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, Vienna University of Technology, 1040 Vienna, Austria
[2] Dipartimento di Matematica e Informatica, Università della Calabria, 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. Recent initiatives such as [6] aim at establishing best practices in computer science evaluations, especially identifying measures to be taken for ensuring repeatability, excluding common pitfalls, and introducing appropriate tools. For instance, Asparagus [1] focusses on maintaining benchmarks and instances thereof. Other known tools such as Runlim (http://fmv.jku.at/runlim/) and Runsolver [11] 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 [12], 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 **(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 (e.g., when the server room air condition fails).

Moreover, a competition platform must be flexible enough to allow for "late" or updated benchmark and solver submissions. It is not uncommon that anomalies arise during the execution, and changing the course of an evaluation after the platform has started is cumbersome and requires manual effort for the competition maintainers.

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 these 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

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

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 (i) GNU Make and GNU M4 for building the track execution workflow, (ii) the HTCondor [14] high throughput computing platform, which provides flexible means to support the requirements of running a competition, like automated job scheduling on a collection of benchmark servers, and (iii) the Directed Acyclic Graph Manager (DAGMan) [5], 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 open source and implemented using standard UNIX tools, thus it runs on every UNIX-like system that has support for those utilities. VCWC is maintained at https://github.com/tkren/vcwc, and an extended version of this paper is available at http://www.kr.tuwien.ac.at/staff/tkren/pub/2013/lpnmr2013-vcwc.pdf.

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 [9].

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. 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 evaluated $k > 1$ times to eliminate outliers and to provide well-founded statistical results.

For example, 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 participant may choose the 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.

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 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*

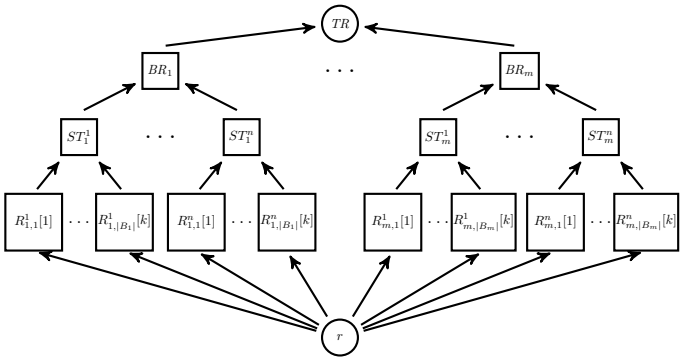

Fig. 1. Competition workflow for a track with m benchmarks and n solvers

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 an acyclic dependency graph constitutes a track execution workflow whose tasks can be possibly executed in parallel using proper job scheduling software.

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 $R_{v,w}^u[i]$ 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_w^u represent the solver summary statistics task of solver S_u in benchmark B_w, i.e., ST_w^u 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_w^u for $1 \leq u \leq n$. The topmost node TR is the

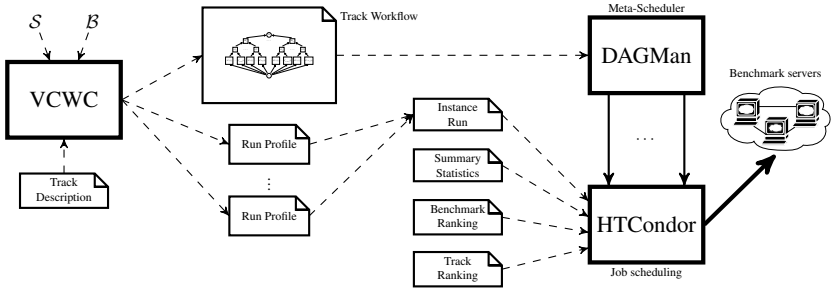

Fig. 2. VCWC System Architecture (dashed lines: data flow, solid lines: call flow)

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 or benchmark verification scripts after the competition start, we can introduce a workflow versioning mechanism for incrementally changing the competition execution workflow. Without details, one can add fresh participants, further benchmarks (or instances), or more runs. Additions and removals do not have impact on previously stored executions of the workflow, and statistics will be updated accordingly.

3 Implementation of the VCWC System

The system architecture of VCWC is shown in Fig. 2. The main components are (i) the VCWC compiler, which generates a competition workflow description and profiles for instance parameters; (ii) DAGMan (Directed Acyclic Graph Manager), a meta-scheduler for managing dependencies between jobs built on top of (iii) HTCondor, a job scheduler for building high-throughput computing environments.

VCWC expects two directories as input: a benchmarks directory with all possible benchmarks $\mathcal{B}$ assigned to track T as subdirectories, and a dedicated `participants` directory containing subdirectories for each possible benchmark of a track; participating solvers $\mathcal{S}$ can then choose which benchmark they want to solve. VCWC further takes a track description file as input that records various parameters of a track.

In practice, the VCWC tool consists of a wrapper shell script that invokes GNU Make 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 generates as output

```
# 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: 001
- timestamp: 2013-04-26 14:34:15+02:00
compiling 90 runs for S/t03/b01/s40/001
[...]
compiling 6 participants for B/t03/b01/001
[...]
linking 26 benchmarks for T/t03/001
```

This will generate a DAG workflow file 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.

Generated workflows can be processed by DAGMan, which submits jobs to HTCondor for execution in the network of benchmark servers. HTCondor is a high-troughput computing framework for distributed computation of computationally intensive tasks. Each task (job) that needs to be executed is first enqueued, 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. The HTCondor job queue is persistent and make administrator intervention unnecessary in case of a reboot or system crash, as interrupted jobs are automatically rescheduled. The correct topological order of job execution is ensured by DAGMan, which—based on the generated workflow—dispatches, monitors, and keeps track of exit codes of jobs. DAGMan requires human intervention only when no further job can be submitted according to the current topological order, because of a previously failed dependency.

4 Discussion and Conclusions

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 in Section 1. 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 23 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 runs. 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.

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) 3rd ASP Competition evaluation platform [4]. An attempt at providing a general purpose platform, serving multiple communities and generalizing specific needs is the StarExec platform [12]. Similar efforts in the neighbor communities are the IPC platform [7], the SMT-Exec platform [2] and the TPTP library and associated infrastructure [13]; the QBF-LIB library and evaluation platform [10], and last but not least the SAT Competitions infrastructure [8]. Future versions of VCWC will provide support for more fine-grained instance runs that allow to parametrize solver heuristics, advanced early diagnostics, and database storage facilities.

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. Couvares, P., Kosar, T., Roy, A., Weber, J., Wenger, K.: Workflows for e-Science, chap. Workflow Management in Condor, pp. 357–375. Springer (2007)
6. Collaboratory on Experimental Evaluation of Software and Systems in Computer Science. http://evaluate.inf.usi.ch/ (2012)
7. The software of the seventh international planning competition (IPC). http://www.plg.inf.uc3m.es/ipc2011-deterministic/FrontPage/Software (2011)
8. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine 33(1) (2012)
9. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
10. 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)
11. Roussel, O.: Controlling a solver execution with the runsolver tool system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 139–144 (2011)
12. Stump, A., Sutcliffe, G., Tinelli, C.: Introducing StarExec: a cross-community infrastructure for logic solving. In: Klebanov, V., Beckert, B., Biere, A., Sutcliffe, G. (eds.) COMPARE. CEUR Workshop Proceedings, vol. 873, p. 2. CEUR-WS.org (2012)
13. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)

14. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency Computat. Pract. Exper. 17(2-4), 323–356 (2005)