Design and Implementation of a Benchmark Testing Infrastructure for the DL System Racer

Atila Kaya, Keno Selzer
Software Systems Department
Technical University of Hamburg-Harburg
Hamburg, Germany
Email: {at.kaya | k.selzer}@tuhh.de

Abstract

We present an approach for developing an infrastructure to standardize the creation of benchmark tests for DL systems. We introduce a tool with a graphical user interface that supports and standardises the creation and execution of test benchmarks for the DL system Racer. We will discuss its functionality and implementation in detail.

1 Motivation

Parallel to the growing importance of description logics and its applications in the recent years, requirements from DL systems are rising continuously. Consequently, the Racer system [1] evolves frequently to support additional functionality. As in any other cyclic software development process, tracking the effects of changes on the system, is of major importance for the developers of Racer.

Besides the developers of the Racer system there are also other people who want to analyse and monitor the system continuously. With respect to their intentions, they can be categorised into three major groups:

- System users are mainly interested in getting to know the Racer system, especially in the exploration of Racer policies. This means that they want to try out different optimisation configurations and evaluate the system.

- Researchers of DL systems want to evaluate and compare the performance of different DL systems. They are interested in the analysis of the system's behaviour, when faced with complex inference tasks.

- Developers of DL systems are interested in the analysis of several aspects of a DL system. They test different functions of the system in order to find areas that need development.

All these groups need a methodology to measure the system's performance for certain functions of interest under defined circumstances. In the past, various test benchmarks suites in proprietary formats have been defined and used for the Racer system.

Typically, a test benchmark contains one or more knowledge bases (A-Boxes and T-Boxes), some queries to this knowledge bases, and some statements that configure or optimise the Racer Server. In addition, a test benchmark may contain some program code with conditionals and loops to create desired circumstances.

Analysis of existing test benchmarks has shown that all test benchmarks contain some common structures, even though they differ greatly in complexity and format. These repeated structures can be termed "benchmark patterns".

Simple benchmarks have the following pattern: First a knowledge base is loaded. Afterwards the loaded knowledge base is queried. More complex benchmarks additionally execute some optimisation operations on the knowledge base and requery it.

Many other (more sophisticated) benchmarks follow the same pattern reiteratively. This means that a sophisticated benchmark can be considered a collection of complex benchmarks that are executed consecutively. Generally the complexity of knowledge bases, queries, and optimization operations rises continuously within a sophisticated benchmark.

Considering different user groups, their goals, and the analysis of existing test benchmarks, requirements to a new benchmark testing infrastructure are these:

- Users without any specific programming language knowledge should be enabled and encouraged to create complex test benchmarks.

- Some researchers and developers of DL systems have test benchmarks suites they already use. Therefore, all existing benchmarks must be supported.

- The realised benchmark patterns should be easily representable and definable. Moreover the users should be supported in using the benchmark patterns.

- A new benchmark definition language, which is easy to use and programming language independent, should be developed. Moreover, this benchmark language should become DL system-independent.

In order to provide a benchmark testing infrastructure that fulfils these requirements, we decided to design and implement a tool with a graphical user interface. Besides fulfilling these requirements the tool should:

- Be platform-independent in order to run on different operating systems.

- Present the output of a benchmark execution visually or prepare the output of it in a format compatible with other visualization tools.

2 Technical Details

Different user profiles of the Racer system and their requirements motivated us to choose an object-oriented and platform-independent programming language, namely Java [2] for the implementation. We used the Java Swing library to develop the graphical user interface for the benchmark tool called Benchee.

It supports the execution of existing test benchmarks and the creation of new ones. First, we will present the features of Benchee briefly:

Benchee can manage Racer servers and execute test benchmarks on them. Test benchmarks created with Benchee can directly be executed with it. Besides the execution of new test benchmarks, Benchee can also execute existing test benchmarks that are not created with Benchee. However, these must be available either as an executable jar file for the Java platform or as an application program running on the operating system platform. Additionally, Benchee can manage Racer Servers that are necessary to execute these benchmarks. Racer Server instances managed by Benchee may be distributed on different physical locations. Furthermore, the instances may be different versions of the Racer system. This enables the execution of the same test benchmarks on different versions of Racer and supports the comparison of the results.

Benchee also can send commands entered by the user in the nRQL [3] language directly to a Racer Server. It has the ability to measure and display the time elapsed for the execution. Being able to send commands lets users to load a knowledge base, and send some statements or queries, without defining a test benchmark. This is helpful, when users try to find out what they want to test and how to properly define it as a benchmark.

Finally, users can create new test benchmarks with Benchee. By using the tool, users don’t need to write a program to define a complex test benchmark. Benchee stores the definition of a test benchmark in a benchee specific format. After the execution of a test benchmark, its results are stored in a special file that can be plotted by gnuplot. Gnuplot is a portable command-line driven interactive data file and function plotting utility for many operating systems [4].

We to present the features of Benchee in more detail and sketch the implementation:

2.1 Management and Execution of Benchmarks

The menu items *Benchmark* and *Racer* enable the management of benchmarks and Racer servers (see Figure 1). They let users add, edit or delete benchmarks and Racer servers. 2

To define a racer server, its version and the path to the server must be entered. Additional start options can also be defined. More information on possible start options can be found in the Racer manual [5].

A benchmark definition must contain at least a name and the path to the benchmark definition file. This file can be:

- a benchee specific file. (Benchmarks generated using Benchee have the file extension ".ben".)
- an executable jar file for the Java platform or an application program running on the operating system platform.

Moreover several optional parameters can be defined using the benchmark menu item. For more information please refer to the Benchee manual [6].

The *Benchmark* tab is used to start racer servers and execute benchmarks (see Figure 1).

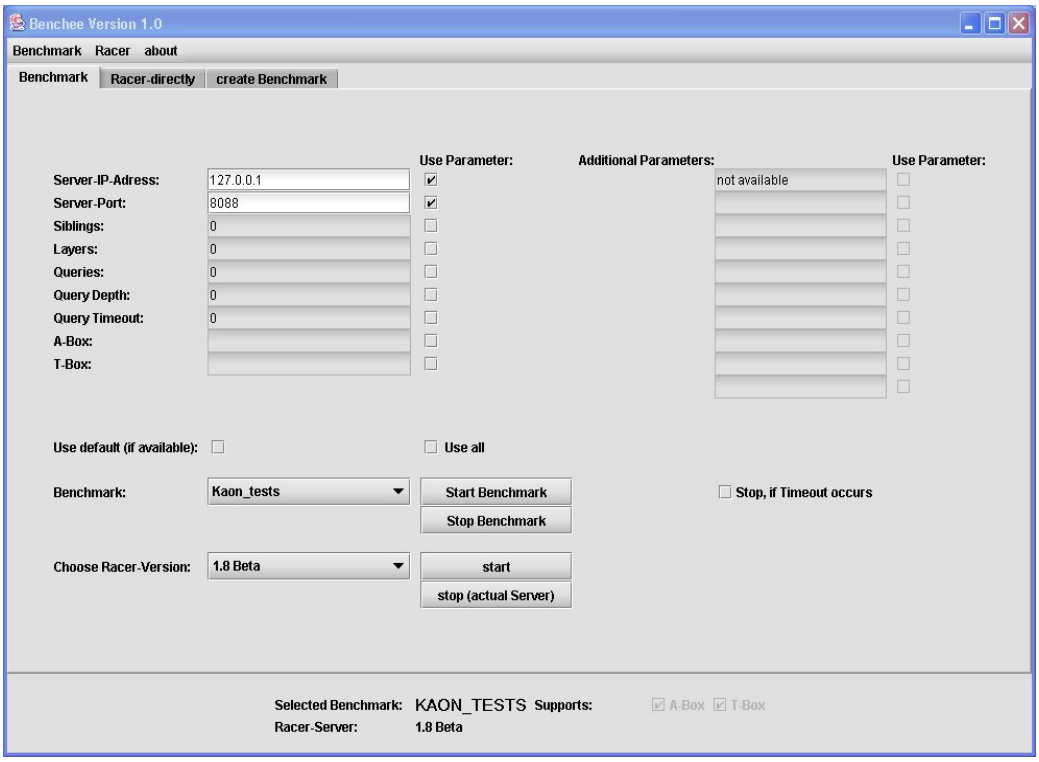

Figure 1: *Benchmark* tab of Benchee

In the *Benchmark* tab a Racer server can be selected, started, and stopped. After starting a Racer server a benchmark can be selected. Depending on the selected benchmark some additional parameters concerning the Racer server or the benchmark can be defined. Later a benchmark can be started. Normally a benchmark terminates automatically, but the user can use the *stop benchmark* button to terminate the execution manually. Moreover, the user may decide to select the option *stop, if timeout occurs*. This is useful if desired circumstances can not be guaranteed after a timeout. In such a situation, a reload of the underlying knowledge base is necessary and the execution of the benchmark must be terminated.

2.2 Direct Communication with Racer Server

The definition of a new test benchmark starts with an experimentation phase. The function of interest may deliver different results depending on the statements executed before. E.g. if a user requires the classification of a T-Box, Racer computes an index for the T-Box to answer queries. Queries send to this T-Box after the classification will be answered more quickly than queries sent before.

The *Racer-directly* tab offers users an easy way to find out the proper benchmark definition (see Figure 2).

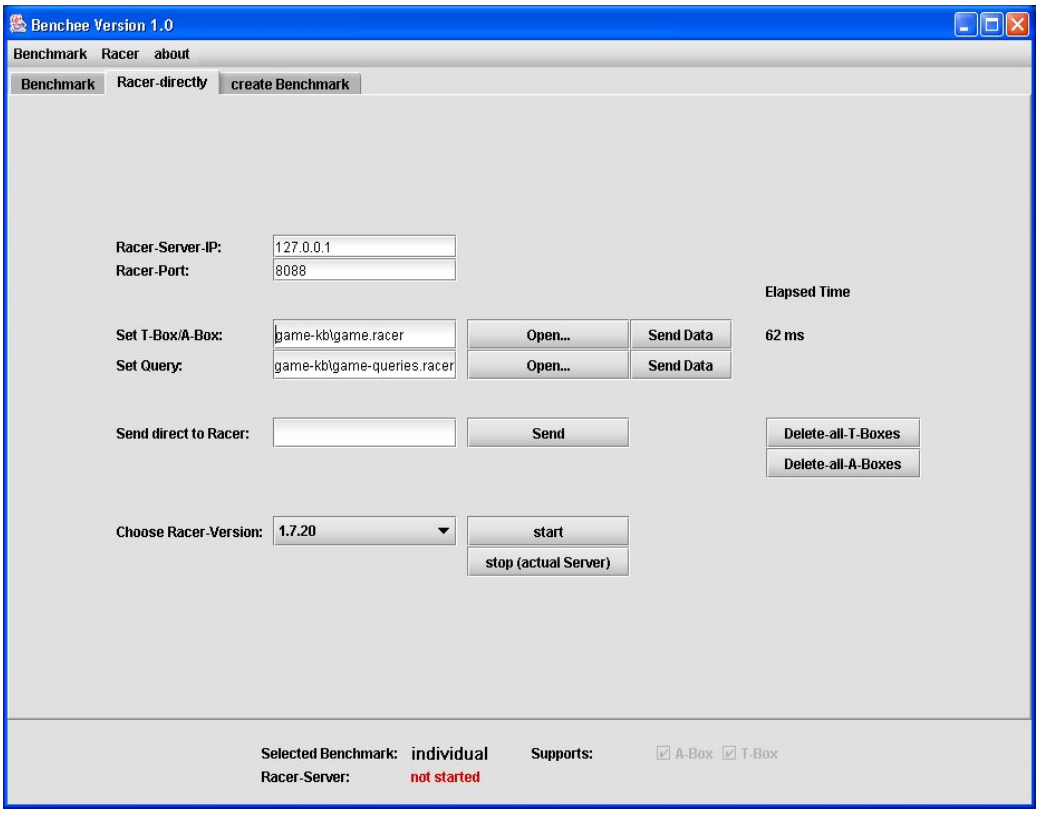

Figure 2: *Racer-directly* tab of Benchee

Typically a user will use this tab in the following way:

1. Send a T-Box or an A-Box to a Racer Server.
2. Send a collection of queries to the T-Box or A-Box.
3. Send a statement to configure the optimization policy.
4. Resend the same collection of queries.

The time elapsed for the execution of each operation sent to a Racer Server will be displayed on the *Racer-directly* tab of Benchee, so that the user can compare the results. Here, statements entered by the user and the displayed execution times are not saved in files as it is the case for test benchmarks.

2.3 Creation of Test Benchmarks

The main task of Benchee is to assist the creation of new test benchmarks. The *Create Benchmark* tab offers the functionality to define new benchmarks (see Figure 3).

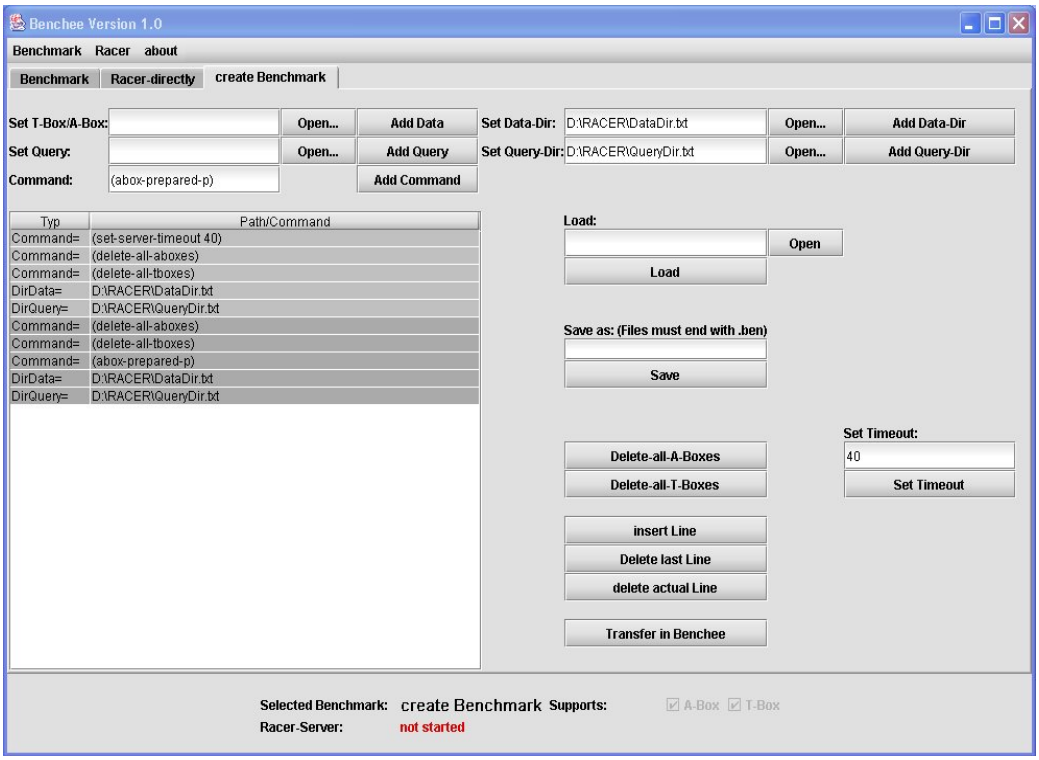

Figure 3: *Create Benchmark* tab of Benchee

Typically, a complex benchmark definition can be created by these functions:

- Using the “Set T-Box/A-Box” field a KB will be loaded.
- Using the “Set Query” field a file containing the queries will be loaded.
- Using the “Command” field a statement will be added.

Each time the *add data*, *query* or *command* button is pressed, the operation will be added to the benchmark definition window as a new line. Each line in this window is editable. The buttons *insert Line*, *delete last Line* and *delete current Line* are added to make editing more comfortable. Moreover, some frequently used commands are available as buttons too; *Delete-all-A-Boxes*, *Delete-all-T-Boxes* and *Set Timeout*.

As explained in the motivation section, sophisticated benchmarks can be considered as a collection of complex benchmarks that increase in the complexity of knowledge bases, queries and optimization operations used. To ease the definition of such benchmarks the *Add Data Dir* and *Add Query Dir* buttons were added to the *Create Benchmark* tab. Using these buttons two text files can be added to the benchmark definition. The data directory text file contains a list of data files (knowledge bases) and the query text file contains a list of query files. It is important that the number of files defined in the two files are equal, because for each data a corresponding query is necessary. As can be seen in Figure 3, the benchmark definition is represented in the benchmark definition window. Each occurence of the line *DirQuery* in the window indicates the beginning of another loop and has a different background colour. The second loop in Figure 3 uses the same data and query files. The only difference is that the query *abox-consistent-p* has to be executed before.

Regardless of the complexity and the method used to define it, a new benchmark displayed in the benchmark definition window can be saved into a file by giving a name and pressing the save button. This file is a special text file containing the benchmark definition and having the suffix "*.ben*". The load button is used to open and edit a benchmark file created with Benchee.

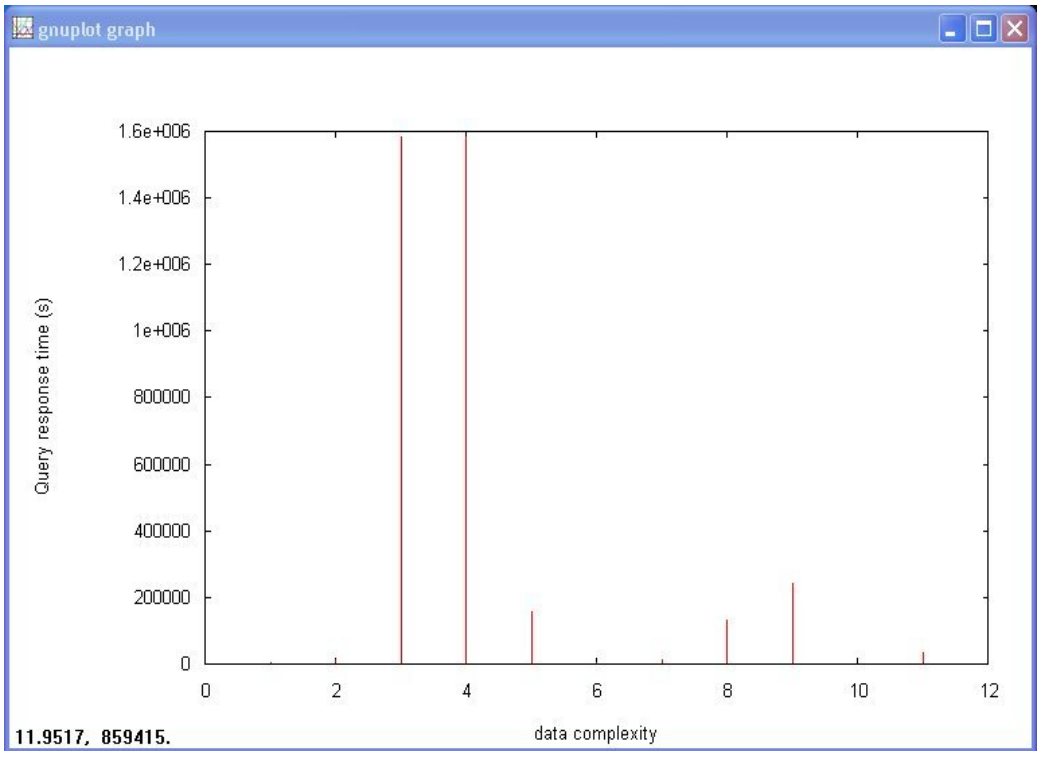

Figure 4: Visualization in gnuplot

The created benchmark can be added to the list of available benchmarks by clicking on the *Transfer in Benchee* button. After doing this, the benchmark can be started using the benchmark tab in Figure 1.

The results of the test benchmarks generated with Benchee are saved in a gnuplot compliant file. Additionally, every operation executed on the Racer server is logged in the same file. Using this file the results of a test benchmark can be visualized with gnuplot (see Figure 4). Experienced users can edit this file to adjust the visualization with gnuplot.

Benchmarks generated by other tools can also be managed and executed by Benchee. Benchee does not alter these, so that the output of these benchmarks remain unchanged.

3 Conclusion

In this paper we presented a benchmark-testing infrastructure that supports and standardises the creation and execution of test benchmarks for the DL system Racer.

The tool presented in this paper, named Benchee, is implemented in the widely used object oriented programming language Java to achieve platform independence. Benchee has been extensively tested and used with the Racer system.

Benchee can manage and execute existing test benchmarks without changing their definition, execution or result representation.

In addition, Benchee supports users in creating test benchmarks, without the necessity to learn a specific programming language.

When a test benchmark created in Benchee is executed, its results are saved in a gnuplot compliant file, so that they can easily be visualized.

The benchmark patterns discussed in the motivation section can easily be used in new test benchmarks created with Benchee. Moreover, through the functionality offered in Benchee's graphical user interface, users are encouraged to use these patterns in creating more sophisticated benchmarks.

With the work presented in this paper, we were able to develop an easy to use and programming language independent tool for maintaining and automating DL test benchmark collections for Racer.

Popular software development approaches such as Extreme Programming (XP) require automated regression testing. In XP unit tests build confidence that the code works correctly. Unit tests are written for any method that has a nontrivial implementation [7]. Even though goals and extents of unit tests differ widely from DL test benchmarks, and the tools used for XP do not support DL systems, automated test execution is also required from DL benchmark test tools. Therefore we want to enhance our tool with an interface for other

systems such as software agents. This will enable them to use our framework to automate benchmark test execution.

Although the benchmark patterns presented in this paper are independent of the DL system used, their implementations in test benchmarks are Racer specific. As a result Benchee is tailored for the benchmark testing of Racer. However, one of the motivations of this paper has been the comparison of different DL systems by using test benchmarks. This requires a common benchmark definition language interpretable by several DL systems. Therefore besides further analysis of benchmark patterns, we aim at enhancing Benchee to support a common benchmark definition language in the future.

References

[1] V. Haarslev and R. Möller, Description of the Racer system and its applications. In *International Workshop on Description Logics* (*DL-2001*), Stanford, August 2001

[2] Java Programming Language, http://java.sun.com

[3] V. Haarslev , R. Möller, R. V. D. Straeten, M. Wessel, Extended query facilities for Racer and an application to software-engineering problems. In the *International Workshop on Description Logics 2004* (*DL-2004*), Whistler, British Columbia, Canada, June 2004.

[4] Gnuplot plotting utility, http://www.gnuplot.info

[5] V. Haarslev and R. Möller, Racer User's Guide and Reference Manual, April 2004

[6] K. Selzer, Benchee User's Guide and Reference Manual, August 2004

[7] Eric M. Burke and Brian M. Coyner, Java Extreme Programming Cookbook. O'Reilly& Associates Inc, Marc 2003