

Declarative Specification of Benchmark Sessions via ASP

Mario Alviano, Bernardo Cuteri, and Francesco Ricca

University of Calabria, Italy
`lastname@mat.unical.it`

Abstract. As a matter of fact, benchmark sessions are easy to describe in a declarative fashion, but are usually painful to implement in practice. In this paper we describe DecBench, a suite of tools conceived for easing the task of configuring and running structured software benchmark analyses of command-line executables. Benchmark sessions are specified in DecBench by means of Answer Set Programming (ASP), a declarative knowledge representation language having its roots in nonmonotonic reasoning and logic programming. The provided declarative specification is fed to an ASP solver, and the output is used to produce a Makefile that controls the execution of the specified benchmark analysis. The paper reports also a complete usecase conceived to validate the presented tool.

1 Introduction

Software benchmarking is an indispensable mean for analyzing the behavior of algorithms, heuristics and optimization techniques, as well as for obtaining empirical data on software efficiency. As a matter of fact, even when the benchmark execution plan is well-defined and clear, researchers end up in spending a lot of time for configuring the execution environment, creating scripts for running executables, extracting and analyzing the results. This is because benchmark sessions are easy to describe in a declarative fashion, but are usually painful to implement in practice. Starting from this observation, we decided to explore the possibility to apply a logic-based declarative paradigm for modeling the entities of a typical benchmark session, with the goal of easing the task of configuring and running structured software benchmark analyses on command-line executables. The logic adopted is Answer Set Programming (ASP) [4], a powerful paradigm for knowledge representation and declarative problem-solving, for which robust implementations are available [2].

In this paper we report on DecBench (<https://bitbucket.org/bernardo-cuteri/decbench-software>), a promising suite of tools for declarative specification sessions. In DecBench, benchmark instances and related data are represented by ASP facts, so that benchmark sessions are then specified by means of ASP rules

Proceedings of the 21st RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2014).
Vienna, Austria, July 17–18, 2014.

defining the execution plan. In a nutshell, the specification is evaluated by an ASP solver, whose output is analyzed in order to generate a Makefile that controls the execution of tested executables as well as the generation of benchmark results. In particular, the results of the execution are stored in XML files, thus allowing further manipulation such as the transformation into a spreadsheet, an HTML page, a PDF document, or other form of reports.

2 Declarative Specification of Benchmarks

This section briefly reports the main entities involved in DecBench, including some examples to further clarify how the specification works. Entities are actually represented by ASP atoms, that is, constructs of the following form: $predicate(arg_1, \dots, arg_n)$, for some $n \geq 0$. A complete description of ASP is out of the scope of this paper, and we refer the reader to [1] for a textbook.

The following atoms constitute the core of DecBench, i.e., they are the minimal set of atoms that are required to specify a benchmark session:

$$run(solverId, domainId) \tag{1}$$

$$solver(solverId, "solver\ command") \tag{2}$$

$$domain(domainId, "domain\ command") \tag{3}$$

$$testcase(testcaseId, domainId, "testcase\ command") \tag{4}$$

Roughly, an atom of the form (1) specifies that a solver must be run on the testcases of a domain. A solver is declared by means of an atom of the form (2), where the second argument is the command-line used for executing the solver. A domain instead is specified by means of an atom of the form (3), where the second argument is a suffix to be appended to the command-line of each solver run on this benchmark. Finally, testcases in a domain are specified by atoms of the form (4), where the third argument is an additional command-line suffix.

Example 1. The following is a specification for a benchmark session involving two ASP solvers and two testcases grouped in a domain named *minimalPath*:

```
run(S, minimalPath) :- solver(S, _).
solver(dlv, "dlv -n=1"). solver(clasp, "clasp.sh").
domain(minimalPath, "minimalPath/enc.asp").
testcase(1, minimalPath, "minimalPath/01.asp").
testcase(2, minimalPath, "minimalPath/02.asp").
```

Note that the declaration of DLV includes a path to the executable and command-line options, while for clasp a bash script has been specified.

In a benchmark session it is often the case that some resources are limited. Within DecBench, such limitations can be specified as follows:

$$resourceLimit(domainId, solverId, resource, value) \tag{5}$$

where *resource* can be *wall* (i.e., real time), *cpu* (i.e., system+user time) or *virtualMemory* (i.e., used memory, including swap).

Example 2. Continuing with Example 1, the following rules:

```
resourceLimit(minimalPath, S, cpu, 600) :- run(S, minimalPath).
resourceLimit(minimalPath, dl原因, virtualMemory, 4096).
```

can be used to specify that every solver run on *minimalPath* instances is allowed to use at most 600 seconds of cpu time. Moreover, the solver DLV is further limited to use at most 4096 MB (i.e., 4 GB) of virtual memory.

Additional information can be specified for each testcase as follows:

```
testcaseData(testcaseId, domainId, attribute, value) (6)
```

```
testcaseExecutionData(solverId, testcaseId, domId, "usage", "value") (7)
```

Intuitively, each atom of the form (6) adds a pair *attribute = value* to a single testcase. These attributes can then be used either in ASP rules to produce other atom instances, or in the postprocessing analysis. Each atom of the form (7), instead, specifies a parameter to be used in the execution of a single testcase.

Example 3. Continuing the previous examples, the following rules:

```
testcaseData(1, minimalPath, numberOfNodes, 20).
testcaseData(2, minimalPath, numberOfNodes, 30).
testcaseExecutionData(dlv, T, minimalPath, "-N=", Nodes) :-
    testcaseData(T, minimalPath, numberOfNodes, Nodes).
```

add information concerning the number of nodes of each testcase in the domain *minimalPath*. This information is then passed to the solver DLV as a command-line option of the form *-N=20* for the first instance, and *-N=30* for the second. To sum up, DLV is run on the two testcases as follows:

```
$ dlv -n=1 -N=20 minimalPath/enc.asp minimalPath/01.asp
$ dlv -n=1 -N=30 minimalPath/enc.asp minimalPath/02.asp
```

The last atom that can be used in DecBench is the following:

```
orderingRule(solverId, domainId, orderingParam, criterion, strength)(8)
```

Intuitively, this atom forces an order in the execution of the instances of a certain domain by a solver. Each testcase in the domain must be associated with an attribute *orderingParam* specified by an atom of the form (6). Testcases are thus ordered according to the values of this attribute, following the direction specified by the argument *criterion*, which can be either *ascending* or *descending*. Finally, the argument *strength* is used to distinguish between *strong* and *weak* ordering dependencies. Roughly, a strong dependency implies that a testcase is run only if all preceding testcases completed their execution within the allotted resources. Weak dependency instead only forces orders in the execution of testcases.

Example 4. In the previous examples, testcases in the domain *minimalPath* can be sorted by number of nodes by means of the following rule:

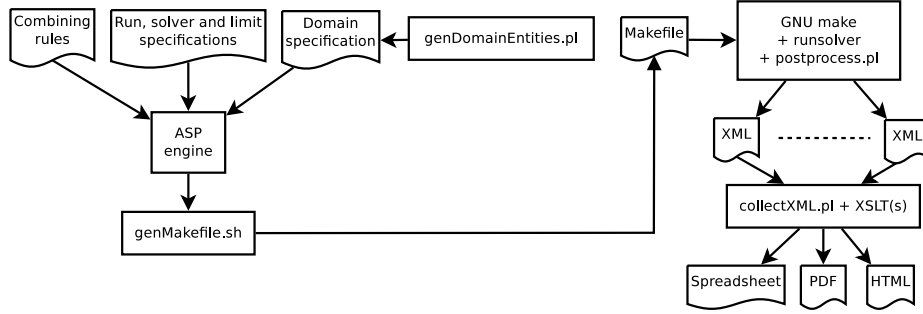


Fig. 1. System architecture

```

orderingRule(S, minimalPath, numberOfNodes, ascending, strong)
:- run(S, minimalPath).

```

Since the ordering dependency is strong, the second testcase is not run by a solver S if S does not terminate within the allotted resources on the first testcase.

3 Implementation

DecBench has been implemented with a particular focus on software modularity in order to ease maintainability and encourage software extensions. A sketch of the architecture of DecBench is reported in Figure 1. Roughly, the user provides entities to an ASP engine, which in our case is DLV. Domain entities can be produced almost automatically by the module *genDomainEntities.pl*, which essentially analyzes a directory for indexing testcases of a domain; for an example usage see the next section. Run, solver and limit entities are usually specified by the user without the help of a specific module, as the flexibility provided by ASP rules is already sufficient in this case; see Example 2. Implicit entities are instead obtained by a fixed set of combining rules. For example, the following rule defines an atom *testcaseCommand* that is used to group all command-line arguments required to run a solver on a testcase of a particular domain:

```

testcaseCommand(S,D,T,Sc,Dc,Tc) :- solver(S,Sc), domain(D,Dc),
run(D,S), testcase(T,D,Tc).

```

The following rules instead are used to materialize ordering dependencies:

```

requires(S,D,T,S,D,T2,St) :- orderRule(S,D,K,ascending,St),
testcaseData(T,D,K,V), testcaseData(T2,D,K,V2), V > V2.
requires(S,D,T,S,D,T2,St) :- orderRule(S,D,K,descending,St),
testcaseData(T,D,K,V), testcaseData(T2,D,K,V2), V < V2.

```

The output of DLV is then processed by the module *genMakefile.pl*, which essentially converts the benchmark specification into a runnable Makefile. In fact, dependencies and execution commands have natural counterparts in a Makefile.

The produced Makefile is processed by GNU make, which supports several interesting features. Among them are parallelization of the process and recover of partially processed targets. Resource usage is measured and limited by the runsolver tool [7], whose output is converted in XML to ease the subsequent analysis of the collected results. Actually, the module *postprocess.pl* also enriches the output of runsolver by appending information from input entities. Finally, the module *collectXML.pl* is combined with one or more XSLT to obtain different representations of the empirical data.

4 Validation

In order to assess both usability and effectiveness of DecBench we implemented a real use case. In particular, we implemented a benchmark that mimics a substantial portion of the system competition track of the 3rd Answer Set Programming Competition [2]. The usual benchmark executed in an ASP competition consists on running several competing solvers on the instances of different computational problems that are modeled in ASP. For each benchmark problem there is a fixed encoding, usually stored in a single file, to be combined with instance specifications, usually stored in separate files. The number of instances solved by participants within a fixed time and the execution times are then measured.

4.1 Benchmark configuration

In this section we report on the configuration of DecBench in order to produce a Makefile for benchmarking three solvers, namely DLV, ClaspD and ClaspD2. First of all, let us declare these solvers:

```
solver(dlv, "./aspcomp2011/executables/dlvRunScript.sh").
solver(claspd, "./aspcomp2011/executables/claspdRunScript.sh").
solver(claspd2, "./aspcomp2011/executables/cd2RunScript.sh").
```

Domain entities can be automatically obtained by running *genDomainEntities.pl* for each problem in the Competition. For example, domain entities for the problem *Labyrinth* are produced as follows:

```
$ genDomainEntities.pl -domain="labyrinth"
  -command="./17-Labyrinth/encoding.asp"
  -testcasesDir="./17-Labyrinth/selectedInstances"
  -regex="(\d+)-0.asp" -regexUsage="dlv:-N="
  -insideRegex="max_steps\\((\\d+)\\)"
  -insideAttribute="maxSteps"
  -insideOccurrences="field connect"
```

The first two arguments cause the production of the following rule:

```
domain(labyrinth, "./17-Labyrinth/encoding.asp").
```

The script then searches on directory `../17-Labyrinth/selectedInstances` for files matching the regular expression `(\d+)-0.asp`. A testcase atom is produced for each matched file. Moreover, the group `(\d+)` is used to create a `testcaseExecutionData` atom for each solver specified in the `-regexUsage` argument. Finally, `testcaseData` atoms are produced by processing the content of each matching file. In particular, the script applies the regular expression `max.steps\((\d+)\)` for obtaining the attribute `maxSteps`, while attributes `field` and `connect` are obtained by counting the occurrences of these predicates. The followings are the atoms produced for a single testcase:

```
testcase(1,labyrinth,
    "../17-Labyrinth/selectedInstances/003-labyrinth-11-0.asp").
testcaseExecutionData(dlv,1,labyrinth,"-N=",11).
testcaseData(1,labyrinth,maxSteps,10).
testcaseData(1,labyrinth,field,100).
testcaseData(1,labyrinth,connect,166).
```

In our setting, all solvers are run on all domains, time is limited to 600 seconds and central memory to 1 GB, as specified by the following rules:

```
run(D,S) :- solver(S,_), domain(D,_).
resourceLimit(D,S,cpu,600) :- run(D,S).
resourceLimit(D,S,virtualMemory,1024) :- run(D,S).
```

A Makefile is then produced by executing `genMakefile.sh` as follows:

```
$ genMakefile.sh specification.dl
```

where `specification.dl` contains all required rules. The script actually runs DLV to materialize implicit atoms, which are then used to produce a Makefile. The Makefile can thus be processed by GNU make, where we require the command-line option `-k` in order to not stop the execution of the benchmark session in case of unsatisfied ordering dependencies. (Unsatisfied strong dependencies are properly handled in the command sections of the Makefile.) Parallelization of the benchmark session can be obtained by means of the command-line option `-j` (for jobs). For example, the following command runs the session with two jobs:

```
$ make -k -j2
```

The benchmark session was configured in around 30 minutes, which is negligible if compared with the time required for handcrafting such a configuration.

4.2 Benchmark execution

The benchmark session was executed on an Intel Xeon X3430 machine 2.4 GHz with 4 cores and 4 GB of RAM. Every testcase was executed under the control of `runtool`, whose output is converted in XML. All XML documents produced during the execution of the benchmark session were grouped by `collectXML.pl` and elaborated using XSLT and back-end scripts. For example, the cactus plot in

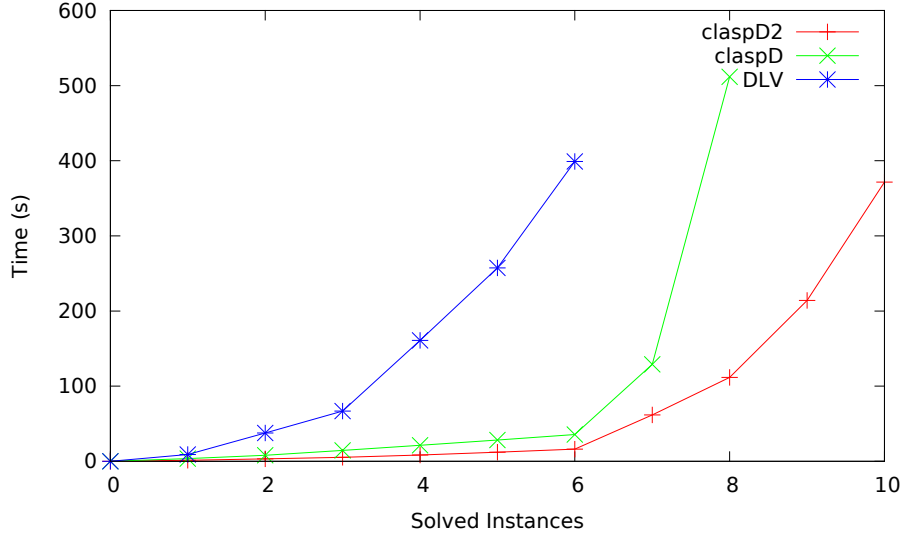


Fig. 2. Cactus plot of the performance of tested ASP systems in SokobanDecision

Figure 2 was obtained by a back-end for gnuplot. Table 1 instead was produced with the help of a spreadsheet obtained with XSLT. Even though an in-depth analysis of the experimental data is out of the scope of this paper, Table 1 gives us an insight on the potentiality of DecBench. In fact, we observe that the speedup of DecBench is almost linear, with minimal differences on the number of solved testcases for each solver. This may happen for particular testcases that are really close to resource limits breaking, thus sensitive to any context change such as jobs parallelization. Clearly, such differences are relevant for an official event, but can be reasonably ignored for daily planned benchmarks.

5 Conclusion

The assessment of the performance of computer programs, algorithms, heuristics and optimization techniques is an important research activity, which is often carried out by resorting to ad-hoc implementations and custom platforms. The systematic analysis of the requirements of an ideal benchmarking platform is

Table 1. Execution times and number of solved testcases

# jobs	Time	Total	DLV	ClaspD	ClaspD2
1	35h 43m 36s	331	102	100	129
2	18h 2m 34s	330	102	99	129
3	12h 11m 20s	330	102	99	129
4	9h 28m 41s	328	101	98	129

subject of study of international organizations and committees [6, 8]. Some execution platforms were already proposed, such as SDEval [5] and VCWC [3], and tools solving specific aspects such as resource consumption measurements are publicly available [7]. However, to the best of our knowledge, no solution for software benchmarking is available that takes advantages of a logic-based declarative specification, and in particular of ASP. DecBench is a proof-of-concept suite of tools fulfilling this lack. Starting from a declarative specification of a benchmark session, DecBench generates a Makefile that controls a possibly parallel execution of the benchmark session. The output of such an execution is an XML document that can be transformed into several media. Another strength of DecBench is its extensibility, as new predicates can be easily added in the specification of a benchmark session. Moreover, the output of the ASP solver could be processed by an alternative script for producing schedules that are compliant with an execution handler different than GNU make.

Concerning future work, we plan to extend the ASP model for the specification of benchmark sessions in order to obtain a more compact representation of experiments. Moreover, we aim at developing a graphical user interface that can further help the user during the benchmark configuration steps.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In *Proc. of LPNMR11.*, pages 388–403. LNCS Springer, 2011.
3. G. Charwat, G. Ianni, T. Krennwallner, M. Kronegger, A. Pfandler, C. Redl, M. Schwengerer, L. K. Spendier, J. P. Wallner, and G. Xiao. Vcwc: A versioning competition workflow compiler. In *LPNMR*, volume 8148 of *Lecture Notes in Computer Science*, pages 233–238. Springer, 2013.
4. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
5. A. Heinle, V. Levandovskyy, and A. Nareike. Symbolicdata: Sdeval - benchmarking for everyone. *CoRR*, abs/1310.5551, 2013.
6. M. Huisman, V. Klebanov, and R. Monahan. On the organisation of program verification competitions. In *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems*, volume 873, pages 50–59. CEUR, 2012.
7. O. Roussel. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation*, (7):139–144, 2011.
8. A. Stump, G. Sutcliffe, and C. Tinelli. Introducing starexec: a cross-community infrastructure for logic solving. In V. Klebanov, B. Beckert, A. Biere, and G. Sutcliffe, editors, *COMPARE*, volume 873 of *CEUR Workshop Proceedings*, page 2. CEUR-WS.org, 2012.