Automated Benchmarking of KR-Systems*

Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
redl@kr.tuwien.ac.at

Abstract. Benchmarking is an important part of scientific work on solving techniques for KR systems. The implementation of hand-crafted scripts for each benchmark problem is cumbersome and repetitive. While most benchmarks are similar such that the process appears to be largely automatable, there are also differences which inhibit a complete reuse of existing scripts, e.g., different parameters to be measured and different aggregation functions to be applied. This calls for a tool which is applicable out of the box for a large range of benchmarks, but still allows for easy customization if needed. In this paper, we present such a system for automated benchmarking, which we base on a formalization of customizable benchmarks. The system captures the whole benchmarking process, including the run of individual instances, extraction of relevant information from the command outputs, aggregation of the results, and generation of the final benchmark table. A single command can then be used to generate the final table in LaTeX format which can conveniently be copied and pasted into a paper. In contrast to existing approaches, which usually focus on standardized settings such as in large-scale competitions, ours focuses on the possibility for customization, i.e., on benchmarks with possibly heterogeneous parameters and values to be measured, such as those which arise when evaluating new evaluation techniques.

Keywords: Benchmarking, Benchmark Description, Experimental Evaluation, System Description

1 Introduction

In this paper we present the ABC system (automated benchmarking based on HTCondor) for automating the experimental evaluation of KR-systems. This includes the execution of single instances, extraction of relevant information from the command outputs, aggregation of the results, and generation of the final LaTeX table. We aim at a system which is easy to use yet customizable and extensible. That is, it should be applicable out of the box for typical benchmarks, while advanced options and extensibility allow for adoption to less standardized cases. To this end, we allow for customizing many system components, but also provide default values and implementations which are good for a large number of experiments. We note that while our system was developed for evaluating systems in KR and AI, it can in fact be used for evaluating any command-line oriented tool.

Our contributions and the organization of this paper are as follows:

* This research has been supported by the Austrian Science Fund (FWF) project P27730.

– Before we present the system, we first *formalize benchmarks in a customizable way*. This is in order to identify the features which are to be supported by the system (Section 2).
– We then implement our formalization in a *script system* which is mainly based on standard Linux tools, the statistics system R[1], and the HTCondor load-balancing tool for parallelization and guaranteeing robust runtimes[2] (Section 3); all required tools are freely available.
– Finally, we discuss related systems and point out the differences to ours (Section 4), and conclude with a summary and an outlook (Section 5).

2 Formalization of Benchmarks

Towards a formalization of benchmark problems, we first discuss their typical structure informally. Our basic assumption is that each benchmark problem consists of a set of *instances* and a set of *configurations* to compare. Configurations can be different settings of a single system or entirely different systems to be compared.

When running the benchmark, the first step is to execute all instances under all configurations. Each run of an instance under a configuration yields several results, consisting of the output produced by the system to be evaluated (e.g, the contents written to the standard output and standard error streams, the exit code) and meta-information produced by the observer (e.g., runtime and memory consumption, timeout or no timeout).

Depending on the benchmark problem at hand, only parts of the results might be relevant. For instance, in many cases not the complete output is needed for the evaluation, but only in the number of lines (which often correspond to the number of results/models when evaluating KR-tools). Also the runtime and the memory consumption might be relevant or not depending on the benchmark scenario. For instance, besides typical resource usage, benchmarks may also measure the number of solutions found by an algorithm within a given time limit, or the quality of the solution rather than the required resources. To capture also these cases, we make use of the concept of *output builder functions* (or just *output builders*), which extract the relevant features of the full output of a single run.

Moreover, benchmark results presented in publications or on the Web usually do not contain results of individual instances, but rather aggregated results for classes of instances (e.g., sets of instances of the same size). Aggregation amounts often to the computation of an average value, but can also be application-dependent: for instance, if one wants to measure the overall runtime needed to solve a class of instances, then the aggregation function might be *sum* instead of *average*. Furthermore, if multiple values are measured, then the aggregation function might be a different one for each column. For instance, one might want to compute the average of the runtimes but the sum of 0/1 timeout flags. Hence, the possibility to customize aggregation functions is an important requirement.

[1] https://www.r-project.org
[2] https://research.cs.wisc.edu/htcondor

Benchmark problems. Based on these consideration, we formalize benchmark problems abstractly. In the following, let $\mathcal{I}$ and $\mathcal{C}$ be domains of instances and configurations, respectively. Usually, the instances are text files (e.g. SAT instances in DIMACS format, ASP programs, ontologies, etc) and the configurations are shell commands (e.g. reasoner calls). The sets $\mathcal{I}$ and $\mathcal{C}$ can thus in most cases be defined as the sets of all files and strings, respectively. Moreover, we assume that $\mathcal{D}$ is a domain of all values which can occur in the final benchmark table; this will frequently be the set of all floating point values (with integers as special cases) and possibly special values such as "n/a", "timeout", "failed", etc.

Based on $\mathcal{I}$, $\mathcal{C}$ and $\mathcal{D}$ and the considerations from above we define a benchmark problem as follows:

Definition 1. *A* benchmark problem *is a tuple* $B = \langle (I_1, \ldots, I_\ell), C, o, a \rangle$ *composed of a list of sets of instances* $I_1, \ldots, I_\ell \subseteq \mathcal{I}$*, a list of configurations* $C \subseteq \mathcal{C}$*, an output builder function* o *and an aggregation function* a*.*

The instances are organized in sublists $I_1, \ldots, I_\ell$, each of which is meant to be aggregated to one entry in the final benchmark table. For instance, sublist I_j might contain all instances of size j which are to be aggregated to a single row. For such a benchmark B, we let I refer to the concatenation of lists $I_1, \ldots, I_\ell$. Note that the instances and configurations are ordered to resemble the desired ordering in the final table, i.e., the aggregated entry for I_i occurs before I_j whenever $i < j$; similarily for configurations.

Example 1. Suppose we want to compare the runtime of multiple SAT-solvers. Then $\mathcal{I}$ is the set of all syntactically wellformed DIMACS files, $\mathcal{C}$ is a set of SAT solver calls, and $\mathcal{D}$ is the set of all floating point values.

Under the assumption that we have two different instance sizes 1 and 2 wrt. the number of variables (for simplicity), we have that I_1, I_2 are sets of SAT-instances to be run. The instances grouped by the number of variables $1 \leq i \leq 2$. For the sake of simplicity we further assume that each group contains only $|I_1| = |I_2| = 2$ instances. The configurations C are a list of SAT-solvers to compare, possibly including command-line parameters, e.g. $C = (minisat^3, clasp^4, manysat^5)$. We assume that we are interested in the runtime and maximum memory usage from each run, and that aggregation should be done by computing the averages of runtimes resp. maximum memory usages of all instances. More details on the definitions of the output builder o and the aggregation function a follow below. □

Evaluating instances. In order to evaluate an instance, we need a (benchmark-independent) *evaluation function* ϵ which maps the instance and the configuration to the output from an abstract *output domain* $\mathcal{O}$. This step corresponds to the execution of the shell command with a given instance. As described above, the elements from the output domain consist of the values produced by the configuration (text output, return code)

[3] http://minisat.se
[4] http://potassco.sourceforge.net
[5] http://www.cril.univ-artois.fr/~jabbour/manysat.htm

and meta-information collected by the observer; the whole output domain is then the set of all possible outputs.

Definition 2. *The* evaluation function ϵ *is of type* $\epsilon\colon \mathcal{I} \times \mathcal{C} \to \mathcal{O}$ *and associates each instance* $i \in \mathcal{I}$ *and configuration* $c \in \mathcal{C}$ *with an output from* $\mathcal{O}$*.*

Selecting the relevant output. For most benchmarks, only parts of the output are relevant. We thus make use of an *output builder* o which extracts these relevant parts. This might be the number of lines in the standard output and the runtime. An output builder basically associates each instance output with a list of relevant output values. The length of this list and the data type depend on the desired final benchmark table.

Definition 3. *For a benchmark with domain* $\mathcal{D}$*, an* output builder o *is a function* $o\colon \mathcal{O} \to \mathcal{D}^n$*, where* n *is the number of values per instance and configuration measured by* o*.*

We assume that n is fixed for a given o. The previous two definitions together allow us to compute for an instance $i \in \mathcal{I}$ and a configuration $c \in \mathcal{C}$ the n relevant output columns $o(\epsilon(i, c))$.

Example 2 (cont'd). Continuing the previous example of benchmarking SAT-solvers, the output domain $\mathcal{O}$ contains all possible outputs, each of which consisting of the standard output (e.g. a satisfiability flag, possibly models), the standard error output (e.g. log information), the return value of the call (e.g. indicating satisfiability), and meta-information (e.g. observed runtime and memory consumption). The output builder o from our scenario extracts from this information the observed runtime and maximum memory usage and returns it as two floating point values, hence $n = 2$.

For a comparison of the three configurations $C = (minisat, clasp, manysat)$ and a concrete instance, say the first instance $i_{2,1} \in I_2$ of size 2, we have that $o(\epsilon(i_{2,1}, c))$ evaluates to a vector of floating point values of length 2 for each $c \in C$, which represents the runtime and maximum memory usage under configuration c. For instance, the results $o(\epsilon(i_{2,1}, minisat)) = (6.44, 2.40)$ $o(\epsilon(i_{2,1}, clasp)) = (3.53, 1.30)$ and $o(\epsilon(i_{2,1}, manysat)) = (1.12, 5.00)$ indicate that the runtimes of the three solvers were 6.44, 3.53 and 1.12 seconds, and the memory usages were 2.40, 1.30 and 5.00 MiB, respectively. □

Note that we assume that all configurations produce the same number n of output values, and all values come from the same domain $\mathcal{D}$. This might not always be the case as some values can be unavailable for some configurations. For instance, if different configurations compare runs for *computing all models* and for *computing the first model*, values such as the *average time between models* are not available in the latter case. Moreover, different output elements might come from different domains, e.g. if one wants to measure both satisfiability (Boolean value) and the runtime (floating point value). However, our assumption is wlog. because superfluous columns can be filled with dedicated "n/a" values and the domain $\mathcal{D}$ can be selected such that all required values are contained.

We can now define the benchmark table with individual instance results. The table consists of one row for each instance (therefore $|I|$ rows) and n columns for each configuration (therefore $|C| \cdot n$ columns in total).

Definition 4 (Instance Results Table). *The* instance results table $T_I(B)$ *associated with a benchmark B as by Definition 1 is the unique table of size $|I| \times |C| \cdot n$ such that $(ti_{u,v\cdot n+1}, \ldots, ti_{u,v\cdot n+n}) = o(\epsilon(I_u, C_{v+1}))$ for all $1 \leq u \leq |I|$, $0 \leq v < |C|$.*

An example is postponed until after Definition 6.

Intuitively, for n output values per instance and configuration, element $ti_{u,v}$ of the table should be an output value of instance u, such that $1 \leq v \leq n$ contain the output values of the first configuration, $n < v \leq 2n$ the values of the second, etc.

Results aggregation. We now turn to the aggregation of the results as the final step of the benchmark table generation. To this end, we make use of *aggregation functions* which map a set of rows from the instance results table to a single row in the aggregated table.

Definition 5. *An* aggregation function *for a benchmark B as by Definition 1 is a function $a\colon 2^{\mathcal{D}^{|C|\cdot n}} \to \mathcal{D}^{|C|\cdot n}$.*

Informally, an aggregation function maps a set of rows of length $|C| \cdot n$ to a single row of the same length. In the following, for a table $T_I(B)$ let $T_I(B)_k$ be its k-th row.

Definition 6 (Aggregated Results Table). *The* aggregated results table $T_A(B)$ *associated with a benchmark B (Definition 1) has rows $r_i = a(\{T_I(B)_{s+1}, \ldots, T_I(B)_{s+|I_i|}\})$ for all $1 \leq i \leq \ell$, where $s = \Sigma_{1 \leq j < i}|I_j|$ is the number of instances preceding instance group i.*

Informally, a row ℓ in the aggregated table is computed by passing all rows of $T_I(B)$, which correspond to the instances in instance group I_ℓ, together to the aggregation function.

Example 3 (cont'd). Continuing the previous example of benchmarking SAT-solvers, each row of $T_I(B)$ consists of $|C| \cdot 2$ columns because the output builder returns two values (runtime and memory consumption) for each instance and configuration.

Suppose the instance results table looks as follows:

$T_I(B)$	minisat		clasp		manysat		
	runtime	memory	runtime	memory	runtime	memory	
$T_I(B)_1$	0.04	0.10	1.21	1.00	0.51	0.40	I_1
$T_I(B)_2$	1.64	0.90	5.23	2.20	0.20	0.20	I_1
$T_I(B)_3$	6.44	2.40	3.53	1.30	1.12	5.00	I_2
$T_I(B)_4$	7.70	2.80	6.11	3.30	8.32	7.20	I_2

The first two rows represent the results of instances of size 1, the latter two from instances of size 2. The aggregation function a is separately applied to $\{T_I(B)_1, T_I(B)_2\}$ and $\{T_I(B)_3, T_I(B)_4\}$ and computes the columnwise average values. As above, for a table $T_A(B)$ let $T_A(B)_k$ be its k-th row. This yields table $T_A(B)$ with two rows:

$T_A(B)$	minisat		clasp		manysat	
	runtime	memory	runtime	memory	runtime	memory
$T_A(B)_1$	0.84	0.50	3.22	1.60	0.36	0.30
$T_A(B)_2$	7.07	2.60	4.82	2.30	4.72	6.10

□

Table $T_A(B)$ is the final result from a theoretical perspective. We turn now to the implementation perspective, where $T_A(B)$ is actually transformed into a LaTeX representation.

3 Implementation

We implemented the ABC system (automated benchmarking based on HTCondor) for computing the aggregated results (Definition 6) as a set of shell scrips which use mainly standard commands. It is available from https://github.com/credl/abcbenchmarking.

The system supports the execution of multiple instances in parallel using the *HTCondor* system[6], but can also be used without HTCondor in sequential mode. However, it is suggested to use it with HTCondor as this does not only exploit multi-core or multi-CPU environments but also guarantees robust runtimes, i.e., multiple runs have negligible derivations (provided that the tool to be evaluated has a deterministic behavior); in previous experiments conducted with the ABC system, the derivations between repeated runs were in the magnitude of 0.1 seconds.

The ABC system runs on all standard Linux systems with the *statistics system R* installed (https://www.r-project.org). Obviously, also HTCondor must be installed in order to run instances on top of it.

We give here a rough overview about the system and present some basic settings, but refer to the system documentation for a full description [5].

3.1 Basic Usage of the System

It is suggested to include the path to the ABC system in the system search path such that all scripts are found automatically without specifying absolute paths. Then the user has to create only a single benchmark-specific `run.sh` file, which is usually stored in the directory of the benchmark problem. It is expected to import a "header file" from the ABC system using

```
source run_header.sh
```

and calling the `run` method, defined in this header, with appropriate parameters. These parameters specify the benchmark problem at hand as by Definition 1, i.e., the instances, the configurations, the output builder and the aggregation function.

The system provides default values and implementations of most settings. For instance, the lists $I_1, \ldots, I_\ell$ of instances are automatically extracted from the overall list I of instances by interpreting the first number found in the filename of each instance as size and grouping the instances accordingly. For example, suppose `col_100_inst_1.dl`, `col_100_inst_2.dl`, `col_200_inst_1.dl` and `col_200_inst_2.dl` are ASP programs which encode 3-colorability instances. Then the filenames are interpreted

[6] https://research.cs.wisc.edu/htcondor

such that `100` resp. `200` specify two instance groups (with 100 and 200 nodes, respectively), each of which is composed of 2 instances. The file `run.sh` further supports several parameters (processing of these parameters is imported from `source run_header.sh`), for instance for overriding the default timeout value of 300 seconds or for executing only a specific instance.

In order to simplify editing and increase interoperability with other software programs, the generic output of the ABC system is a table in CSV format, but it comes with further scripts for conversion to LaTeX.

Example 4. In this example, our instances are given by all files of type `*.dlv` (DLV[7] programs) in the directory `instances`. The ABC system automatically iterates over these files.

We compare the configurations `dlv` and `dlv -n=1` to compare the computation of all and of the first solution. The configurations are specified as semicolon-separated list. The definition `combine="CONF INST"` specifies how a configuration and an instance are combined to the overall command, where the tokens `CONF` and `INST` represent the current configuration and the instance, respectively.

Optionally, one can define a benchmark name; if it is not defined, the current directory's name is used. In the example we further use the default output builder, which measures the overall runtime and the number of timeout instances, and the default aggregation script, which computes the average runtime and the count of timeout instances.

This is implemented in the following file `run.sh`:

```
source run_header.sh

# mandatory
instances="instances/*.dlv"
configurations="dlv;dlv␣-n=1"
combine="CONF␣INST"

# optional
benchmarkname="dlv"
aggregationfunc=""
outputbuilder=""

run "$instances" "$configurations" "$combine" \
    "$benchmarkname" "$aggregationfunc" "$outputbuilder"
```

Assuming that there are three groups of 10 instances of sizes 1, 2 and 3, the output of the call `./run.sh` is a table of the following form:

```
1 10 0.12 0 0.07 0
2 10 1.08 0 43.15 1
3 10 22.81 0 270.01 9
```

The rows read as follows: instance size 3 consists of 10 instances, the first configuration yields an average runtime of 22.81 seconds where 0 instances had timeouts, the

[7] www.dlvsystem.com

instance	dlv	dlv -n=1
1 (10)	0.12 (0)	0.07 (0)
2 (10)	1.08 (0)	43.15 (1)
3 (10)	22.81 (0)	270.01 (9)

Table 1: Benchmark Results: Final Appearance

second configuration yields an average runtime of 270.01 seconds where 9 instances had timeouts (aborted after 300.00 seconds).

This ABC system allows for an automatic translation of this table to LaTeX code as shown in Figure 1, which compiles to Table 1. □

```
\begin{table}[t]
\scriptsize
\centering
\begin{tabular}[t]{r|rrr}
\hline
instance & \verb+dlv+ & \verb+dlv -n=1+ \\
\hline
1 (10) & 0.12 (0) & 0.07 (0) \\
2 (10) & 1.08 (0) & 43.15 (1) \\
3 (10) & 22.81 (0) & 270.01 (9) \\
\hline
\end{tabular}
\caption{Benchmark Results}
\label{tab:results}
\end{table}
```

Fig. 1: Benchmark Results: LaTeX Code

For more custom use-cases, a different output builder and/or aggregation function can be specified in form of the filename of a separate script with a certain input-output behavior (cf. [5]). The ABC system however comes with a set of predefined output builders and aggregation scripts which are good for many use cases.

Our system computes the aggregated results table as by Definition 6:

Proposition 1. *For a benchmark as by Definition 1, an output builder as by Definition 3 and an aggregation function as by Definition 5, the ABC system computes the aggregated results table T_A as by Definition 6.*

3.2 Customization of the Output Builder and the Aggregation Function

A custom output builder as by Definition 3 is implemented as a shell script which extracts the relevant parts from the standard output and standard error of the command

and meta-information (the return code, the measured time and memory consumption). To this end, standard output, standard error output and meta-information are redirected to temporary files, whose filenames are passed to the output builder. The implementation of the output builder accesses these temporary files and extracts the relevant information, e.g. by the use of the `grep` command or similar.

Aggregation functions as by Definition 5 have full access to the instance results table T_I. The implementation supports fully customized scripts that transform table T_I into the final table T_A by arbitrary means. However, in many cases, users who implement custom aggregation scripts just want to specify the aggregation function (such as *avg*, *sum*, *max*, etc) for each column. Thus, the default aggregation script supports parameters which allow for selection of the aggregation function individually for each column and a custom aggregation script can just redirect the call to the default one with an appropriate selection of the parameters.

3.3 Additional Features

The ABC system provides additional scripts for post-processing and for conversion of the output format. This includes scripts for reordering or projecting columns or joining multiple tables and for formatting the LaTeX table using different packages.

Also email notifications to the user upon finishing the benchmark runs are supported. The notifications come with a textual representation of the benchmark results.

Finally, also a comparison of the current results to previous ones is possible. Although HTCondor guarantees robust runtimes, such that for deterministic reasoners the differences between mutliple runs are usually in the magnitude of 0.1 seconds, small differences are unavoidable. Therefore, the results must be statistically compared such that significant changes are discriminated from normal ones. Per default, the comparison method is based on a treshold both for relative and absolute changes of the results, but custom comparison methods are also supported. The system can be configured such that significant changes yield a warning, which can be used, for instance, for nightly tests which do not only identify traditional bugs (abnormal termination and wrong results) but also performance bugs (unintended performance decrease due to code modifications).

For details we refer to the system documentation [5].

3.4 System Architecture

The architecture of the ABC system is visualized in Figure 2. The benchmark-specific file `run.sh` largely delegates the call to the run method imported from `source run_header.sh` with appropriate parameters.

Internally, this method uses the scripts `runinsts.sh` and `runconfigs.sh` from the ABC system (those are not benchmark-specific) to schedule all instances for all configurations which are to be compared. Those scripts make callbacks to `run.sh` in a recursive fashion for evaluating single instances (appropriate parameters make this script run a single instance rather than all instances). Note that the two `run.sh` occurrences in Figure 2 refer to a single physical file (symbolized by their connection).

For each finished instance, the system calls an *output builder* (either the default one or a custom one specified in `run.sh`) 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* (either the default one or a custom one specified in `run.sh`), which generates the final benchmark table from the results of individual instances.

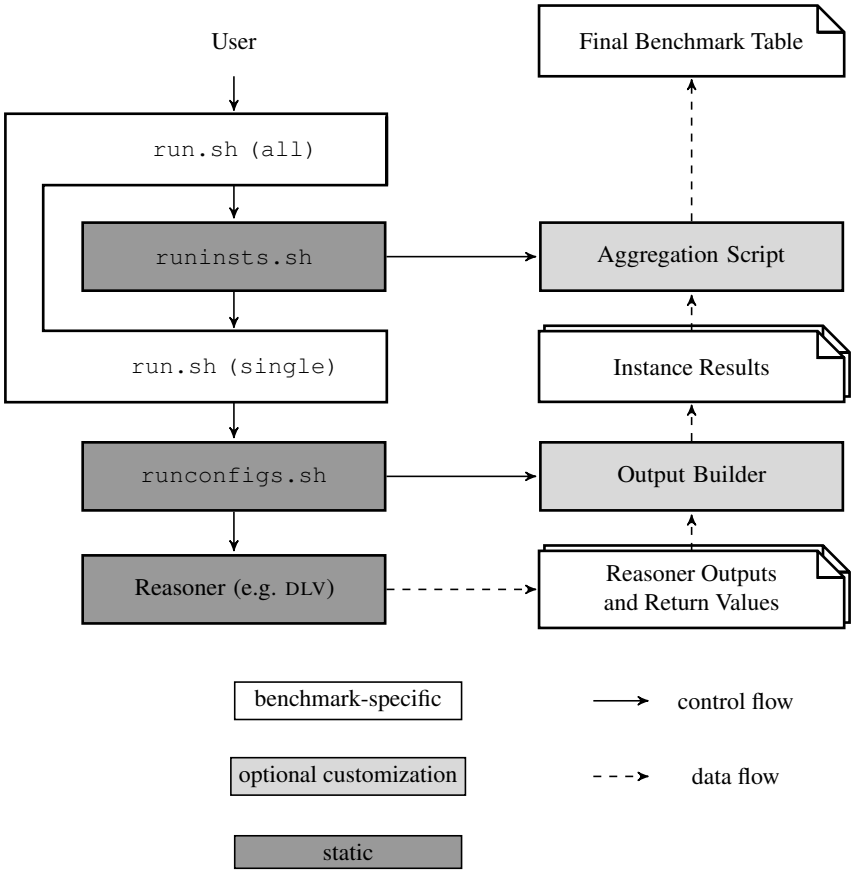

Fig. 2: ABC System Architecture

4 Discussion and Related Work

The ABC system is related to the VCWC workflow compiler, which was developed for running system competitions such as the ASP Competition [2]. Both systems are used to schedule problem instances, aggregate results, and compare systems in a largely automated workflow. However, our system focuses on individual experiments as part

of the development of new evaluation techniques for KR tasks. This differs from competitions in several points: most importantly, different aspects are considered static or dynamic. In competitions, the systems to be compared are much more standardized because organizers specify the interfaces to be provided by participating systems very precisely. In contrast, experiments presented in publications are often carried out with third-party systems which are not standardized at all and might be in an experimental state. It is not always feasible to standardize interfaces while core components are still under development. Therefore, the systems may be called in a much more heterogeneous way, i.e., command-line calls and the input might need to be adapted to single systems.

Also the experiments are less standardized. Competitions focus on a certain domain (such as ASP or SAT solving) and the number of types of benchmark problems is limited. In case of ASP (cf. e.g. [3]), this might be e.g. answer set computation, satisfiability checking and optimization problems. In contrast, a general benchmarking system cannot make such assumptions about the structure of the output. Each system provides a generic standard and error output (plus meta-information observed from outside such as the wall-clock time), and it depends on the experiment which values are relevant. Finally, the computation of the final statistics is also not hard-coded. Depending on the benchmark, the aggregation over multiple instances might use a different function. For instance, assume each instance provides the wall-clock time, a boolean timeout value (timeout or no timeout) and the memory consumption. When aggregating instances of the same size, one might want to compute the *average* runtime, the *sum* of binary timeouts, and the *maximum* memory consumption. On the other hand, due to a much larger amount of system runs in a competition, the requirements for recovery after software or hardware failures are possibly higher than with individual benchmarking.

The script system BMTOOL is related to our tool [4], but is more focused on the comparison of configurations as it does not support loop conditions such that automated iteration over the instances; also aggregation of the results is not implemented.

Related is also the DecBench system [1], which allows for the specification of benchmarks using answer set programs. Facts over special predicates specify solvers, test domains and instances, and derived atoms relate these entities by specifying which solvers are run with which instances. While the technical realization is different (ASP instead of shell scripts), the basic goal is similar to the one of the ABC system. However, DecBench's flexibility ends with the execution of the benchmarks. In contrast to the ABC system, the values to be included in the result table (cf. output builders) and the aggregation function cannot easily be customized (only data export in different formats is possible, while postprocessing is intended to be done by e.g. spreadsheet programs). Also support for high-throughput computing systems such as HTCondor is missing.

Concerning declarativity, we note that despite the use of shell scripts, benchmarks in our system are in standard cases specified by setting parameters and thus it can be considered largely declarative. We did intentionally not use a purely declarative language like ASP in DecBench because scripts provide more flexibility for customization. This is because declarative benchmark descriptions need to be compiled into an executable format (such as makefiles as in DecBench) anyway; but then the room for customization of a benchmark description is limited by the compiler, e.g., by the predicates which are interpreted. However, it is an interesting starting point for future work to provide an ASP frontend for standard use cases.

5 Conclusion

We presented the ABC system for automating benchmark experiments in the area of KR and AI research. To this end, we first formalized benchmarks in an abstract and customizable way. We then implemented a system based on shell scripts and the free Linux tools HTCondor and the statistics system R. Different from existing systems, we did not focus on the support for large-scale competitions but on experiments to be carried out as part of scientific work. While the former requires e.g. automated restarts and the possibility to add instances on-the-fly (cf. e.g. [2]), the interfaces of participating solvers are standardized by precisely specified input and output formats. In contrast, experiments for evaluating brand-new techniques have a much more ad-hoc character. Different from competitions, when comparing newly developed to third-party software as part of scientific work, the systems can usually not be assumed to have standardized interfaces. Also the parameters to be measured (runtime, number of models, etc) should be easily updatable since intermediate results while writing a paper might hint that further parameters are also relevant. In contrast, the valuation method for competitions is fixed before benchmark runs start. Therefore, a design goal when developing the ABC system was maximal flexibility, which was realized by the possibility for customized feature extraction scripts (called output builders) and aggregation scripts.

For future work, the use of a declarative language as a frontend is an interesting starting point. However, as declarative languages are often less suited for numeric computations (as needed when defining aggregation functions), it is challenging to retain flexibility. Other possible extensions are the support for alternative backends (e.g. makefiles with parallelization option instead of HTCondor) and the support for further output file formats in addition to LaTeX (e.g. XML, HTML tables).

References

1. Alviano, M., Cuteri, B., Ricca, F.: Declarative specification of benchmark sessions via asp. In: Proceedings of the Twenty-First RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (July 2014)
2. Charwat, G., Ianni, G., Krennwallner, T., Kronegger, M., Pfandler, A., Redl, C., Schwengerer, M., Spendier, L., Wallner, J.P., Xiao, G.: VCWC: a versioning competition workflow compiler. In: Cabalar, P., Son, T.C. (eds.) Proceedings of the Twelfth ernational Conference on Logic Programming and Nonmonotonic Reasoning, Corunna, Spain, September 15-19, 2013. LNCS, vol. 8148, pp. 233–238. Springer (September 2013), http://www.kr.tuwien.ac.at/staff/tkren/pub/2013/lpnmr2013-vcwc.pdf
3. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer. In: 5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009. LNCS, vol. 5689, pp. 40–110. Springer (2009), http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf
4. Faber, W.: A tool for benchmarking command-line systems. Tech. Rep. CS-2005-03, Computer Science Group (2005)
5. Redl, C.: The abc benchmarking system. Tech. Rep. INFSYS RR-1843-15-07, Vienna University of Technology, Institute for Information Systems (October 2015)