Using SWISH to realise interactive web based tutorials for logic based languages

JAN WIELEMAKER
Centrum Wiskunde & Informatica, Amsterdam, Netherlands
(*e-mail:* J.Wielemaker@cwi.nl)

FABRIZIO RIGUZZI
(*e-mail:* fabrizio.riguzzi@unife.it)

BOB KOWALSKI
Imperial College, London
(*e-mail:* r.kowalski@imperial.ac.uk)

TORBJÖRN LAGER
University of Gothenburg
(*e-mail:* lager@ling.gu.se)

FARIBA SADRI
Imperial College, London
(*e-mail:* fs@doc.ic.ac.uk)

MIGUEL CALEJO
Logical Contracts
(*e-mail:* mc@logicalcontracts.com)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Programming environments have evolved from purely text based to using graphical user interfaces, and now we see a move towards web based interfaces, such as Jupyter. Web based interfaces allow for the creation of interactive documents that consist of text and programs, as well as their output. The output can be rendered using web technology as, e.g., text, tables, charts or graphs. This approach is particularly suitable for capturing data analysis workflows and creating interactive educational material. This article describes SWISH, a web front-end for Prolog that consists of a web server implemented in SWI-Prolog and a client web application written in JavaScript. SWISH provides a web server where multiple users can manipulate and run the same material, and it can be adapted to support Prolog extensions. In this paper we describe the architecture of SWISH, and describe two case studies of extensions of Prolog, namely Probabilistic Logic Programming (PLP) and Logic Production System (LPS), which have used SWISH to provide tutorial sites.

KEYWORDS: Prolog, logic programming system, notebook interface, web

1 Introduction

Interactive languages such as Prolog traditionally provide a *console* to which one types commands (queries) and where the output (answers) appear in textual form. The programming environment consists of this interactive session, a code editor and optionally tools for cross-referencing, profiling, testing, etc. These may be integrated in a single front end called an *Integrated Development Environment* (IDE) such as Emacs, Eclipse or Visual Studio.

Literate programming combines text about a program with the program itself in a single document. This field was pioneered by Knuth (1984), who created high quality documentation and a Pascal program for TEX from a single document. Note that in Knuth's vision the text was more design documentation than merely simple comments. For program development purposes literate programming evolved into generating program documentation from source code based on properties of the code itself such as class names or function names and *structured comments*. Structured comments use the comment syntax of the target language, avoiding the need for special programs to extract the program from the document. Well known examples of this approach are the JavaDoc and Doxygen.

Notably for domains where programs are used to perform calculations on raw (experimental) data and produce derived data and charts, the traditional literate programming approach has been extended to include the result of the program such as a table or chart in the document. The first interactive version of this approach, called *notebook interfaces*, *computational notebooks* or *data science notebooks* was introduced in 1988 by Mathematica 1.0 on the Macintosh.[1]

Modern web technologies such as HTML5, CSS3 and JavaScript allow for web based versions of notebooks, simplifying their deployment. A good example of this is Jupyter.[2] The notebook interface paradigm is claimed (Worlfram 2016) to be particularly suited to improve reproducibility in e.g., data science workflows and to provide interactive educational material.

The benefits of interactive web based development environments and notebooks for capturing workflows and education also apply to Prolog. How should Prolog be part of these developments? There are two obvious options: (1) provide support in an existing language independent environment such as Jupyter or (2) develop a dedicated environment for Prolog and related languages. Prolog has a number of distinctive features that are hard to support in systems that are designed for imperative object oriented languages. First, the program-is-data feature, combined with the absence of keywords complicates syntax highlighting, completion and explanations when *hovering*[3], which are found in modern editors and expected by today's users. Second, non-determinism allows the generation of multiple answers for one query, which requires a different query/answer interface. Third, a large subset of queries is side effect free, which allows multiple users to use a single server through

[1] `https://en.wikipedia.org/wiki/Notebook_interface`
[2] `http://jupyter.org/`
[3] Positioning the mouse cursor over an area.

a web based interface. These observations motivated us to explore the development of a pure Prolog oriented web based programming environment.

This article discusses SWISH,[4] an acronym for *SWI*-Prolog for *SH*aring or *SWI*-Prolog *SH*ell. SWISH consists of a web server written in SWI-Prolog and a client application written in JavaScript. In this sense the design is similar to that of Jupyter. SWISH however is designed as a multi-user notebook that supports Prolog's distinctive features and can be extended to facilitate extensions of the Prolog language such as CHR, PLP and LPS.

This article describes the architecture of SWISH, how it is being deployed to support Prolog education and how it has been extended to support PLP and LPS. The main contributions of this article are:

- Provide a detailed description of a novel web-based Prolog environment for data analysis and education.
- Describe the potential of web-based Prolog. Notably demonstrate how SWISH can be used to provide tutorial environments for derived languages. The paper discusses two such environments: `cplint` on SWISH (Section 4.1) for PLP and LPS on SWISH (Section 4.2) for LPS.
- Resolve technical challenges involved in running a web based IDE over low-bandwidth networks with high latency. This notably poses challenges for server-assisted semantic syntax highlighting.

This article is organised as follows. After this introduction we provide an overview of the functionality provided by SWISH in Section 2. In Section 3 we elaborate on education support. Section 4 discusses how SWISH can be used to provide a tutorial site for two Prolog derived languages and is followed by a description of the overall architecture (Section 5) and details about some of its critical components. Section 6 presents related work and Section 7 discusses the impact, lessons learned and portability. We then conclude and present future directions.

2 The SWISH application

Before going into a technical description of the architecture in Section 5 we present the functionality of the key components of SWISH as it is exposed to the user. SWISH has two 'faces'. The first is an IDE 'face' where it shows a program editor, a query editor and query answers (see Figure 1). This is the original interface and it is the most suitable for program development and testing. Programs created in this way may be saved and reused in other programs or notebooks using the standard Prolog **include/1** directive (Section 5.4). The second 'face' is the notebook interface described in Section 2.3. The notebook interface is particularly suitable for educational applications and recording data science workflows. First we describe the components of the IDE interface. The notebook interface reuses these components.

[4] `https://swish.swi-prolog.org`

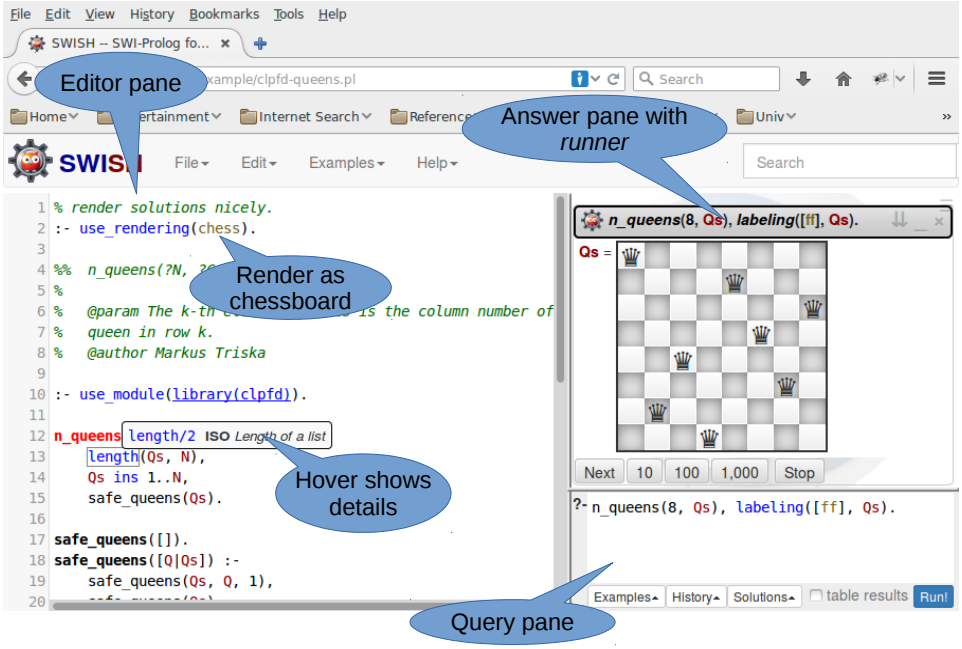

Fig. 1. Screendump of SWISH in IDE mode. The left pane holds the source code editor, while the top-right pane holds a query *runner* that employs the current selected answer *renderer* and buttons for continuing after the first answer. The bottom-right pane holds the query editor with access to example queries stored in the source, query history, solution modifiers, result presentation and a Run! button to start the query.

2.1 The program and query editor

The *program editor* uses CodeMirror,[5] a JavaScript text and code editor that we extended with a Prolog mode. It provides semantic highlighting, template insertion and tooltips providing summary information from the manual (Section 5.3) and access to the SWISH infrastructure for loading, storing and versioning of documents (Section 5.4).

The *query editor* is based on the same jQuery plugin that realises the code editor and thus profits from syntax highlighting, template insertion and tooltips. In addition, it provides three popup menus:

Examples This menu is filled from structured comments in the program. Such a comment is formatted as below:

```
/** <examples>
?- reverse([a,b], L).
?- reverse([a,b], [b,a]).
*/
```

The examples menu provides a menu entry for adding the current query to the

[5] https://codemirror.net/

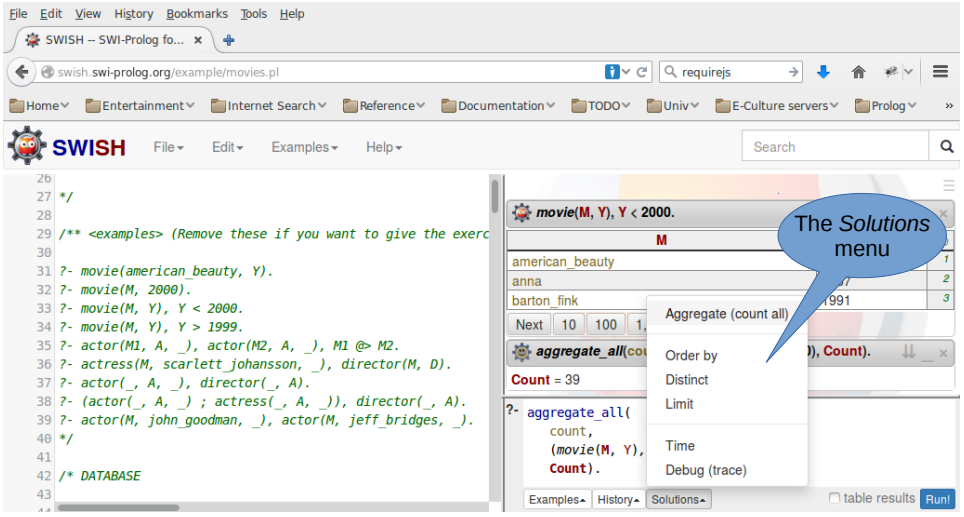

Fig. 2. The Solutions menu can be used to count results, order them, filter duplicates, etc. The upper runner shows answers to the query as a table.

program, either by creating a structured comment block as above or adding the query to an existing block.

History This menu provides access to previously executed queries.

Solutions This menu embeds an existing query in a meta-call to alter the result. Currently provided operations are *Aggregate (count all)*, *Order by*, *Distinct*, *Limit*, *Time* and *Debug (trace)*. Figure 2 shows how the menu is used to count the number of solutions of a goal.

SWISH allows the user to save the program in the code editor. The user can assign the program a meaningful name or accept a randomly created name. Once a program is saved, it is associated to an URL of the form

```
https://swish.swi-prolog.org/p/<name>
```

that is persistent: it can be used to retrieve the program later and to share it with other users. Saved programs are versioned using technology inspired by the GIT SCM,[6] where documents (programs) and their meta-data (author, time, relation to previous version etc.) are identified by a cryptographic hash of the data. This approach guarantees integrity of the data and simplifies concurrent and distributed management of versions. The versioning system allows for using previous versions as well as examining changes. The user can log into the system: in this case the identity of the user is associated with the saved program.

Each program is associated with a chat room where users can discuss the current

[6] `https://git-scm.com/`

program. SWISH has a general chat room for discussions not related to a specific program.

After the user enters a query, the Run! button (bottom right hand corner) can be used to execute the query. Pressing Run! collects the current program and query and creates a *runner* instance which is discussed in the next section.

2.2 Running a query: runners in the answer pane

The answer pane (top-right in Figure 1) is a placeholder for *runners*. The answer pane provides a menu in the upper right corner for operating on all runners inside it. Possible actions are Collapse all, Expand all, Stop all, and Clear. Each runner in the answer pane represents a query. The query may be completed, running or waiting for user input. SWISH can manage multiple active queries at the same time, up to an application defined maximum (default 3). Once a runner has been created, its execution no longer depends on the program and query editors. While the query is executing the user may prepare and start more queries.

Each runner provides its own set of controls to manage its query. During execution, a runner provides an Abort button. After query evaluation completes with an answer and more answers may be available, the runner allows the user to ask the next 1, 10, 100 or 1,000 results or to Stop the query. In addition, the runner shows a *text input* field if the application wants to read input and may show debugger interaction buttons if the tracer is being used (see Section 2.2.1).

A runner can render answers in two modes. The default classical Prolog mode renders the variable bindings as a sequence of *Var* = *Value* statements. The *tabled* mode displays a table where each column represents a variable. The tabled mode is particularly appealing for querying datasets (see Figure 2), while the default mode is more suitable for rendering small amounts of complex answers such as the chessboard position in Figure 1. By default, Prolog terms are rendered as a nested HTML `<span>` element where the rendered text is the same as the output generated by Prolog's **writeq/1** predicate. The functor of a compound term may be clicked to collapse the term to ellipses (...).

The server can provide *rendering libraries* that render Prolog terms using dedicated HTML. In Figure 1, the 'chess' renderer translates a list of length N holding integers in the range $1 \ldots N$ as a chessboard with queens. The interface allows for switching the selected rendering as illustrated in Figure 3. The technical details are discussed in Section 5.6.

2.2.1 Debugging

The SWISH debugger is based on the traditional 4-port debugging model for Prolog (Byrd 1980). The ports are events happening to goals in the procedural interpretation of the program:

call starting to prove the goal,
exit the goal was successfully proved,
fail the goal cannot be proved,

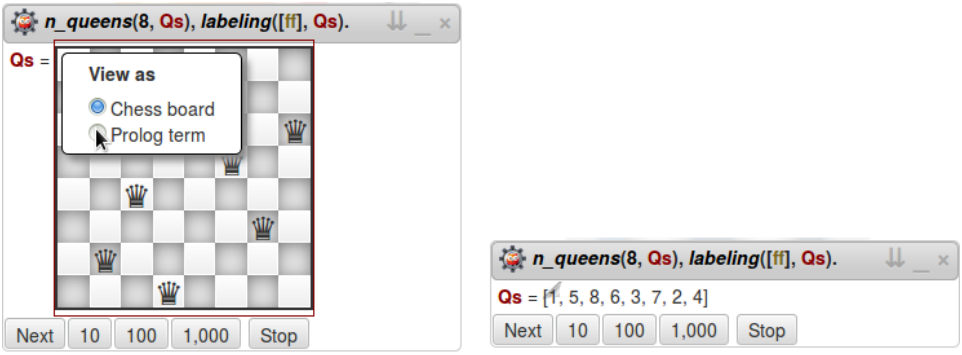

Fig. 3. With the 'chess' render library, a list of integers is interpreted as queens on a chessboard. The numbers in the list are the indexes of the columns of the queens in each of the rows. The user can select rendering as a 'Prolog term' to see the actual term.

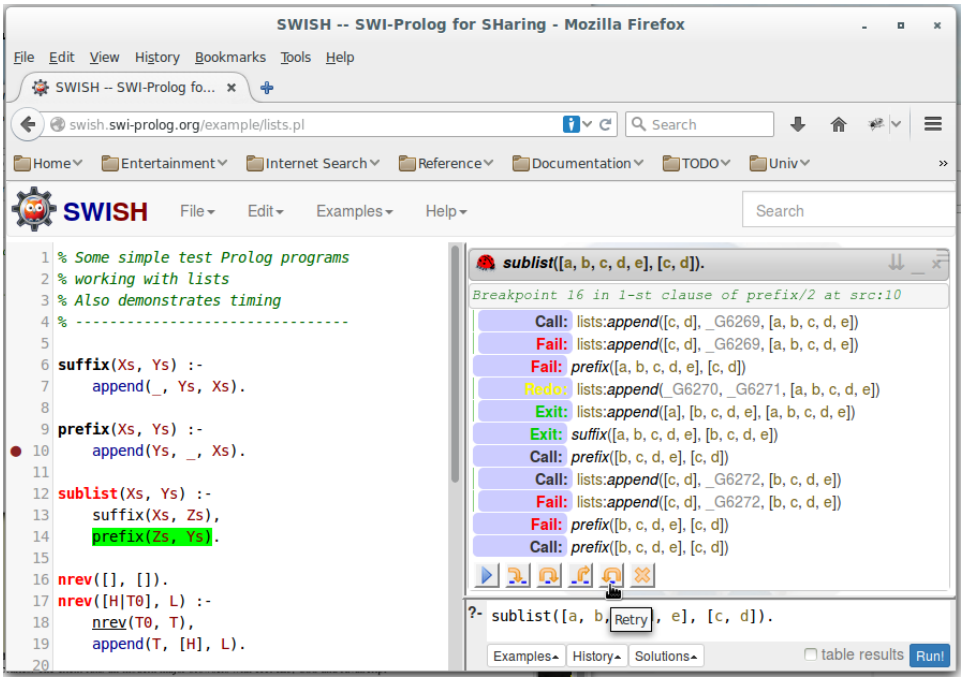

Fig. 4. Debugging applications in SWISH

redo backtracking for further solutions of the goal.

Figure 4 shows the tracer in action on **sublist/2** from the *Lists* example source. The debugger was triggered by a break-point on line 10 set by clicking on the line-number in the code editor. The debugging interaction is deliberately kept simple and similar to traditional programming environments. A *retry* button is added to the commonly seen 'step into', 'step over' and 'step out' for highlighting the feature of Prolog of allowing the re-evaluation of a goal.

2.3 The notebook interface

A notebook is, inspired by Jupyter notebooks, a series of *cells* that are displayed in page order (top-down), see Figure 5. The notebook menu allows cells to be added, copy/pasted, deleted or reordered. SWISH supports four cell types: *program*, *query*, *markdown* and *HTML*. The program and query cells are instances of the program and query editors. The markdown and HTML cells have two modes. By default they are displayed in rendered mode. Double clicking them transforms them into a CodeMirror editor instance with syntax highlighting. Clicking outside the cell re-enables the rendered mode. Cells have a dynamic height that fits their content. Program cells have a maximum height and may be collapsed to a single line.

Notebooks have two types of programs, *local* and *global*. A query that is being executed creates a *runner* using the first local program cell above it and all global program cells on the notebook. This is commonly used to realise variants of a query. In this scenario, global program cells are placed (usually) at the bottom of the notebook and provide the shared code for all queries. Just above the query cell we place a program cell that completes the global program. For example, suppose we want to find paths between nodes in various graphs. In this case we write generic code for finding paths in graphs and we flag it as global. Then we may include in different local programs the description of the different graphs in terms of their arcs: a query below a local program can then perform pathfinding in the individual graph represented by the local program.

Notebook queries have a settings button that determines whether the results are shown as a table or as a traditional set of bindings, whether or not the query should be executed when the page is loaded and how many results to show initially.

3 Using SWISH in Prolog education

The educational potential of SWISH was obvious from its inception. SWISH provides a rich Prolog environment for basic educational needs without installing software. The relatively simple and modern web interface is appealing to students. The possibility of saving programs assigning them an URL allows students to save their assignment and send the link to their teacher.

An obvious application is to exploit the ability to embed SWISH using an `<iframe>` element into existing online course material. The first example was created by Jan Wielemaker for Learn Prolog Now![7], see Figure 6. Learn Prolog Now! is an online version of a Prolog book by Patrick Blackburn, Johan Bos, and Kristina Striegnitz (Blackburn et al. 2006). We established a proof of concept that embeds SWISH in the online course material.[8] The current version rewrites the Learn Prolog Now! HTML on the fly, recognising source code and example queries. It is not yet good at recognising the relations between source code fragments and queries.

[7] `http://www.learnprolognow.org`
[8] `http://lpn.swi-prolog.org`

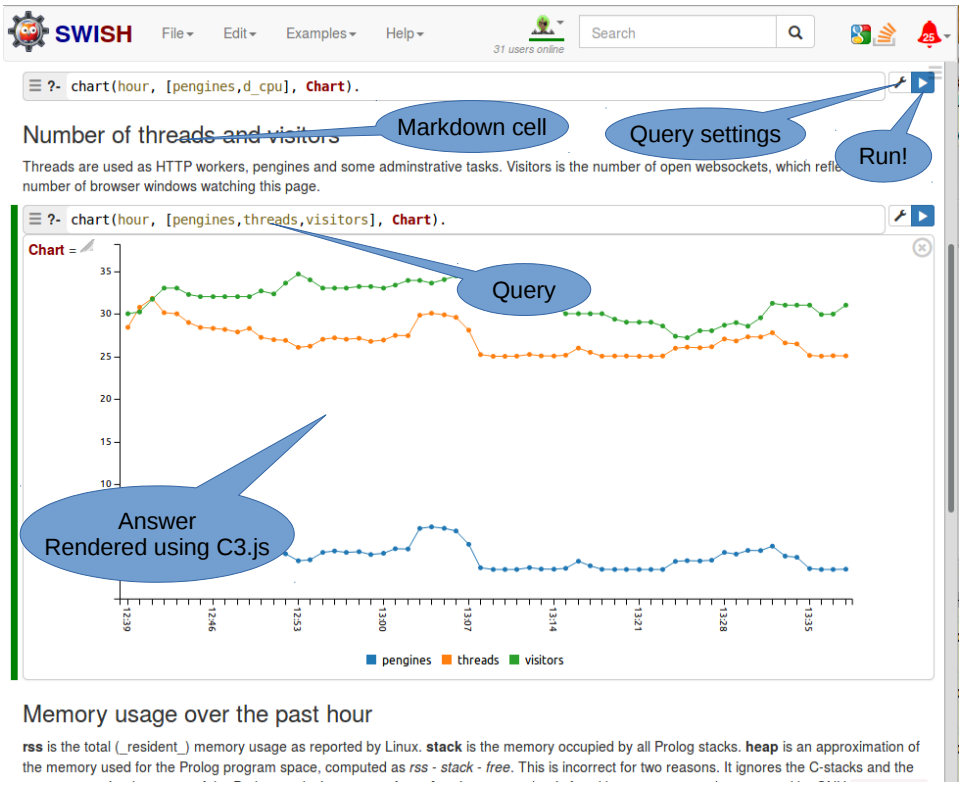

Fig. 5. Screendump of SWISH in 'notebook' mode. The displayed notebook is from the Examples menu, showing statistics from the running SWISH server. The visible part contains two queries, the bottom one being executed. The result is a chart showing the number of running queries (Pengines), Prolog threads and visitors of the website. The result is rendered using C3.js.

Also Learn Prolog Now! needs some updating to be more compatible with SWI-Prolog. The server served 304,205 pages (user pages, excluding javascript, css and images) during 2016.

A recent example of an online book embedding SWISH is *Simply Logical* by Peter Flach.[9]

From private mail exchange we know that several universities run local copies of SWISH to support education. For example, at the University of Ferrara SWISH is used to teach Prolog to the students of the Programming Languages course. The students are given assignments in the form of specifications and they have to write Prolog code implementing them. The students then submit the assignment by sending the link of the saved program or notebook to the teacher. The chatroom is one of the feature most liked by the students: they use it mainly by exchanging messages about the assignments among themselves. SWISH also considerably simplifies laboratory management because it removes the need to install and keep updated

[9] `http://book.simply-logical.space/`

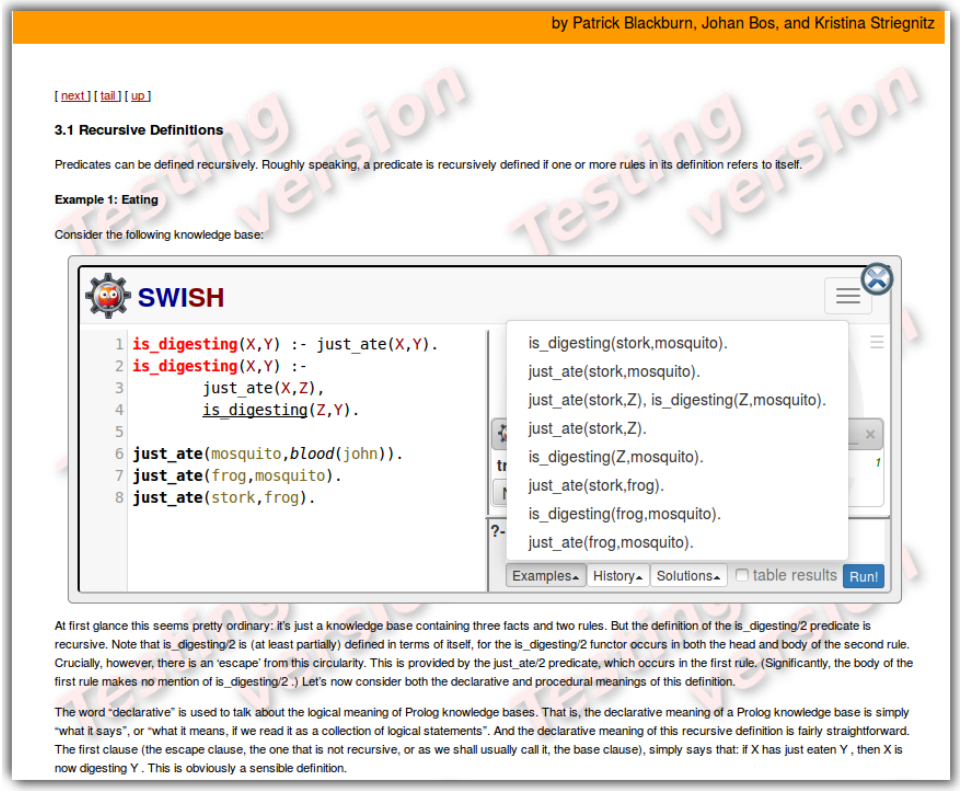

Fig. 6. Screendump of Learn Prolog Now with an opened SWISH instance that shows the collected source as well as example queries from the following text that are classified as relating to this source. The embedded SWISH provides all functionality available in the stand-alone SWISH. If the user presses the close button, SWISH is removed and the original code re-appears.

the Prolog stack on the laboratory machines. Moreover, it makes Prolog accessible from the students' laptop straight away, bypassing installation problems due to the wide variety of operating systems and distributions in use by the students.

`cplint` on SWISH is also used at the University of Ferrara for teaching Statistical Relational Artificial Intelligence in the Data Mining & Analytics course: students are given an assignment whose solution is a program in PLP, which they can develop in `cplint` on SWISH and submit by saving it in the system and sending the URL to the instructor. For this course, the graphics capabilities of SWISH are really important, as the output of probabilistic inference and learning can be best visualized using graphs and diagrams. Therefore SWISH plays a very important role in the dissemination and promotion of PLP in general and `cplint` in particular.

Finally, SWISH is also used in online education. "Introduction to SWI-Prolog" is an online course taught by Anne Ogborn that run for 8 weeks from June 8th to August 6th 2018. It had 389 enrolled participants. The course uses a variety of tools: the Moodle e-learning platform, Youtube broadcasts, Slack channels, Twitter, forums and SWISH. The instructors hands out weekly assignments, some of which

are given as SWISH notebooks with exercises that the student can solve by modifying the notebooks and saving them with a different name. The saved notebooks are tagged by the student with the `prologclass` tag so that the instructor and fellow students can easily locate them. As of August 2nd 2018 the system includes 63 files tagged as `prologclass`.

4 Extending SWISH

SWISH can be extended to support languages built on top of Prolog. We assume that the new language uses Prolog syntax, typically extended with additional operators. We also assume we still have a *program* and queries about this program that have zero, one or more answers. To build an extension one needs to:

- Provide syntax highlighting support for the target language. This implies using the SWI-Prolog library `prolog_colour.pl`.
- Provide additional *answer rendering* plugins. See Section 5.6.
- Change the program or notebook *profiles*. Profiles are skeleton or template programs and notebooks that are stored in a directory `profiles`. The user may create a new program or notebook from one of these profiles.
- Add notebooks and programs that provide examples. Examples are added to the directory `examples`. The directory contains a JSON index file that organises the content of the Examples menu in the SWISH navigation bar. Typically only a few notebooks with a single markdown cell describing the available examples are added to the menu.

4.1 The `cplint` on SWISH system

`cplint` on SWISH[10] (Riguzzi et al. 2016; Alberti et al. 2017) is an extension of SWISH for experimenting with Probabilistic Logic Programming (PLP) (De Raedt and Kimmig 2015), the integration of logic programming with probability theory. `cplint` on SWISH provides a web interface to the `cplint` system (Riguzzi and Swift 2013; Riguzzi 2013; Bellodi and Riguzzi 2013; Bellodi and Riguzzi 2015).

PLP is still a relatively new field and its user base is still growing. One obstacle to its widespread adoption is the difficulty in the installation of the various PLP systems. For example, `cplint` is distributed as a SWI-Prolog pack but its installation requires the compilation of an external C library together with the CUDD library[11] for handling Binary Decision Diagrams (BDDs). The compilation is not automatic on some platforms, such as Windows, and requires manual intervention. This may present a hurdle for some users. `cplint` on SWISH allows users to try `cplint` without installing anything on their machines, thus offering the opportunity of quickly getting a feeling of what PLP is and what `cplint` can do. Moreover, the

[10] `http://cplint.ml.unife.it`
[11] `http://vlsi.colorado.edu/~fabio/`

possibility of saving, sharing and discussing programs also allows users to experiment with the system in a collaborative way, which is particularly important when learning a new technology.

`cplint` on SWISH handles PLP languages that follow the distribution semantics (Sato 1995): a probabilistic program defines a probability distribution over normal programs called *instances*, and the probability of a query is the sum of the probabilities of the instances where the query is true. This approach has recently gained prominence and it is adopted by many languages such as Logic Programs with Annotated Disjuctions (LPADs) (Vennekens et al. 2004), ProbLog (De Raedt et al. 2007) and PRISM (Sato 1995). Programs in any of these languages can be written in `cplint` on SWISH.

Each language offers the possibility of expressing probabilistic choices: for example, LPADs allow disjunctions in the head of clauses where each disjunct is annotated with a probability, the meaning of a clause of this form being that, if the body is true, then one of the head atoms is true with its corresponding probability.

In PLP the reasoning tasks are inference, parameter learning and structure learning. In inference, the aim is to compute the probability of a query, possibly given some evidence, from a program. Generating all possible instances is not viable because there is an exponential number of them so smarter algorithms must be adopted. One of the most successful approaches is *knowledge compilation*, where the program is compiled into an intermediate language from which the computation of the probability is easy. `cplint` on SWISH includes the PITA algorithm (Riguzzi and Swift 2013) that compiles the program to a BDD and computes the probability with a dynamic programming algorithm. However, exact inference is intractable in general so approximate algorithms have been developed. `cplint` on SWISH also includes MCINTYRE (Riguzzi 2013), which uses a Monte Carlo approach: the algorithm repeatedly samples the truth value of the query and the probability of the query is given by the relative frequency of the true value.

In parameter learning, the user has a program for which he does not know the probabilistic parameters and wants to induce them from data. In structure learning, the user wants to induce both the structure of the clauses and the parameters from data. In `cplint` on SWISH learning can be performed from sets of interpretations, each expressing a possible state of the world. Some predicates are identified as target, and true and false atoms of the target predicates in the interpretations are used as positive and negative examples respectively.

EMBLEM (Bellodi and Riguzzi 2013) performs parameter learning in `cplint` on SWISH using a special dynamic programming algorithm operating on BDDs. For structure learning, `cplint` on SWISH includes SLIPCOVER (Bellodi and Riguzzi 2015), which performs a search in the space of clauses and scores them using the likelihood of the data after parameter learning by EMBLEM, and LEMUR (Di Mauro et al. 2015), which is similar to SLIPCOVER but uses Monte Carlo tree search.

`cplint` on SWISH includes many examples of inference from various domains: reasoning about actions, random walks, marketing, natural language, biology, puzzles, genetics, model checking, medicine, games, social networks, filtering, Bayesian

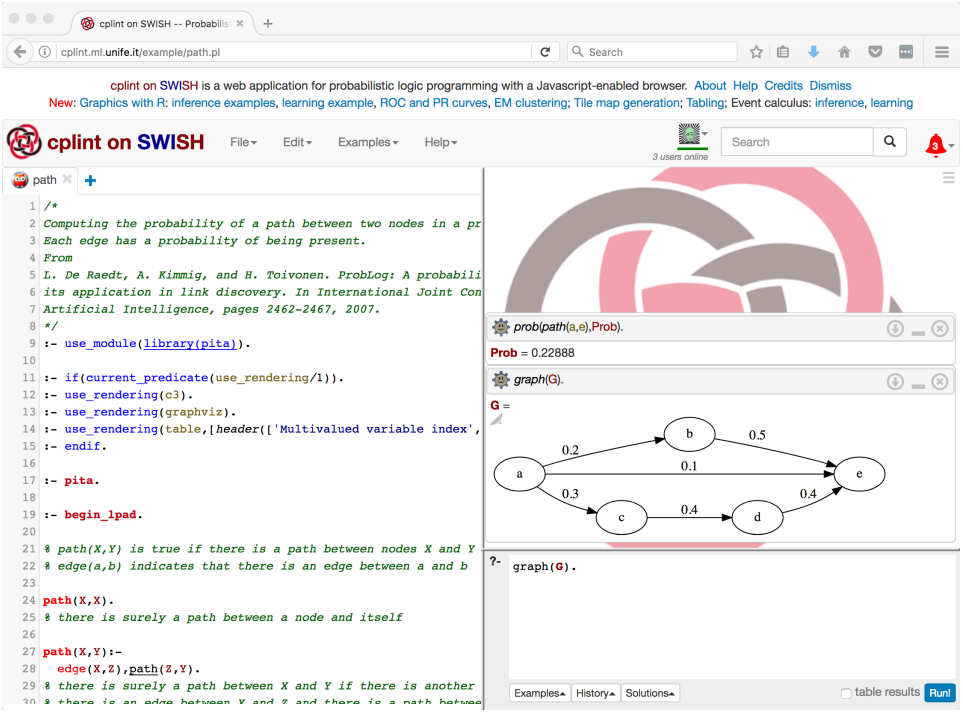

Fig. 7. Screendump of `cplint` on SWISH with opened probabilistic graph example.

estimation and regression. These examples show the various features of the inference modules: encoding Markov Logic Networks and stochastic logic programs, continuous random variables, particle filtering, likelihood weighting, Metropolis-Hastings sampling, rejection sampling, argument sampling, causal inference and computation of expectations, see (Riguzzi et al. 2016; Alberti et al. 2017; Nguembang Fadja and Riguzzi 2017) for more details on the examples.

`cplint` on SWISH exploits the graphics capabilities of SWISH. For example, Figure 7 shows a program for the computation of the probability of the existence of a path between two nodes in a probabilistic graph (also known as the *network reliability problem*). The graphviz rendering library of SWISH is used to draw the probabilistic graph. The same library can be used to draw the BDD that is built by the PITA algorithm for answering a query.

`cplint` on SWISH also uses the R for SWISH library, which provides an interface to R both for computation and for graphics. Figure 8 presents a one-dimensional Kalman filter example with a graph that shows how the probability density on the position of the target changes with time.

SWISH also offers the possibility of adding rendering plugins. `cplint` on SWISH for example adds a plugin for drawing two dimensional tile maps. Figure 9 shows an example for the random generation of a two-dimensional tile map for a video game.

`cplint` on SWISH also includes learning examples, both for parameter learning

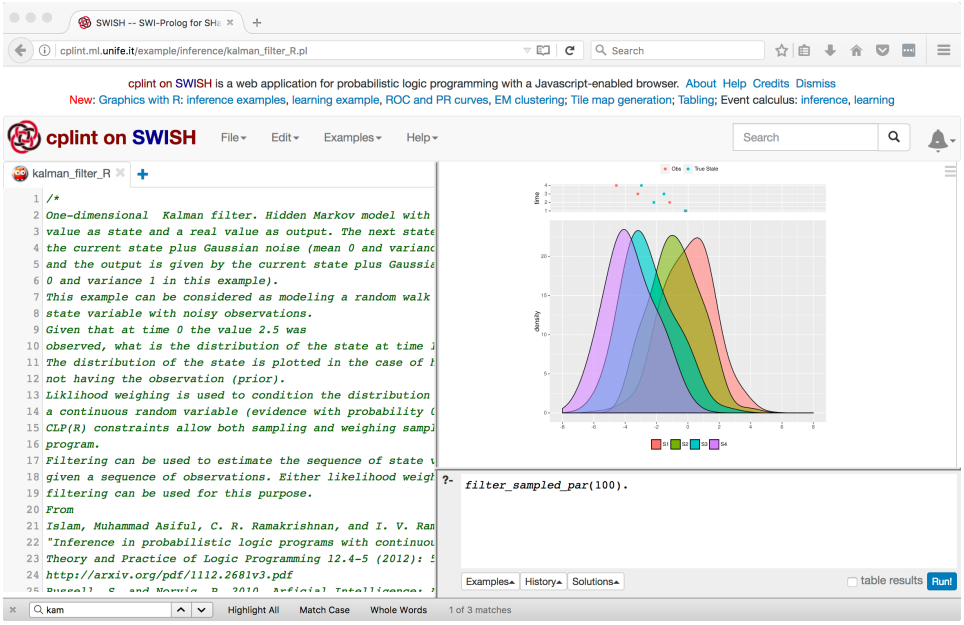

Fig. 8. Screendump of **cplint** on SWISH with opened Kalman filter example together with the results of a MCINTYRE query graphed with R.

Fig. 9. Screendump of **cplint** on SWISH with opened tile map generation example.

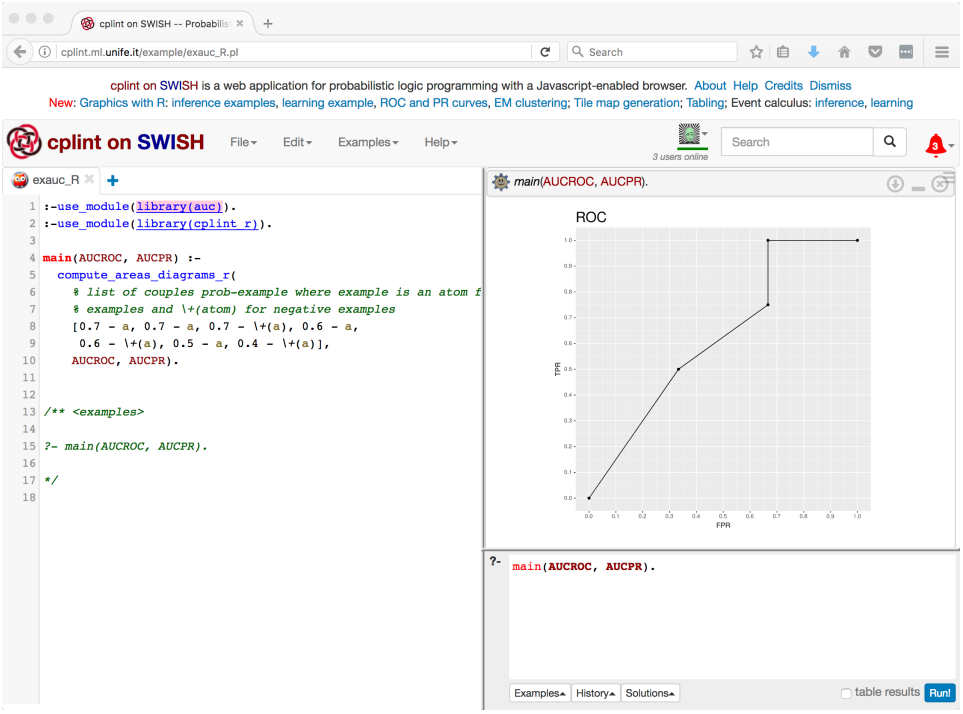

Fig. 10. Screendump of `cplint` on SWISH with opened ROC and PR curves example.

and for structure learning with SLIPCOVER and LEMUR: predicting whether a machine should be fixed, classifying pictures containing geometrical objects (Bongard problems), predicting the shopping behaviour of people, learning the parameters of hidden Markov models, predicting whether a molecule is an active mutagenic agent (the famous mutagenesis benchmark dataset (Srinivasan et al. 1996)), predicting the rating of courses of a university and learning the probabilistic effects of actions in the Event Calculus.

The learned programs are shown in the answer pane using a specific rendering library and can be applied to a testing set. Data analysts often find it useful to draw the Received Operating Characteristic (ROC) and Precision Recall (PR) curves of the performance of a model on the test set and to compute the areas under these curves. `cplint` on SWISH includes the `auc` SWI-Prolog pack for such purpose and can draw the curves using either R or C3.js: for example, Figure 10 shows a ROC curve drawn with R.

`cplint` on SWISH also includes Aleph (Srinivasan 2007), one of the most influential Inductive Logic Programming (ILP) systems. It learns normal logic programs from examples and background knowledge and can be used as a baseline for probabilistic ILP systems. Aleph has been ported from Yap to SWI-Prolog and transformed into a module with conditional compilation directives as the `cplint` algorithms in order to work in SWISH. It is now available as a SWI-Prolog pack.

The inclusion of these machine learning systems makes `cplint` on SWISH an example of the use of SWISH for data analysis.

`cplint` on SWISH offers program and notebook profiles for each of the inference and learning algorithms containing code skeletons for using the algorithms. The syntax highlighter of SWISH was extended to correctly emphasize the probabilistic constructs.

4.2 The LPS on SWISH system

LPS (Logic-based Production System) (Kowalski and Sadri 2015; Kowalski and Sadri 2016) is an extension of logic programming, which gives a logical interpretation to imperative sentences, by interpreting them as goals in first-order logic. Computation in LPS satisfies goals by generating a model that makes them true. It generates models for *reactive rules* of the form *if antecedent then consequent*, by performing actions to make *consequents* true whenever *antecedents* become true.

Reactive rules in LPS behave like production rules in production systems and like active rules (or triggers) in active databases. But, unlike production rules and active rules, which have only an imperative reading, reactive rules in LPS have a combined declarative and imperative interpretation:

To **make** *if antecedent then consequent* true,
make *consequent* true whenever *antecedent* becomes true.

Models in LPS are defined by logic programs, which represent beliefs, and are generated incrementally, starting from an initial state, observing a stream of external events, executing a stream of actions, and updating the current state. Updates are performed using causal laws, which define the time-varying facts (or fluents) initiated and terminated by events, including both external events and actions. Causal laws are based on the Event Calculus (Kowalski and Sergot 1986), but are implemented destructively, as in imperative computer languages. Analogously, events are processed as streams, without remembering their past.

The LPS interpreter previously ran on XSB and (SWISH-less) SWI Prolog. The SWISH implementation of LPS was developed to provide an easy-to-use online tool for teaching logic, computing and computational thinking. SWISH was chosen because it was the only Prolog system that provided the web-based interface and program editor needed for the task. Thanks to the SWISH and SWI-Prolog infrastructures, it was possible to develop the implementation with a limited budget of about four person-months. The open-source prototype is accessible from Imperial College London.[12]

The multi-pane IDE interface and the notebook facility were not part of the original requirement, but were additional bonuses. The notebook facility of SWISH encouraged us to create an interactive tutorial, combining explanatory text with runnable and editable examples and queries. This is illustrated in Figure 11 by the first notebook in the First Steps with LPS notebook.

[12] http://lps.doc.ic.ac.uk

Computation in LPS

Computation in LPS generates a sequence of states and events, using laws of cause and effect. States are sets of facts (also called fluents, because they can change with time). Events cause state transitions by initiating facts and terminating facts. For example:

```
1  % This is a comment, preceeded by %.
2  % The following two sentences declare the fluents and events in this example.
3  %
4  fluents lightOn, lightOff.
5  events switch.
6
7  % The following sentence describes the facts that are true at time 1.
8  %
9  initially lightOff.
10
11 % The following three sentences describe events taking place
12 % in the transition from one time point to the next.
13 %
14 observe switch from 1 to 2.
15 observe switch from 3 to 4.
16 observe switch from 10 to 11.
17
18 % The following four sentences describe the laws of cause and effect for this
19 %
20 switch initiates    lightOn     if lightOff.
21 switch terminates   lightOff    if lightOff.
22 switch initiates    lightOff    if lightOn.
23 switch terminates   lightOn     if lightOn.
```

To run this program, put `go(Timeline)`. into the query window.

```
?- go(Timeline).
```

Fig. 11. The first notebook in the Firststeps notebook in the Examples menu. The SWISH editor was extended, higlighting LPS-specific keywords, fluents and events.

```
3  fluents     ffire.
4  actions     eliminate, escape.
5  events      deal_with_fire.
6
7  initially   fire.
8
9  if      fire at T1
   Syntax error: holds(fire,'T1')' is not a valid reactive rule
   conjunct'
10 then    deal_with_fire from T1 to T2.
```

Fig. 12. SWISH editor showing a LPS-specific error.

Lines 4 and 5 of the program in the notebook are declarations. Line 4 declares that lightOn and lightOff are fluents, and line 5 declares that switch is an event. To aid in understanding the program, fluents are highlighted blue/grey in the SWISH program pane, and events are highlighted orange.

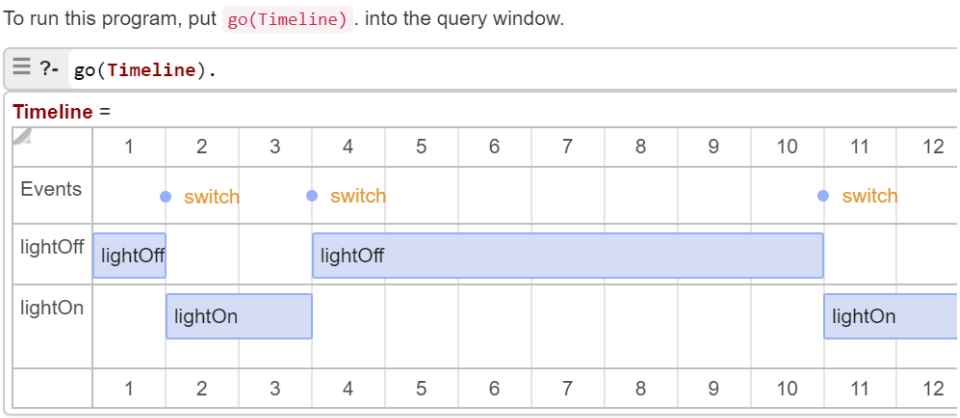

Fig. 13. LPS execution timeline.

Line 9 states that the fluent lightOff is true in the initial state, at time 1. Lines 14-16 record observations of external events, which occur in the transition from one state to the next. These events are timestamped by the times of the previous state and the next state. Lines 20-23 are causal laws, which specify the fluents that are initiated by an event and added to the current state, and the fluents that are terminated and deleted from the state. Events can also include actions generated by the program.

The fluent and event declarations help to detect syntactic errors, which are highlighted in red. The error message in Figure 12 reports that fire at $T1$ (internally represented as $holds(fire, T1)$) is not a valid conjunct for the reactive rule in lines 9 and 10, because fire has not been declared as a fluent.

Given an initial state, and stream of external events and actions, computation in LPS updates the current state, using the causal laws. This is illustrated by the execution of the program in Figure 11, which visualises the model as the timeline in Figure 13. The timeline is a graphical rendering of the term generated as an answer to the Prolog query $go(Timeline)$. Alternatively, models can also be presented in textual form as a list of states and events, using the nullary query go.

Timelines display models in their totality, from the initial state to the final state, including all states and events in between. In theory, time can be infinite. But in the implementation, the final state, which can be overridden, is at time 20 by default.

The program in Figure 11 illustrates a passive agent without any goals and without the ability to perform any actions. The second notebook in the same First Steps notebook illustrates an active agent, which can perform the action of switching the light (on or off) for itself, and which has the goal of switching the light as soon as it observes that the light is off:

```
        if lightOff at T1 then switch from T1 to T2.
```

Here $T1$ and $T2$ are "time" (or state) variables. $T1$ is implicitly universally quantified and $T2$ is implicitly existentially quantified. The quantifiers are not written

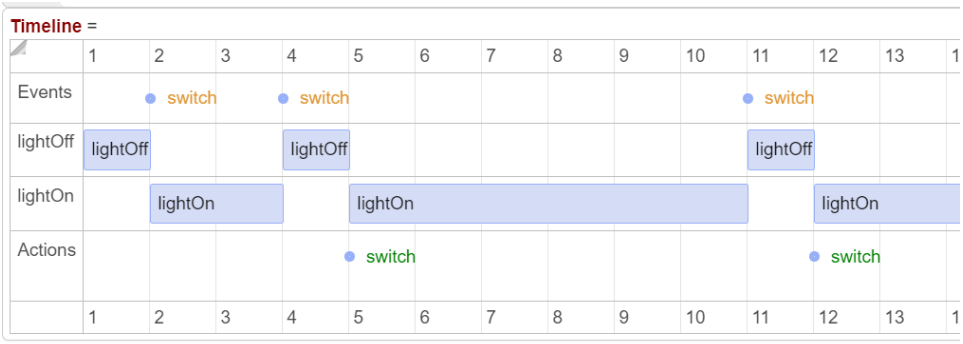

Fig. 14. LPS execution timeline for the same program and same external events, but with a reactive rule if *lightOff* at $T1$ then *switch* from $T1$ to $T2$.

explicitly, because all variables in the antecedent of a rule are universally quantified, and all variables that are only in the consequent are existentially quantified. Because switch is an atomic action, the only instantiation of the variable *T2* that solves the goal *switch from T1 to T2* is where *T2* is the next time after *T1*.

The explicit representation of time clarifies the meaning of sentences. However, as a shortcut, time can often be left implicit. For example, the rule in this example can also be written more simply in the form:

```
if lightOff then switch.
```

No matter how the rule is written, for the rule to be effective, switch needs to be declared as an action. As a result, when the light is observed to be off at 4, the rule generates an action of switching the light between 4 and 5. See Figure 14.

The first two notebooks illustrate agents whose only "beliefs" are "facts" or causal laws. However, beliefs in LPS can also include logic programs that define abstract concepts in terms of more concrete concepts, and logic programs that define complex events and plans in terms of other events, actions and fluents. The third notebook in First Steps illustrates a logic program defining an intensional predicate *happy* in terms of extensional predicates *lightOn* and *lightOff*:

```
happy(bob) if lightOn.
happy(dad) if lightOff.
```

In LPS, only extensional predicates are updated explicitly. Intensional predicates become true or false indirectly as ramifications of updates of extensional predicates. Times in the definition of an intensional predicate do not need to be written explicitly, because they are all identical.

In theory, the logical semantics of LPS allows any sentence of first-order logic to play the role of a goal. In this respect, the resulting combination of logic programs and first-order logic is like their use for model expansion in FO(ID) (Denecker 2000). The main difference is the focus in LPS on generating models that are sequences of states and events, using destructive updates as in imperative computer languages.

In contrast with the logical semantics of LPS, which allows arbitrary goals in

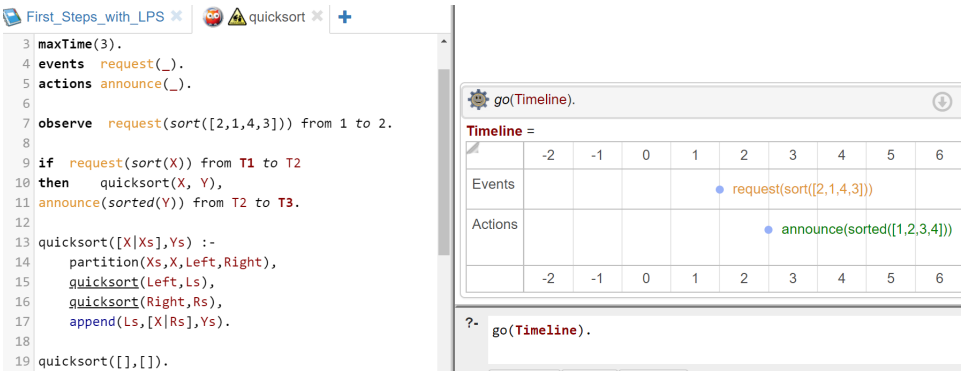

Fig. 15. Timeless, auxiliary predicates can be defined using Prolog.

first-order logic, the SWISH implementation allows only reactive rules and constraints. Constraints are written in the form *false conditions*, where *conditions* is a conjunction of events that all occur at the same time and fluents that hold before or after the events. In the fourth notebook, this is illustrated by the constraint:

```
false goto(dad, Place1), goto(dad, Place2), Place1 \= Place2.
```

which prevents dad from going to more than one place at a time. The constraint is necessary, because dad's behaviour is governed by reactive rules that make him go to a room whenever there is a light on in the room, and to turn the light off if it is still on. In this example, times do not need to be written explicitly, because of the restrictions on times in constraints.

Logic programs in LPS are pure Prolog programs, but are represented by Prolog facts, and are executed by a Prolog meta-interpreter. The meta-interpreter implements destructive updates, with its logical model generation semantics, using Prolog's assert and retract, which do not have a logical semantics. The use of Prolog to implement LPS means that auxiliary predicates in LPS can be defined by a compiled Prolog program, as in the example in Figure 15. The example also shows how LPS gives a simple, logical semantics to input-output, representing inputs as observations of external events, outputs as actions, and the relationship between inputs and outputs as a reactive rule. If the input is a request, then the rule may optionally include additional arguments and conditions to determine, for example, whether to fulfil the sender's request, reply with a polite refusal, or block future requests from the sender.

Models in LPS have two forms: A static form, which is needed for the logical semantics, and in which fluents and events are included in one, all-encompassing structure with timestamps; and a dynamic form, which is needed for practical implementation, and in which only the current state and events are stored without timestamps. Timelines display models in their static form. But models can also be displayed in their dynamic form, as animations.

For this purpose, we have used the rendering facilities of SWISH to develop a preliminary, declarative language for users to generate their own 2D animations.

Fig. 16. LPS 2D animation frame for the Game of Life.

Figure 16 illustrates an animation frame representing a single, current state in Conway's game of life, in the LPS examples notebook in the examples menu.

To obtain the animation, it suffices to specify the visual representation of fluents and their 2D co-ordinates. For example, the Prolog clause:

```
d(X-Y,[center:[XX,YY], radius:5, type:circle, fillColor:green]) :-
XX is X*10, YY is Y*10.
```

expresses that the fluent X-Y, which represents the presence of a live cell at location X-Y, is displayed as a green circle centred at $[X * 10, Y * 10]$ with radius 5. The transition from one state to the next is specified by reactive rules, such as the rule that a live cell dies if it has less than two live neighbors:

```
if X-Y at T, aliveNeighbors(X-Y,N) at T, N<2 then die(X-Y) from T.
```

2D animation can be used for a wide range of examples. Figure 17 illustrates an animation frame for a bank transfer simulation in the same LPS example notebook. The video-like controls allow the user to stop and restart the animation at any time. In fact, as the figure suggests, time in LPS is more like a state during which time stands still in an observe-think-act agent cycle.

The changing state of bob's and fariba's bank accounts is controlled by reactive rules, designed to enrich bob at fariba's expense:

```
if   transfer(fariba, bob, X) from T1 to T2,
     balance(bob, A) at T2,   A >= 10
then transfer(bob, fariba, 10) from T2 to T3.

if   transfer(bob, fariba, X) from T1 to T2,
     balance(fariba, A) at T2,  A >= 20
then transfer(fariba, bob, 20) from  T2 to T3.
```

The SWISH implementation of LPS is currently being used in an introductory AI course at Imperial College, and to provide self-study tutorials for more advanced students, some of whom have gone on to reimplement LPS in Python and JavaScript, and to use LPS in such applications as games.

The online prototype has also attracted a wide range of external users. The bank transfer example, in particular, has attracted users with interests in financial technology and smart contracts. Partly in response to this interest and building on previous work (Sergot et al. 1986) on representing legal documents as logic programs,

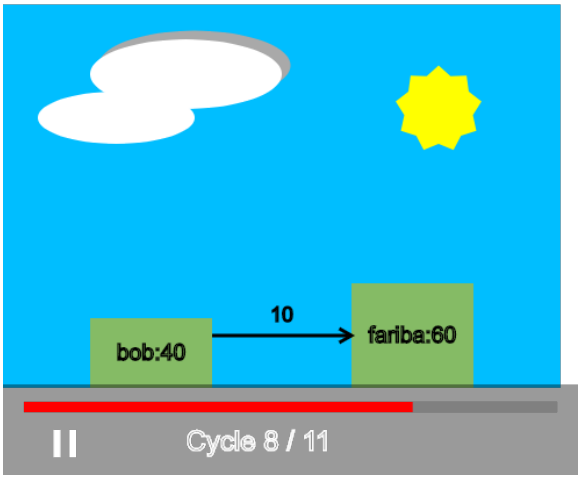

Fig. 17. Animation of the bank account example.

the LPS team has created a start-up company, Logical Contracts[13] to represent legal documents in logical, executable form. LPS is well suited for such legal applications, because there is a natural correspondence between logic programs and constitutive rules, which define legal concepts, between reactive rules and regulative rules, which enforce legal obligations, and between constraints and prohibitions.

5 Architecture

SWISH consists of two parts. The *client side*, running in a browser, is implemented as a series of jQuery plugins, using Bootstrap[14] for styling and RequireJS[15] for package management. The *server side* is completely implemented in SWI-Prolog (Wielemaker et al. 2008). It builds on top of the SWI-Prolog HTTP server libraries (Wielemaker et al. 2008), the Pengines library (Lager and Wielemaker 2014) and the IDE support libraries that provide data for auto completion, documentation and highlighting. In the remainder of this section we first give a brief overview of the available components, next we elaborate on some of these components and finally we discuss the portability of the design.

5.1 The client components

The client is written as a series of JavaScript jQuery plugins. RequireJS is used to resolve the dependencies and, for production installations, to compile all jQuery components into a single minified file. Currently, `swish-min.js` is 1.1Mb (308Kb compressed) and `swish-min.css` 188Kb (53Kb compressed).

[13] http://logicalcontracts.com
[14] `https://getbootstrap.com/`
[15] `http://requirejs.org/`

- The **program editor** is used to edit programs, queries, HTML and Markdown. It is described in in Section 5.3.
- The **query editor** is derived from the code editor, providing additional access to examples, history and modifying the query (count solutions, time, order, distinct, etc.)
- A **runner** instance takes care of actually running a query. It is created from zero or more source code fragments and a query. It allows for controlling the query (ask for more solutions, abort or discard it) and collects the *answers*. It can be asked to organise the answers in the traditional Prolog way as a series of *Var* = *Value* statements or as a *table*, providing a solution as a row of values.
- An **answer** instance is responsible for rendering a term that is part of the answer. The baseline is a nested HTML `<span>` reflecting the syntactic components of the term that allows for folding and unfolding terms. SWISH allows for alternative term rendering using, e.g., tables, charts, graphs, etc. This may be compared to the traditional **portray/1** hook, but SWISH rendering can exploit the full capabilities of HTML5, CSS, SVG and JavaScript. See Section 5.6.
- A **notebook** instance is a list of *cells* built from the above components, see Section 2.3. The notebook itself allows for adding, deleting and organising cells and provides the necessary code to serialise the content as an HTML document on the server and import other notebooks.
- The social infrastructure makes other users aware that they are operating on the same program using an *avatar*. The avatar has a notification area that indicates when a document is opened, closed or saved. Each document has an associated **chat room** that allows users to discuss code and share queries.

5.2 The SWISH server components

The SWISH server is a monolithic web server written entirely in SWI-Prolog. The code heavily depends on the SWI-Prolog web libraries (see Section 7.1). The current 20K source lines are distributed over 67 files. Below we describe the key components:

- The client side code editor is supported by a **highlight server** that produces semantically enriched highlighting information based on cross-reference analysis. In addition, the *mode declarations* from the documentation, e.g., "`thread_create(:Goal, -Id, +Options) is det`" are compiled into a JSON object to support CodeMirror insertion of *templates*. See Section 5.3.
- Running a query is managed by a **Pengine** or Prolog engine. The Pengine library is part of SWI-Prolog (Lager and Wielemaker 2014). *Sandboxing* queries is one of the optional facilities of the Pengine service and is discussed in Section 5.7.
- Query answers are processed by the **rendering service** that is described in Section 5.6. A rendering plugin recognises a Prolog term (like **portray/1**) and emits an HTML node. Adding rendering plugins is a common way to customise SWISH. See Section 4.

- Notebook **markdown** cells are supported by a server-side markdown engine that is borrowed from PlDoc, the SWI-Prolog documentation system (Wielemaker and Anjewierden 2007).
- The **storage** component stores documents in a format that is largely compatible with the GIT SCM. The main difference is that documents are treated individually and not organized in a *tree*. The storage component is discussed in Section 5.4.
- **Authentication** and **authorisation** can be used to provide optional or obligatory login using HTTP authentication (basic or digest) or oauth2 federated login. Authorised users can be assigned rights to documents. Authorization can also be used to disable the **sandbox** restrictions on permitted queries for certain users.
- The **data source** plugin based service can be used to import data into SWISH, typically from the web. The default plugins allow loading CSV files, scraping web pages and running SPARQL queries. In addition, Pengines can be used to access remote Pengine (or SWISH) servers.

5.3 The program editor

A proper editor is the most important component of a usable programming environment. The editor must support the language, including syntax highlighting, auto indentation, code completion based on templates and already existing code, highlighting of errors and warnings from the compiler and providing access to the documentation.

Prolog is a difficult language to support in code editors due to the lack of reserved keywords, the fact that there is no distinction between code and data and the ability to extend the syntax using new operators. For example, the word `if` in C starts an if-statement if not embedded in a comment or a string, but the word `is` in Prolog can refer to the built-in predicate **is/2**, but also to some predicate with a different arity, to just a constant, etc. Another example is `X-Y` which can both be an arithmetic expression or a *pair* as used within, e.g., **keysort/2**. SWI-Prolog's native editor, called PceEmacs, solves this problem by closely integrating Prolog with the editor. While typing, the current term (clause or directive) is parsed and analysed in the context of the current file and its imports after each keystroke. If the term has valid syntax, all tokens are coloured according to their syntactic category as well as their semantic role in the program. For example, a call to a non-existing predicate is coloured red, a call to a built-in or imported predicate is blue and a recursive call is underlined. The libraries that implement this analysis have been decoupled from the native IDE, both to support source colouring for the SWI-Prolog documentation system PlDoc (Wielemaker and Anjewierden 2007) and ProDT.[16]

There are two dominant open source and actively maintained in-browser code

[16] `http://prodevtools.sourceforge.net`, these libraries are not yet used by ProDT.

editors available: ACE and CodeMirror. We opted for CodeMirror because its highlighting is based on raw JavaScript code rather than a regular expression-based template language as used by ACE. The low level implementation allows for a novel highlighting implementation. The highlighter consists of a Prolog *tokeniser* in JavaScript. Tokenising Prolog is sufficient to colour comments, quoted material (strings, quoted atoms), variables and constants (atoms and numbers). The tokeniser also keeps track of bracket nesting to support smart indentation. CodeMirror's token-based highlighting does *not* allow for lookahead over line breaks. As a result we are not able to count the *arity* of terms that are broken over multiple lines. As terms with the same name but different arity are not related in Prolog this prevents unambiguous meaningful colouring.

We resolve this issue using server-side support to realise *semantic* highlighting. First, the changes to the content of the editor are forwarded to the server which maintains a mirror of the editor content. Second, the editor asks the server to produce a list of semantically enriched tokens for the source. The tokens are returned as a list-of-lists, where each inner list represents the tokens for a source term (clause or directive). Grouping the tokens per source term allows for incremental update (not implemented) as well as re-synchronisation (see below). For example, consider the following program.

```
go :-
    non_existing(X).
```

The JavaScript client produces the tokens [*atom*, *atom*, *functor*, *punct*, *var*, *punct*, *fullstop*]. The server analyses the program text and produces [*head (not called)*, *neck*, *goal (undefined)*, *punct*, *var (singleton)*, *punct*, *fullstop*]. The JavaScript tokeniser examines both its own tokens and the server token list. If the tokens are compatible (for example 'atom' is consistent with 'head'), it uses the enriched server information (head of a not-called predicate) to determine the colour and font. If the tokens are not compatible (for example the client finds *var* and the server reports *atom*, it schedules a request to the server for a revised list of enriched tokens. This request is sent if the user pauses typing for two seconds. While waiting for up-to-date enriched tokens, the JavaScript highlighting code heuristically tries to re-synchronise and either uses the uncertain results or falls back to the plain tokens. Heuristic re-synchronisation checks whether a single added, deleted or modified token gets the token stream in-sync. If this fails it tries to re-synchronise on a full-stop with the next clause or directive.

A CodeMirror *hover* plugin is used to show basic information about tokens if the pointer hovers over it. For goals, this includes where the goal is defined (ISO, SWI-Prolog built-in, a library, locally) and the documentation summary information if available. This information is requested from the server.

A CodeMirror *template* plugin is configured from templates (e.g., `atom_length(+Atom, -Length)`) extracted from the SWI-Prolog manual and PlDoc documentation of imported libraries. This plugin shows a menu of applicable predicates with their templates when the user presses Control-Space.

Finally, if the compiler generates errors or warnings, these are inserted as notes into the source code.

5.4 Document storage and version management

SWISH explicitly targets the cooperative development of Prolog programs and queries, optionally organised in notebooks. Cooperative development requires version management. The storage service is implemented in Prolog and inspired by the GIT SCM. Unlike GIT, each file is versioned independently rather than maintaining the version of a hierarchy of files because files are generally independent. The object representation of our storage system is binary compatible with GIT. The content of a *commit* object is a SWI-Prolog *dict* object linking the current version to the previous one and providing metadata such as the time, author and commit message. Saving a new version adds a record to a journal file that provides the file name, its previous commit hash and the new commit hash. The file-based repository can be shared between multiple SWISH processes that may run on different nodes if a shared file system is used and the file system supports advisory locks for synchronising writes to the journal file. The journal file facilitates fast discovery of the available heads and fast tracking of changes by other SWISH instances. The ability to run multiple SWISH servers on a shared or distributed file system improves its scalability.

Files can be referenced by content using their hash or by name. The name can be considered a version *head* and refers to the latest version with that name. The file interface provides the following operations:

- Saving a file anonymously, which produces a randomly generated URL.
- Saving a file by name.
- Organise files using *tags*.
- Saving a new version.
- Merge changes if another user saved the same file.
- Show the available versions and modifications between versions.
- Forking a file under a new name. The history remains linked to the original. Both the old and new name act as a revision *head*.

Prolog source files can include other programs on the same server using `:- include(filename).`, including the latest version or `:- include(hash).` to include a specific version.

5.5 Server side execution of the query

Server-side execution of a query is supported by the Pengines (Lager and Wielemaker 2014) library. The Pengines library allows for creating a Prolog engine possibly on a remote server represented by a Prolog thread. Optionally, the Pengine is handed a Prolog program that is loaded into the Pengine's workspace, a Prolog

module. The Pengine may be asked queries through HTTP, similar to a traditional Prolog user interacting with Prolog running in a terminal.[17]

5.6 Rendering answers using HTML

The *rendering* infrastructure exploits one of the key benefits of using web technology, the ability to use arbitrary HTML, CSS, Javascript and SVG to visualise answers. Figure 3 and Figure 13 provide examples. A rendering *plugin* is a Prolog module that exports a Prolog grammar rule (DCG) named **term_rendering//3**. This rule is passed the term to be rendered, a list of *Name* = *Var* bindings providing the names of variables as provided by **read_term/2** and a list of options. If the renderer recognises a Prolog term, it produces a list of HTML tokens following the rules of SWI-Prolog DCG-based HTML generation (Wielemaker et al. 2008). The rendering plugin is registered using a directive **register_renderer/2**, which defines the name of the renderer and a short description. A rendering plugin is activated for a program using the directive below. The *Options* term is optional. If provided it is passed to **term_rendering//3** in its third argument.

```
:- use_rendering(Renderer, Options).
```

Each term is handed to all active renderers and finally to the built-in generic term renderer that represents any term as **writeq/1** using nested `<span>` elements. If one or more of the special purpose renderers succeed, the output of the first is displayed, together with a hover-menu that provides access to the other representations.

Currently, rendering plugins are not executed inside the Pengine, but by the HTTP handler that returns a Pengine answer to the client. This is a consequence of the Pengine interface that will be addressed in future versions as it prevents the renderer from accessing global data, bypasses resource limitations imposed on the Pengine and prevents supporting rendering hooks as part of the user program.

5.7 Sandboxing queries

Prolog systems provide a programming environment that is capable of changing itself permanently and which exposes a potentially dangerous interface to the operating system. Both for education purposes and data analysis though, one can write meaningful programs without making permanent changes to the server or the server's file system. Preventing the user from performing dangerous operations is the task of the sandbox library. The sandbox is active while loading a program into the Pengine's program space, where it refuses to add clauses to other modules and where it only accepts a restricted set of *directives*, also aimed at keeping all changes local to the workspace. Prior to execution, the sandbox unfolds the query and compares all reachable goals with a whitelist. It fails under one of these conditions:

[17] `https://www.youtube.com/watch?v=G_eYTctGZw8`

- It discovers a (meta-) goal for which it cannot deduce the called code. The traditional example is `read(X), call(X)`. If such a goal is encountered, it signals an *instantiation error*, together with a trace that explains how the insufficiently instantiated goal can be reached. Note that it can deal with normal high-order predicates if the meta-argument is specified. For example, the following goal is accepted as safe.

```
?- maplist(plus(1), [1,2,3], List).
```

- It discovers a goal that is not whitelisted. In this case it signals a *permission error*, again accompanied with a trace that explains how the goal can be reached. Note that pure Prolog predicates are unfolded even if they are built-in or library predicates. Unfolding stops if a predicate is in the whitelist.
- It discovers a cross-module call (*M:Goal*) to a predicate that is not public. Normally, SWI-Prolog, in the tradition of Quintus Prolog, allows for this. Allowing it in SWISH would imply that no data can be kept secret. With this prohibition, libraries can keep data in local dynamic predicates that remain invisible to non-authorised users.

5.8 Remote access to SWISH

SWISH can be integrated in workflows with external tools and data. The integration options are embedding, downloading query results as CSV (comma separated values), accessing SWISH as a service and accessing other services from SWISH.

SWISH can be *embedded* in a web page using an `<iframe>`. This is particularly appealing for educational deployment: textual material can be interleaved with SWISH frames. The SWISH interface can be preloaded with a *background program* (an invisible program that is sent along with each query), a *program* (that appears in the code editor), a *query* and *examples* that appear in the query editor. It can also be preloaded with a *notebook*. See Section 3.

Both a *runner* and the notebook query editor provide a menu to download the query answers as CSV. The subsequent dialogue allows for setting a limit on the number of results and defining the *projection*, the variables to be included in the CSV and the order of these variables.

SWISH offers the Pengine API (Application Programming Interface) to clients in Prolog, JavaScript, Java, Ruby and Erlang. This API can be used to execute queries against programs that are provided by the client or available as saved documents on the SWISH server.

The client for Prolog offers a predicate for making non-deterministic RPC (remote procedure calls), for example:

```
?- use_module(library(pengines)).
true.
?- pengine_rpc('https://swish.swi-prolog.org', member(X,[a,b,c])).
X = a ;
X = b ;
X = c.
?-
```

5.9 SWISH Implementation of cplint

All the `cplint` on SWISH algorithms are available as modules to be loaded in a regular Prolog input file. Each module offers directives to separate various sections of the source with different roles: for example, for holding the probabilistic logic program for inference or parameter learning or the background knowledge for parameter or structure learning. These directives are handled using **term_expansion/2** and activate or deactivate other **term_expansion/2** clauses that process the probabilistic clauses and transform them into Prolog clauses to be used by the algorithms.

The modules were adapted for the multi-user context of `cplint` on SWISH. The modules use asserts to store information in the database: these asserts are now performed in the module that is created for the individual Prolog engine (see Section 5). During the execution of the algorithms all asserts are performed in the temporary Prolog engine module and discarded at the end of the computation.

5.10 SWISH Implementation of LPS

The LPS program is preprocessed into Prolog using the system predicate **term_expansion/2**. In addition, the interpreter was made thread-safe (wrt to SWISH's multithreaded server environment) by declaring relevant dynamic predicates with SWI Prolog's **thread_local/1** directive.

To help with reading and understanding an LPS program, the LPS specific syntax (of fluents, events and actions, as well as keywords such as if, $then$ and $false$) is highlighted by plugging into and extending SWISH's syntax colouring framework. The LPS pre-processor also checks for semantic errors and flags them when they occur, as in Figure 12. For this purpose, the pre-processor keeps a record of LPS source positions, and, when an error occurs, it delegates a report of the error to SWI Prolog's **print_message/2** predicate, passing an error term with the position. The SWISH machinery displays such messages as notes in the source code[18]

Each execution of a query $go(Timeline)$ generates a post-mortem trace as a set of Prolog facts, which is converted into a JSON representation, visualised by a JavaScript module.[19] The timeline is implemented as a SWISH custom renderer. Similarly, 2D animations are displayed via a SWISH custom renderer, developed using a 2D object library.[20]

More recently, the start-up Logical Contracts has further enhanced the SWISH implementation to allow the execution of LPS programs in the background, providing web services to client applications, which can query the background server for access to the server's fluents, receive events injected by the server, and delegate actions to other systems as REST requests. These events and actions can include interaction with the Ethereum blockchain.

[18] Such notes are displayed by CodeMirror, but they are *not* part of the user document.
[19] http://visjs.org/docs/timeline
[20] http://paperjs.org

6 Related work

We are not aware of other initiatives that aim at developing a rich web-based development environment for Prolog. There are web sites that allow running Prolog online such as Tutorial Points.[21] These sites either run your Prolog program as a batch job or provide a classical terminal to access Prolog. None of the environments we visited provide SWISH features such as notebooks or rendering Prolog answers using web technology such as charts.

We do not compare SWISH with traditional editor or GUI based development environments for Prolog such as GNU-Emacs or ProDT,[22] because web-based environments provide both new opportunities such as shared access and pose new challenges such as a more limited interface, limited bandwidth and high latency. Instead, we discuss three applications that served as inspiration for SWISH: JSFiddle,[23] R-Studio[24] and Jupyter Notebooks.[25]

- The initial inspiration for SWISH was JSFiddle. JSFiddle is an environment for testing browser technology (HTML, CSS, JavaScript) and thus it is naturally executed inside the user's browser. SWISH programs are executed on the server. For educational purposes, client side execution is probably feasible. For data analysis purposes remote execution allows users to control large Prolog jobs on big servers from their browser.
- R-Studio (Gandrud 2013) is an interface to the R statistical package. Although not a web application, it is based on the Qt webkit framework and uses web based technology in the background. The R-studio interface has a similar layout as SWISH, providing a source window, a console and an output window that typically shows results as tables or charts.
- Jupyter notebooks (formerly IPython notebooks) (Rossant 2013) allow mixing markdown or HTML text with Python code. When executed, the notebook shows the text, sources and possible results as numbers, tables or charts.

SWISH embodies many of the ideas behind JSFiddle, R-Studio and Jupyter. SWISH provides shared anonymous access like JSFiddle. Both the R-Studio 'program and output window' and Jupyter notebook interfaces are available as they serve different user communities in our experience.

Both R-Studio and Jupyter notebooks work on the basis of *authenticated access*. Once access is obtained, any command may be executed. SWISH can operate both as a public service granting access to non-intrusive queries that may be executed concurrently on shared (typically read-only) data and as an authenticated service to run arbitrary queries. The authenticated mode is often used for maintenance purposes such as updating the server software or loading data for further shared analysis.

[21] `https://www.tutorialspoint.com/execute_prolog_online.php`
[22] `http://prodevtools.sourceforge.net/`
[23] `https://jsfiddle.net/`
[24] `http://www.rstudio.com/`
[25] `http://jupyter.org`

Where Jupyter supports other languages using the notion of Jupyter *kernels*, such support is not provided by SWISH. We believe that Prolog is sufficiently different from the languages targeted by Jupyter to justify dedicated support. SWISH has little to offer to compete with Jupyter on the languages that Jupyter supports properly though.

Unlike JSFiddle, R-Studio and Jupyter, SWISH can easily be embedded into web pages using an `<iframe>` element where the program, examples and queries can be provided in the URL that loads SWISH. This feature can be used to transform source code in a static HTML page into a SWISH instance using a simple JavaScript call. This is demonstrated in Section 3.

`cplint` on SWISH is a system for PLP that uses SWISH to provide a web interface to `cplint`. A related PLP system is ProbLog2 (Fierens et al. 2015), which has also an online interface[26] (Dries et al. 2015). The main difference between the web interfaces of `cplint` on SWISH and ProbLog2 is that the first uses a Prolog-only software stack, whereas the latter relies on several different technologies, including Python 3 and external knowledge compilers. ProbLog2 can also be used in Jupyter notebooks but, differently from SWISH, cannot interface to R.

7 Discussion

We have not performed any direct user evaluations of SWISH. Given the pace of development, different configurations and user communities, it is not feasible to perform such a study with limited resources. Below we summarise the impact of the system, including some lessons learned regarding using Prolog through a web app, and discuss portability.

SWISH has been online (September 2017) for three years. It is extensively being used for Prolog education, as discussed in Section 3 and Section 4. Since its launch, at least three independently developed educational sites have been launched: `cplint` and LPS (described in this article) and the book *Simply Logical* (Flach 1994). The main server now has 35,420 programs stored with 68,806 revisions. Usage is strongly correlated with academic holidays, ranging from 36,415 queries per week (July 31 - August 6, 2017) to 125,946 (May 7-14, 2017). Google link search (`link:swish.swi-prolog.org`) reports 8,490 links. The popularity of the Github repository, 198 stars and 48 forks compared to the main SWI-Prolog repository with 223 stars and 53 forks is another indication of the impact of SWISH.

The development of SWISH and its usage as a public service demonstrates SWI-Prolog's ability for writing web services. Moreover, it represents an excellent stress-test for SWI-Prolog. We have been forced to extend resource management, e.g., by limiting the program space associated with the temporary modules to avoid extensive use of **assert/1** causing the server to run out of memory. Timeout and exception handling needed to be improved to avoid runaway queries. In particular, resource exceptions such as stack overflows are frequent and need to be handled

[26] `https://dtai.cs.kuleuven.be/problog/`

reliably. Both scalability and reliability of multi-threaded execution needed to be improved. We have now reached a level of stability were restarts due to software upgrades (every one to two weeks) are more frequent than restarts due to crashes, memory leaks or runaway queries. We have shown that SWI-Prolog-based servers are capable of concurrently running arbitrary queries with bounds on resources.

The current browser infrastructure has proven to be sufficiently rich, portable and performant for developing a web IDE for Prolog. Server-assisted semantic highlighting has proven to be feasible, but the mechanism is rather fragile because it requires synchronisation of two independent highlighting processes, one at the client and one on the server. Besides providing shared access, a web front end allows for transforming Prolog answers into rich graphical representations such as tables, charts, graphs or the LPS timelines.

7.1 Portability

The SWISH client libraries are based on mature JavaScript libraries. The client runs in all modern major browsers with HTML5, CSS and JavaScript support. It is frequently tested on Firefox, Chrome, Safari and Internet Explorer 11.

The server code is basically non-portable. Many of the required libraries and features are shared with at least one other Prolog implementation, but none is capable to support the full range. Below we summarise the main issues.

- The scale of the involved Prolog libraries demands a closely compatible Prolog module system. Probably only SICStus and YAP can be used without significant restucturing.
- The HTTP server libraries are heavily based on C code that interacts with the SWI-Prolog foreign language interface to Prolog streams. YAP has copied the low-level libraries and is capable to run an old version of these libraries.
- The Pengines library depends on the HTTP library and the multi-thread interface. The SWI-Prolog thread API is also provided by YAP and XSB.
- The sandbox library (Section 5.7) assumes that whitelisted predicates both have no side effects and are robust against stack overflows, cyclic terms, etc. Few Prolog systems can satisfy this requirement. SICStus Prolog would be a candidate.
- The semantic syntax highlighting depends on detailed source layout information provided by **read_term/3**. SWI-Prolog's support for term layout is an extended version of the Quintus Prolog term layout functionality.
- Significant parts of the code rely on SWI-Prolog version 7 extensions, notably the *dict* and *string* types that facilitate a natural mapping between Prolog and JSON data.

From the above list it should be clear that a fully functional port of SWISH to another Prolog system is not immediately feasible. There is a more realistic scenario though. In this setup, SWI-Prolog provides the web interface and most of the development tools and another language, not even necessarily Prolog, provides the query solving. The interface between the two can be based on inter-process

communication or, if the target system is safe and capable of supporting threads, by linking the target system into the process and using the C interface.

Inter-process communication is already used in SWISH for embedding R.[27] This embedding is based on Rserve[28]. Rserve consists of an R instance that is fully initialised and optionally contains pre-loaded libraries. The R server provides a TCP/IP or Unix socket and a binary protocol to manage sessions and R commands. A new connection causes the server to fork a new child that handles the client. Resource management and isolation are provided by running Rserve inside a Docker based OS sandbox.[29]

7.2 Scalability

The SWISH server is a monolithic SWI-Prolog program. Some parts of the interaction with SWISH are by nature stateful, notably query execution and syntax highlighting, which maintains a clone of the user's editor on the server. These parts can be distributed over a pool of servers using a proxy server with support for *sticky sessions*. Sticky session support implies that the proxy assigns a new user to a random server and subsequently forwards all trafic from this user to the same server. Other parts, such as serving the main page, icons, CSS, JavaScript and manual pages can easily be offloaded from the Prolog server by using a caching reverse proxy or serving them from another server. Secondary servers may also be used to offload the chat and storage services. Extrapolating from resource usage on the main public server we estimate that a single SWISH server can handle up to about 1,000 concurrent users (dual Intel Xeon E5-2650) for a typical educational workload.

8 Conclusions

This article presented SWISH, a web service that supports the execution of Prolog and domain specific languages (DSLs) defined on top of Prolog from a browser. SWISH provides an interface that is primarily targeted at program development and testing as well as a *notebook* interface that is argeted at capturing workflows for data analysis and tutorials. The SWISH server is a SWI-Prolog program where user queries are executed by Pengines which are built from a Prolog thread and an isolated temporary Prolog module. The monolithic approach provides easy deployment as well as shared access to, e.g., large pre-loaded Prolog knowledge bases.

The current interface provides a better interaction with and more control for managing queries as well as small Prolog programs than the traditional Prolog command line and Prolog IDEs. This makes it suitable for education and data analysis. It is not yet suitable for the development of large applications. For these

[27] `http://www.swi-prolog.org/pack/list?p=rserve_client`
[28] `https://cran.r-project.org/web/packages/Rserve/`
[29] `https://github.com/JanWielemaker/rserve-sandbox`

scenarios, users expect support for modular program development, a richer editor and a better debugger.

The current system does not provide much support for animating progress, such as showing the moves while solving the towers of Hanoi puzzle. One option used in LPS is to generate the entire animation and then use JavaScript to play the animation in the user's browser. Future versions may exploit the Pengine *prompt* mechanism that is also used to implement **read/1** to read from the browser, for creating interactive animations. Currently, SWISH rendering plugins are executed in the context of the HTTP reply mechanism rather than by the Pengine itself. This implies that they are not sandboxed, they must be provided by the SWISH installation and we cannot support user defined rendering plugins.

In data analysis scenarios some queries take long to execute and, if we provide a result, we would like to be able to reliably reproduce this result and apply exactly the same version of the analysis program to new data. For this purpose we are developing a persistent answer cache based on a hash of the called predicate and its dependencies.

Acknowledgements

This research was partially supported by the VRE4EIC project, a project that has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement №676247. The SWISH implementation of LPS was developed under an EPSRC grant administered by Imperial College London. We thank the referees for their helpful comments.

References

Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., and Zese, R. 2017. `cplint` on SWISH: Probabilistic logical inference with a web browser. *Intelligenza Artificiale 11,* 1, 47–64.

Bellodi, E. and Riguzzi, F. 2013. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis 17,* 2, 343–363.

Bellodi, E. and Riguzzi, F. 2015. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming 15,* 2, 169–212.

Blackburn, P., Bos, J., and Striegnitz, K. 2006. *Learn prolog now!* Vol. 7. College Publications, London, UK.

Byrd, L. 1980. Understanding the Control Flow of Prolog Programs. In *Logic Programming Workshop*. Department of Artificial Intelligence, University of Edinburgh, Debrecen, Hungary.

De Raedt, L. and Kimmig, A. 2015. Probabilistic (logic) programming concepts. *Machine Learning 100,* 1, 5–47.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *20th International Joint Conference on Artificial Intelligence, Hyderabad, India (IJCAI-07)*, M. M. Veloso, Ed. Vol. 7. AAAI Press, Palo Alto, California USA, 2462–2467.

Denecker, M. 2000. Extending classical logic with inductive definitions. In *Computational logicCL 2000*. Springer, Cham, 703–717.

Di Mauro, N., Bellodi, E., and Riguzzi, F. 2015. Bandit-based monte-carlo structure learning of probabilistic logic programs. *Machine Learning 100,* 1, 127–156.

Dries, A., Kimmig, A., Meert, W., Renkens, J., Van den Broeck, G., Vlasselaer, J., and De Raedt, L. 2015. Problog2: Probabilistic logic programming. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*, A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, and M. Spiliopoulou, Eds. Springer International Publishing, Cham, 312–315.

Fierens, D., den Broeck, G. V., Renkens, J., Shterionov, D. S., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming 15,* 3, 358–401.

Flach, P. 1994. *Simply Logical: Intelligent Reasoning by Example.* Wiley, Chichester, UK.

Gandrud, C. 2013. *Reproducible Research with R and R Studio.* CRC Press, Boca Raton, FL.

Knuth, D. E. 1984. Literate programming. *The Computer Journal 27,* 2, 97–111.

Kowalski, R. and Sadri, F. 2015. Reactive computing as model generation. *New Generation Computing 33,* 1, 33–67.

Kowalski, R. and Sadri, F. 2016. Programming in logic without logic programming. *Theory and Practice of Logic Programming 16,* 3, 269–295.

Kowalski, R. and Sergot, M. 1986. A logic-based calculus of events. *New Generation Computing 4,* 1, 67–95.

Lager, T. and Wielemaker, J. 2014. Pengines: Web logic programming made easy. *TPLP 14,* 4-5, 539–552.

Nguembang Fadja, A. and Riguzzi, F. 2017. Probabilistic logic programming in action. In *Towards Integrative Machine Learning and Knowledge Extraction*, A. Holzinger, R. Goebel, M. Ferri, and V. Palade, Eds. Springer, vol. 10344. Springer, Heidelberg, Germany.

Riguzzi, F. 2013. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae 124,* 4, 521–541.

Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., and Cota, G. 2016. Probabilistic logic programming on the web. *Software: Practice and Experience 46,* 10 (October), 1381–1396.

Riguzzi, F. and Swift, T. 2013. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory Pract. Log. Program. 13,* Special Issue 02 - 25th Annual GULP Conference (March), 279–302.

Rossant, C. 2013. *Learning IPython for interactive computing and data visualization.* Packt Publishing Ltd, Birmingham, UK.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *12th International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, MA, 715–729.

Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., and Cory, H. T. 1986. The british nationality act as a logic program. *Communications of the ACM 29,* 5, 370–386.

Srinivasan, A. 2007. Aleph. `http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html`.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., and King, R. D. 1996. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence 85,* 1-2, 277–299.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004. Logic Programs With Annotated Disjunctions. In *Logic Programming: 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004. Proceedings*, B. Demoen and V. Lifschitz, Eds. LNCS, vol. 3132. Springer Berlin Heidelberg, Berlin Heidelberg, Germany, 431–445.

Wielemaker, J. and Anjewierden, A. 2007. PlDoc: Wiki style literate programming for Prolog. In *Proceedings of the 17th Workshop on Logic-Based methods in Programming Environments*, P. Hill and W. Vanhoof, Eds. Cornell University Library, Ithaca, NY, 16–30.

Wielemaker, J., Huang, Z., and van der Meij, L. 2008. Swi-prolog and the web. *TPLP 8,* 3, 363–392.

Worlfram, S. 2016. How to teach computational thinking. `http://blog.stephenwolfram.com/2016/09/how-to-teach-computational-thinking/`.