Towards an Automated Test Bench Environment for Prolog Systems*

Ricardo Gonçalves[1], Miguel Areias[2], and Ricardo Rocha[3]

1 **CRACS & INESC TEC and Faculty of Sciences, University of Porto, Porto, Portugal**
rgoncalves@dcc.fc.up.pt
2 **CRACS & INESC TEC and Faculty of Sciences, University of Porto, Porto, Portugal**
miguel-areias@dcc.fc.up.pt
3 **CRACS & INESC TEC and Faculty of Sciences, University of Porto, Porto, Portugal**
ricroc@dcc.fc.up.pt

Abstract

Software testing and benchmarking is a key component of the software development process. Nowadays, a good practice in big software projects is the *Continuous Integration (CI)* software development technique. The key idea of CI is to let developers integrate their work as they produce it, instead of doing the integration at the end of each software module. In this paper, we extend a previous work on a benchmark suite for the Yap Prolog system and we propose a fully automated test bench environment for Prolog systems, named *Yet Another Prolog Test Bench Environment (YAPTBE)*, aimed to assist developers in the development and CI of Prolog systems. YAPTBE is based on a cloud computing architecture and relies on the Jenkins framework and in a set of new Jenkins plugins to manage the underneath infrastructure. We present the key design and implementation aspects of YAPTBE and show its most important features, such as its graphical user interface and the automated process that builds and runs Prolog systems and benchmarks.

1998 ACM Subject Classification D.2.5 Testing and Debugging, D.1.6 Logic Programming

Keywords and phrases Software Engineering, Program Correctness, Benchmarking, Prolog

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.2

1 Introduction

In the early years of software development, it was a well-known rule of thumb that in a typical software project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were spent in benchmarking the components of the software under development. Nowadays, despite the new development systems and languages with built-in tools, benchmarking still plays an important role in any software development project. Software benchmarking is a process, or a series of processes, designed to make sure that computer code does what it was designed to do and, conversely, that it does not do anything

* This work was funded by the ERDF (European Regional Development Fund) through Project 9471 – *Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação (Projeto 9471-RIDTI)* – and through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013. Miguel Areias was funded by the FCT grant SFRH/BPD/108018/2015.

unintended [15]. Software benchmarking techniques can be broadly classified into *white-box benchmarking* and *black-box benchmarking*. The former refers to the structural benchmarking technique that designs test cases based on the information derived from source code. The latter, also called data-driven or input/output driven benchmarking, views the program as a black box, and its goal is to be completely unconcerned about the internal behavior and structure of the program and, instead, it concentrates on finding circumstances in which the program does not behave according to the specifications [15]. Nowadays, a good practice in big software projects is the *Continuous Integration (CI)* software development technique [9]. The key idea of CI is to let developers integrate their work as they produce it, instead of doing the integration at the end of each software module. Each integration is then verified by an automated benchmark environment which ensures the correctness of the integration or detect the integration errors. One of the greatest advantages of the CI is an earlier detection of errors, leading to smaller and less complex error corrections.

Prolog is a language with a long history whose community has seen a large number of implementations which evolved independently. This situation is totally different from more recent languages, such as, Java, Python or Perl, that either have a single implementation (Python, Perl) or are controlled centrally (Java implementations can only be called Java if they satisfy certain standards). The international standard for Prolog ISO/IEC 13211 [11] was created to standardize Prolog implementations. However, due to the different sources of development, the standard is not completely implemented in most Prolog systems. The Prolog community knows that different Prolog systems have different dialects with different syntax and different semantics for common features. A good example is Wielemaker's recent work on dictionaries and new string extensions to Prolog [22], which are not part of the ISO/IEC 13211. A different direction is the one followed by Wielemaker and Santos Costa [20, 21], where they studied the status of the standardization of Prolog systems and gave a first step towards a new era of Prolog, where all systems are fully compliant with each other. While this new era is not reached yet, every publicly available significant piece of Prolog code must be carefully examined for portability issues before it can be used in any Prolog system. This creates a significant obstacle, if one wants to compare Prolog systems in performance and/or correctness measurements.

Benchmark suite frameworks for Prolog have been around for some time [6, 10] and several still exist that are specially aimed to evaluate Prolog systems. Two good examples are China [5] and OpenRuleBench [12]. China is a data-flow analyzer for constraint logic programming languages written in C++ which performs bottom-up analysis deriving information on both call-patterns and success-patterns by means of program transformations and optimized fix-point computation techniques. OpenRuleBench is an open community resource designed to analyze the performance and scalability of different rule engines in a set of semantic web information benchmarks.

In previous work, we have also developed a first benchmark suite framework based in the CI and black-box approaches to support the development of the Yap Prolog system [17]. This framework was very important for our work [1, 2], mainly to ensure Yap's correctness in the context of several improvements and new features added to its tabling engine. The framework handles the comparison of outputs obtained through the run of benchmarks for general Prolog queries and for the answers stored in the table space if using tabled evaluation. It also supports the different Prolog dialects of the XSB Prolog [16] and B-Prolog [23] systems. However, the framework still lacks important user productive features such as automation and a powerful graphical user interface.

In this paper, we extend such a previous work and we propose a fully automated test bench environment for Prolog systems, named *Yet Another Prolog Test Bench Environment*

(YAPTBE), aimed to assist developers in the development and integration of Prolog systems. YAPTBE is based in a cloud computing architecture and relies in Jenkins [18] and in a set of new Jenkins plugins to manage the underneath infrastructure. Arguably, Jenkins is one of the most successful open source automation tools to manage a CI infrastructure. Jenkins, originally called Hudson, is written in Java, provides hundreds of plugins to support building, deploying and automating any project, and is used by software teams of all sizes, for projects in a wide variety of languages and technologies.

YAPTBE includes the following features: (i) a graphical user interface which coordinates all the interactions with the test bench environment; (ii) the definition of a cloud computing environment including different computing nodes running different operating systems; (iii) an automated process to synchronize, compile and run Prolog systems against sets of benchmarks; (iv) an automated process to handle the comparison of output results and store them for future reference; (v) a smooth integration with state-of-the-art version control systems such as GIT; (vi) a publicly available online version that allows anonymous users to interact with the environment to follow the state of the several Prolog systems. To be best of our knowledge, YAPTBE is the first environment specially aimed for Prolog systems that supports all such features. For simplicity of presentation, we will focus our description in the Yap Prolog system, but YAPTBE can be used with any other system.

The remainder of the paper is organized as follows. First, we briefly introduce some background about Prolog and tabled evaluation. Next, we discuss the key ideas of YAPTBE and how it can be used to support the development and evaluation of Prolog systems. Then, we present the key design and implementation details and we show a small test-drive over YAPTBE. Finally, we outline some conclusions and indicate further working directions.

2 Background

Arguably, one of the most popular logic programming languages is the Prolog language. Prolog has its origins in a software tool proposed by Colmerauer in 1972 at *Université de Aix-Marseille* which was named *PROgramation en LOGic* [8]. In 1977, David H. D. Warren made Prolog a viable language by developing the first compiler for Prolog. This helped to attract a wider following to Prolog and made the syntax used in this implementation the *de facto* Prolog standard. In 1983, Warren proposed a new abstract machine for executing compiled Prolog code that has come to be known as the Warren Abstract Machine, or simply WAM [19]. The WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM's technology.

A logic program consists of a collection of Horn clauses. Using Prolog's notation, each clause may be a rule of the form:

$$a(\vec{X_0}) :- b_1(\vec{X_1}),\ b_2(\vec{X_2}),\ ...,\ b_n(\vec{X_n}).$$

where $a(\vec{X})$ is the head of the rule, $b_i(\vec{X_i})$ are the body subgoals and $\vec{X_i}$ are the subgoals' arguments, or it may be a fact (without body subgoals) and simply written as:

$$a(\vec{X_0}).$$

The symbol :− represents the logic implication and the comma (,) between subgoals represents logic conjunction, i.e., rules define the expression:

$$b_1(\vec{X_1}) \wedge b_2(\vec{X_2}) \wedge ... \wedge b_n(\vec{X_n}) \Rightarrow ga(\vec{X_0})$$

while facts assert $a(\vec{X_0})$ as true.

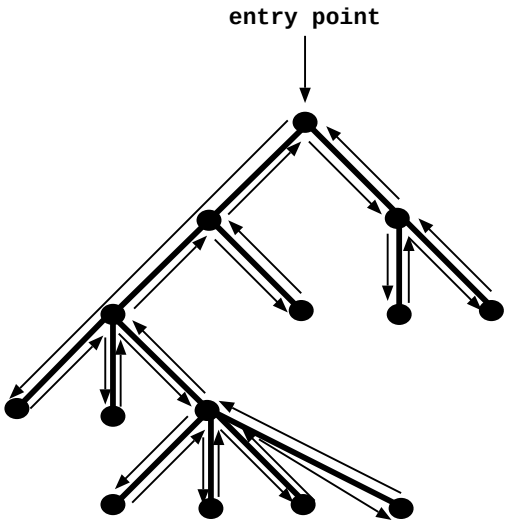


Figure 1 Depth-first left-to-right search with backtracking in Prolog.

Information from a logic program is retrieved through query execution. Execution of a query Q with respect to a program P proceeds by reducing the initial conjunction of subgoals in Q to subsequent conjunctions of subgoals according to a refutation procedure called SLD resolution [13]. Figure 1 shows a pure and sequential SLD evaluation in Prolog, which consists in traversing a search space in a *depth-first left-to-right* form. Non-leaf nodes of the search space represent stages of computation (*choice points*) where alternative branches (clauses) can be explored to satisfy the program's query, while leaf nodes represent solution or failed paths. When the computation reaches a failed path, Prolog starts the *backtracking* mechanism, which consists in restoring the computation up to the previous non-leaf node and schedule an alternative unexplored branch.

SLD resolution allows for efficient implementations but suffers from some fundamental limitations in dealing with recursion and redundant sub-computations. Tabling [7] is a refinement of Prolog's SLD resolution that overcomes some of those limitations. Tabling is a kind of dynamic programming implementation technique that stems from one simple idea: save intermediate answers for current computations in an appropriate data area, called the *table space*, so that they can be reused when a *similar computation* appears during the resolution process. With tabling, similar calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in the corresponding table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all similar calls. Figure 2 shows the evaluation of a tabled program.

The top left corner of the figure shows the program code and the top right corner shows the final state of the table space. The program defines a small directed graph, represented by two *edge/2* facts, with a relation of reachability, defined by a *path/2* tabled predicate. The bottom of the figure shows the evaluation sequence for the query goal *path(a,Z)*. Note that traditional Prolog would immediately enter an infinite loop because the first clause of *path/2* leads to a similar call (*path(a,Y)* at step 1).

First calls to tabled subgoals correspond to *generator nodes* (depicted by white oval boxes) and, for first calls, a new entry representing the subgoal is added to the table space (step 0). Next, *path(a,Z)* is resolved against the first *path/2* clause calling, in the continuation,

```
:- table path/2.

path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Table

subgoals	answers
0. path(a,Z)	3. Z=b 5. Z=a

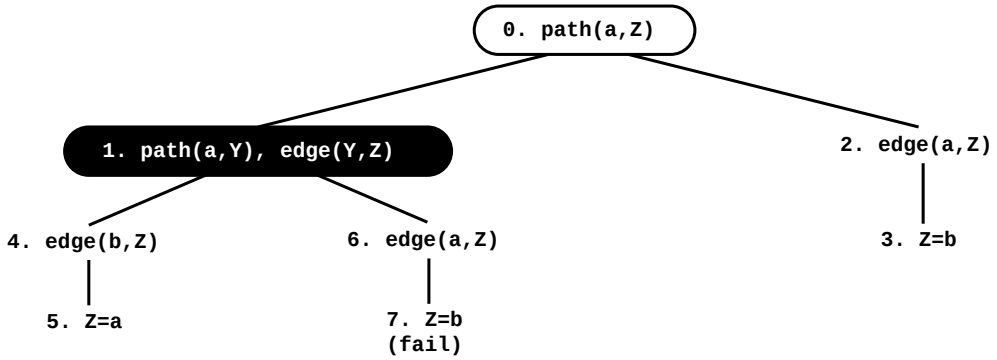


Figure 2 An example of a tabled evaluation.

path(a,Y). Since *path(a,Y)* is a similar call to *path(a,Z)*, the engine does not evaluate the subgoal against the program clauses, instead it consumes answers from the table space. Such nodes are called *consumer nodes* (depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended (step 1). The only possible move after suspending is to backtrack and try the second clause for *path/2* (step 2). This originates the answer $\{Z{=}b\}$, which is then stored in the table space (step 3). At this point, the computation at node 1 can be resumed with the newly found answer (step 4), giving rise to one more answer, $\{Z{=}a\}$ (step 5). This second answer is then also inserted in the table space and propagated to the consumer node (step 6), which originates the answer $\{Z{=}b\}$ (step 7). This answer had already been found at step 3. Tabling does not store duplicate answers in the table space and, instead, repeated answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. Since there are no more answers to consume nor more clauses left to try, the evaluation ends and the table entry for *path(a,Z)* can be marked as *completed*.

For our test bench environment, tabling plays an important role because, with tabling, we might want to handle not only the comparison of outputs obtained through the run of general Prolog queries, but also the comparison of the structure/configuration of the tables stored during such executions. Moreover, if we table all predicates involved in a computation, we can use tabling as a way to keep track of all intermediate subcomputations that are done for a particular top query goal. Tabling can thus be used as a built-in powerful tool to check and ensure the correctness of the Prolog engine internals. To take advantage of tabling, we thus need to design our test bench environment to take into account the kind of output given by tabling.

3 Yet Another Prolog Test Bench Environment

In this section, we introduce the key concepts about YAPTBE's design.

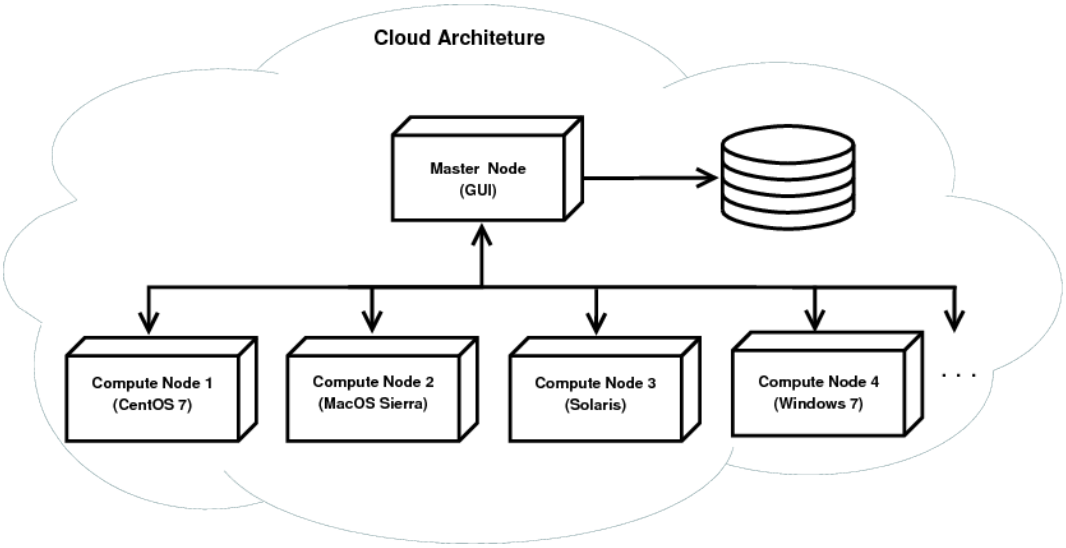


Figure 3 The cloud-based architecture of YAPTBE.

3.1 Cloud-Based Architecture

In the early years of software development, a piece of software was designed having in mind a specific operating system and hardware architecture. As time passed by, operating systems and hardware architectures became more sophisticated and branched out into a multiplicity of platforms and versions, which are often variations of the same software or hardware component. In order to go along with this new reality, nowadays, whenever a new piece of software is designed, developers must ensure that it will work correctly in different operating systems and hardware architectures. Fortunately, cloud computing has emerged as an excellent alternative for software development. Cloud computing is very powerful because it provides ubiquitous access to multiple operating systems, heterogeneous and non-heterogeneous hardware architectures, which can be seen and manipulated as being similar resources. In what follows, we explain how we tried to bring the advantages of cloud computing into YAPTBE's design.

Figure 3 shows a general perspective of YAPTBE's cloud-based architecture. At the entry point, a *master node* with a *Graphical User Interface (GUI)* allows users to interact with YAPTBE's cloud-based infrastructure. The master node is then connected, through an intranet connection, to a *storage device* (shown at right in Fig. 3), which stores and backups all relevant information, and to several *computing nodes* (or *slave nodes*) which can be connected through an intranet or internet connection, depending if they are or not close enough to the master node. Each computing node has its own version of an operating system. In Fig. 3, we can see four computing nodes running the *CentOS 7*, *MacOS Sierra*, *Solaris* and *Windows 7* operating systems. Each computing node is then organized in a working space specially aimed to store the resources available in the node and to store the files generated by the users during the usage of the node. Figure 4 shows an example of a tree hierarchy for the working space of a computing node.

At the top of the hierarchy, we have the root folder named *'Computing Node'*. The root folder is divided in two sub-folders, the *Developers* and the *Resources* folders.

The *Resources* folder is then used to store the sources of the Prolog systems and the sources of the test bench suites available in the computing node. Figure 4 shows two Prolog

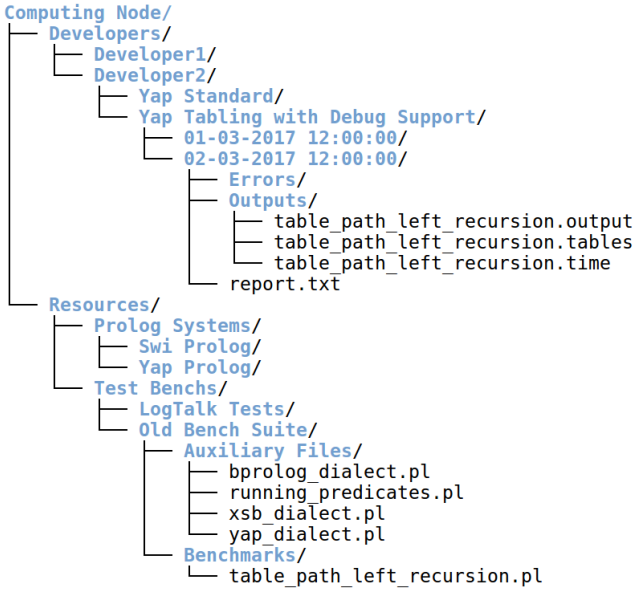


Figure 4 Working space of a computing node.

systems (represented by the *'Swi Prolog'* and the *'Yap Prolog'* folders) and two test bench suits (represented by the *'LogTalk Tests'* and the *'Old Bench Suite'* folders) available in the computing node. The folder structure under each particular resource is then independent from YAPTBE. Figure 4 shows the specific structure for the *'Old Bench Suite'* resource. The *'Old Bench Suite'* resource corresponds to the benchmark suite we have developed in previous work to ensure Yap's correctness in the context of several improvements and new features added to its tabling engine [1, 2]. It contains two sub-folders, one named *'Auxiliary Files'* and another *Benchmarks*. The *Benchmarks* folder stores the Prolog files for the benchmarks (such as *table_path_left_recursion.pl* representing the example in Fig. 2). The *'Auxiliary Files'* folder holds the files related with the dialects specificities of each Prolog system and with the running of the benchmarks (used to launch/terminate a run; obtain the running time; print outputs; print internal statistics about the run; or print the results stored in the tables, if using tabling).

The *Developers* folder stores the information for the *builds* and the *jobs* of each developer (Fig. 4 shows the folder structure for *Developer2*). The first level sub-folders represent the developer's builds and the second level sub-folders represent the developer's jobs for a particular build. Prolog sources can be configured and/or compiled in different fashions. Each build folder corresponds to such a configuration and holds all the files required to launch the Prolog system. In Fig. 4, we can see that *Developer2* has two builds, one named *'Yap Standard'*, holding the binaries required to run Yap compiled with the default compilation flags, and another named *'Yap Tabling with Debug Support'*, holding the binaries required to run Yap compiled with tabling and debugging support. Finally, each job folder stores the outputs obtained in a particular run of a build. In Fig. 4, we can see that the *'Yap Tabling with Debug Support'* build has two jobs (by default, jobs are named with the time when they were created). The folder structure under each particular job is then independent from YAPTBE. For the job named *'02-03-2017 12:00:00'*, we can see a *report.txt* file with a summary of the run and two folders used to stored auxiliary error and output information about the run, in this case, the query output, the structure of the table space and the execution time.

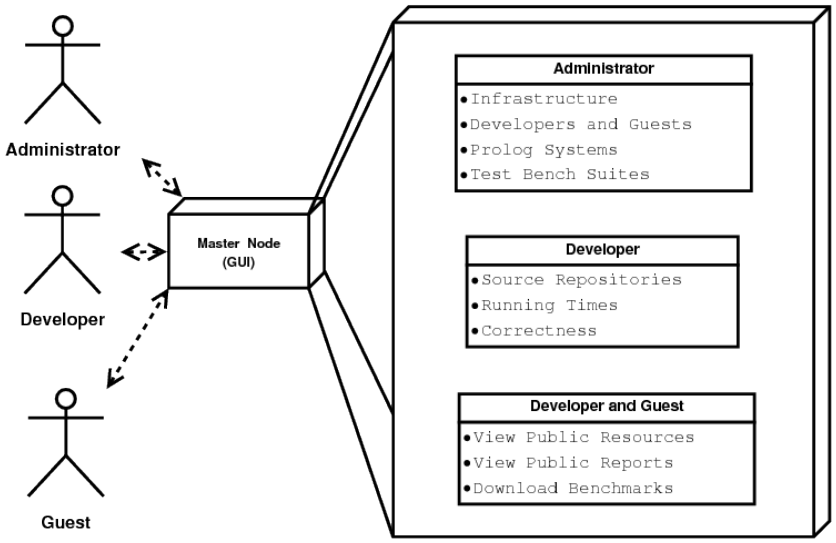


Figure 5 Users and services provided.

3.2 Services

YAPTBE is designed to provide different services to different types of users. We consider three different types of users: (i) the system administrators; (ii) the developers; and (iii) the guest users. Figure 5 shows the key services provided to each user.

The system administrators will manage the infrastructure and configure the several aspects of the test bench environment. They can manage the infrastructure by adding/removing computing nodes, manage the accounts and access permissions for developers and guests, and manage the available resources by setting up the source repositories for the Prolog systems and for the test bench suites.

The developers will use the environment for performance measurements and for ensuring the correctness of the integration of the code being developed. They can manage all features related with the source repositories, such as merging, branching, pulling, configure and compile, run benchmarks and compare the running times obtained in different dates with different Prolog systems, test the correctness of the Prolog systems and check specific features, such as tabling or multithreading.

The guest users can use the environment to check and follow the state of the several Prolog systems. They can view the resources, navigate in the existent reports from previous runs, and download the available benchmarks.

Since YAPTBE's main target users are the developers, they will have a special access to the infrastructure. They will be allowed to include their machines into the cloud in such a way that they can develop and deploy their work in an computing node where they can control the environment of the run. This special feature is important because, often, developers want to quickly access what went wrong with their integration. As expected, the machine of the developer will be protected against abusive workloads by other users. We allow developers to define if their computing nodes are private or public and, in the latter case, we also allow them to define the resources that they want to share with other developers. The public resources that can be defined vary from the maximum disk space to be used, to the maximum number of cores to be used and to the maximum number of jobs to be accepted.

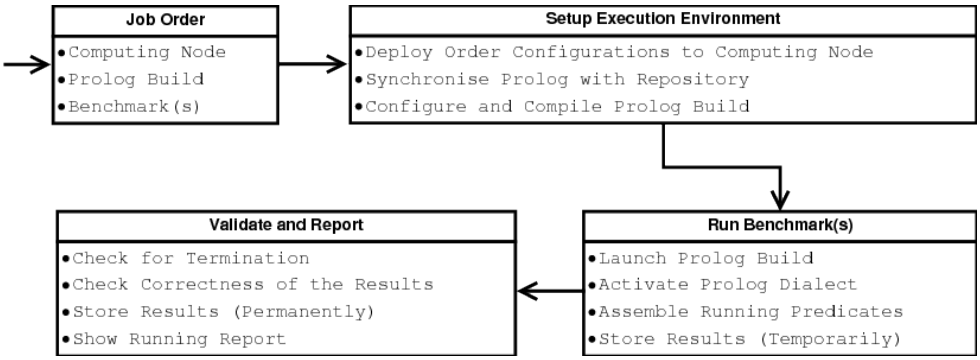


Figure 6 Pipeline of a job request to test the correctness of a Prolog system build.

4 Implementation Details

In this section, we introduce some extra details about YAPTBE's implementation, which relies on the Jenkins framework [18] to manage the cloud-based architecture. Jenkins has some important advantages: (i) the user interface is simple, intuitive, visually appealing, and with a very low learning curve; (ii) it is extremely flexible and easy to adapt to multiple purposes; and (iii) has several open source plugins available, which cover a wide range of features, such as, version control systems, build tools, code quality metrics, build notifiers, integration with external systems, and user interface customization.

In a nutshell, we use Jenkins to manage the GUI, the computing nodes and the scheduling of jobs. The master node has a main Jenkins agent that runs the GUI and connects the master node with the computing nodes. Jobs are deployed by the master node to the computing nodes and, to run a job, each computing node has a Jenkins slave agent that manages the run. At the end of a run, the slave agent sends back a minor report with the results obtained to the main agent. The full details of the run are stored locally in the computing node. If a storage device is available, it can be used to backup the results. Next, we give more details about the scheduling of a job.

4.1 Job Scheduling

Job scheduling is one of the most important features of YAPTBE. We consider a job to be any automated service that can be provided by the environment. Jobs can vary from downloading and installing a Prolog system in a computing node to executing a run order from a developer. Figure 6 shows the pipeline for running a job request made by a developer. For the sake of simplicity, we will assume that a developer has all the permissions necessary to run the job and wants to run the latest committed version in the repository of the Prolog system.

On the initial stage of the pipeline, the developer creates an order for a job through the GUI of the master node. The order defines the computing node, the Prolog system build and the (set of) benchmark(s) to be run. The scheduling of the order is managed by Jenkins, which will insert the order in the computing node pool. When the computing node is ready to execute the order, the pipeline moves to the next stage to setup the execution environment. In this stage, Jenkins activates a set of internal scripts that will deploy the configurations of the order to the computing node. These scripts will synchronize the Prolog system with its repository, configure and compile the corresponding build in the computing node.

(a) Adding Yap Prolog as a resource.

(b) Installing Yap Prolog in a computing node.

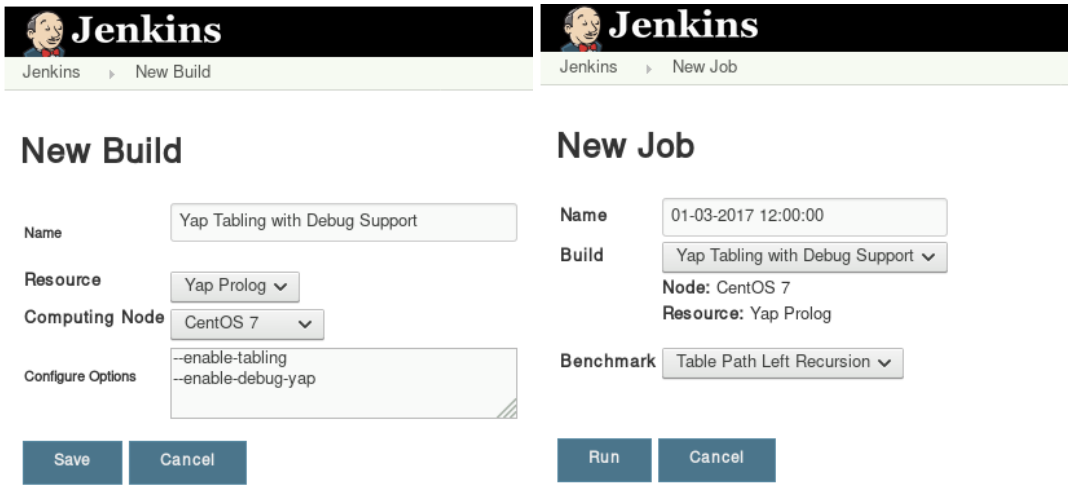
Figure 7 Resource management GUI for administrators.

On the next stage of the pipeline, the *Run Benchmark(s)* stage, the Prolog build is launched and the (set of) benchmark(s) is ran. This can include selecting the Prolog dialect, which will activate a set of compatibility predicates that will be used to run the benchmark, and selecting specific running predicates to obtain specific outputs, such as the structure of the table space, if using tabling. Afterwards, the results are stored temporarily within a folder structure similar to the one described in Fig. 4, which can be used to store auxiliary error and/or output information, such as output answers, tabled answers, execution time, the structure of the table space, or internal Prolog statistics.

On the last stage, to validate the results, YAPTBE searches for execution failures, such as segmentation fault errors, and if no failures exist, it checks the correctness of the results. We assume that results are correct if at least two Prolog system give the same solutions. For our old bench suite, we are using the Yap Prolog and the SWI Prolog for standard benchmarks and the Yap Prolog and the XSB Prolog for tabled benchmarks (in this case, we store the output results and the answers stored in the tables). Thus, at this stage, we compare the results obtained in the run with the results pre-stored and assumed to be correct. If the results match, then the run is considered to be correct, otherwise the run is considered to be a error. Finally, the results are stored in an permanent and unique location, and a report with information is sent to the Jenkins master agent. The report has the status of the run, the execution times and the folder locations for the full output and error details.

4.2 Test-Driving YAPTBE

In this section, we show a small test-drive of YAPTBE. Jenkins is already packed with a huge amount of tools and has also several highly valued plugins that can be easily installed on demand. Even so, to allow administrators, developers and guests to use YAPTBE in an easier fashion, we have developed a new custom made plugin that was integrated in Jenkins. The following figures illustrate a scenario where a developer wants to use the Yap Prolog system and a computing node running the CentOS 7 operating system.



(a) Creating a new build for Yap Prolog system. **(b)** Running a job with a previously defined build.

Figure 8 Job management GUI for developers.

Figure 7 shows the resource management GUI for administrators for adding Yap Prolog as a resource (Fig. 7a) and to install it in the computing node running the CentOS 7 operating system (Fig. 7b). In both cases, the GUI is quite simple. To add a new resource (Fig. 7a), the administrator has to define the name of the resource, the link to the repository with the source code, and a template with the commands to build the binary for the resource. The template can include optional arguments to be defined by the developers. For example, in Fig. 7a, the build template starts with a configure command which includes optional arguments (*'Configure Options'*) to be later defined by the developers when building a specific build of this resource. To install a resource in a specific computing node (Fig. 7b), the administrator defines the desired computing node and resource and then presses the *Install* button. If the resource installs correctly, it becomes immediately available in the computing node.

Figure 8 then shows the job management GUI for developers for creating a new build for the Yap Prolog system in the CentOS 7 computing node (Fig. 8a) and to deploy a job using such build (Fig. 8b). Again, in both cases, the GUI is quite simple. To create a new build (Fig. 8a), the developer has to define the name of the build, the resource and computing node to be used and, if the administrator has defined optional arguments in the build template commands, then such optional commands can be included here. This is the case of the *'Configure Options'* entry as previously defined in Fig. 7a. In this particular example, the developer is building Yap with tabling and debug support. After the build be saved, it becomes available for the developer to use it in future orders for a job. To deploy a job (Fig. 8b), the developer sets a name for the job and defines the build to be used (up on the definition, the computing node and resource will automatically appear in a non-editable fashion, thus that the developer can see if it is using the correct build settings). At the end, the developer defines the benchmark or set of benchmarks to be run and presses the *Run* button to launch the corresponding job. The job will enter in the job scheduler and follow the pipeline described in the previous subsection.

Although we have already implemented all the features shown, there are still many other important features that are undergoing, such as: (i) implementation of a storage node to

backup all important data; (ii) design and implement a GUI for guest users; (iii) implement a set of strict security policies for all users; (iv) increase significantly the number of tests and benchmarks available. We expect to conclude these features soon and to have the first version of YAPTBE available online in the near future.

5 Conclusions and Further Work

Software testing and benchmarking is a key component of the software development process. In this paper, we extended a previous work on a benchmark suite for the Yap Prolog system and we proposed a fully automated test bench environment for Prolog systems, named *Yet Another Prolog Test Bench Environment (YAPTBE)*, aimed to assist developers in the development and integration of Prolog systems. YAPTBE is based in a cloud computing architecture and relies in Jenkins and in a set of new Jenkins plugins to manage the underneath infrastructure. We presented the key design and implementation aspects of YAPTBE and showed several of its most important features, such as its graphical user interface and the automated process that builds and runs Prolog systems and benchmarks.

Besides assisting in the development of Prolog systems, we hope that YAPTBE may, in the future, contribute to reduce the gap between different Prolog dialects and to create a salutary competition between Prolog systems in different benchmarks.

In the recent past, multiple features have been added to Prolog's world. One such feature is the ISO Prolog multithreading standardization proposal [14], which currently is implemented in several Prolog systems including Ciao, SWI Prolog, XSB Prolog and Yap Prolog, providing a highly portable solution given the number of operating systems supported by these systems. Arguably, one of the features that promises to have a significant impact is the combination of multithreading with tabling [3, 4], since Prolog users will be able to exploit the combination of higher procedural control with higher declarative semantics. Future work plans include the extension of YAPTBE to support the execution and output analysis of standard and tabled multithreaded Prolog runs.

References

1 Miguel Areias and Ricardo Rocha. On combining linear-based strategies for tabled evaluation of logic programs. *Journal of Theory and Practice of Logic Programming, (Special Issue, International Conference on Logic Programming)*, 11(4–5):681–696, July 2011.

2 Miguel Areias and Ricardo Rocha. On extending a linear tabling framework to support batched scheduling. In Alberto Simões, Ricardo Queirós, and Daniela da Cruz, editors, *Symposium on Languages, Applications and Technologies (SLATE 2012)*, pages 9–24, June 2012.

3 Miguel Areias and Ricardo Rocha. Towards multi-threaded local tabling using a common table space. *Journal of Theory and Practice of Logic Programming, (Special Issue, International Conference on Logic Programming)*, 12(4–5):427–443, September 2012.

4 Miguel Areias and Ricardo Rocha. On scaling dynamic programming problems with a multithreaded tabling system. *Journal of Systems and Software*, 125:417–426, 2017.

5 Roberto Bagnara. China – A Data-Flow Analyzer for CLP Languages. Available: `http://www.cs.unipr.it/China/` (accessed April 2017).

6 Klaus Bothe. A Prolog space benchmark suite: A new tool to compare Prolog implementations. *SIGPLAN Notices*, 25(12):54–60, 1990.

7 Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

8 Alain Colmerauer, Henry Kanoui, Robert Pasero, and Philippe Roussel. Un système de communication homme–machine en francais. Technical report, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1973.

9 Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

10 Ralph Haygood. A Prolog benchmark suite for aquarius. Technical report, University of California at Berkeley, 1989.

11 ISO/IEC 13211-1:1995: Information technology – Programming languages – Prolog – Part 1: General core, 1995.

12 Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *Internacional World Wide Web Conference*, pages 601–610. ACM, April 2009.

13 John Wylie Lloyd. *Foundations of Logic Programming*. Springer, 1987.

14 Paulo Moura. ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates, 2008. URL: `http://logtalk.org/plstd/threads.pdf`.

15 Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

16 Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.

17 Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

18 John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011.

19 David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

20 Jan Wielemaker and Vítor Santos Costa. Portability of Prolog programs: theory and case-studies. *CoRR*, abs/1009.3796, 2010. `http://arxiv.org/abs/1009.3796`.

21 Jan Wielemaker and Vítor Santos Costa. On the portability of Prolog applications. In Ricardo Rocha and John Launchbury, editors, *13th International Symposium on Practical Aspects of Declarative Languages (PADL2011)*, pages 69–83. Springer Berlin Heidelberg, 2011.

22 Jan Wielemakers. Swi-prolog version 7 extensions. In *International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments*, pages 109–123, 2014.

23 Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*, number 1753 in LNCS, pages 109–123. Springer, 2000.