information

MDPI

Article

On the Implementation of a Cloud-Based Computing Test Bench Environment for Prolog Systems †

Ricardo Gonçalves, Miguel Areias * and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto, Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal; rgoncalves@fc.up.pt (R.G.); miguel-areias@dcc.fc.up.pt (M.A.); ricroc@dcc.fc.up.pt (R.R.)

* Correspondence: miguel-areias@dcc.fc.up.pt

† This paper is an extended version of our paper published in the Symposium on Languages, Applications and Technologies 2017.

Received: 13 September 2017; Accepted: 13 October 2017; Published: date

Abstract: Software testing and benchmarking are key components of the software development process. Nowadays, a good practice in large 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 performing 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 as well as a new Jenkins plugin to manage the underlying infrastructure. We present the key design and implementation aspects of YAPTBE and show its most important features, such as its graphical user interface (GUI) and the automated process that builds and runs Prolog systems and benchmarks.

Keywords: software engineering; program correctness; benchmarking; Prolog

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% of the elapsed time and more than 50% of the total cost were spent in benchmarking the components of the software under development. Nowadays, despite the new developments in 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, likewise, that it does not do anything unintended [1]. 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, concentrate on finding circumstances in which the program does not behave according to the specifications [1]. Nowadays, a good practice in large software projects is the *continuous integration (CI)* software development technique [2]. The key idea of CI is to let developers integrate their work as they produce it, instead of performing 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 detects the integration errors. One of the greatest advantages of 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 many implementations, which have evolved independently. Some representative examples of Prolog implementations include

systems such as B-Prolog [3], Ciao Prolog [4], Mercury [5], Picat [6], Sicstus [7], SWI-Prolog [8], XSB Prolog [9] and YAP Prolog [10]. This situation is entirely different from more recent languages, such as Java, Python or Perl, that either have a single implementation (Python and Perl) or are controlled centrally (Java implementations are only Java if they satisfy certain standards). The international standard for Prolog ISO/IEC 13211 [11] was created to standardize Prolog implementations. However, because of 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 [12], which are not part of the ISO/IEC 13211. A different direction is that followed by Wielemaker and Santos Costa [13,14], who studied the status of the standardization of Prolog systems and gave a first step towards a new era of Prolog, for which all systems are fully compliant with each other. While this new era has not been 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 [15,16], and several still exist that are specifically aimed to evaluate Prolog systems. Two good examples are China [17] and OpenRuleBench [18]. China is a data-flow analyzer for constraint logic programming languages written in C++ that performs bottom-up analysis, deriving information on both call patterns and success patterns by means of program transformations and optimized fixed-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 a previous work, we also developed a first benchmark suite framework based on the CI and black-box approaches to support the development of the YAP Prolog system [10]. This framework was very important, mainly to ensure YAP's correctness in the context of several improvements and new features added to its tabling engine [19,20]. The framework handles the comparison of outputs obtained through the run of benchmarks for specific Prolog queries and for the answers stored in the table space if using tabled evaluation. It also supports the different dialects of the B-Prolog and XSB Prolog systems. However, the framework still lacks important user productive features, such as automation and a powerful graphical user interface (GUI).

In this paper, we extend the previous work and propose a fully automated test bench environment for Prolog systems, named Yet Another Prolog Test Bench Environment (YAPTBE), which aims to assist developers in the development and integration of Prolog systems [21]. YAPTBE is based on a cloud computing architecture [22] and relies on the Jenkins framework [23] and on a new Jenkins plugin to manage the underlying 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 and provides hundreds of plugins to support the building, deploying and automating of any project; it 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 GUI that 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 [24]; and (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 specifically aimed at Prolog systems that supports all such features. For simplicity of presentation, we focus our description on 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 on Prolog, tabled evaluations and the Jenkins plugin development. 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

This section introduces the basic concepts needed to better understand the rest of the paper.

2.1. Logic Programming, Prolog and Tabling

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 [25]. 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 [26]. Although other abstract machines, such as the tree-oriented abstract machine [27], exist, the WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM's technology or its variants [28,29].

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

$$a(\vec{X}_0).$$

The symbol $:-$ represents the logic implication and the comma (,) between subgoals represents logic conjunction; that is, rules define the expression:

$$b_1(\vec{X}_1) \wedge b_2(\vec{X}_2) \wedge ... \wedge b_n(\vec{X}_n) \Rightarrow a(\vec{X}_0)$$

while facts assert $a(\vec{X}_0)$ as true.

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 (Selective Linear Definite clause resolution)* [30]. Figure 1 shows a pure and sequential SLD evaluation in Prolog, which consists of 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 of restoring the computation up to the previous non-leaf node and scheduling an alternative unexplored branch.

SLD resolution allows for efficient implementations but suffers from some fundamental limitations in dealing with infinite loops and redundant sub-computations. Tabling [31] is a refinement of Prolog's SLD resolution that overcomes some of these 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

a small example of how infinite loops and redundant sub-computations can be avoided with the use of tabling.

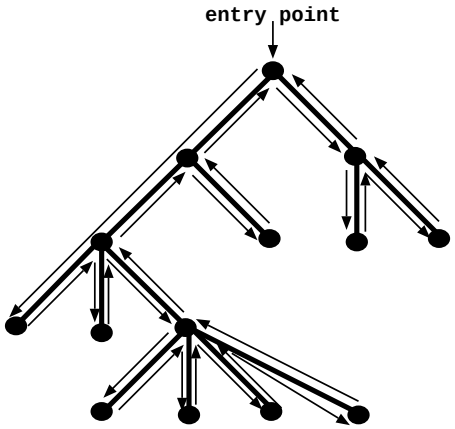


Figure 1. Depth-first left-to-right search with backtracking in Prolog.

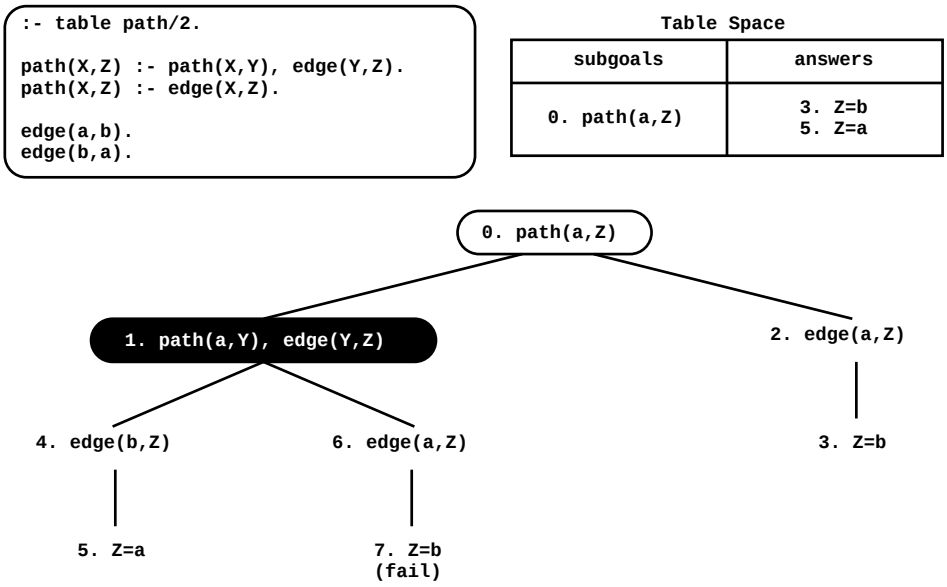


Figure 2. An example of a tabled evaluation.

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)*. We 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). With tabling, as we will see next, the infinite loop is avoided.

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, *path(a,Y)* (step 1). Because *path(a,Y)* is a similar call to *path(a,Z)*, the tabling 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 into the table space and propagated to the consumer node (step 6), which produces 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. As 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*.

As tabled resolution methods can considerably reduce the search space, avoid looping and have better termination properties than SLD resolution-based methods [31], the interest of the Prolog community in tabling has grown rapidly since the ground-breaking and pioneering work in the XSB Prolog system [32]. Several Prolog systems that initially did not support tabling are, nowadays, converging to support some kind of tabling mechanism. Implementations of tabling are currently available in systems such as B-Prolog [33], Ciao Prolog [34], Mercury [5], SWI-Prolog [8], XSB Prolog [32] and YAP Prolog [35]. For our test bench environment, it is then particularly important to support the evaluation of tabled programs. Tabling plays also an important role because, with tabling, one may want to handle not only the comparison of outputs obtained through the run of specific Prolog queries, but also the comparison of the structure/configuration of the tables stored during such executions. Moreover, if one tables all predicates involved in a computation, one can use tabling as a way to keep track of all intermediate sub-computations that are made 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 when designing a test bench environment, one thus needs to take into account the kind of output given by tabling.

2.2. Plugin Development and Security in Jenkins

Our work relies on the Jenkins framework [23] 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) it 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.

The development of plugins for Jenkins follows a strict guideline of policy rules (available at https://wiki.jenkins.io/display/JENKINS/Plugin+tutorial). To begin the plugin development, one should start by running the following command in a terminal:

```
$ {mvn hpi:create -Pjenkins}
```

This command triggers a set of instructions that will initialize the plugin working space. During the initialization, the command will ask for the *group* and the *artifact* identifiers. For the group identifier, one can use the default value provided by Jenkins, and for the artifact identifier, one can choose the name for the plugin.

After the completion of the command, the working space of the plugin has a hierarchy tree similar to that shown in Figure 3, which represents the working space of a Jenkins plugin named *HelloWorld*. The *pom.xml* file is used to define the specifications of the Jenkins version that one wants to build and run the plugin with and to define the plugin dependencies. Under the *src/main* folder, there are two sub-folders, the *java* folder and the *resources* folder. The java folder stores the algorithms, written in Java, that define the plugin execution, while the resources folder stores the resources used by the plugin and all the information related with its appearance.

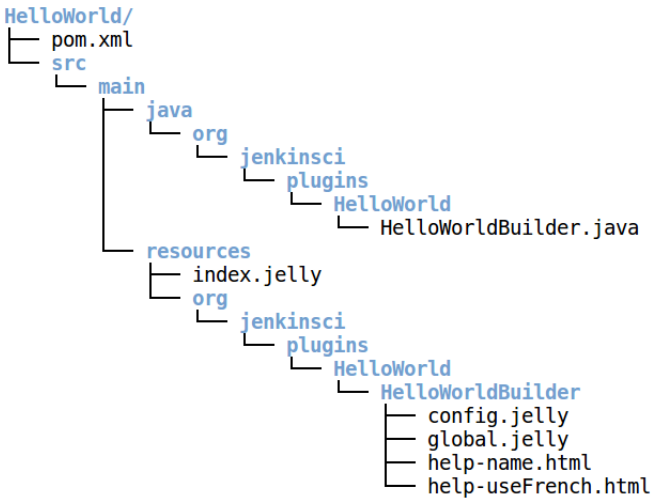


Figure 3. Working space of a Jenkins plugin named *HelloWorld*.

To deploy the plugin in a running Jenkins instance, one should execute a command similar to the following:

```
$ mvn hpi:run -Djetty.port=9090 -Pjenkins
```

In this example, we have chosen a command that runs the Jenkins system in port 9090 of a *localhost* machine. By default, if we have not defined any port, Jenkins runs in the predefined port 8080. Afterwards, when the Jenkins system is ready, one can start using Jenkins with the plugin by visiting in a Web browser the link http://localhost:9090/jenkins.

When the plugin is ready to be used, one can distribute it on the installation of any Jenkins instance. To do so, one must create an image of the plugin and run the following command:

```
$ mvn package
```

The command creates a *target/*.hpi* file with the complete details about the plugin. The file can then be uploaded in any Jenkins instance via the GUI.

Finally, before beginning to use the plugin, the guideline of the policy rules recommends that one should enable Jenkins Security controllers. In order to set up the correct configurations, one should use the administrator dashboard and configure *Global Security* with the following options:

- *Security*: enable.
- *Jenkins' own user database*: allow users to sign up.
- *Project-based Matrix Authorization Strategy*: add the users with the desired permissions.

3. Yet Another Prolog Test Bench Environment

This section introduces the key concepts of YAPTBE's design.

3.1. Cloud-Based Architecture

In the early years of software development, a piece of software was designed with a specific operating system and hardware architecture in mind. As time passed, operating systems and hardware architectures became more sophisticated and branched out into a multiplicity of platforms and versions, which were often variations of the same software or hardware components. In order to progress with this new reality, whenever a new piece of software was designed, developers had to ensure that it would work correctly in different operating systems and hardware architectures.

Cloud computing [22] is a powerful and respected alternative for software development that has emerged and is being used in multiple different contexts. Cloud computing is very powerful because it

provides ubiquitous access to multiple operating systems and heterogeneous and non-heterogeneous hardware architectures, which can be seen and manipulated as being similar resources. In what follows, we explain how we aimed to bring the advantages of cloud computing into YAPTBE's design.

Figure 4 shows a general perspective of YAPTBE's cloud-based architecture. At the entry point, a *master node* with a 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 the right in Figure 4), 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 on whether they are close enough or not to the master node. Each computing node has its own version of an operating system. In Figure 4, one can observe four computing nodes running the CentOS 7, MacOS Sierra, Solaris and Windows 7 operating systems.

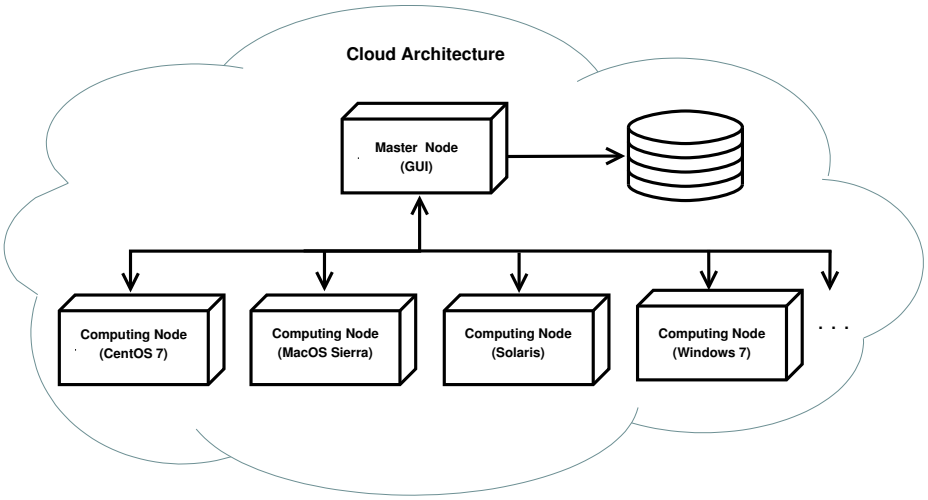


Figure 4. The cloud-based architecture of Yet Another Prolog Test Bench Environment (YAPTBE).

Each computing node is then organized in a working space specifically aimed at storing the resources available in the node and at storing the files generated by the users during the usage of the node. Figure 5 shows an example of a tree hierarchy for the working space of a computing node.

```
Computing Node/
├── Developers
│   ├── Developer1
│   └── Developer2
│       ├── Yap Standard
│       └── Yap Tabling with Debug Support
│           ├── 01-03-2017 12:00:00
│           └── 02-03-2017 12:00:00
│               ├── Errors
│               ├── Outputs
│               │   ├── table_path_left_recursion.output
│               │   ├── table_path_left_recursion.tables
│               │   └── table_path_left_recursion.time
│               └── report.txt
└── Resources
    ├── Prolog Systems
    │   ├── Swi Prolog
    │   └── Yap Prolog
    └── Test Benchs
        ├── LogTalk Tests
        └── Old Bench Suite
            ├── Auxiliary Files
            │   ├── running_predicates.pl
            │   ├── swi_dialect.pl
            │   └── yap_dialect.pl
            └── Benchmarks
                └── table_path_left_recursion.pl
```

Figure 5. 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 into 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 5 shows two Prolog 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 5 shows the specific structure for the *Old Bench Suite* resource. The *Old Bench Suite* resource corresponds to the benchmark suite we have developed in a previous work to ensure YAP's correctness in the context of several improvements and new features added to its tabling engine [19,20]. It contains two sub-folders, one named *Auxiliary Files* and another named *Benchmarks*. The *Benchmarks* folder stores the Prolog files for the benchmarks (such as *table_path_left_recursion.pl*, representing the example in Figure 2). The *Auxiliary Files* folder holds the files related with the dialects specific to 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 (Figure 5 shows the folder structure for *Developer2*). 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 a build configuration and holds all the files required to launch the Prolog system with such a configuration. In Figure 5, one can observe 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. Additionally, each job folder stores the outputs obtained in a particular run of a build. In Figure 5, one can observe 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*, one can observe 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.

3.2. Services

YAPTBE is designed to provide different services to different types of users. We consider three different types of users: (i) the administrators, (ii) the developers, and (iii) the guest users. Figure 6 shows the key services provided to each user.

The administrators will manage the infrastructure and configure 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 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; configuring and compiling; running benchmarks and comparing the running times obtained at different dates with different Prolog systems; testing the correctness of the Prolog systems; and checking 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.

Because YAPTBE's main target users are the developers, they will have special permission to access 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 a 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 whether 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 publicly in the cloud. Additionally, the developers can define the maximum amount of local resources that can be publicly shared in the cloud, such as the disk space, the number of cores and the number of jobs that can be accepted per day.

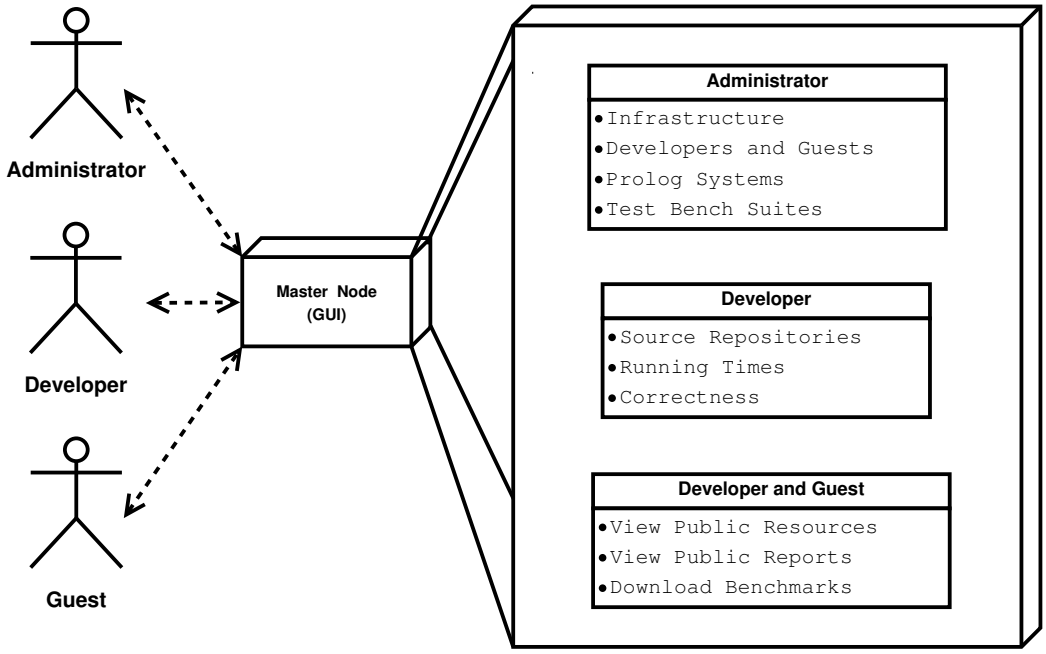


Figure 6. Users and services provided.

4. Implementation Details

This section introduces some extra details about YAPTBE's implementation. In short, we use Jenkins to manage the GUI, the computing nodes and the scheduling of jobs. The master node has a Jenkins main agent that runs the GUI and manages the connections between the master node and 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 back up 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 7 shows the pipeline for running a job request made by a developer. For the sake of simplicity, we assume that the 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.

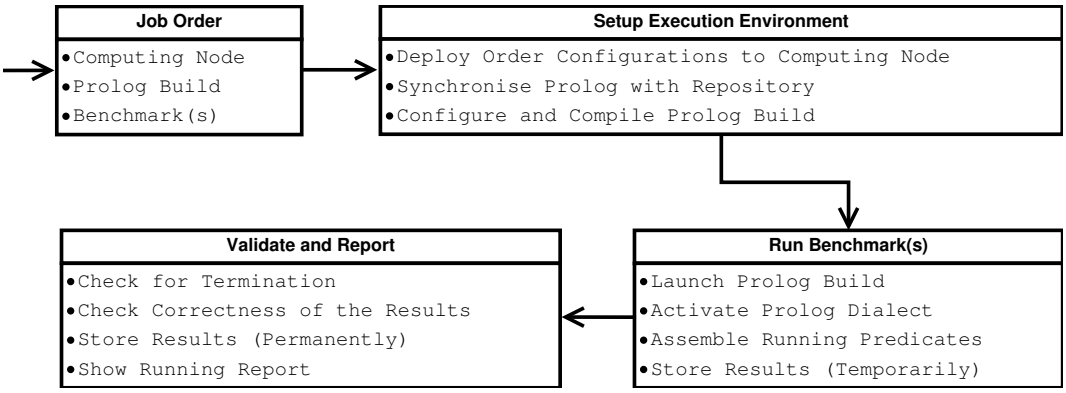


Figure 7. Pipeline of a job request to test the correctness of a Prolog system build.

On the initial stage of the pipeline, the *Job Order* stage, 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 inserts the order into the computing node pool.

When the computing node is ready to execute the order, the pipeline moves to the next stage to set up the execution environment, the *Setup Execution Environment* stage. In this stage, Jenkins creates a folder in the computing node and activates a set of internal scripts that will deploy the configurations of the order to the folder of the computing node. These scripts will synchronize the Prolog system with its repository, and configure and compile the corresponding build in the computing node.

On the next stage of the pipeline, the *Run Benchmark(s)* stage, the build of the Prolog is launched and the (set of) benchmark(s) is run. This can include selecting the Prolog dialect, which will activate a set of compatibility predicates, which 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. After launching the run, YAPTBE simply waits for the run to be complete. The execution time of the run is conditioned by multiple factors, such as the computing node workload or the operating system scheduler and memory manager. When the execution is complete, YAPTBE stores the results temporarily within a folder structure similar to that described in Figure 5, which can be used to store auxiliary error and/or output information, such as output answers, tabled answers, the execution time, the structure of the table space, or the internal Prolog statistics.

At the last stage, the *Validate and Report* stage, the results are validated. 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 systems give the same solutions. For our previous bench suite, we used 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 an error. Finally, the results are stored in a permanent and unique location, and a report with information is sent to the Jenkins master agent. The report contains the status of the run, the execution times, and the folder locations of the full output and error details.

4.2. Our Plugin Approach to Integrate Jenkins with YAPTBE

Jenkins already possesses a huge number of tools and also has 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 to be integrated in Jenkins. This subsection describes in more detail the internals of our new Jenkins plugin. The plugin

includes some new dashboards that are specific to the needs of the YAPTBE users. We consider two main dashboards, the *administrator dashboard* and the *developer dashboard*.

4.2.1. Administrator Dashboard

The administrator dashboard extends the *ManagementLink* class, which means that, if the *Global Security* policy configuration is set to use the authorization scheme *Project-based Matrix Authorization Strategy* as mentioned in Section 2.2, by using our plugin, one can define the user permissions individually, instead of defining an overall security polity valid for all users. Additionally, users are able to be set with administrative privileges, thus allowing them to access the boards under the *ManagementLink* extensions, which are shown under the *Manage Jenkins* section and are only available to the users with administrative privileges.

However, we noticed that if someone knows the URLs that trigger the administrative boards, then the security policy of Jenkins can be bypassed and the administrative features freely accessed. To address this issue, we implemented an extra security layer, for which administrator users have in their *user identifier* the word *admin*. Upon accessing the administrator boards, our plugin checks not only for the permissions in the *Matrix Authorization Strategy*, but also for the keyword *admin* in the user identifier. As the administrator users can only be created by a single Jenkins administrator, this extra security layer is enough to ensure that only fully granted users can access the administrative boards. Thus, if somebody tries to access an administrative URL without permission, our plugin will deny the access and automatically redirect to Jenkins' default entry page.

The administrator dashboard includes four different boards; three of these, the *Add Resource*, *Install Resource* and *Manage Resources* boards, were designed de novo for our plugin, and the fourth, named *Configure Nodes*, simply redirects the plugin to the Jenkins' original node configuration board, where administrators are able to create, edit and remove the computing nodes to be used by the plugin.

The *Add Resource* board allows the defining of new resources. Resources are stored in the master node under the folder *admin* in an XML (eXtensible Markup Language) file named *resources.xml*. The definition of a resource includes the resource name, the GIT repository location, and the set of commands to be used in the build process by the developer to install the resource. Figure 8 shows an example for which two resources, named *Yap Prolog* and *SWI Prolog*, are stored in the *resources.xml* file. Each resource has a corresponding GIT repository location (tag element *source*) and a set of build commands (tag element *commands*).

```
<resources>
  <resource name="Yap Prolog">
    <source>https://github.com/vscosta/yap-6.3.git</source>
    <commands>./configure $Configure Options$; make install</commands>
    <computing_nodes/>
  </resource>
  <resource name="SWI Prolog">
    <source>https://github.com/SWI-Prolog/swipl-devel.git</source>
    <commands>./configure $Configure Options$; make install</commands>
    <computing_nodes/>
  </resource>
</resources>
```

Figure 8. An example of the *resources.xml* file.

Both build commands use a *configure* script to set up the general configurations of the Prolog systems using specific compilation flags, and, at the end, they create and install the Prolog binary file in the host operating system. The macro *$Configure Options$* specifies where the specific compilation flags of the developers choice (to be defined later when building and installing a resource) should be mapped in the sequence of build commands. In this example, both resources end with the *make install* instruction, which installs the Prolog system in the computing node.

The *Install Resource* board allows the installation of an already added resource (from the previous *Add Resource* board) in a computing node. The administrator selects a resource from the existing resources list and then chooses the desired computing node where the resource is going to be installed. At the GUI level, our plugin shows only the computing nodes where the desired resource is not yet installed, thus avoiding potential errors, such as if an administrator tries to install a resource in a machine where the resource already exists.

When the administrator activates the installation process, the Jenkins Web page will wait until the process completes. The installation process starts by cloning all the resource files to the selected node through a GIT command, and ends by updating the *resources.xml* file with the association between the computing node and the resource in installation. For the first resource installed, the plugin will create a new tag element called *node* within the *computing_nodes* element in the *resources.xml* file. The *node* element will contain the information about the computing node, that is, the label used by Jenkins to uniquely identify the computing node in the context of the YAPTBE framework. For the following resources to be installed, the plugin will simply create new *node* elements for each resource within the *computing_nodes* element where it is being installed. Figure 9 shows an example of the state of the *resources.xml* file after the successful installation of the resource *Yap Prolog* in two different computing nodes, the *CentOS 7* and the *MacOS Sierra* computing nodes. In this example, the resource *SWI Prolog* is not installed in any computing node.

```
▼<resources>
  ▼<resource name="Yap Prolog">
     <source>https://github.com/vscosta/yap-6.3.git</source>
     <commands>./configure $Configure Options$; make install</commands>
    ▼<computing_nodes>
       <node>CentOS 7</node>
       <node>MacOS Sierra</node>
     </computing_nodes>
   </resource>
  ▼<resource name="SWI Prolog">
     <source>https://github.com/SWI-Prolog/swipl-devel.git</source>
     <commands>./configure $Configure Options$; make install</commands>
     <computing_nodes/>
   </resource>
 </resources>
```

Figure 9. Installing the *Yap Prolog* resource in the *CentOS 7* and *MacOS Sierra* computing nodes.

Finally, the *Manage Resources* board allows the listing of the existing resources and the checking of their properties, such as their name, the computing nodes where they are installed, the existent sources, and the build commands. Additionally, it allows administrators to remove a resource from a computing node and to remove a resource from YAPTBE (this latter case is only possible if the resource is not installed in any computing node). To simplify the plugin and avoid inconsistencies in the installed resources, our plugin does not allow the editing of resources; that is, once the resource is installed, it must remain unchanged until it is removed. Consequently, to make a change in a resource, one must first delete the resource and then make a fresh install of the resource with the desired changes.

4.2.2. Developer Dashboard

The developer dashboard is available for all users and includes three different boards designed de novo for our plugin, the *New Build*, *Manage Builds* and *New Job* boards.

The *New Build* board allows developers to create new builds. In YAPTBE, a build is defined as the set of parameters to be used in a Jenkins job creation. For example, the developer starts by selecting a resource from the list of available resources in the computing node where a benchmark is to be tested. After choosing the resource, the defined build commands (set by the administrator in the *Add Resource* board) will be displayed to the developer, such that the desired missing commands can be set. When

this process is completed, an XML file named *builds_list.xml* is stored in the master node under the developer's project folder.

Figure 10 shows an example in which a new build named *Yap Tabling with Debug Support* is stored in the *builds_list.xml* file. The new build uses the *Yap Prolog* resource from the *CentOS 7* computing node with the macro *$Configure Options$* (detailed in Figure 8) replaced with the specific compilation flags - *-enable-tabling* - *-enable-debug-yap*. We only allow developers to define configuration options because we want to prevent developers from injecting command line instructions directly into the computing nodes, as an incorrect command line instruction might lead to irreparable inconsistencies in YAPTBE. With this approach, we are able to narrow down the iterations of developers to command lines that are already predefined in the framework. In the example shown in Figure 10, one can observe that an incorrect command line made by the developer will lead only to an erroneous configuration of the resource in the computing node.

```
▼<builds>
  ▼<build name="Yap Tabling with Debug Support">
     <resource path="../../Resources/yap-6.3">Yap Prolog</resource>
     <computing_node>CentOS 7</computing_node>
    ▼<commands>
       ./configure --enable-tabling --enable-debug-yap; make install
     </commands>
   </build>
 </builds>
```

Figure 10. An example of the *builds_list.xml* file.

The *Manage Builds* board allows for the listing of the existing builds, checking of how they were originally defined, and the removing of undesirable builds. As in the case of the *Manage Resources* from the administrator board, developers are not allowed to edit builds.

Finally, the *New Job* board allows developers to combine a build, from the existing builds list, with the desired benchmark to be tested by the selected build. A benchmark is a set of one or more test files, specifically designed to test whatever the developer is implementing. Benchmarks are defined under the administrator dashboards as a resource and are placed in the *admin* folder alongside with the *resources.xml* file. However, in the future, we may extend this feature to allow each developer to specify their own benchmarks. To do so, we intend to have a new board where the developers can define the benchmarks and save their details alongside with the *builds_list.xml* file specific to the developer's project folder.

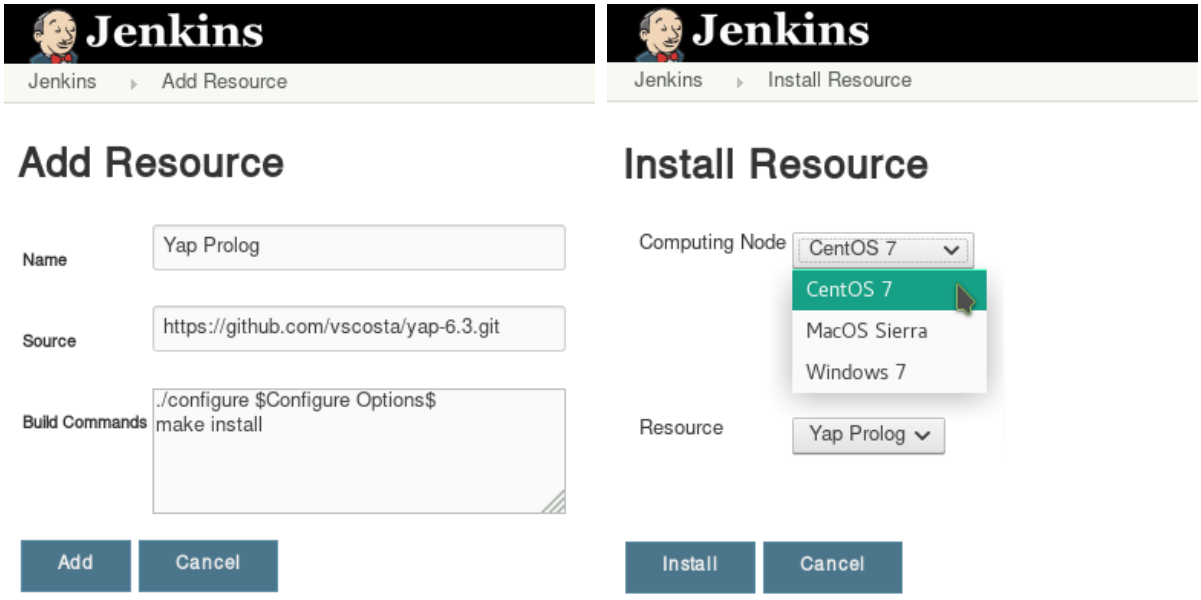
5. Test-Driving YAPTBE

In this section, we show a small test-drive of YAPTBE. We have designed the dashboards of the new plugin in such a way that users would not feel the difference between running Jenkins individually or combined with YAPTBE. The following figures illustrate a scenario in which a developer wants to use the YAP Prolog system and a computing node running the CentOS 7 operating system.

Figure 11 shows the resource management GUI for administrators for adding YAP Prolog as a resource (Figure 11a) and to install it in the computing node running the CentOS 7 operating system (Figure 11b).

In both cases, the GUI is quite simple. To add a new resource (Figure 11a), 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 Figure 11a, the build template starts with a configuration 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 (Figure 11b), the administrator defines the desired computing node and resource and

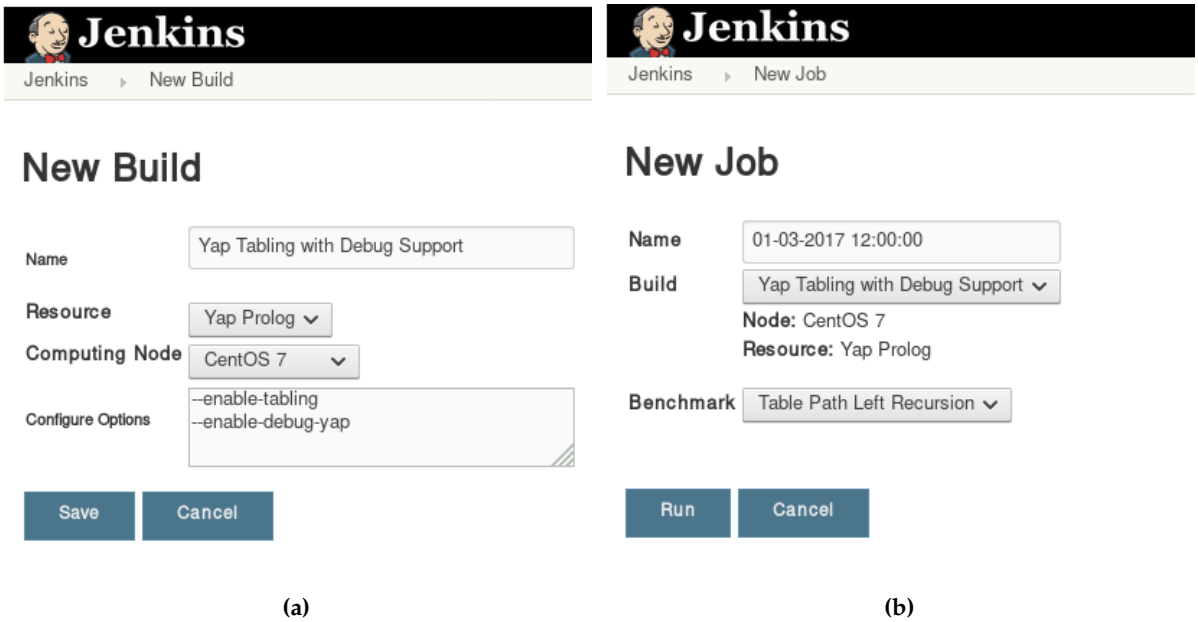
then presses the *Install* button. If the resource installs correctly, it becomes immediately available in the computing node.



(a) **(b)**

Figure 11. Resource management graphical user interface (GUI) for administrators. (**a**) Adding YAP Prolog as a resource. (**b**) Installing YAP Prolog in a computing node.

Figure 12 then shows the job management GUI for developers for creating a new build for the YAP Prolog system in the CentOS 7 computing node (Figure 12a) and to deploy a job using this build (Figure 12b).



(a) **(b)**

Figure 12. Job management graphical user interface (GUI) for developers. (**a**) Creating a new build for the YAP Prolog system. (**b**) Running a job with a previously defined build.

Again, in both cases, the GUI is quite simple. To create a new build (Figure 12a), 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 Figure 11a. In this particular example, the developer is building YAP with tabling and debug support. After the build is saved, it becomes available for the developer to use in future orders for a job. To deploy a job (Figure 12b), the developer sets a name for the job and defines the build to be used (upon the definition, the computing node and resource will automatically appear in a non-editable fashion; thus the developer can see whether it is using the correct build settings). At the end, the developer defines the (set of) benchmark(s) to be run and presses the *Run* button to launch the corresponding job. The job will enter into the job scheduler and follow the pipeline described previously.

6. Further Work

Although we have already implemented all of the features shown in Figures 11 and 12, there are some other important features that are still undergoing development, such as (i) the implementation of user security and access policies for monitoring and controlling the user's activity on the computing nodes; (ii) the implementation of a storage node to back up the overall system configurations and the results of the jobs' executions; (iii) the design and implementation of the GUI for guest users; and (iv) the design of a logging mechanism that collects information regarding YAPTBE's performance, accuracy and reliability.

As further work, we also intend to make a tighter integration between YAPTBE and the way benchmarks are handled. Currently, YAPTBE sees the benchmarks' results as a set of output files without knowing much about their contents. Our plan is to extend YAPTBE to include a reflexive engine with a well-defined protocol that will allow YAPTBE to have some extra knowledge about the accuracy and reliability of the benchmarks' results. In particular, we are interested in taking advantage of tabling as a way to keep track of all intermediate sub-computations that are made during the execution of a benchmark. To do so, YAPTBE must be extended with an internal communication protocol, which will allow it to understand the outputs given by a tabling engine. The same idea can be applied to support the execution and output analysis of multithreaded Prolog runs, as is discussed next.

Recently, multiple features have been added to Prolog's world. One such feature is the ISO Prolog multithreading standardization proposal [36], 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 [37,38], as Prolog users will be able to exploit the combination of a 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.

7. Conclusions

Software testing and benchmarking are key components of the software development process. However, as Prolog systems have different dialects with different syntax and different semantics for common features, the comparison between them is more difficult than in other programming languages. 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 at assisting developers in the development and integration of Prolog systems. YAPTBE is based on a cloud computing architecture and relies on Jenkins as well as a new plugin that manages the underlying infrastructure and integrates YAPTBE with Jenkins. We discussed the most relevant design and implementation points of YAPTBE and

showed several of its most important features, such as its GUI and the automated process that builds and runs Prolog systems and benchmarks.

Currently, YAPTBE is still under development; thus some important features are not yet implemented. We expect to conclude these features and have the first version of YAPTBE available online in the near future. As well as assisting in the development of Prolog systems, we hope that YAPTBE may, in the future, contribute to reducing the gap between different Prolog dialects and to create a salutary competition between Prolog systems in different benchmarks.

Acknowledgments: 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)*—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.

Author Contributions: All authors contributed fairly to the research described in this work. Miguel Areias and Ricardo Rocha wrote most of the paper and contributed with the initial research and main design decisions of the YAPTBE framework. Ricardo Gonçalves implemented the Jenkins plugin and made contributions to the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Myers, G.J.; Sandler, C.; Badgett, T. *The Art of Software Testing*, 3rd ed.; Wiley Publishing: Hoboken, NJ, USA, 2011.
2. Duvall, P.; Matyas, S.M.; Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*; The Addison-Wesley Signature Series; Addison-Wesley Professional: Boston, MA, USA, 2007.
3. Zhou, N.F. The Language Features and Architecture of B-Prolog. *Theory Pract. Log. Program.* **2012**, *12*, 189–218.
4. Hermenegildo, M.V.; Bueno, F.; Carro, M.; López-García, P.; Mera, E.; Morales, J.F.; Puebla, G. An Overview of Ciao and its Design Philosophy. *Theory Pract. Log. Program.* **2012**, *12*, 219–252.
5. Somogyi, Z.; Sagonas, K. Tabling in Mercury: Design and Implementation. In Proceedings of the International Symposium on Practical Aspects of Declarative Languages, Charleston, SC, USA, 9–10 January 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 150–167.
6. Zhou, N.F.; Kjellerstrand, H.; Fruhman, J. *Constraint Solving and Planning with Picat*; Springer: Berlin/Heidelberg, Germany, 2015.
7. Carlsson, M.; Mildner, P. SICStus Prolog—The first 25 years. *Theory Pract. Log. Program.* **2012**, *12*, 35–66.
8. Wielemaker, J.; Schrijvers, T.; Triska, M.; Lager, T. SWI-Prolog. *Theory Pract. Log. Program.* **2012**, *12*, 67–96.
9. Swift, T.; Warren, D.S. XSB: Extending Prolog with Tabled Logic Programming. *Theory Pract Log. Program.* **2012**, *12*, 157–187.
10. Santos Costa, V.; Rocha, R.; Damas, L. The YAP Prolog System. *Theory Pract. Log. Program.* **2012**, *12*, 5–34.
11. ISO. *ISO/IEC 13211-1:1995: Information Technology—Programming Languages—Prolog—Part 1: General Core*; ISO: Geneva, Switzerland, 1995; pp. 1–199.
12. Wielemaker, J. SWI-Prolog version 7 extensions. In Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-Based Methods in Programming Environments, Vienna, Austria, 17–18 July 2014; pp. 109–123.
13. Wielemaker, J.; Costa, V.S. Portability of Prolog programs: Theory and case-studies. *CoRR* **2010**, arXiv:1009.3796.
14. Wielemaker, J.; Costa, V.S. On the Portability of Prolog Applications. In Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages , Austin, TX, USA, 24–25 January 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 69–83.
15. Bothe, K. A Prolog Space Benchmark Suite: A New Tool to Compare Prolog Implementations. *SIGPLAN Not.* **1990**, *25*, 54–60.
16. Haygood, R. *A Prolog Benchmark Suite for Aquarius*; Technical Report; University of California at Berkeley: Berkeley, CA, USA, 1989.
17. Bagnara, R. China—A Data-Flow Analyzer for CLP Languages. Available online: http://www.cs.unipr.it/China/ (accessed on 5 March 2017).

18. Liang, S.; Fodor, P.; Wan, H.; Kifer, M. OpenRuleBench: An Analysis of the Performance of Rule Engines. In Proceedings of the Internacional World Wide Web Conference, Madrid, Spain, 20–24 April 2009; ACM: New York, NY, USA, 2009; pp. 601–610.
19. Areias, M.; Rocha, R. On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs. *Theory Pract. Log. Program.* **2011**, *11*, 681–696.
20. Areias, M.; Rocha, R. On Extending a Linear Tabling Framework to Support Batched Scheduling. In Proceedings of the Symposium on Languages, Applications and Technologies, Braga, Portugal, 21–22 June 2012; Simões, A., Queirós, R., Cruz, D., Eds.; 2012; pp. 9–24.
21. Gonçalves, R.; Areias, M.; Rocha, R. Towards an Automated Test Bench Environment for Prolog Systems. In Proceedings of the Symposium on Languages, Applications and Technologies, Vila do Conde, Portugal, 26–27 June 2017.
22. Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A.D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M. A View of Cloud Computing. *Commun. ACM* **2010**, *53*, 50–58.
23. Smart, J.F. *Jenkins: The Definitive Guide*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2011.
24. Torvalds, L. Git—Version Control System. Available online: https://git-scm.com/ (accessed on 5 March 2017).
25. Colmerauer, A.; Kanoui, H.; Pasero, R.; Roussel, P. *Un système de Communication Homme-Machine en Français*; Technical Report cri 72-18; Groupe Intelligence Artificielle, Université Aix-Marseille II: Marseille, France, 1973.
26. Warren, D.H.D. *An Abstract Prolog Instruction Set*; Technical Note 309; SRI International: Menlo Park, CA, USA, 1983.
27. Zhou, N.F. Parameter Passing and Control Stack Management in Prolog Implementation Revisited. *ACM Trans. Program. Lang. Syst.* **1996**, *18*, 752–779.
28. Demoen, B.; Nguyen, P. So Many WAM Variations, So Little Time. In *Computational Logic*; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1861, pp. 1240–1254.
29. Roy, P.V. 1983–1993: The wonder years of sequential Prolog implementation. *J. Log. Program.* **1994**, *19*, 385–441.
30. Lloyd, J.W. *Foundations of Logic Programming*; Springer: Berlin/Heidelberg, Germany, 1987.
31. Chen, W.; Warren, D.S. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM* **1996**, *43*, 20–74.
32. Sagonas, K.; Swift, T. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Trans. Program. Lang. Syst.* **1998**, *20*, 586–634.
33. Zhou, N.F.; Shen, Y.D.; Yuan, L.Y.; You, J.H. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 109–123.
34. De Guzmán, P.C.; Carro, M.; Hermenegildo, M.V. Towards a Complete Scheme for Tabled Execution Based on Program Transformation. In Proceedings of the International Symposium on Practical Aspects of Declarative Languages, Savannah, GA, USA, 19–20 January 2009; Springer: Berlin/Heidelberg, Germany, 2009; pp. 224–238.
35. Rocha, R.; Silva, F.; Santos Costa, V. YapTab: A Tabling Engine Designed to Support Parallelism. In Proceedings of the Conference on Tabulation in Parsing and Deduction, Vigo, Spain, 19–21 September 2000; pp. 77–87.
36. Moura, P. *ISO/IEC DTR 13211-5:2007 Prolog Multi-Threading Predicates*; Universidade da Beira Interior: Covilhã, Portugal, 2008.
37. Areias, M.; Rocha, R. Towards Multi-Threaded Local Tabling Using a Common Table Space. *J. Theory Pract. Log. Program.* **2012**, *12*, 427–443.
38. Areias, M.; Rocha, R. On Scaling Dynamic Programming Problems with a Multithreaded Tabling System. *Journal of Systems and Software* **2017**, *125*, 417–426.