The DLVHEX Benchmark System

Christoph Redl

February 21, 2015

1 Introduction

This readme file describes the benchmark system in the repository `https://github.com/hexhex/benchmarks`. The repository contains a directory `scripts` for benchmarking, i.e., comparing systems or system configurations to each other. In the following we refer to this directory by `benchmarks/scripts`.

Although the scripts have been developed for the HEX-program reasoner DLVHEX they can be used for experiments with other systems as well. The script system is designed to be easy to use yet customizable. The user is required to provide no more than one script for each benchmark problem, which essentially specifies problem-specific parameters and then delegates the call to generic scripts provided by this repository.

The system is intended to run benchmark instances with multiple system configurations and extract benchmark parameters (such as grounding and solving time, number of answer sets) for each. It supports automated scheduling of instances, parameter extraction, aggregating the parameters, generating a table in text or LaTeX format, and comparison of the results with previous runs. For scheduling benchmark instances, the system supports sequential runs using shell scripts only, or the HTCondor system[1].

2 Basic Usage of the System

In order to use the benchmark system for a concrete benchmark problem, the file `run_template.sh` from the `benchmarks/scripts` directory should be copied to `run.sh` within the benchmark problem's directory. The file needs then to be adopted to the problem by modifying the lines marked with (1), (2) and (3) as follows.

(1) The loop condition which iterates over the instances of the benchmark needs to be configured. Usually this condition will iterate over files, but sometimes also an iteration in a certain range can be useful.

Example: `"instances/*.hex"`
iterates over all `.hex` files in directory `instances`.

Example: `"{1..20}"`
iterates over the integers from 1 to 20.

[1] `http://research.cs.wisc.edu/htcondor/`

(2) Here, the commands (e.g. calls of DLVHEX or another software systems) to be compared should be entered in a semicolon-separated list.

When comparing entirely different systems, the configurations specified at (2) should contain the command strings *including* the executable name and all options.

Example: `"dlv;clasp;dlv -n=1;clasp -n=1"`
compares DLV and CLASP runs for computing
all and the first (`-n=1`) answer set, respectively.

If all configurations compare the same system (e.g. DLVHEX) but with different options, then as a shortcut the static part can (but does not need to) be part of the string at (3) (see below), while (2) may contain only the changing parts of the command.

Example: `"flpcheck=explicit;--flpcheck=ufs"`
compares two DLVHEX configurations, where the actual reasoner call is specified at (3) (see below).

(3) Here, the static part of the actual shell command for a reasoner call needs to be inserted (i.e., without configuration-specific options).

The tokens `INST` and `CONF`[2] will be substituted by the instance name and the current configuration, respectively.

If the string at (2) contains full commands, then the string at (3) should be `"CONF INST"`. Since `CONF` in (3) will be substituted by the elements from the configuration string before executing the command, this will produce complete reasoner calls.

Example: Let `"dlv;clasp;dlv -n=1;clasp -n=1"` be the
configuration string at (2).
Then the string `"CONF INST"` at (3) will evaluate:

```
dlv INST
clasp INST
dlv -n=1 INST
clasp -n=1 INST
```

for all instances `INST`.

If the string at (2) contains only the changing part of the command, then the static part must contained in the string at (3).

Example: Let `"flpcheck=explicit;--flpcheck=ufs"` be the
configuration string at (2).
Then the string `"dlvhex2 -n=1 CONF INST"` at (3) will evaluate:

```
dlvhex2 -n=1 --flpcheck=explitit INST
dlvhex2 -n=1 --flpcheck=ufs INST
```

for all instances `INST`.

[2] Alternatively to `INST` and `CONF`, the meta-variables `\$instance` and `\$config` within the command string can be used (e.g. `dlvhex2 -n=1 \$config \$instance`); note the backslash in front of the $ characters which makes sure that the variables are not immediately resolved but are part of the string and are to be resolved later when an individual instance and configuration is to be inserted.

Note that the scripts `runinsts.sh` and `runconfs.sh` support more parameters than shown in `run_template.sh`. However, the default values are good for most benchmarks. If the aggregation of the results or the extraction of benchmark parameters from the runs needs to be customized, then Section 3 is a good starting point to learn about the parameters.

It is convenient to add the path to the `benchmarks/scripts` directory to the `PATH` variable. In this case, scheduling the benchmark instances is possible by simply calling `./run.sh`, otherwise the path to this directory needs to be passed as parameter to `run.sh` (cf. Section 3.1).

Moreover, the beginnings of `run.sh` scripts are in many cases exactly like in `run_template.sh`. Thus this part can be directly imported by

```
source dlvhex_run_header.sh
```

provided that the benchmark script directory is the `PATH` variable. This also defines the aliases `all=0`, `instance=$2`, `to=$3`, `bmscripts=$4` if `$1` is `single`, and `all=1`, `to=$2`, `bmscripts=$3` otherwise (cf. Section 3.1). Moreover, `mydir` will be set to the absolute path of the directory where the current `run.sh` script is located and `req` to the requirements file to be used.

3 System Architecture and Customization

The basic system architecture is visualized in Figure 1. There is a single benchmark-specific file `run.sh`, which is usually stored in the benchmark's directory (different from the `benchmarks/scripts` directory). This file can be used for scheduling all instances and for evaluating a single instance (controlled by parameters). Normally, the user calls this file without any parameters, which schedules all instances (due to the default values for the parameters). Internally, `run.sh` uses scripts from the benchmark scripts directory (`runinsts.sh` and `runconfs.sh`; those are not benchmark-specific) to schedule all instances for all configurations which are to be compared. Those scripts will make callbacks to `run.sh` for evaluating single instances (appropriate parameters make this script run a single instance rather than all instances in the initial call). Note that the two `run.sh` occurrences in Figure 1 refer to a single physical file (symbolized by their connection).

For each finished instance, the script system will call an *output builder* which extracts the actual benchmark parameters the user is interested in, e.g. time information and number of answer sets, from the reasoner output (standard output and standard error). The result is stored in a separate result file for each instance. When all instances have finished, the script system further calls an *aggregation script*, which generates the final benchmark table from the results of individual instances. The script system provides a standard output builder `outputbuilder.sh` and a standard aggregation script `aggregateresults.sh`, which are good for many benchmark problems, but customization is possible by providing alternative scripts (which need to be specified within `run.sh` as described in Sections 3.2 and 3.3).

The following subsections describe the scripts of the benchmark system in more detail. Note that except for `run_template.sh`, which serves as an example for writing a benchmark-specific file `run.sh`, none of these files need to be modified by the user.

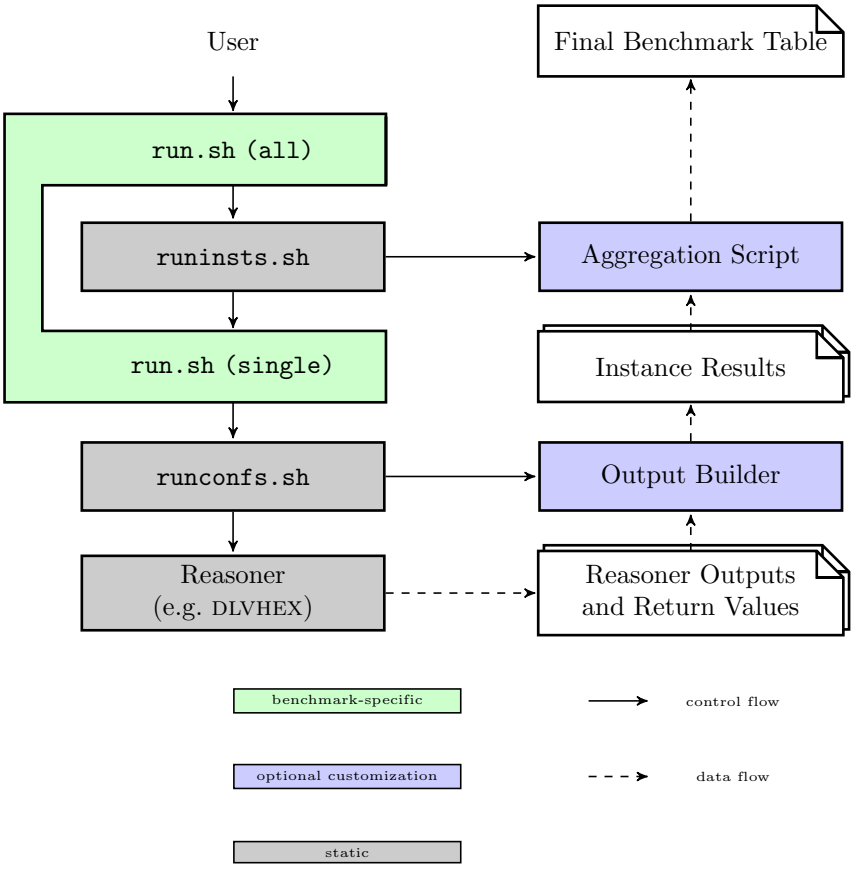

Figure 1: Benchmark System Architecture

3.1 run_template.sh

This file is an example for an implementation of `run.sh`. It specifies how to execute a single instance under various configurations and how to iterate over the instances of the benchmark.

The parameters mainly specify whether to execute all instances of the benchmark (using further scripts from the benchmark system) or a single instance. Note that this script is (indirectly) recursively called: usually, the first call is to evaluate all instances, which internally calls `runinsts.sh`, which in turn makes another (recursive) call of `run.sh` for evaluating every single instance. A single instance is then usually evaluated under a number of configurations, for which the script `runconfs.sh` is available.

However, this recursion is managed by the benchmark system. The user only needs to adopt `run.sh` as described in Section 2.

Parameters:

`$1`: (optional) `all`, `allseq` or `single`, default is `all`.

- If `$1` is `all` then the instances are evaluated in parallel using HTCondor. There are no further mandatory parameters.

  `$2`: (optional) Timeout in seconds, default is 300.

  `$3`: (optional) Directory with the benchmark scripts.
- If `$1` is `allseq` then the instances are evaluated sequentially. There are no further mandatory parameters.

  `$2`: (optional) Timeout in seconds, default is 300.

  `$3`: (optional) Directory with the benchmark scripts.
- If `$1` is `single` then a single instance is evaluated. There are three more parameters, one of which is optional:

  `$2`: Instance name.

  `$3`: Timeout in seconds.

  `$4`: (optional) Directory with the benchmark scripts.

3.2 runinsts.sh

This script implements a loop over instances and schedules them for execution, which is done either sequentially or in parallel using HTCondor. It is usually called from `run.sh` of a concrete benchmark problem.

The parameters of the script control the loop condition and the command used for executing a single instance. This script will also call the aggregation script (cf. Section 3.5) after execution of all instance to produce the final benchmark table.

Parameters:

`$1`: (optional) Instance loop condition, default is `*.hex`.

`$2`: (optional) Single benchmark command, default is `./run.sh`.

`$3`: (optional) Working directory, default is `$PWD`.

`$4`: (optional) Timeout in seconds, default is 300.

`$5`: (optional) Custom aggregation script, default is `aggregateresults.sh` in the `benchmarks/scripts` directory. Custom aggregation scripts need to behave as described in Section 3.5.

`$6`: (optional) Name of the benchmark, default is the name of the working directory.

`$7`: (optional) Requirements file (cf. Section 3.6).

When the script makes a call for executing a single benchmark instance as specified by `$2`, it will run

`$2 single INST TO SCRIPTS`,

where `INST` is the current instance (an element of the loop specified in `$1`), `TO` is the timeout specified in `$4` and `SCRIPTS` is the path of the benchmark scripts directly.

3.3 `runconfs.sh`

This script implements the evaluation of a single instance under various configurations. It is usually called from `run.sh` of a concrete benchmark problem that needs to evaluate a single instance.

The parameters control the configurations to be compared and the name of an output builder.

Parameters:

`$1`: Command for executing an instance, which may contain the constant `CONF` as a placeholder for options to be inserted and `INST` as a placeholder for the instance.

`$2`: A semicolon-separated list of configuration strings to be substituted for `CONF` in `$1`.

`$3`: Instance to be inserted in the command string in `$1` for `INST`.

`$4`: Timeout in seconds.

`$5`: (optional) Custom output builder name of a script to build the output of a run, which gets the parameters, default is `timeoutputbuilder.sh` in the `benchmarks/scripts` directory. A custom output builder needs to behave as described in Section 3.4.

3.4 Output Builders

An output builder is a script which extracts the actual benchmark parameters from a run. This is the information the user is interest in. Usually it includes timing information (e.g. grounding and solving time), the number of answer sets, etc.

An output builder gets the following parameters:

`$1`: Return value of the reasoner call (0 if success, 124 if timeout, any other value $\neq 0$ if failed).

`$2`: File with the overall runtime of the reasoner call.

`$3`: File with standard output of the reasoner call.

`$4`: File with standard error of the reasoner call.

The output builder needs to return 0 if it succeeded (note that the output builder can succeed in extracting information even if the reasoner failed!) and $\neq 0$ if the output builder itself failed.

It further needs to write a space-separated list of the actual benchmark parameters to standard output. This is usually a list of numbers to represent timing information, timeouts, number of answer sets, etc. For time values, the special values `---` and `===` for representing timeouts and memory outs, respectively, are supported. If `---` or `===` is output for some columns, the output builder can still return 0, i.e., output building counts as succeeded. In contrast, if the output builder completely fails to extract some information, it should output `FAIL` for the according column and return $\neq 0$; detailed error descriptions are allowed on standard error.

The benchmark script directory provides the following predefined output builders:

- `timeoutbuilder.sh` extracts the overall time (one time column and a column 0/1 for tagging successful/timeout instances).
- `gstimeoutbuilder.sh` extracts the overall time, the grounding and the solving time (each accompanied by a column 0/1 for tagging successful/timeout instances).
- `ansctimeoutputbuilder.sh` extracts the overall time (accompanied by a column 0/1 for tagging successful/timeout instances) and the number of answer sets.

3.5 Aggregation Scripts

An aggregation script generates the final output of a benchmark as a table in text format, given the results of individual instances. As standard input the file gets the concatenation of the results for all instances produced by the output builder (cf. Section 3.4).

In the input, column 1 is interpreted as instance size. The script is expected to aggregate over all instances of the same size and to output a single row for each instances size.

Note that a custom aggregation script only needs to read from standard input and write to standard output and is not expected to interpret any parameters. However, the default aggregation script `aggregateresults.sh` in the `benchmarks/scripts` directory provies some optional parameters, which allow in many cases to write custom aggregation scripts that simply delegate the call to the default one with appropriate parameters.

Parameters:

`$1`: (optional) Timeout value (in seconds) to be used for `---` and `===` instances.

`$2`,`$3`: (optional) Start position and length of instance size information in the instance filenames (index origin 0), or 0 and 0 for auto-detection, default is auto-detection.

`$4`: (optional) Comma-separated list of column indexes to compute means.

`$5`: (optional) Comma-separated list of column indexes to compute maxima.

`$6`: (optional) Comma-separated list of column indexes to minima.

`$7`: (optional) Comma-separated list of column indexes to sums.

The default for `$4-$7` is to compute means of odd and sums of even columns (interpreted as times and timeouts, respectively).

If a customized aggregation script is specified in the call of `runinsts.sh` (cf. Section 3.2), then this script can delegate the call to the default script `aggregateresults.sh` with appropriate parameters.

3.6 Requirements File

The requirements file specifies the hardware resources to be allocated for the benchmark run. An example is available in `req_template`.

The file is roughly a resource specification to be added to HTCondor job files[3]. In addition, the following specification may be part of a resource file:

- `# sequential`
  Evaluate all instances in sequence (not using HTCondor).
- `# ExtendedNotification = mail@address.com`
  Send the benchmark results by e-mail.

Remark: The leading # is important to make sure that HTCondor ignores these lines as they are not valid HTCondor syntax. However, the script system will still interpret them.

The script system will use as requirements file the first one of the following which exists:

1. The file passed as argument `$7` to `runinsts.sh`.
2. The file `req` in the same directory as the `run.sh` script.
3. The file `req` in the `benchmarks/scripts` directory.

4 Advanced Features

This section discusses the advanced features of the benchmark system, which are needed for table formatting and comparison of multiple benchmark runs.

[3] `http://research.cs.wisc.edu/htcondor/`

4.1 Table Editing

Multiple tables generated by the benchmark system can be merged by calling

```
mergetables.sh table1.dat table2.dat ...,
```

where `table1.dat`, `table2.dat`, ... are tables with the same number of rows. Each row of the resulting table will be the concatenation of the respective rows of the input tables, keeping the order of the tables specified in the call.

A table `table.dat` (which can also be a merged or previously projected table) can be projected to certain columns by calling

```
cat table.dat | colselect.sh i1 i2 ...,
```

where `i1`, `i2`, ... are 1-based column indexes which specify the columns to select. The same index can occur multiple times. The order of the output columns corresponds to the order specified on the command line.

4.2 Generating LaTeX-Tables

A benchmark table `table.dat` in text format may be sent (as standard input) to the script `tolatextable.sh` to produce valid LaTeX code. The script can produce the table header using different LaTeX packages or the table body only.

Parameters:

`$1`: (optional) LaTeX package to use for creating table header.
Valid values: `none`, `standard`, `booktabs`, default is `none` (only table body)

`$2`: (optional) Pattern how to format each line
The string needs to consist of constant parts and expressions of form `${val[i]}` to refer to the value of the i-th colum or `${fill[i]}` to refer to a sequence of double-tilde ($\sim\sim$) required to align this column.
Default:
`${val[0]} ${fill[1]}(${val[1]}) &` ↩
`  ${val[2]} ${fill[3]}(${val[3]}) & ...`
(pairs of columns are put in the same cell with even columns being interpreted as times and odd ones as timeouts which are put in parentheses)
The pattern needs to be specified without final line break (\\).

`$3`: (optional) `subst` (as a constant string)
If specified, then `$2` supports a more convenient syntax: `${val[i]}` and `${fill[i]}` can be written as `val[i]` and `fill[i]`, respectively.

4.3 Running Multiple Benchmarks

The system supports to run multiple benchmarks and a final script for post-processing with a single call. To this end, the script `multibenchmark.sh` should be used.

Parameters:

`$1`: A file which defines the benchmarks to be executed. Each line of this file is of form

`BENCHMARKNAME=PATH`,

where `PATH` needs to point to the `run.sh` script of the according benchmark.

`$2`: Output directory for the overall benchmark results.

`$3`: Requirements file to be used for all benchmarks (cf. Section 3.6).

`$4`: (optional) Either script to be executed after benchmarks have been finished (working directory will be `$2`), or a directory with reference benchmark results.

After completion of all benchmarks, `$4` will be called (if existing); it gets the names of all benchmarks as parameters and is intended to compare the new results to reference results.

4.4 Comparison to Previous Results

The benchmark system supports to compare benchmark results to previous runs. To this end, `multibenchmark.sh` supports to specify in `$4` a directory with reference output. It is expected that for each benchmark with name `BM`, there is a reference output in `$4/BM/BM.dat`.

By default, the comparison procedure checks if there is an instance and configuration with a difference in runtime of more than 5 seconds and more than 10%. However, this behavior can be overridden for each benchmark by providing a custom comparison file `compare.sh` in the same directory as the `run.sh` file of the according benchmark.

Script `compare.sh` gets as parameters:

`$1`: File (table) with the current benchmark results.

`$2`: File (table) with the reference benchmark results.

It is expected to return 0 if there were no significant changes, 1 if comparison failed, and 2 if there were significant changes.

4.5 Running the DLVHEX Standard Benchmarks

The script `runall.sh` allows for running all standard benchmarks which are currently available, provided that the DLVHEX core and all required plugins are installed.

To prepare this process, the script `setup.sh` downloads and installs all required DLVHEX software packages from scratch.

Parameters:

`$1`: Destination directory where all software packages are built and installed and where benchmark results are stored.

`$2`: (optional) URL of the benchmark instances, default is a Web server at Vienna University of Technology.

`$3`: (optional) Custom options for configuring DLVHEX and its plugins.

`$4`: (optional) Custom options for building DLVHEX and its plugins.

`$5`: (optional) Custom requirements file, cf. Section 3.6.

The whole process from checkout to the final benchmark tables can therefore be automated by calling

```
setup.sh PATH
```

for some `PATH` to prepare the benchmarks, and then calling

```
PATH/benchmarks/scripts/runall.sh
```

to run them.