VCWC: A Versioning Competition Workflow Compiler*

Günther Charwat[1], Giovambattista Ianni[2], Thomas Krennwallner[1], Martin Kronegger[1], Andreas Pfandler[1], Christoph Redl[1], Martin Schwengerer[1], Lara Spendier[1], Johannes Peter Wallner[1], and Guohui Xiao[1]

[1] Institute of Information Systems, Technische Universität Wien, A-1040 Vienna, Austria
[2] Dipartimento di Matematica, Università della Calabria, I-87036 Rende (CS), Italy

1 Introduction

System competitions evaluate solvers and compare state-of-the-art implementations on benchmark sets in a dedicated and controlled computing environment, usually comprising of multiple machines. Running a competition is a time-consuming task that involves execution, maintenance, and data extraction of thousands of small tasks and runs producing a vast amount of log information for statistical and post mortem analysis. Each subtask in a competition could have several dependencies on other tasks before it can be executed, thus competition software must be aware of such intrinsic task prerequisites. For instance, before data extraction, we have to run a solver on a particular benchmark instance which is followed by verifying the result of the run using solution verification software. Only when the statistical data has been extracted for all runs of a benchmark instance for a particular solver, the next job will be to compute summary statistics of all those runs, and so forth.

Performing experimental evaluations in computer science like competitions is not a straightforward task: recent initiatives like [7] aim at establishing best practices in computer science evaluations, especially identifying measures to be taken for ensuring repeatability and classifying common pitfalls. Several programs exist that help running a competition, controlling its parameters and the surrounding environment, but they mainly focus on certain subtasks necessary for such an effort. For instance, Asparagus [1] focusses on maintaining benchmarks and instances thereof. Other known tools such as Runlim [20] and Runsolver [19] help to limit resources and measure CPU time and memory usage of solver runs. Other systems are tailored at specific needs of specific communities: the not publicly accessible ASP Competition evaluation platform for the 3rd ASP Competition 2011 [4] implements a framework for running a ASP competition. Another more general platform is StarExec [22], which aims at providing a generic framework for competition maintainers. The last two systems are similar in spirit, but each have restrictions that reduce the possibility of general usage: the StarExec platform does not provide support for generic solver input and has no scripting support, while the ASP Competition evaluation platform has no support for fault-tolerant execution of instance runs. Moreover, benchmark statistics and ranking can only be computed after all solver runs for all benchmark instances have been completed.

A robust job execution platform is a basic requirement for a competition. During benchmark evaluation, several different kinds of failures may happen, mainly

* This research is supported by the Austrian Science Fund (FWF) project P20841 and P24090.

(a) programming errors in the participant software;
(b) software bugs in the solution verification programs; or
(c) hardware failures during a run, which may be local to a machine (e.g., harddisk or memory failure) or global to the network (for instance, when the server room air condition fails).

The likelihood of failure (a) is very high when running a competition with several participants, and it is treated as failing a benchmark instance when ranking solvers and benchmarks. Participant solvers can fail for a variety of reasons: the execution environment must be aware of such potential problems, like waste of CPU or memory resources, and reap all left-over processes of crashed solvers, in order to prevent interference with follow-up jobs running on the same machine. Also, processes running wild may fill up harddisk space, and administrative intervention can be necessary to bring the system in a usable state again. Failure (b) is more subtle, as it involves the integrity of the competition. Nevertheless, no competition can be immune to such events, and detecting and treating such problems as early as possible is imperative. Problem (c) on the other hand might appear to be something nobody can foresee and might be impossible to prevent. The proper handling of such situations must be an integral part of the system, e.g., by reverting the execution to its last consistent state and properly resuming.

Moreover, a competition platform must be flexible enough to allow for "late" or updated benchmark and solver submissions. It is not uncommon that certain problems arise during the execution of the competition that have not been spotted while selecting benchmarks or when participants have submitted their final versions of their solvers. Changing the course of the competition after the platform has started executing is cumbersome and requires further effort for the competition maintainers.

However, there is no standard software that can address the problems and requirements mentioned above and on the same hand allows for a controlled and flexible evaluation of solvers on benchmark instances. Finding and preventing errors during execution cannot be guaranteed, but a fault-tolerant design helps the competition maintainers to perform all steps and minimizes the action required to come back to a safe state. To address this issues, we introduce the Versioning Competition Workflow Compiler (VCWC) system. VCWC uses a two-step approach: first, a workflow for a competition track is generated; a workflow is a dependency description of jobs that need to be executed in order to come to a ranking of solvers that participate in a competition track. Then, a versatile job scheduling system takes this workflow and executes it. Specifically, VCWC is based on

- GNU Make and GNU M4 for building the track execution workflow,
- the HTCondor [25] high throughput computing platform, which provides flexible means to support the requirements of running a competition, like job scheduling on a collection of benchmark servers, and
- the Directed Acyclic Graph Manager (DAGMan) [6], a meta-scheduler for HTCondor that maintains the dependencies between jobs and provides facilities for a reliable, fault-tolerant, and self-healing execution of benchmarking workflows.

VCWC is implemented using standard UNIX tools, thus it runs on every UNIX-like system that has support for those utilities.[3]

This paper has the following contributions:

- we first identify the major tasks and concepts that arise in a solver competition and then provide an overview of task dependencies and the resulting workflow for a competition track (Section 2);
- based on the requirements and tasks described before, we give in-detail information on the VCWC system and show how it has been implemented (Section 3); and
- we show how VCWC has been used in the ASP Competition 2013 in Section 4.

VCWC is open source and can be downloaded from https://github.com/tkren/vcwc.

2 Modeling a Competition

In this section, we describe the basic building blocks of a solver competition. We assume familiarity with the notion of *(computational) problem*, *instance*, and *solution* for a problem; an overview is given, e.g., in [17].

A *benchmark* B is a set of instances I from a well-defined computational problem, where all instances are represented in a standardized format (e.g., as logic programs or as CNF clauses). A *solver* S is an implementation for an algorithm that computes the solution for a given instance I from a benchmark B, where solutions are represented in a standardized format such as [13].

Given as set of benchmarks $\mathcal{B}$ and a set of solvers $\mathcal{S}$, we define a *track* T as a subset of $\mathcal{B} \times \mathcal{S}$ that is both left-total and right-total, i.e., for each $B \in \mathcal{B}$ there exists an $S \in \mathcal{S}$ such that $(B, S) \in T$, and for every $S \in \mathcal{S}$ there exists a $B \in \mathcal{B}$ such that $(B, S) \in T$. Intuitively, $(B, S) \in T$ means that solver S participates in track T in solving benchmark B. Each track has an associated computation environment $env(T)$ with a fixed number of CPUs, memory size, and available disk space. The set of all participating solvers to a track T is $\mathcal{S}(T) = \{S \mid (S, B) \in T\}$ and the set of all benchmarks is $\mathcal{B}(T) = \{B \mid (S, B) \in T\}$. Then, a *competition* is a collection of tracks. A *run* R of solver S on instance I in track T is the evaluation of S with instance I within the limits of the computation environment $env(T)$. A run has an associated solution $sol(R)$ and performance measurements for evaluation metrics such as runtime and memory usage. In a competition track, every instance is usually evaluated $k > 1$ times to eliminate outliers and to provide well-founded statistical results.

We exemplify now the notions shown above. In the ASP Competition series [3], a *system track* T forms a complete bipartite graph $(\mathcal{B} \cup \mathcal{S}, T)$, i.e., every solver participates in solving all benchmarks. On the other hand, the *model & solve* track does not have this restriction, a participating solver may choose which benchmarks to solve. Furthermore, tracks are usually classified as *sequential* or *parallel*, which means that their computation environment has exactly one CPU in case of sequential tracks, or more than one CPU in case of parallel tracks.

In a competition, several tasks need to be performed in order to evaluate a solver's performance relative to other solvers that participate in a certain track. The outcome

[3] The execution of the runs on the other hand is not limited to UNIX-like systems, as HTCondor and DAGMan exists for almost all major operating systems

of a competition is a ranking of the participating solvers, which should summarize the performance of a solver S on benchmark $\mathcal{B}$ relative to the other solvers that participate in a track. A *solution verification* $ver(R)$ of run R is a mapping $ver(R) \in \{0, 1, 2\}$ such that $ver(R) = 0$ whenever $sol(R)$ is not a solution for I, $ver(R) = 1$ for $sol(R)$ being a correct solution for I, and $ver(R) = 2$ otherwise. Note that $ver(R)$ might implement an incomplete verification algorithm, as solution verification could be a computationally hard task. The *solver summary statistics* $sumstat(S, B)$ computes for all runs $R_1, R_2, \ldots$ of solver S on instances I from benchmark B the performance measurements of those runs as summary statistics such as means, median, etc., for all instances $I \in B$. Based on $sumstat(S, B)$, the *benchmark ranking* $bmrank(B)$ of a benchmark B ranks each solver $S \in \mathcal{S}$ based on a predefined benchmark scoring function. Then, the *track ranking* $trackrank(T)$ generates a combined performance evaluation of a track T based on scoring function for $bmrank(B)$ for all benchmarks $B \in \mathcal{B}$.

Modeling the Dependencies in a Competition. As described above, several steps are necessary to generate the outcome $trackrank(T)$ of a competition track T. When combining all the tasks in a dependency graph, where nodes represent tasks and an edges (u, v) represent a dependency between u and v such that u must be executed before v, we get a task model of the competition track, which, when executed in sequence, computes all prerequisite information for each task properly and generates the desired outcome. Such a dependency graph forms a directed acyclic graph on the conceptual level, and can be seen as a track execution workflow for a competition track. Workflow execution platforms such as DAGMan [6] can take such a workflow and produce—using dependency resolution mechanisms—a sequence of tasks that can be executed in the right order. A welcoming side-effect is that job scheduling software such as HTCondor [25] can run each individual task potentially in parallel on several benchmark servers, thus speeding up the execution of the competition.

Based on the competition tasks introduced before, we explicitly outline in Fig. 1 the implicit dependencies of the tasks and show a competition workflow that can be used to perform all necessary computational tasks in a competition. Let $n = |\mathcal{S}|$, $m = |\mathcal{B}|$, and k be the number of runs per instance. Nodes $R^u_{v,w}[i]$ stand for the tasks associated with the i-th run, $1 \leq i \leq k$, of solver S_u on instance I_v of benchmark B_w. These tasks are comprehensive of computing the solution and perform the respective verification. The nodes ST^u_w represent the solver summary statistics task of solver S_u in benchmark B_w, i.e., ST^u_w takes all runs executed and verified on S_u that are associated with instances from B_w and creates summary statistics. Then, nodes BR_w represent the benchmark ranking jobs that are connected to all ST^u_w for $1 \leq u \leq n$. The topmost node TR is the track ranking task in a competition, while the lowest node r gives us the computation root, a unique entry point in the workflow without associated task.

Workflow Versioning. A further benefit of modeling a competition track as a workflow is to have a graph-based representation of tasks that can be easily modified and updated when basic constituents of a track change. To address the problem of late participant submissions or fixing broken benchmark instances after competition has already started, we introduce a workflow versioning mechanism.

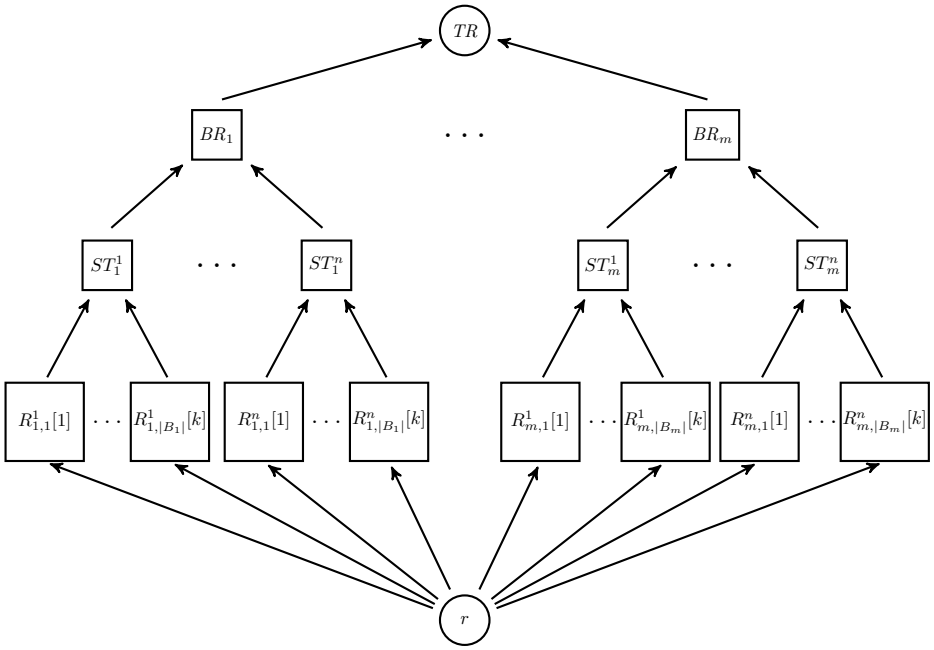

Fig. 1. Competition workflow for a track with m benchmarks and n solvers

Using these principles, one can add updated or fresh participants, benchmark (or instances thereof), or further runs. Additions and removals do not have impact on previously stored executions of the workflow, and broken runs do not appear in generated statistics. We identify the following use cases for changing the workflow of a track T:

1. adding a new benchmark B_{m+1} to T;
2. adding a new solver S_{n+1} to T; or
3. adding a new instance $I_{|B_j|+1}$ to a a particular benchmark B_j from T.

3 Implementation of the VCWC System

The system architecture of VCWC is shown in Fig. 2. The main components are

- the VCWC compiler, which generates a competition workflow description and profiles for instance parameters;
- DAGMan (Directed Acyclic Graph Manager), a meta-scheduler for managing dependencies between jobs built on top of
- HTCondor, a job scheduler for building high-throughput computing environments.

We show now the basic steps that are required to come from the principal components of a competition track T, i.e., benchmarks and participating solvers, to an executable workflow for T.

The first step before T can be executed is to prepare benchmark sets and to select instances for each benchmark. If track T is of the "model and solve" type, participating

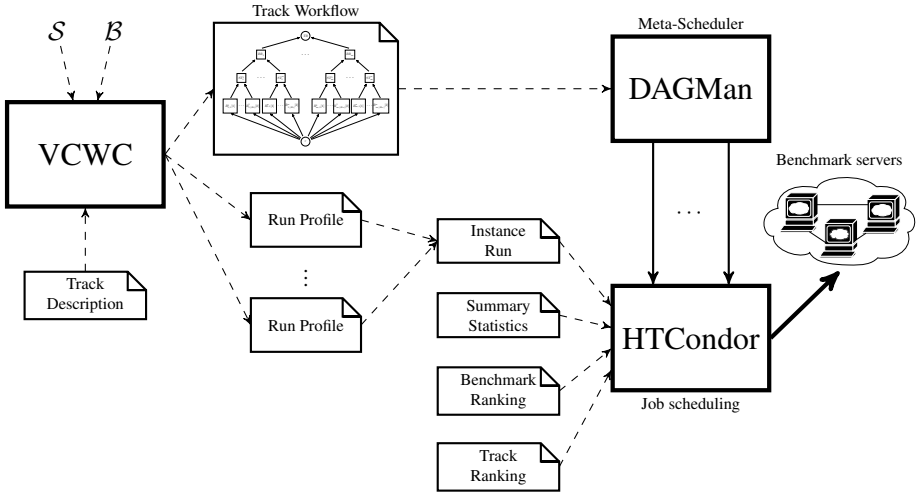

Fig. 2. VCWC System Architecture (dashed lines: data flow, solid lines: call flow)

solvers can choose which benchmarks they solve and prepare a dedicated tool for each benchmark. Alternatively, if T is of the strict "system track" type, all participating solvers are run on all benchmarks. Based on this, VCWC expects a benchmarks directory with all possible benchmarks $\mathcal{B}$ having corresponding subdirectories assigned to track T. The selected instances of a benchmark are stored in those subdirectories. Furthermore, additional information is required for mapping benchmarks to participating solvers. VCWC thus expects a dedicated `participants` directory of participating solvers $\mathcal{S}$, with subdirectories for each possible benchmark of a track. If one particular solver S signed up to solve benchmark B, the subdirectory called B will contain a directory S with the software solution for B for that particular solver S.

Then, VCWC requires a track description file that records the location of the `benchmarks` and `participants` folders, the track name, the workflow output directory, and other parameters. Using this description file as commandline argument, VCWC will then generate a workflow (called DAG file in DAGMan jargon) for T that can be processed by DAGMan, which submits jobs for execution in the network of benchmark servers. Further files that are required for track execution are job submission files for HTCondor. Job submission files describe the location of executables and arguments for particular jobs.

HTCondor is a high-troughput computing framework for distributed computation of computationally intensive tasks. Each task (job) that needs to be executed is first sent into a job queue, and based on priority management and job requirements (such as number of CPUs or memory) it is scheduled to run on one of the target machines that are free for new jobs and fulfill all job requirements. Using this framework, it is easy to pool several dedicated benchmark machines to form a powerful cluster for competition execution. The HTCondor job queue is persistent, i.e., once a job has been submitted for running, it will stay in the queue until it has been served by one of the available benchmark

machines and the computation result is available. This "fire-and-forget" mechanism survives machine reboots and system crashes, i.e., no administrative intervention is necessary after an unexpected system shutdown. Once HTCondor comes up again, it will re-schedule previously interrupted jobs and starts the computation from scratch again.

HTCondor itself does not keep track of job dependencies, this is DAGMan's responsibility. The DAGMan meta-scheduler takes one argument: the DAG file that has been generated by VCWC. When we start the execution for track T, DAGMan deploys a job to HTCondor that continuously runs and dispatches jobs in the workflow in the correct topological sort. DAGMan monitors the submitted jobs, keeps track of their exit codes, and only finishes successfully once all jobs have been successfully executed. Otherwise, it stops processing whenever a job fails and no further job can be submitted according to the current topological ordering. In this situation, DAGMan records successful and unsuccessful runs and exits: in this case, administrative intervention is required to bring the track execution back to life. After the reason for the job failure has been found and eliminated, a simple re-deploy of DAGMan on HTCondor resumes processing the workflow at the previously failed job.

In the following subsections, we will provide information how each component is implemented and provide details on how they are run.

VCWC. The VCWC tool consists of a wrapper shell script that invokes GNU Make [9] on a `Makefile`. First, this `Makefile` reads the track description, which references the `benchmarks` and `participants` folders as input, and generates lists of benchmark instances and solvers. Based on this information, the `Makefile` instantiates rules that tell GNU Make how to generate the DAGMan workflow.

For instance, a typical VCWC call and its output looks like this:

```
# vcwc trackinfo-t03.mk
Welcome to vcwc 0.1
generating workflow for track t03 with following setup:
- benchmarks: b01 b02 b04 b05 b06 b07 b08 b09 b10 b11 b12 [...]
- participants: s40 s42 s44 s60 s62 s63
- benchmarks/participants: b18/s40 b18/s60 b18/s42 b18/s63 [...]
- runs: r000 r001 r002
- workflow version: 000
- timestamp: 2013-04-26 14:34:15+02:00
compiling 90 runs for S/t03/b01/s40/000
[...]
compiling 78 runs for S/t03/b04/s40/000
[...]
compiling 6 participants for B/t03/b01/000
compiling 6 participants for B/t03/b02/000
[...]
linking 26 benchmarks for T/t03/001
```

This will generate a DAG file for DAGMan and run profiles for each individual instance run. The generated DAG workflow has always the same shape as Fig. 1. Each node in this DAG encodes the job type, which is an instance run, a solver summary statistics, a benchmark ranking, or the track ranking job. VCWC uses the GNU M4 macro processing language to instantiate workflow templates and run profiles based on the names of benchmarks, solvers, instances, and runs.

Adding new instances, benchmarks, or solvers to a track execution workflow is implemented as described in Section **??**. This can be done by simply adding instance or solver files and their corresponding directories to the `benchmarks` and `participants`

folders. GNU Make will automatically recognize that the input has changed, and will only regenerate those parts of the workflow that requires changes.

DAGMan and HTCondor. In this subsection, we exemplarily show how to execute a track using the VCWC system. Once the VCWC compiler generated a workflow file, we can submit the corresponding DAG for execution. This is simply done using a call to

```
# vcwc trackinfo-t03.mk submit
[...]
Running rescue DAG 2
[...]
Log of the life of condor_dagman itself : [...]/t03/track.dag.dagman.log
```

which calls `condor_submit_dag`. In the example output above we can see that DAGMan has successfully deployed its workflow manager to HTCondor. The message `Running rescue DAG 2` shows that DAGMan did not successfully execute the whole track before, either because a job failed and no further job could be run therefore, because the execution was forced to stop, or even because the system had to be restarted. DAGMan thus attempts to continue to run from a previous state stored in a `rescue` file with version 2 (hence, this is the third attempt to execute this track). A look into the HTCondor execution queue shows the following picture:

```
# condor_q -dag
-- Submitter: lion.kr.tuwien.ac.at : <10.0.0.100:52610> : lion.kr.tuwien.ac.at
ID          OWNER/NODENAME  SUBMITTED    RUN_TIME   ST PRI SIZE CMD
260226.0    aspstat         4/28 18:04   0+00:22:38 R  0   0.3  condor_dagman -f -
260347.0     |-R/t02/b05/s  4/28 18:07   0+00:02:00 R  0   0.0  run_instance.sh as
260348.0     |-R/t02/b05/s  4/28 18:07   0+00:02:00 R  0   0.0  run_instance.sh as
[...]
260358.0     |-R/t02/b01/s  4/28 18:09   0+00:01:39 R  0   0.0  run_instance.sh as
260359.0     |-R/t02/b01/s  4/28 18:09   0+00:00:00 I  0   0.0  run_instance.sh as
260360.0     |-R/t02/b01/s  4/28 18:09   0+00:00:00 I  0   0.0  run_instance.sh as
[...]
106 jobs; 0 completed, 0 removed, 93 idle, 13 running, 0 held, 0 suspended
```

In this situation, we see that there is one DAGMan instance executing our track, and 12 jobs are being run on the benchmark servers in parallel, while 93 jobs are pending to be executed on one of the machines connected to the benchmark cluster.

4 Use Case: ASP Competition 2013

VCWC has been developed as part of the ASP Competition 2013 evaluation software. A lot of experience had been gained when running the former competition, and the design of VCWC has profit from this. Special care has been given to have a versatile system that allows to address the failure sources (a)–(c) described above. Even though very unlikely, fatal hardware failures (c) do occur, in fact, during the execution of the ASP competition 2013, a broken valve actuator prevented to distribute chilled water from the backup cooling system, thus excess heat continued to warm up the data center to an ambient temperature of 45 degrees Celsius, and all server machines had to shut down. After the cooling loop was working again, starting up the benchmark servers automatically re-scheduled all unfinished jobs, and the track workflows continued to run without administrative intervention.

VCWC can easily handle thousands of benchmark runs. With 21 participants among two main tracks and 27 benchmark problems, VCWC has been put under intensive testing: The system track workflow consists of over 18000 jobs, and the size of the DAG file is about 3 MiB. It took about a minute to generate this file, mainly because a lot of small intermediate files had to be written to the harddisk during the compilation. While setting up the Competition, the incremental versioning system allowed to make fixes with no impact in the ongoing run. We got further mileage out of using GNU Make for the implementation of VCWC by using its parallel execution mechanism. In this scenario, we could profit from an immediate 4-fold speedup for compiling the workflows just by turning on parallel make execution on our benchmark servers with two 12-core AMD Opteron Processor 6176 SE processors and 128GiB RAM.

5 Conclusion and Related Work

Scientific competitions and evaluations are traditionally run on custom evaluation platforms. The degree of complexity and the type of service of such platforms might vary, ranging from simple benchmark repositories to fully-fledged evaluation systems. In the neighbouring communities it is worth citing the IPC Script collection [12] used throughout the International Planning Competition series up to its seventh edition [5]; the SMT-Exec platform [2, 21] adopted in Satisfiability Modulo Theories Competitions; the TPTP library and associated infrastructure [23] used in Automated theorem proving competitions [24]; the Quantified Boolean Formulas satisfiability community with its QBF-LIB library [8] and evaluation platform [18]; and, last but not least, the very active satisfiability community, with its large SATLIB collection [11] and longstanding experience in competition evaluation systems (see [14]). To the best of our knowledge none of the aforementioned platforms focus on workflow and workload management issues.

In the ASP community, our VCWC platform follows chronologically and is inspired by the Asparagus Web-based Benchmarking Environment [1] and the (not publicly accessible) Third ASP Competition evaluation platform [4]. An attempt at providing a general purpose platform, serving multiple communities and generalizing specific needs is the StarExec platform [22].

Among interdisciplinary initiatives aimed at fostering benchmark best practices we recall the Evaluate initiative and the Compare workshop [7, 15].

Future versions of VCWC may provide support for more fine-grained instance runs that allow to parametrize solver heuristics; currently, each solver setting requires a dedicated solver. While the filesystem-based data storage is good enough for most purposes, more powerful alternatives such as SQL storage might be of interest. Support for other job scheduling platforms such as Oracle Grid Engine [16] or Hadoop [10] could be of interest when using VCWC in environments that do not have HTCondor installed.

References

1. Asparagus Web-based Benchmarking Environment. http://asparagus.cs.uni-potsdam.de/
2. Barrett, C., Deters, M., Moura, L., Oliveras, A., Stump, A.: 6 years of smt-comp. Journal of Automated Reasoning 50(3), 243–277 (2013), http://dx.doi.org/10.1007/s10817-012-9246-5

3. Calimeri, F., Ianni, G., Krennwallner, T., Ricca, F.: The Answer Set Programming Competition. AI Magazine 33(4), 114–118 (December 2012), http://www.kr.tuwien.ac.at/staff/tkren/pub/2012/aimag2012-aspcomp.pdf
4. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. Theory and Practice of Logic Programming FirstView, 1–19 (2012)
5. Coles, A.J., Coles, A., Olaya, A.G., Jiménez, S., López, C.L., Sanner, S., Yoon, S.: A survey of the seventh international planning competition. AI Magazine 33(1) (2012)
6. Couvares, P., Kosar, T., Roy, A., Weber, J., Wenger, K.: Workflows for e-Science, chap. Workflow Management in Condor, pp. 357–375. Springer (2007)
7. Collaboratory on Experimental Evaluation of Software and Systems in Computer Science. http://evaluate.inf.usi.ch/ (2012)
8. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001), www.qbflib.org
9. GNU Make. https://www.gnu.org/software/make/
10. Hadoop. https://hadoop.apache.org/
11. Holger, H.H., Stützle, T.: SATLIB: An online resource for research on SAT. In: Proceedings of Theory and Applications of Satisfiability Testing, 4th International Conference (SAT 2000). pp. 283–292. IOS Press (2000)
12. The software of the seventh international planning competition (IPC). http://www.plg.inf.uc3m.es/ipc2011-deterministic/FrontPage/Software (2011)
13. Järvisalo, M., Berre, D.L., Roussel, O.: Rules of the 2011 sat competition. http://www.satcompetition.org/2011/rules.pdf (2011)
14. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine 33(1) (2012)
15. Klebanov, V., Beckert, B., Biere, A., Sutcliffe, G. (eds.): Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012, CEUR Workshop Proceedings, vol. 873. CEUR-WS.org (2012)
16. Oracle grid engine. http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html
17. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
18. Peschiera, C., Pulina, L., Tacchella, A.: Designing a solver competition: the QBFEVAL'10 case study. In: Stump, A., Sutcliffe, G., Tinelli, C. (eds.) Workshop on Evaluation Methods for Solvers, and Quality Metrics for Solutions (EMSQMS) 2010. EasyChair Proceedings in Computing, vol. 6, pp. 19–32. EasyChair (2012)
19. Roussel, O.: Controlling a solver execution with the runsolver tool system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 139–144 (2011)
20. Runlim. http://fmv.jku.at/runlim/
21. SMT Exec. http://www.smtexec.org/
22. Stump, A., Sutcliffe, G., Tinelli, C.: Introducing StarExec: a cross-community infrastructure for logic solving. In: Klebanov et al. [15], p. 2
23. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
24. Sutcliffe, G.: The CADE-23 automated theorem proving system competition–CASC-23. AI Communications 25(1), 49–63 (2012)
25. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency Computat. Pract. Exper. 17(2-4), 323–356 (2005)