a, b, ... This includes all values for array indexed variables with these names. This command should typically be inserted at the end of a loop after using UNIQUE or INARRAY, to make sure that no old variables are present anymore during the next iteration.

`REP(<expr> count [,<command> separator]) [<command>...] END`

`REPI(<variable> i, <expr> count [,<command> separator]) [<command>...] END`

Repeat the commands between the 'REP() ... END' statements count times and optionally match 'separator' command (count-1) times in between. The value of count must fit in an unsigned 32 bit int. The second command 'REPI' does the same, but also stores the current iteration (counting from zero) in the variable 'i'.

`WHILE(<test> condition [,<command> separator]) [<command>...]  END`

`WHILEI(<variable> i, <test> condition [,<command> separator]) [<command>...]  END`

Repeat the commands as long as 'condition' is true. Optionally match 'separator' command between two consecutive iterations. The second command 'WHILEI' does the same, but also stores the current iteration (counting from zero) in the variable 'i'.

`IF(<test> cond) [<command> cmds1...]  [ELSE [<command> cmds2...]]  END`

Executes cmds1 if cond is true. Otherwise, executes cmds2 if the else statement is available.

B DOMjudge problem format

DOMjudge supports the import and export of problems in a zip-bundle format. This zip file contains the following files in its base directory:

`domjudge-problem.ini`

: This file has a simple INI-syntax and contains problem metadata, see below.

`problem.{pdf,html,txt}`

: The full problem statement as distributed to participants. The file extension determines any of three supported formats. If multiple files matching this pattern are available, any one of those will be used.

`<testdata-file>.in` / `<testdata-file>.out`

: Each pair of `<testdata-file>.{in,out}` contains the input and correct/reference output for a single test case. Single files without their corresponding `in` or `out` counterpart are ignored. The order of the files in the zip archive determines the initial ordering of the testcases after import.

`<solution>.<ext>`

: Submits code of reference solution as team 'domjudge' if <ext> is a known language extension. The contest, the problem, and the language have to be enabled. The contest must be started. If you include a comment starting with '`@EXPECTED_RESULTS@:` ' followed by the possible outcomes, you can use the *judging verifier* in the admin interface to verify the results.

When importing a zip file into DOMjudge, any other files are ignored.

The file `domjudge-problem.ini` contains key-value pairs, one pair per line, of the form `key = value`. The `=` can optionally be surrounded by whitespace and the value may be quoted, which allows it to contain newlines. The following keys are supported (these correspond directly to the problem settings in the jury web interface):

- `probid` - the problem identifier
- `cid` - the associated contest identifier
- `name` - the problem displayed name
- `allow_submit` - allow submissions to this problem, disabling this also makes the problem invisible to teams and public
- `allow_judge` - allow judging of this problem
- `timelimit` - time limit in seconds per test case
- `special_run` - suffix tag of a special run script
- `special_compare` - suffix tag of a special compare script
- `color` - CSS color specification for this problem

The `probid` key is required when importing a new problem from the `jury/problems.php` overview page, while it is ignored when uploading into an existing problem. All other keys are optional. If they are present, the respective value will be overwritten; if not present, then the value will not be changed or a default chosen when creating a new problem. Test data files are added to set of test cases already present. Thus, one can easily add test cases to a configured problem by uploading a zip file that contains only `*.{in,out}` files.