RITS System Architecture

The Rule-based Intelligent Tutoring System (RITS) contains and controls all logic for interacting with users. Each client (which we use as a synonym for "user" throughout) sends user actions to RITS, and then queries RITS for the next action to be performed. To use RITS, add the following line to your Prolog code:

:- use_module(rits).

RITS is accessible via the following API calls:

- **rits_start(-Start)**
  Start is unified with a Prolog term that represents the initial state of RITS, before the system has received any client (i.e., user) actions.
- **rits_next_action(+A0, -A, +S0, -S)**
  This predicate relates a client action **A0** and a RITS state **S0** to the next RITS action **A** and next state **S**.
- **rits_history(+S, -Hist)**
  Hist is unified with a list of Prolog terms that represent the history of client interactions up to and including state **S**.

These predicates are completely *pure*: They do not emit any output, do not write any files, and do not query the user in any way. Therefore, they can be quite safely executed in a hosting environment.

Client actions are described in Table 1. The most important RITS actions are described in Table 2. RITS clients are expected to interpret RITS actions "appropriately", such as: displaying messages on the terminal or via HTML, allowing users to give answers etc.

solve(Task)	Guide the student through solving **Task**.
student_answers(T)	Tell RITS that the student has responded with the Prolog term **T**. This action is only admissible if the directly preceding RITS action was **read_answer**.
next	Query RITS for its next action.
skip	The student wants to skip solving the subproblem that is currently in progress. (*TODO*)

Table 1: Admissible client actions

enter	Start of a subproblem that the student must solve.
exit	The most recently spawned subproblem was solved.
format(F)	The string **F** is to be displayed.
format(F,As)	The format string **F** is to be displayed. **As** is a list of arguments that are output, in sequence, in place of each `~w` that appears in **F**.
read_answer	The student is to be queried for a an answer.
student_answers(T)	The Prolog term **T** is the *parsed* student answer.
solve(Task)	The student should be given the task **Task**.
done	There are no further actions.

Table 2: The most important RITS actions

Due to their purity, these predicates are also very amenable to regression testing and further analysis. (*more to follow*)

The RITS engine may be accessed directly via Prolog, remotely via JavaScript and *pengines* using JSON encoding of Prolog terms, or via any other language embedding.

Extending RITS

Internally, RITS uses the following predicates to decide what to do. All of them are DCG rules that describe a list of RITS actions that need to be performed in response to certain client actions.

- **rits:solve//1**: this is called with argument **Task** for client actions of the form **solve(Task)**.
- **rits:actions//3**: This is called with arguments **Task** and **Answer** after the user (i.e., client) answered the question posed after **solve(Task)** with **Answer**. The third argument is a Prolog representation of previous interactions of the form **T=A**, in reverse chronological order.

RITS can be extended in a *modular* way to handle new domains. To teach RITS additional rules, define a Prolog module that provides its own (additional) DCG rules for **rits:solve//1** and **rits:actions//3**.

In these DCG rules, modules may provide their own custom RITS actions as Prolog terms that must be handled by the client when they appear as a RITS action upon calling **rits_next_action/4**.

See the file **rits_multiple_choice.pl** for the definition of simple multiple choice tests that are repeated when wrong answers are given.

LORITS: The extensible language of RITS

Preparing teaching material and exercises for students is one of the most common and essential tasks that teachers need to perform every day. It requires judicious human creativity and knowledge and cannot be automated.

Describing the material that is to be taught is a *declarative* task: We primarily want to describe *what* should be presented to students. *How* it is presented can vary by circumstances. For example, on a text terminal, we expect a presentation of the material that differs from that on mobile phones, Braille terminals, and sheets of paper.

RITS makes it easy for you to declaratively describe *what* should be done. RITS comes with built-in modules that describe what it means for students to "solve" a task like canceling fractions. In addition, you can supply your own rules for RITS, thereby stating what you mean by custom tasks that you define for your students.

The language of RITS is called LORITS and lets you state actions that need to be performed in order for students to "solve" a task. For example, it is easy to declaratively describe your "Lesson 1", which may consist of showing a video that contains some facts, and then have students automatically go through a custom multiple-choice test that asks them questions about the video in order to assess their understanding:

```
rits:solve(lesson_1) -->
        [video("http://www.youtube.com/VideoAboutMozambique"),
         solve(mchoice("What is the capitol of Mozambique?\n",
                       ['Maputo','Pretoria',
                        'Nairobi','Vienna'], 1)),
         done].
```

LORITS provides predefined actions such as formatting texts, going through multiple-choice tests, playing videos etc., which makes it easy to put together custom lessons that students can work through.

Here, it is our task to provide teachers with suitable building blocks of RITS actions, so that custom lessons can be put together quickly, possibly with our initial assistance. The task here is to find good and versatile actions to cover a large variety of interaction styles.

UTRITS: Expressing Unit Tests for RITS

UTRITS is a domain-specific language that lets you express *unit tests* for RITS. A unit test is a sequence of RITS actions and client responses, with some possibilities for abbreviations. The elements of UTRITS are described in Table 3.

String	True when RITS "emits" (via RITS action **format**) a string that contains *String*.
solve(Task)	Start solving **Task**. This can be initiated either by RITS itself (for example, when spawning a subproblem), or by the client.
=>(T)	Respond to the preceding RITS action, which must have been **read_answer**, with the client action **student_answers(T)**.
*	The precise meaning of * depends on the next UTRITS element E in the sequence. In all cases, * means to *ignore* all RITS actions, until: • If E has the form **=>(T)** and the next RITS action is **read_answer**, in which case **student_answers(T)** is sent to RITS and normal interaction is resumed. • If E is a literal string and RITS emits a string that contains E, in which case normal interaction is resumed. • If there is no next UTRITS element, then all further interaction is ignored.

Table 3: Admissible elements of UTRITS

A sample unit test and its result is shown in Fig. 1. Notice how messaging flips between server and client, just like in a real-world example of using RITS.

Due to the side-effect free and declarative nature of LORITS, it is easy to detect when users must be queried for a response, and to simulate user input. Unit tests that cover many possible interaction patterns can therefore be formulated quickly and conveniently with UTRITS.

```
?- rits:rits_run_test([solve(1/2 + 3/4),*,=>(4/6),*,solve(_),*,
                       =>(4),*,solve(_),*,=>(5/4),"nice"]).

    RITS says:      format("Please solve:\n\n~t~10+")
    Client says:       next
    RITS says:      fraction_layout(1/2+3/4)
    Client says:       next
    RITS says:      read_answer
    Client says:       student_answers(4/6)
    RITS says:      format("This is wrong.\n")
    Client says:       next
    RITS says:      format("You cannot just sum the numerators ...")
    Client says:       next
    RITS says:      enter
    Client says:       next
    RITS says:      format("Let us first find a common multiple of ...")
    Client says:       next
    RITS says:      solve(cm(2,4))
    Client says:       solve(cm(2,4))
    RITS says:      format("Please enter a common multiple of ...")
    Client says:       next
    RITS says:      read_answer
    Client says:       student_answers(4)
    RITS says:      format("Good, the solution is correct")
    Client says:       next
    RITS says:      format(" and also minimal. Very nice!\n\n")
    Client says:       next
    RITS says:      format("Now apply this knowledge ...")
    Client says:       next
    RITS says:      exit
    Client says:       next
    RITS says:      solve(1/2+3/4)
    Client says:       solve(1/2+3/4)
    RITS says:      format("Please solve:\n\n~t~10+")
    Client says:       next
    RITS says:      fraction_layout(1/2+3/4)
    Client says:       next
    RITS says:      read_answer
    Client says:       student_answers(5/4)
    RITS says:      format("Good, the solution is correct")
    Client says:       next
    RITS says:      format(" and also minimal. Very nice!\n\n")
    Client says:       next
    RITS says:      done
    true.
```

Figure 1: Sample unit test with UTRITS and the resulting interaction