Testing Provers on a Grid

Framework Description

Carlos Areces[1], Daniel Gorín[2,4],
Alejandra Lorenzo[3], and Mariano Pérez Rodríguez[4]

[1] INRIA Nancy Grand Est, France
[2] Université Henri Poincaré, France
[3] Université Nancy 2, France
[4] Universidad de Buenos Aires, Argentina

Abstract. GridTest is a framework for testing automated theorem provers using randomly generated formulas. It can be used to run tests locally, in a single computer, or in a computer grid. It automatically generates a report as a PostScript file which, among others, includes graphs for time comparison.
We have found GridTest extremely useful for testing and comparing the performance of different automated theorem provers (for hybrid, modal, first order and description logics). We present GridTest in this framework description in the hope that it might be useful for the general community working in automated deduction.

1 Testing Automated Provers

Testing a system is an invaluable source of information about its behaviour. But testing can be difficult and time consuming. This is particularly the case for systems that have to deal with very diverse input (where carrying out exhaustive testing would be impossible), and which need complex computation (that is, systems which are trying to solve problems which are known to be computationally hard). Automated theorem provers are prime examples of systems with these characteristics.

The issue of how to perform suitable testing and comparison of automated theorem provers has been largely debated, and different proposals have been presented (see, e.g., [1–10]) for logics ranging from propositional to first-order. Which kind of testing is most adequate — i.e., whether the problems are randomly generated, hand tailored, or coming from real world applications; how is the potential input space covered; etc. — is difficult to evaluate. Probably, the only safe thing to say is that the more you test the best it is (and still, it might be that no testing is better than bad testing, depending on the claims that are put forward in the basis of the tests performed).

In this article we will not discuss the issue, and we will focus on one particular kind of testing which is well suited to be distributed in a computer grid. In particular, we will present GridTest, a framework for testing automated theorem provers using randomly generated formulas. GridTest has been developed in python [11], and can be used to run tests locally, in a single computer or in a computer grid. It automatically generates a report as a PostScript file (generated via LaTeX). It can compile statistics provided by the provers (e.g., running time, number of applications of a particular rule, open/closed branches, etc.) and produce graphs generated using GnuPlot. Even if the prover does not provide any kind of statistics, GridTest will use the `time` command (available in all POSIX-conformant operating systems) to obtain running times to plot in the final report.

The framework is released under the GNU General Public License and is available at http://glyc.dc.uba.ar/intohylo/gridtest.php.

GridTest has been originally designed for automatizing tests as those described in [8]. That is, we use a random generator of formulas in conjunctive normal form to obtain batches of formulas with an increasing number of conjunctions. In this way, we can explore the average behaviour of the provers in a spectrum of formulas that runs from mostly satisfiable to mostly unsatisfiable, aiming to hit the point of maximum uncertainty (i.e., where the chances of a randomly generated formula being satisfiable or unsatisfiable is balanced).

We have used GridTest mostly to test theorem provers for hybrid logics and hence the current framework uses hGen as its random formula generator [12]. hGen (also distributed under the GNU General Public License and available at http://glyc.dc.uba.ar/intohylo/hgen.php) is a generator of random formulas in the hybrid language $\mathcal{H}(@, \downarrow, \mathsf{A}, \mathsf{D}, \mathsf{P})$ (that is, the basic modal logic extended with nominals, the satisfiability operator @, the ↓ binder, the universal modality A, the difference modality D, and the past modality P, see [13]). But because we were interested in the comparison of provers for different logics (e.g., description and first order logics) the framework has been designed to properly handle translations between the output format of the random generator and the input format of the different provers. In particular, timing graphs will discriminate between the time used for the translation and the time used by the prover. A number of translations from the output format of hGen to the input format of different provers is provided with the source code (e.g., the `tptp` format for first-order provers, the standard input format for description logic provers, etc.), together with the drivers for different provers (e.g., E, SPASS, Bliksem, Vampire, Racer, FaCT++, HTab, HyLoRes, etc.).

In order to run on a single machine, GridTest requires a python interpreter and some typical POSIX tools (`bash`, `time`, `tar`, etc.). Its output is a collection of GnuPlot and LaTeX scripts that are automatically compiled into a PostScript file reporting the results. These requirements are fairly typical and are available on almost every platform.

Unfortunately, unlike the POSIX standard for operating systems, there is as yet no standard batch scheduling mechanism for computer clusters or grids. GridTest currently supports only one backend, namely the OAR [14] batch scheduler, to distribute the test on a computer cluster. We designed GridTest so as to use only very basic services that most batch schedulers should provide, but still, porting it to other systems could be the most difficult challenge when trying to use GridTest elsewhere.

We have found GridTest extremely useful for testing and comparing the performance of different automated theorem provers. Thus, we present this framework description, and we freely release the source code, in the hope that it might be useful for the general community working in automated deduction.

2 Testing on the Basis of Random Formulas

The testing methodology implemented in GridTest uses a customizable random formula generator. We currently use hGen [12], although it should be easy to add support for additional generators.

Similarly to other random formula generators (e.g., [1, 8]), hGen generates formulas in a conjunctive normal form: each formula is a conjunction of disjunctive clauses. Since each disjunctive clause can be seen as an additional constraint on satisfying models, random formulas with a small number of clauses tend to be satisfiable while a large enough number of random clauses will be unsatisfiable.

By generating formulas with an increasing number of disjunctive clauses, we can generate tests that start with formulas with a high chance of being satisfiable, and progresively obtain formulas with a high chance of unsatisfiability, going through the point where these probabilities are roughly the same. Formulas at this spot tend to be difficult for most provers, regardless of whether they are naturally biased towards satisfiable or unsatisfiable formulas. Of course, the precise number of clauses needed to reach this point varies depending on other parameters, such as the number of proposition symbols, the number and kind of modalities, the maximum modal depth, etc.

With a random formula generator like hGen, we can set up the following benchmark: i) generate random formulas $\varphi_1 \dots \varphi_n$ where φ_i has exactly i conjunctions (each conjunct being a disjunctive clause) and the rest of the parameters are fixed, ii) run provers $p_1 \dots p_k$ over each of the n random formulas, using a fixed time limit per formula, iii) collect data of interest about each run (execution time, answer, number of rules fired if available, etc.) and plot it for comparison. Of course, this experiment is not statistically relevant by itself (because the input formula used in each data point has been generated randomly). However, by repeating it a sufficiently large number of times (or, equivalently, using batches of formulas sufficiently large for each data point) and using a statistical estimator on the sampled data (e.g., average, median, etc.) statistically relevant results can be obtained.

One can use this testing methodology for different purposes. For example, by comparing the response of every prover on a particular formula, we have discovered inconsistencies which, in turn, allowed us to find and correct implementation errors in the provers we develop. We have also used this tests to assess the effectiveness of optimizations; this was done by comparing alternative versions of the same prover and looking at running times, number of rules fired, clauses generated, etc. Finally, one can also use this methodology to evaluate the relative performance of different provers. Of course, given the artificial nature of the formulas used in the test, one must be careful about the conclusions drawn.

This methodology is conceptually simple, but it presents a clear drawback: even for tests of a moderate size, if we generate non-trivial formulas and allow each prover to run for a reasonable amount of time, the total running time on a single computer can quickly become large. Tests with running times measured in days (or weeks) become common. This is especially true if some of the provers involved in the test tend to time out often. If we are interested in using this form of testing as part of the development process of a prover, rapid availability of the results is crucial.

Notice though, that because of the nature of the tests (and specially, if we are more interested in qualitative data, rather than in quantitative data), we are not obliged to run all the tests serially in the same computer. We can, as well, obtain statistical relevance by distributing the tests on a computer cluster: each machine runs the complete tests on batches of smaller size and the data is pulled together for statistical analysis when all the runs are completed.

Concretely, instead of running a test with batches of size b on a single computer, we could alternatively run n tests on n different computers, each having to process a batch of size b/n, obtaining a linear reduction on the time required.

Although large computer clusters are not ubiquitous, the recent emergence of *grid computing* technologies is giving researchers access to a very large number of computing resources for bounded periods of time. In this scenario, it is not unreasonable to assume the simultaneous availability of such a number of computers.

As we mentioned, even if the grid is composed of heterogeneous machines (different processors, clock speeds, cache memory sizes, etc.), qualitative result (i.e., the relative performance of the provers under evaluation) would not be affected. A computer cluster seems perfectly suited to run this kind of tests.

3 Installing and Using GridTest

The directory structure of a GridTest installation looks like this:

```
gridtest
 ├─ bin ........................... binaries (provers, etc.) to be run locally
 ├─ sbin ...................... binaries (provers, etc.) to be run on the grid
 ├─ configuration ................................... test configurations
 └─ drivers .............................. drivers for the different provers
```

A more detailed description of what each directory is supposed to contain is as follows:

bin. Binaries for all the provers, translators and for the formula generator used in a test to be run on the local machine.

sbin. Binaries involved in a test on a grid. The architecture and/or operating system on the grid may differ from that on the local machine and therefore it is better to keep those binaries in a separate directory. Observe that even if the architecture and operating system matches, there still can be differences in the particular versions of the libraries installed; therefore it is convenient to use statically-linked binaries for testing on the grid. The `makeEnv.sh` script (see below) takes the binaries from this directory.

configurations. The configuration of the test to run typically resides here. The installation comes with several examples that can be taken as basis for new tests.

drivers. This directory contains python driver files for the different provers. The installation comes with drivers for various theorem provers (e.g., Vampire, SPASS, E, Bliksem, Racer, FaCT++, HTab, HyLoRes, etc.). A new driver should be written for each additional prover to be used with the testing framework.
A driver specifies how the prover should be ran on a particular input file (for example, which command line parameters the prover takes) and how the answers from the prover should be understood. Typically the latter involves parsing the provers output (saved to a file) and collecting relevant data.

Suppose that we have the executables for the provers for hybrid logics HyLoRes [15, 16] and HTab [17] in the directory `gridtest/bin`. Suitable drivers for these two provers are already provided in `gridtest/drivers` as the files `htab.py` and `hyloresv2_5.py`. An example of a possible test configuration using these two provers is provided in `gridtest/configurations` as the file `template.py` (shown in Figure 1). With all this in place, we can launch a local test simply with the command:

```
./testRunner.py configurations/template.py
```

The result will be the creation of a directory called `scratch-`*`BENCHMARK_NAME.RANDOM_SUFFIX`*. The *`RANDOM_SUFFIX`* is added to avoid file name clashes when running the same configuration more than once. Before explaining the structure of this directory, let's have a look at the `template.py` file.

The first line in Figure 1 imports the class `RandomHCNFTest`. A configuration file is expected to define a function `configureTest()` that returns a properly configured instance of this class. Most of the parameters of the test can be set using the constructor. Many of them will be used by hGen to generate formulas (e.g., the number and type of atoms that can appear, and their relative frequency; the kind of modalities that can appear, their maximum nesting and their relative frequency). The parameters under the "Test structure" label define the general configuration of the test: the number of formulas per batch, the range of clauses to generate (together with the step used to increase the number of clauses in each batch) and the time limit used for each run of a prover on one formula in the benchmark.

The final lines in the `configureTest()` function respectively define the output directory name and set the provers to be run. The latter is done by passing a list of drivers to the `setProvers()` method. In this example, the list of drivers is built in the `proverConfiguration()` function: `HTab` and `HyLoResV2_5` are the classes of the drivers for each prover and their constructors expect an id for the prover, the directory where the binary is located (`dirStructure.binDir` defaults to the `bin` directory) and the name of the binary.

As we mentioned, the benchmark will generate a directory called `scratch-Template.`*`RANDOM_SUFFIX`*. The general structure of the resulting directory is as follows:

```
scratch-BENCHMARK_NAME.RANDOM_SUFFIX
|__tests.................... the batches of formulas generated for the test
|  |__batch-1
|  |__...
|  |__batch-n
|__reposonses...................... the responses of the different provers
|  |__prover_1
|  |  |__batch-1
|  |  |__...
|  |  |__batch-n
|  |__...
|  |__prover_k
|     |__batch-1
|     |__...
|     |__batch-n
|__results........................... the report and its intermediate files
```

In the `tests` directory we find all the input formulas generated for the test. These formulas are stored so that the test can be reproduced in the future, by running the prover on the exact same benchmark. In the `responses` directory we find the responses of the different provers (captured by redirecting `stdout` to a file) when ran on each input formula. In addition, for each run, we will find there also the running time obtained using the `time` command (both for the prover and for the translation, if a translation is needed). The information in the `responses` directory is used to create the report found in the `results` directory.

As discussed in Section 2, the report will contain at least a graph corresponding to the satisfiable/unsatisfiable/timeout functions for each of the provers tested, together with graphs corresponding to real and user median time (showing deviation) for all the provers (an example of these graphs are shown in Figures 3 and 4). In addition, the report will also show graphs corresponding to cases of *unique responses* (i.e., formulas that where solved by only one of the provers), and it will explicitly list formulas where provers have provided contradictory responses (disregarding timeouts). These formulas are particularly interesting since they typically witness to implementation errors on some of the provers. Other graphs can easily be generated (average number of clauses generated by batch, average number of open/closed branches, etc.), depending on the statistics collected by each prover driver.

```
from randomHCNFTest import RandomHCNFTest

from htab        import HTab
from hyloresv2_5 import HyLoResV2_5

def configureTest():
    test  = RandomHCNFTest(testId         = 'Template',
                           # Number of atoms
                           numOfProps     = 2, freqOfProps = 2,
                           numOfNoms      = 2, freqOfNoms  = 2,
                           numOfSVars     = 0, freqOfSVars = 0,
                           # Number and depth of modalities
                           numOfRels      = 2, maxDepth    = 2,
                           diamDepth      = 1, freqOfDiam  = 1,
                           atDepth        = 1, freqOfAt    = 1,
                           downDepth      = 0, freqOfDown  = 0,
                           invDepth       = 0, freqOfInv   = 0,
                           univDepth      = 1, freqOfUniv  = 1,
                           diffDepth      = 0, freqOfDiff  = 0,
                           # Test structure
                           batchSize      = 5,
                           fromNumClauses = 10,
                           toNumClauses   = 20,
                           step           = 2,
                           timeout        = 10
                          )

    test.dirStructure.setNewScratchDirFor(test.testId)
    test.setProvers(proverConfiguration(test.dirStructure))
    return test

def proverConfiguration(dirStructure):
    binpath = dirStructure.binDir
    htab    = HTab       ('htab',        binpath, 'htab')
    hylores = HyLoResV2_5('hylores-2.5', binpath, 'hylores')

    return [htab, hylores]
```

Fig. 1. Test configuration `template.py`

It is also possible to take the resulting directory of a test, parse again the responses of the provers and regenerate the report. This is useful when updating the driver of a prover included in the test, or when adding a new kind of plot. The command is simply:

```
./testRunner.py scratch-BENCHMARK_NAME.RANDOM_SUFFIX
```

One can take the exact same test configuration `template.py` we discussed above and run it in a computer cluster based on the batch scheduler OAR. Two provisions must be taken: i) suitable binaries must be put in the `sbin` directory, and ii) the `makeEnv.sh` script must be ran to create the `environment.tar.gz` file which contains the binaries, plus all the required python scripts The actual command to run the test on a grid is then:

```
./clusterTestRunner.py configurations/template.py user@host #_of_nodes
```

Here, `user@host` identifies the login account on the cluster's access node and `#_of_nodes` determines the number of computers requested. It is also possible to indicate in the command line additional requirements, such as the minimum number of cores on each node, etc. The first thing the `clusterTestRunner` program does is to upload the file `environment.tar.gz` to the grid. Therefore, if the same test will be performed twice, it is not neccesary to execute the `makeEnv.sh` script again. After uploading `environment.tar.gz` the script will automatically reserve the required number of nodes in the cluster (with the specified characteristics) using OAR, and then distribute the tests among them. Once all the distributed tests are finished, it pulls together the responses and compiles the report. The result of the test is again returned as a directory:

```
scratch-BENCHMARK_NAME.RANDOM_SUFFIX
└─ tests....................the batches of formulas generated for the test
   ├─ batch-1
   ├─ ...
   └─ batch-n
```

The current version of `clusterTestRunner.py` does not return the files obtained by redirecting the `stdout` of the execution of each prover over each test, to minimize file transfer from the cluster (that is, the `responses` directory is not present inside the `scratch` directory). Only the already compiled statistics are returned. The report is generated in the `scratch` directory itself.

As we explained above, when a test is ran in the cluster, the formulas in each batch are evenly distributed among the requested nodes. Because each run of a prover on an input formula is independent of the others we can safely distribute the test on all the requested nodes in the computer cluster, obtaining an effective linear speed up. That is, a test that takes time t when run locally, will roughly run in time t/n, for n the number of nodes reserved in the cluster.

4 A Complete Example

To finish this framework description, we present a concrete test, both with a local run and a run on the grid. The test compares three provers: FaCT++, HTab and HyLoRes. The test configuration file is shown in Figure 2. Notice that in order to run FaCT++ on the input provided by hGen we had to implement a suitable translation (which we called `sthl-fact`: standard translation from hybrid logics formulas to FaCT++ input). This translation is not optimized and, in particular, naively translates the @ operator using a master modality [18]. Hence the concrete times obtained from the test should not be considered a fair evaluation of the performance of FaCT++.

The graphs in Figure 3 were taken from the report for a run of the test described above that was performed on a 2.4 GHz Pentium IV computer with 1 Gb of memory running GNU/Linux (kernel 2.6.27). The total running time was of approximately 22 hours.

The graphs in Figure 4 come from the report for the same test ran on 120 nodes of the *Grid'5000* [19]. The nodes were all on the *Bordeaux* site of the Grid; this site contained a mix of Opteron 2218, Opteron 248 and Xeon EMT64T based computers running GNU/Linux. The total running time was in this case of approximately 22 minutes (including file transfers over the network and job queueing by the batch manager).

If we compare both figures, we will see small variations on the curves. This is reasonable since the formulas of each test were not the same (recall that they were randomly generated from the same parameters) and the provers were run on different processors. However it is clear that they are qualitatively equivalent.

5 Conclusions

We have presented a framework for testing automated theorem provers. The methodology is based on random formula generation. An unusual feature of our framework is that it can be run on large computer clusters, uniformly distributing the test load among nodes. This way, we achieve a massive decrease in the total execution time of the tests. However, when testing on heterogeneous clusters the quantitative results may differ from those obtained on a single machine.

Random formula based testing has proven to be very useful in exposing subtle bugs and corner cases that would otherwise have gone unnoticed. It is also a rather inexpensive way of testing new language features for which there are no real-world examples. These kind of examples, on the other hand, are invaluable when tuning and benchmarking provers. For example, the system described in [9, 10] can be used to collect real-life ontologies and test DL provers on them. We believe these kind of systems can also benefit from exploiting Grid computing: by assigning each example to a unique node the same linear speed-up we obtained should be expected.

```
from randomHCNFTest import RandomHCNFTest

from fact        import FACT
from translator  import Translator
from htab        import HTab
from hyloresv2_5 import HyLoResV2_5

def configureTest():
    test  = RandomHCNFTest(testId         = 'dltest-120',
                           # Number of atoms
                           numOfProps     = 6, freqOfProps = 2,
                           numOfNoms      = 4, freqOfNoms  = 2,
                           numOfSVars     = 0, freqOfSVars = 0,
                           # Number and depth of modalities
                           numOfRels      = 3, maxDepth    = 8,
                           diamDepth      = 4, freqOfDiam  = 3,
                           atDepth        = 4, freqOfAt    = 2,
                           downDepth      = 0, freqOfDown  = 0,
                           invDepth       = 0, freqOfInv   = 0,
                           univDepth      = 0, freqOfUniv  = 0,
                           diffDepth      = 0, freqOfDiff  = 0,
                           # Test structure
                           batchSize      = 120,
                           fromNumClauses = 10,
                           toNumClauses   = 111,
                           step           = 5,
                           timeout        = 50,
                          )

    test.dirStructure.setNewScratchDirFor(test.testId)
    test.setProvers(proverConfiguration(test.dirStructure))
    return test

def proverConfiguration(dirStructure):
    binpath = dirStructure.binDir
    htab    = HTab       ('htab',        binpath, 'htab')
    fact    = Translator ('sthl-fact',   binpath, 'sthl-fact',
                     FACT('FaCT++',      binpath))
    hylores = HyLoResV2_5('hylores_2.5', binpath, 'hylores')

    return [htab,fact,hylores]
```

Fig. 2. Example test configuration

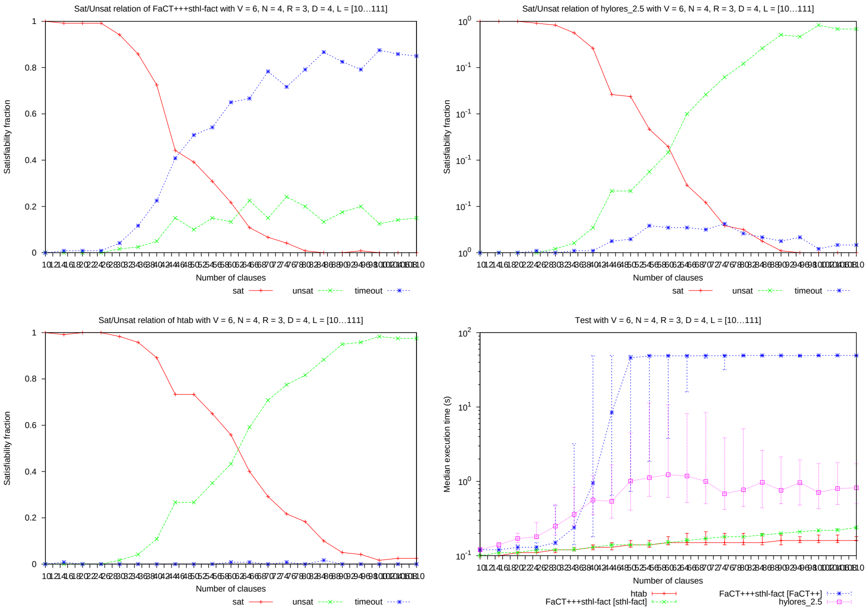

Fig. 3. Test ran on a local machine (test running time: ∼22 hours)

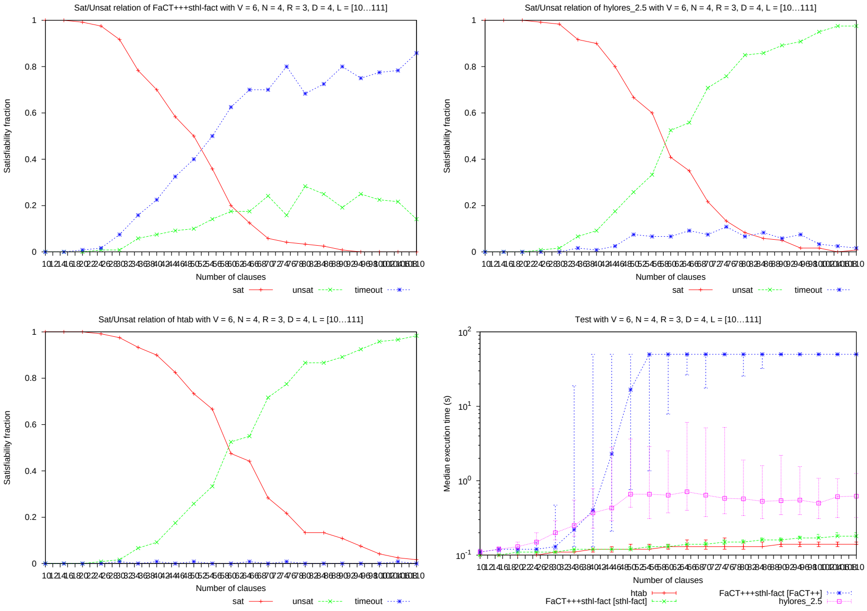

Fig. 4. Test ran on 120 nodes in a computer cluster (test running time: ∼22 minutes)

References

1. van Gelder, A.: Problem generator `mkcnf` (1993) Contributed to the DIMACS 1993 Challenge archive.
2. Sutcliffe, G., Suttner, C., Yemenis, T.: The TPTP problem library. In Bundy, A., ed.: Proc. of CADE-12. Number 814 in LNAI, Nancy, France (1994) 252–266
3. Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, and S4. Technical report IAM-96-015, University of Bern, Switzerland (1996)
4. Hustadt, U., Schmidt, R.: On evaluating decision procedures for modal logic. In Pollack, M., ed.: Proc. of the 15th IJCAI. (1997) 202–207
5. Giunchiglia, E., Giunchiglia, F., Sebastiani, R., Tacchella, A.: More evaluation of decision procedures for modal logics. In: KR'98: Principles of Knowledge Representation and Reasoning. Morgan Kaufmann (1998) 626–635
6. Massacci, F.: Design and results of the Tableaux-99 non-classical (modal) system competition. In: Proc. Tableaux'99. (1999)
7. Horrocks, I., Patel-Schneider, P., Sebastiani, R.: An analysis of empirical testing for modal decision procedures. Logic Journal of the IGPL **8** (2000) 293–323
8. Patel-Schneider, P., Sebastiani, R.: A new general method to generate random modal formulae for testing decision procedures. Journal of Artificial Intelligence Research **18** (May 2003) 351–389
9. Gardiner, T., Horrocks, I., Tsarkov, D.: Automated benchmarking of description logic reasoners. In: Proc. of the 2006 Description Logic Workshop (DL 2006). Volume 189 of CEUR (http://ceur-ws.org/). (2006)
10. Gardiner, T., Tsarkov, D., Horrocks, I.: Framework for an automated comparison of description logic reasoners. In: Proc. of the 5th International Semantic Web Conference (ISWC 2006). Volume 4273 of Lecture Notes in Computer Science., Springer (2006) 654–667
11. van Rossum, G.: Python reference manual. Technical Report CS-R9525, Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands (May 1995)
12. Areces, C., Heguiabehere, J.: hgen: A random cnf formula generator for hybrid languages. In: Methods for Modalities 3 - M4M-3, Nancy, France, Nancy, France (September 2003)
13. Areces, C., ten Cate, B.: Hybrid logics. In Blackburn, P., Wolter, F., van Benthem, J., eds.: Handbook of Modal Logics. Elsevier (2006) 821–868
14. The OAR team: The OAR resource manager. http://oar.imag.fr
15. Areces, C., Heguiabehere, J.: Hylores 1.0: Direct resolution for hybrid logics. In Voronkov, A., ed.: Proceedings of CADE-18, Copenhagen, Denmark (July 2002) 156–160
16. Areces, C., Gorín, D.: Ordered resolution with selection for h(@). In Baader, F., Voronkov, A., eds.: Proceedings of LPAR 2004. Volume 3452 of LNCS., Montevideo, Uruguay, Springer (2005) 125–141
17. Hoffmann, G., Areces, C.: Htab: A terminating tableaux system for hybrid logic. In: Proceedings of Methods for Modalities 5. (November 2007)
18. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)
19. Bolze, R., Cappello, F., Caron, E., Daydé , M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Irena, T.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications **20**(4) (November 2006) 481–494