Collaborative Benchmarking Rule-Reasoners with B-Runner

Federico Ulliana, Pierre Bisquert, Akira Charoensit, Renaud Colin, Florent Tornil, Quentin Yeche

Collaborative Benchmarking Rule-Reasoners with B-Runner

Federico Ulliana[1], Pierre Bisquert[2,1], Akira Charoensit[1],
Renaud Colin[2], Florent Tornil[1], Quentin Yeche[2]
[1]Inria, LIRMM, Univ Montpellier, CNRS, Montpellier, France
[2]INRAE, Montpellier, France

Abstract

Conducting experimental analysis on rule reasoners is a mainstream task for validating novel algorithms and systems. Nevertheless, providing robust, verifiable, and reproducible experiments can still raise a sensible challenge. We propose to demonstrate B-Runner, an open library for *collaborative benchmarking* focusing on the deployment of articulate tests for *knowledge and rule-based systems* with low cost and high robustness. B-Runner reduces the benchmarking setup time while guaranteeing experiment repeatability. At the same time, it improves the scrutability of experimental protocols thereby enhancing their robustness as well as fairness of system comparisons. This demonstration proposes to showcase the use of the tool for systematically testing a number of systems as well as to introduce its architecture and its extensibility to novel tools and experimental protocols.

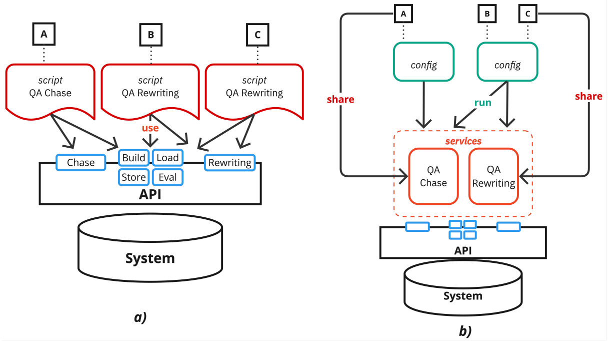


Figure 1. Benchmarking: *a*) Independent vs *b*) Collaborative.

1 The Benchmarking Issue

Scientific experiments are essential for validating and improving systems, but they come with numerous challenges. These can be classified depending on whether they are related to planning, reporting, or conducting experiments.

Planning consists in writing down all design choices for the experiments. Notably, the benchmarks to use, the competitors to consider, the testing environment (hardware/software) and the target measures.

Reporting includes retrieving and analyzing the results, then choosing the most meaningful data as well as the most informative and succinct graphical representations.

Conducting consists in faithfully implementing what has been planned. This includes setting up the test environment, coding (and verifying) the experimental protocols, and then running the tests while handling potential failures.

While all phases of experimental analysis can cause issues, the experiment conduction is perhaps the least accessible part. Accessibility here is intended as the *time and effort it takes a user to conduct a robust experimental analysis.* Of course, with high accessibility experimental analysis can be deeper, which directly translates to more thoughtful validation of novel approaches.

Conducting is also intimately related to *repeatability*. Repeatability means that another user equipped with the appropriate software and hardware can repeat the same experiments. Repeatability is easily attained when experiment conduction is simple, reliable, and documented. Conversely, it may be a burden to prepare a test suite and write sophisticated instructions on how to configure and run a benchmark starting from ad-hoc scripts, not to mention debugging them to make them portable on different systems. Experiment repeatability has become increasingly relevant during the last decades. Nowadays many conferences consider the availability of test artifacts in the review of scientific work, and have established reproducibility[1] tracks.

1.1 Independent vs Collaborative Benchmarking

In this work, we argue that the underlying issue with benchmarking conduction and repeatability lies in the lack of a platform for *collaborative benchmarking*. Indeed, the most common benchmarking scenario is one where activities are conducted *independently*, which in practice results in a plethora of hard-to-maintain scripts and ad hoc methods for running tests. While this consideration may apply to many fields, we stress that our focus is on the case of benchmarking *knowledge and rule-based reasoners* for which we introduce a dedicated collaborative benchmarking library.

Independent Benchmarking. Figure 1.a) illustrates a set of users (here A, B, C) conducting an experimental analysis to understand the performances of a given reasoning system (which may or may not have been coded by one of the users). The reasoning system supports query answering (QA) tasks on a knowledge base. In particular, users B and C want to

[1] reproducibility is stricter than repeatability: not only it ensures that the experiment can be re-run, but that it also yields the same result.

test the performances of QA via query rewriting while user A is interested in the performances of QA via the chase.

Let us explain in detail this scenario. A knowledge base (KB) can be seen as the extension of a database (also referred to as a *factbase*) with a set of rules modelling semantic constraints on the data. These semantic constraints can be used to both enrich and ensure the consistency of the data [5]. The reasoner can be commanded through an API. The API also offers a number of advanced features. For instance, *loading* and *building* the knowledge base starting from plain data and/or data-integration mappings [9], and rules. But also, deploying a certain type of internal *storage* for the data (graph, relational, triplestore, in-memory/disk, etc.). And finally, being able to *evaluate* queries on a factbase (without considering the rules). The API also includes a number of advanced facilities. This includes techniques which accounts for rules using *saturation* (also called *chase*) and *query rewriting*. The chase approach essentially consists at extending the KB data (i.e., the factbase) with the result of the application of rules [4]. Query rewriting in contrast compiles the rules into the input query thereby yielding a *reformulated* query which provides the same answers (as the input rules and query) on any database [8]. Query answering via the chase consists in evaluating the input query on the saturated knowledge base. Query answering via rewriting consists in evaluating the reformulated query on the input data.[2]

As Figure 1.a) shows, to conduct the test, every user writes a script coding an experimental protocol by leveraging on the API. Typically, this is done *from scratch*. Firstly, this is *time consuming*, as it requires learning the API and internals of the system as well as coding the measurements for the API calls. Secondly, this is *error prone*. Indeed, since scripts are *independently* written, for more complex tests it is not uncommon for the test protocols to even diverge - with an impact on measures.[3] All of this makes that setting up an experiment can have a significant cost and compromised robustness, resulting in low accessibility.

Collaborative Benchmarking. A principled solution to this issue is to provide users with a platform for collaborative-benchmarking, which simplifies the conduction of extended test trials. Collaborative benchmarking captures the idea that robust and repeatable experimental analysis can be achieved if *a community of users work together* on *i*) consolidating a set of *test protocols* which *ii*) can run on a *reliable framework* and *iii*) without *unnecessary complexity* at the user level. Figure 1.b) illustrates the case for collaborative benchmarking. Again, we consider users B and C testing QA via rewriting and user A testing QA via the chase. The first characteristic of this approach is that it allows users to share *benchmarkable services* (or simply services) coded in a common programming language.[4] Every such service implements a self-contained *testing protocol* which allows one to measure the performances (time, throughput, etc.) of a certain task. This test protocol is meant to be deployed on a variety of input scenarios and algorithm configurations - and not on a single one. By definition, sharing test implementations can benefit from code reviews in a collaborative environment, thus increasing the robustness of the experimental analysis and avoiding the divergence of testing protocols. Figure 1.b) illustrates user A sharing a service for running QA via the chase, while user C shares a service for QA via rewriting. By fostering test reuse, user B can perform an experimental analysis without coding by reusing the service shared by C. This results in significant time savings. Besides test sharing, another key characteristic of this approach is that test trials are specified by users via *configurations*. This makes the specification of test trials more *declarative* than programmatic, and hence independent from any programming languages. In this aspect, collaborative benchmarking is strongly opposed to independent benchmarking, the latter allowing each script to be potentially written in a different language. Crucially, configuration files can also be shared, as illustrated for users B and C. Not only does this save time, but it allows users to *communicate* and *document* the content and aim of their experiments in a more standard and intelligible way.

1.2 Novelty and Contributions.

We propose to demonstrate B-Runner: *a Java tool for collaborative benchmarking on knowledge and rule-based reasoners*.[5] The distinctive elements of the tool are the following.

1) Collaborative B-Runner allows users to share and reuse test protocols and configurations. Collaborative benchmarking has not been considered so far for rule-based reasoners. To the best of our knowledge, B-Runner is the only tool implementing this approach for rule-based reasoners.

2) Simple and Robust B-Runner allows to define trial specifications through declarative configurations which require minimal coding and learning of test systems. B-Runner proposes a design pattern for writing test protocols that favors their scrutability. The execution of tests is controlled via the Java Microbenchmarking Harness (JMH) library. This favors converging towards robust error-free testing protocols and measures.

3) Extensible and Portable. B-Runner's generic architecture allows to easily include novel systems and testing protocols. B-Runner is Java-based, which makes it portable, yet still able to support benchmarking of non-Java systems.

[2] To illustrate, let us consider the factbase $F = \{Prof(Alice)\}$ the rule $\forall x.Prof(x) \rightarrow Faculty(x)$ and the query $Q = \exists y.Faculty(y)$. The chase will yields $\{Prof(Alice), Faculty(Alice)\}$ on which Q answers true. Query rewriting will yield $\exists y.Faculty(y) \vee Prof(y)$ which answers true on F.

[3] typical errors are including/omitting optimizations in the tool, parsing time, writing on disk or standard output (logging, result export), improper use of cold/warm measures, misplaced timers or timeouts.

[4] While in principle services can be written using different languages, a single language very much improves protocol readability.

[5] B-Runner is an open source tool available at gitlab.inria.fr/rules/brunner.

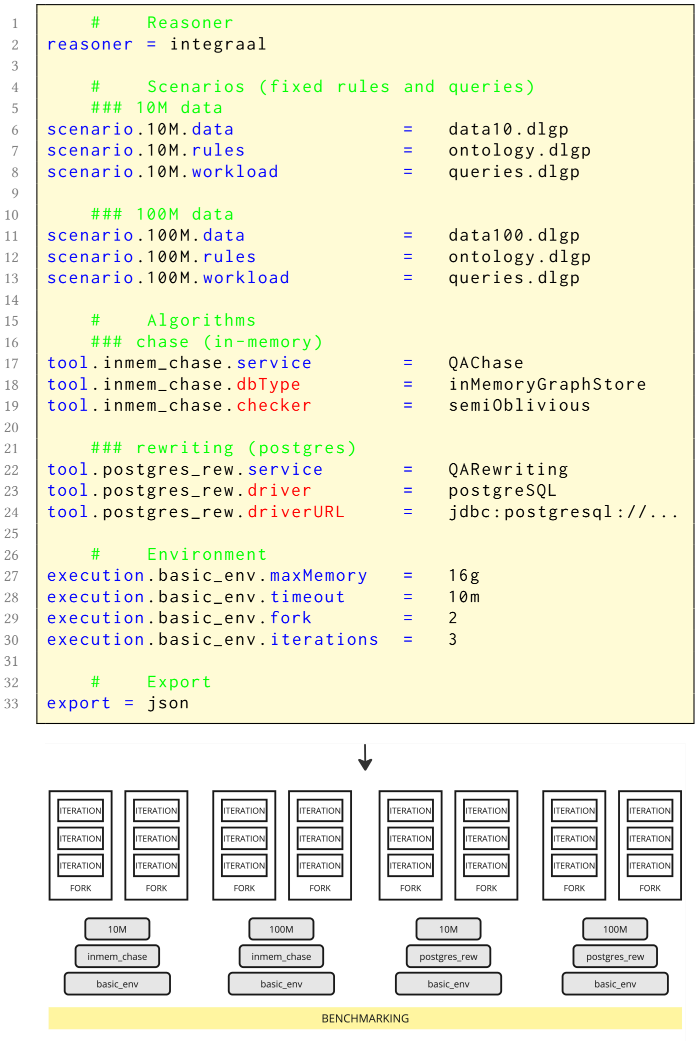
Related Work. Collaborative benchmarking is important in the context of *scientific workflows* [7]. These have been introduced for the reproducibility and automation of large scale scientific computations in domains such as genomics, biology, and astronomy [7]. A scientific workflow is expressed as a directed graph whose nodes are computations and edges dependencies. Research work in this area concerns the design of languages for workflow specification, as well as the reproducibility of experiments (or part of experiments) involving large data masses. The optimization of scheduling computations across different distributed and cloud platforms has also been considered, with the goal to obtain results faster and with less cloud computation [7].

User Groups. B-Runner can be used by two types of users: *testers* and *providers*. Testers define experiments from available services using configuration files (for instance, user B in Figure 1.b). Providers share benchmarking services (for instance, users A and C in Figure 1.b that share services for testing respectively KB chase and QA chase). In the remainder of this paper, Section 2 presents how a tester can declare a benchmarking activity through a configuration file, and how B-Runner responds to it. Section 3 shows what needs to be done on the provider side to offer a benchmarking protocol. Section 4 delves into features and limits of B-Runner. Section 5 outlines the scenarios proposed for this demonstration.

2 Benchmarking in a Hurry!

The first distinctive feature of B-Runner is the possibility of conducting experiments at small cost for *tester* users. Let us introduce the benchmarking activities we consider. A benchmarking activity B is a sequence of service executions $B = (e_1, \ldots, e_k)$. Every service execution is a triple $e = (s, c, r)$ where s is a service, c is a configuration for the service, and r the repetition parameters. A configuration $c = (n, a, d)$ is a combination of an execution environment n on top of which the service s is executed by taking as input an algorithm configuration a and an input scenario d. The repetition parameter is a couple $r = (f, i)$ where f is the number of forks and i the number of the iterations for the execution of s given c. Figure 2 illustrates a benchmarking activity made of four service executions (with their corresponding configurations and repetitions parameters).

The goal of a *fork* is twofold. Firstly, it creates a "cold" execution environment. Secondly, it sets up the reasoning system for running a number of repetitions (of the same service). An *iteration* takes places within a fork, and consists in the actual execution of the service itself. It is worth pointing out that every fork in B-Runner triggers the creation of a new Java Virtual Machine (JVM) where the test runs. More specifically, this is accomplished by leveraging on Java Microbenchmark Harness (JMH), and happens to be useful to eliminate measure bias (due to cache, Just-In-Time compilation, etc.).



```
 1      #    Reasoner
 2  reasoner = integraal
 3
 4      #    Scenarios (fixed rules and queries)
 5      ### 10M data
 6  scenario.10M.data               =    data10.dlgp
 7  scenario.10M.rules              =    ontology.dlgp
 8  scenario.10M.workload           =    queries.dlgp
 9
10      ### 100M data
11  scenario.100M.data              =    data100.dlgp
12  scenario.100M.rules             =    ontology.dlgp
13  scenario.100M.workload          =    queries.dlgp
14
15      #    Algorithms
16      ### chase (in-memory)
17  tool.inmem_chase.service        =    QAChase
18  tool.inmem_chase.dbType         =    inMemoryGraphStore
19  tool.inmem_chase.checker        =    semiOblivious
20
21      ### rewriting (postgres)
22  tool.postgres_rew.service       =    QARewriting
23  tool.postgres_rew.driver        =    postgreSQL
24  tool.postgres_rew.driverURL     =    jdbc:postgresql://...
25
26      #    Environment
27  execution.basic_env.maxMemory   =    16g
28  execution.basic_env.timeout     =    10m
29  execution.basic_env.fork        =    2
30  execution.basic_env.iterations  =    3
31
32      #    Export
33  export = json
```

Figure 2. Trial Configuration Example

Figure 2 illustrates a configuration written by a tester and inspired by the example given in the introduction (Figure 1).[6] In the example, reserved keywords for B-Runner are highlighted in blue. As indicated by line 2, the configuration applies to the InteGraal reasoner [3].[7] Keywords proper to the configuration of InteGraal are highlighted in red. The configuration then continues with the specification of two test scenarios "10M" (lines 7-9) and "100M" (lines 12-14). To declare different datasets, we use the reserved keyword `scenario`. Both scenarios consider the same ontology-rules

[6] The syntax represents a collection of properties of the form *name* = *value* where *name* is a sequence of keys of the form $a.b.c$ and *value* is a string. This notation is used in our case as a notation for *trees*. For instance the set of properties $\{a.d = 1, a.e = 2\}$ can be equivalently expressed as an XML tree <a><b>1</b><c>2</c></a> or a JSON record {a:{b:1,c:2}}.

[7] In B-Runner, every configuration file can be associated to a single reasoner. Although in principle multiple reasoners could be managed from the same configuration, it is unlikely that their parameters would be the same. Hence, a good practice is to keep configurations for different reasoners separated.

```
1  public void serviceOperations() {
2    setup(DATA_LOADING,    this::setData);
3    setup(RULE_LOADING,    this::setRules);
4
5    operation(QUERY_LOADING,     this::setQuery);
6    operation(BUILD_REWRITER,    this::buildRewriter);
7    operation(QUERY_REWRITING,   this::rewrite);
8
9    operation(BUILD_EVALUATOR,   this::buildEvaluator);
10   operation(QUERY_EVALUATION,  this::evalRewriting);
11  }
```

Figure 3. Protocol for Rewriting-based Query Answering

and queries but differ in terms of data (here, .dlgp denotes the Datalog-Plus language [2]. Then, two different algorithmic strategies for query answering are used. This is done with properties prefixed by the keyword `tool`. The first is via the chase procedure (line 17). Note also that the data is stored in a native InteGraal in-memory graph store (line 18) and that a specific variant of the chase, called *semi-oblivious* [5] is chosen (line 19). Similarly, a second strategy based on rewriting is defined (line 22). In this case, note that the data is stored on a (local or remote) PosgtreSQL server (line 23) which is accessible via the given connection URL (line 24). These options illustrates the richness and simplicity of the algorithm configuration that can be achieved. Finally, the execution environment for the tests is set. This is done to properties prefixed by the keyword `execution`. Options here include setting the maximum JVM size to 16GB (line 27), setting the timeout for the execution of a *single* iteration to 10 minutes (line 28), and setting the number of forks and repetitions (lines 29-31). The concluding command (line 33) concerns the export of the results in JSON.

From this specification file, B-Runner automatically conducts a benchmarking activity on a sequence of configurations generated by combining *every* input scenario with *every* algorithm and *every* environment configuration. The resulting benchmarking activity is illustrated in Figure 2. Selective test execution is also possible in B-Runner [1].

3 Sharing a New Benchmarkable Service

The second distinctive feature of B-Runner is the possibility of sharing new testing protocol for *provider* users. In contrast with configuration files, protocols must be *programmatically written.* B-Runner adopts Java, yet the methodology we present is transposable to other programming languages.

In short, B-Runner proposes a design pattern for implementing protocols which aims at making a *service equivalent to an array of method references.* To illustrate, Figure 3 shows a testing protocol for query answering via query rewriting inspired by the example given in the introduction (Figure 1). The service measures the overall time it takes to answer a query by using this technique, and also provides details for each step of the task. The B-Runner API offers the possibility of specifying new testing protocols by simply implementing the `serviceOperations` method. In turn, this uses two methods for specifying the test steps. The first is the setup method, which corresponds to a step that has to be performed *for every fork.* The second is the operation method, which corresponds to a step that has to be performed *for every iteration.* As Figure 3 illustrates, typical examples of setup steps include the loading of the scenario, notably data and rules which are considered fixed (lines 2-3). Then, operation steps include creating objects and executing query rewriting (lines 5-7), followed by evaluation (lines 9-10). Note that describing a step via setup and operation using the design-pattern proposed by B-Runner systematically requires two inputs. The first is a description of the step, which is instrumental to associate a measure to an operation. The second is a *method reference.* E.g., this::setData refers to the setData() method of the protocol class. Each method is then meant to include a compound block of instructions. Making a service equivalent to an array of method references results in code which is free from ad-hoc instructions for measuring (as this code can be factorized). We understand that these are mostly engineering aspects, but we believe that these are crucial for achieving a collection of readable protocols. For space reasons, we refer to [1] for details on B-Runner architecture.

4 Effective Experiment Conduction

We conclude the presentation of B-Runner by emphasizing two key aspects of the tool: the outputs and its limitations.

Interpreting Benchmark Results. The goal of benchmarking is to yield data to be analyzed. Yet, monitoring test execution must not be underestimated.

Benchmarking progress as well as errors must be visible. Tests can take a long time (minutes, hours, days, or weeks) and users need to be able to check the state of their evaluation. Tests can also *fail* during benchmarking. This can be due to the inputs, the tool, the execution environment, the network, timeouts, etc. In all of these cases, users need to see meaningful errors in order to correct and eventually restart them. It is important to clearly distinguish failures from timeouts. While providing meaningful errors requires a bit more effort in coding the experimental protocol, we argue that providing context for failures, and their nature, is an invaluable help. In B-Runner, benchmarking progress and errors are provided both via logging and on the test results.

Benchmarking results must come in a structured format. Results need to be automatically exploitable, in an easy way. Specifically, a set of console lines may not always be handy. B-Runner exports benchmarking results in both JSON and XML, and other formats can be easily added. Experimental results must also come with precise context about where the measure was made (scenario, algorithm, environment, operation). The availability of structured data helps with setting up routines for data aggregation, which saves time and benefits consistency of data processing. Also, a rich context helps users spotting anomalies in the data.

Recognizing and Overcoming Limitations. While B-Runner provides a flexible and extensible setting to automate some extended testing scenarios, it also has some limits which are important to keep in mind. We will discuss three limitations that mainly pertain to the use of Java, and ways to circumvent them.

Monitoring Memory. This is a feature that B-Runner does not currently handle but which is planned for future releases. The strategy we plan to pursue for this consists at using the JMH library facilities which allows one to run a built-in profiler, add an external profiler, or a custom profiler to report metrics - notably memory usage at the desired level.

The API Wall. The precision of a measurement is directly proportional to the richness of the API of the tested reasoner. Being a Java tool, B-Runner is naturally more at ease with benchmarking Java APIs. That being said, it is important to note that it can still be used for tools implemented in other languages that can be commanded either via Java Native Interface (JNI) or system calls.

Consider then an API which exposes only high level features such as a compound operation including both data loading and query evaluation. In this case, it could become more difficult to measure the time taken by the two operations separately. Nevertheless, it should be noted that, by definition, this situation would be impacting *all* experimental analysis involving such API. Therefore, it is still very useful to point out the issue and find an alternative way of measuring the two subtask in a uniform way for all testers - like by retrieving information provided by the tested tool.

Comparing Versions. It is sometimes desirable to compare the performance of a tool to one of its previous versions. This raises the risk of creating a conflict of Java dependencies between the versions. To avoid such a situation, a good practice consists in *i*) making sure that every release of a reasoner under test has its dedicated B-Rrunner project as well as *ii*) performing two different executions of the benchmarking activity; every execution must include in the Java classpath only one of the version to compare. Such a workflow for comparing versions can furthermore be automatized.

5 Demonstration Scenario

During the demonstration, we will show how to use our tool for multiple types of tasks. First, we focus on the definition of tests using configurations. This will showcase the simplicity of the tool for defining extended test trials. Second is the extension of the tool with new services. This will showcase the simplicity of the tool for defining new test protocols. Finally, we will outline the logging features provided by the tool which are helpful for monitoring test progress, failures, as well as the test result exports. We will focus on testing scenarios by using InteGraal [3] and Rulewerk/VLog [6] on standard benchmarks [5]. Details on use cases will be available at gitlab.inria.fr/rules-demo/brunner-demo.

6 Conclusion and Outline

Principled benchmarking paradigms can be instrumental to perform robust experimental analysis at small cost. *We advocate for the development of collaborative benchmarking for knowledge and rule-based reasoners.* As an answer to this need, we introduced B-Runner, an open library for testing rule-based reasoners. B-Runner leverages on Java Microbenchmarking Harnessing (JMH) for experiment conduction. B-Runner is Java-based - which makes it portable - yet still able to support non-Java systems.

Our library implements an architecture for *collaborative benchmarking* which is service-oriented and configuration-driven. We argue that sharing reusable services dramatically reduces experiment setup thereby increasing their reproducibility. More generally, this provides testing accessibility to larger audiences. Also, making test protocols transparent through a simple yet well-defined design pattern enhances their robustness and scrutability. Crucially, this can contribute to increase the fairness of comparisons when these span across different tools. Using a configuration-based approach - as opposed to scripting - constitutes a simple way to plan extended benchmarking activities, which also improves readability. Finally, while our work focuses on reasoning systems, these principles can be applied to other contexts.

B-Runner is available online [1]. The tool supports a number of reasoners and is currently under active development. Future work also involves the creation of a library for automatic chart creation from trial results.

References

[1] 2024. B-Runner Repository. gitlab.inria.fr/rules/brunner.

[2] Jean-François Baget, Alain Gutierrez, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Datalog+, RuleML and OWL 2: Formats and Translations for Existential Rules. In *RuleML*.

[3] Jean-François Baget, Pierre Bisquert, Michel Leclère, Marie-Laure Mugnier, Guillaume Pérution-Kihli, Florent Tornil, and Federico Ulliana. 2023. InteGraal: a Tool for Data-Integration and Reasoning on Heterogeneous and Federated Sources. Repository gitlab.inria.fr/rules/integraal. In *BDA 2023*. Montpellier, France.

[4] Catriel Beeri and Moshe Y. Vardi. 1984. A Proof Procedure for Data Dependencies. *J. ACM* (1984).

[5] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *PODS*.

[6] David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch, and Jacopo Urbani. 2019. Vlog: A rule engine for knowledge graphs. Repository github.com/knowsys/rulewerk. In *ISWC*.

[7] Sarah Cohen-Boulakia, Khalid Belhajjame, Olivier Collin, Jérôme Chopard, Christine Froidevaux, Alban Gaignard, Konrad Hinsen, Pierre Larmande, Yvan Le Bras, Frédéric Lemoine, Fabien Mareuil, Hervé Ménager, Christophe Pradal, and Christophe Blanchet. 2017. Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems* 75 (2017), 284–298. https://doi.org/10.1016/j.future.2017.01.012

[8] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. 2015. Sound, complete and minimal UCQ-rewriting for existential rules. *Semantic Web* (2015).

[9] Maurizio Lenzerini. 2002. Data integration: A theoretical perspective.