tuProlog IDE Guide

tuProlog IDE version: 1.2.0

Document revision: 003
Creation date: 2002-09-17
Last Changes date: 2004-09-27

DEIS, Università di Bologna a Cesena, Italy

Contents

Chapter 1

The IDE

The tuProlog system comes with a simple application providing an user friendly integrated development environment to interact with a tuProlog engine, manipulate its knowledge base, make queries and explore solutions. In addition, means to dynamically manage the loading and unloading of tuProlog libraries are provided. After a proper installation of the tuProlog distribution, the application is spawned by launching the executable class `alice.tuprologx.ide.GUILauncher`. The console user interface version, providing a command-line shell, can be accessed by launching the executable class `alice.tuprologx.ide.CUIConsole`.

The main window of the tuProlog IDE is shown in Figure **??**. It is divided in two sections:

- an editing area on the middle, providing means to edit the engine's current theory;
- a console on the bottom, providing means to ask queries and display their solutions.

In the main window there also are: a toolbar at the top, providing facilities to manage theories, such as load, save as well as create a new theory, to load and unload libraries into and from the tuProlog engine, and to view in a separate window the debug informations activated by means of the `spy/0` predicate; and a status bar at the very bottom, providing status informations for the IDE and the engine.

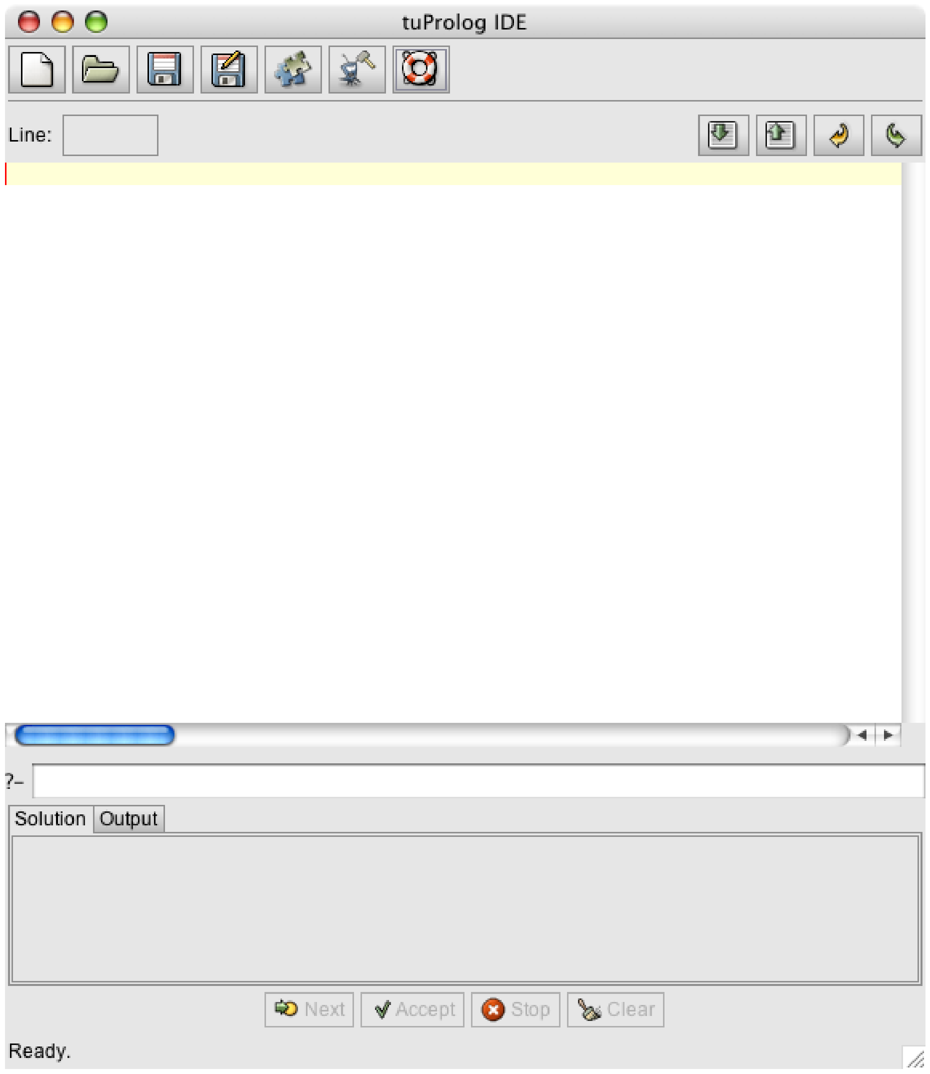


Figure 1.1: tuProlog IDE.

1.1 Editing the theory

The editing area allows multiple theories to be created and modified at the same time, by allocating a tab with a new text area for each theory. The text area provides syntax highlighting for comments, string and list literals, and predefined predicates. Undo and Redo actions are supported through the usual **Ctrl+Z** and **Ctrl+Shift+Z** key bindings.

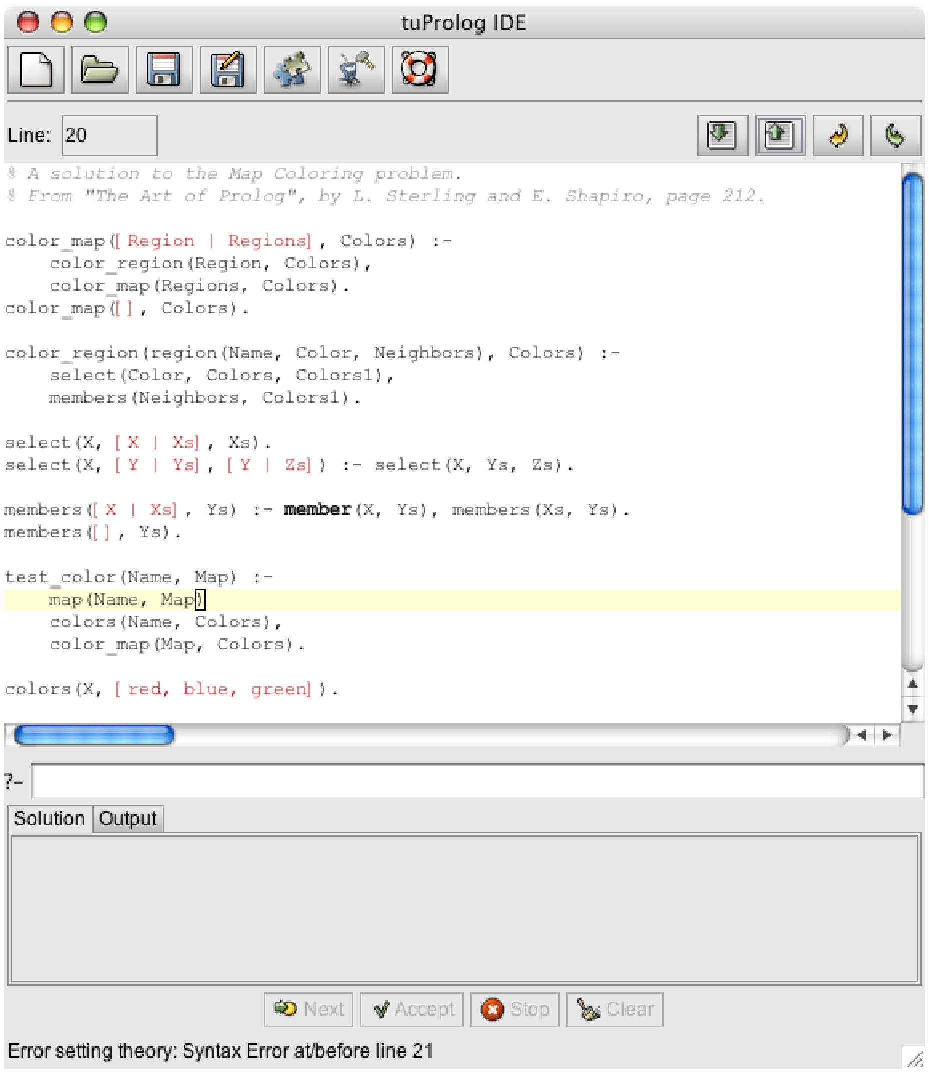
Above the editing tabs, a control area is found, where two buttons are provided to get the text of the engine's current theory into a new tab and to set the text contained in the editor of the selected tab as a new theory for the engine, and two buttons are provided for mouse-clicking support of Undo and Redo actions. An apposite action for retrieving the engine's current theory in an editor (shown in Figure **??**) is needed because whenever that theory gets modified by other means, such as calling the `consult/1` predicate, the changes are not automatically reflected in any text area. On the left side of the control area, there also is an indicator of the line where the caret is currently positioned in the edit area. Informations about the result of the action issued by the control area are provided in the status bar at the very bottom of the IDE's window: for instance, when setting an invalid theory to the engine because of syntax errors, details about the error are provided.

1.2 Solving goals

The console at the bottom of the tuProlog IDE's window is subdivided in two logical panes:

- a query pane composed by a textfield where queries can be inserted, and two buttons to trigger the solving process. The leftmost (Solve) button asks the engine to find the first solution to the query, allowing the user to possibly navigate through further solutions; the rightmost (Solve All) button forces the engine to find all solutions to the given query. Pressing the **Enter** key in the textfield has the same effect as pressing the Solve button.

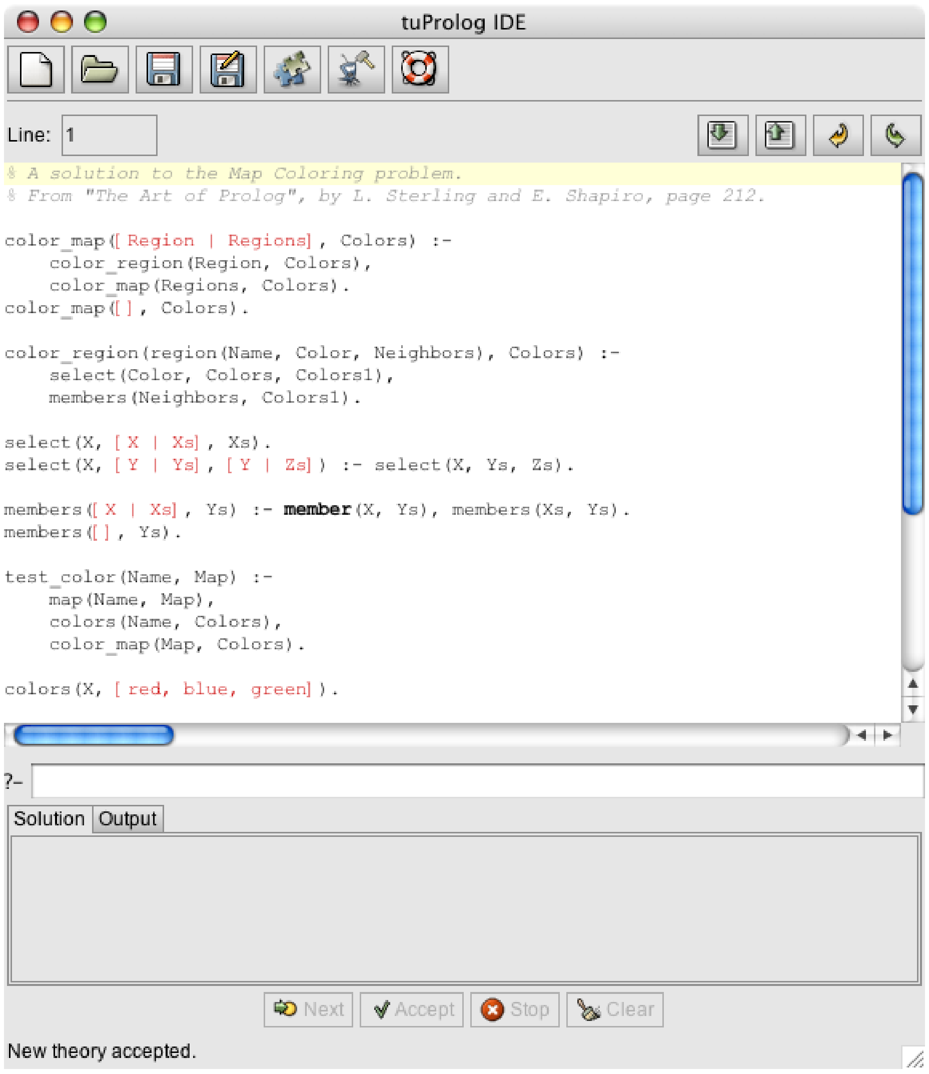
- an answer pane, where answers and output informations are visualized. Answers to Prolog queries are composed by both solutions, showed in a free form within a read-only text area, and bindings, displayed in tabular form. The output tab provides a read-only view on the standard output where informations are possibly written by Prolog



```
% A solution to the Map Coloring problem.
% From "The Art of Prolog", by L. Sterling and E. Shapiro, page 212.

color_map([ Region | Regions] , Colors) :-
    color_region(Region, Colors),
    color_map(Regions, Colors).
color_map([ ] , Colors).

color_region(region(Name, Color, Neighbors), Colors) :-
    select(Color, Colors, Colors1),
    members(Neighbors, Colors1).

select(X, [ X | Xs] , Xs).
select(X, [ Y | Ys] , [ Y | Zs] ) :- select(X, Ys, Zs).

members([ X | Xs] , Ys) :- member(X, Ys), members(Xs, Ys).
members([ ] , Ys).

test_color(Name, Map) :-
    map(Name, Map)
    colors(Name, Colors),
    color_map(Map, Colors).

colors(X, [ red, blue, green] ).
```

```
Error setting theory: Syntax Error at/before line 21
```

Figure 1.2: A syntax error is found when setting the content of the editor area as the new engine's theory.



```
% A solution to the Map Coloring problem.
% From "The Art of Prolog", by L. Sterling and E. Shapiro, page 212.

color_map([ Region | Regions] , Colors) :-
    color_region(Region, Colors),
    color_map(Regions, Colors).
color_map([ ] , Colors).

color_region(region(Name, Color, Neighbors), Colors) :-
    select(Color, Colors, Colors1),
    members(Neighbors, Colors1).

select(X, [ X | Xs] , Xs).
select(X, [ Y | Ys] , [ Y | Zs] ) :- select(X, Ys, Zs).

members([ X | Xs] , Ys) :- member(X, Ys), members(Xs, Ys).
members([ ] , Ys).

test_color(Name, Map) :-
    map(Name, Map),
    colors(Name, Colors),
    color_map(Map, Colors).

colors(X, [ red, blue, green] ).
```

New theory accepted.

Figure 1.3: The syntax error is removed and the Set Theory operation succeeds.

programs, by means of the I/O predicates supplied by the `IOLibrary`.[1] Control buttons are also provided to iterate through possibly multiple solutions, clear the bindings and output panes, and export tabular data in a convenient CSV format.

Goals are asked through the query input box, and answers (bindings and solutions) are provided in the related text area. Query and answers are traced in a proper chronological history, that can be explored by means of **Up** and **Down** arrow keys from the query input textfield. When open alternatives are found solving a goal, the Next and Accept buttons are enabled in the answer pane to interact with the engine, in order to let the user specify if the current solution is accepted or if other alternatives need to be explored.

Note that the theory contained in the currently selected edit pane does not have to be explicitly feeded to the Prolog engine before it could be possible to solve queries against that theory's knowledge base. In fact, any time a goal is asked to be solved, the theory contained in the active edit area is automatically feeded to the engine if its knowledge base has been modified since the last solved goal. (This obviously happens also on the first time a query is asked.) However, whenever the engine's theory is modified by other means than the editor, it does need to be explicitly acquired and presented to the programmer in the text area. In fact, if the theory in the engine is augmented by a call to the predicate `consult/1` issued from a query, for example, the contents of the newly consulted theory will not be automatically inserted in the editor: when the programmer needs an up-to-date view of the knowledge base contained in the underlying tuProlog engine, its display has to be explicitly triggered by means of the GetTheory button, available in the editing area.

An example of the user interaction involving multiple solutions is shown in the following sequence of figures: in Figure **??**, the user issued the query `test_color(test, X).`, using the knowledge base written in the edit area (a solution to the Map Coloring problem,[2] with a test map composed of six areas). The first solution is displayed, and multiple open alternatives can be explored: in Figure **??**, the user asked to get the next possible solution by pressing the Next button, and another solution is provided; finally, in Figure **??**, the user, after having explored the first two solutions, accepts the

[1] The information written on standard output by methods invoked on Java objects from the `JavaLibrary` – for instance using the `stdout` object – are not displayed on this view.

[2] The problem is to color a planar map so that no two adjoining regions have the same color. A famous conjecture was proved in 1976 showing that four colors are sufficient to color any planar map.

third one by pressing the Accept button. During the resolution of a goal, all the theory-related buttons are disabled, included the Library Manager button, since each library can have its theory to be feeded into the engine.

Near to the Next and Accept buttons, a Stop button is found, providing the user with a means to halt the engine if a computation takes too long or a bug in the knowledge base feeded to the engine results in an infinite loop.

Finally, a Clear button is provided, with the aim of allowing the user to clear the bindings and output panes when they get overfull with informations. The button is enabled only when the proper tabs are selected.

1.3 Debug Informations

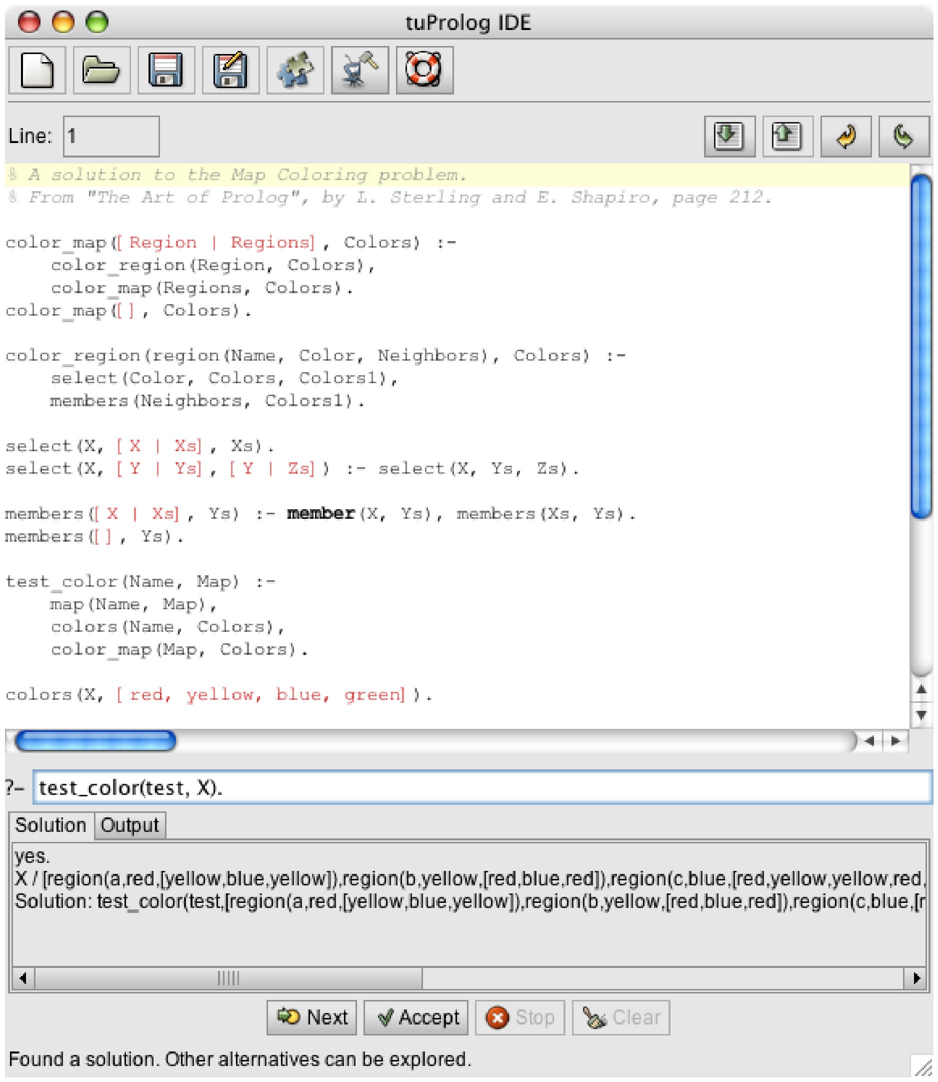
By pressing the View Debug Information button, a new window is opened, providing a view on the warnings, produced by events such as the attempt at redefining a library predicate, and the spy information, concerning basic steps of the engine computation and state, possibly supplied by the engine during a goal demonstration: Warnings are always active; in order to activate the spy information notification, the `spy/0` built-in predicate (provided by `BasicLibrary`) must be issued; `nospy/0` can be used to stop this notification. As an example, Figure **??** shows the content of the spy information view after the execution of a goal involving the activation of spy inspection.

It is worth noting that a computation may contain a huge number of traced steps. For this reason, a toolbar at the top of the window allows to collapse and expand all nodes in the spy information pane, or to expand and collapse selected nodes only. Finally, the content of the warnings and spy panes can be cleared using the Clear button at the leftmost end of the toolbar.

1.4 Dynamic library management

A tuProlog engine can be extended by loading any number of libraries, each provinding a specific set of built-in predicates and functors, and a related theory. The tuProlog IDE allows a dynamic management of libraries through a GUI dialog, which can be displayed by pressing the Open Library Manager button in the toolbar. The Library Manager dialog is shown in Figure **??**.

This dialog displays a list of the libraries currently loaded into the tuProlog engine. For a new instance of the engine, that list will typically contain the four standard libraries coming with the application core, that is `BasicLibrary`, `IOLibrary`, `ISOLibrary`, `JavaLibrary`, along with their



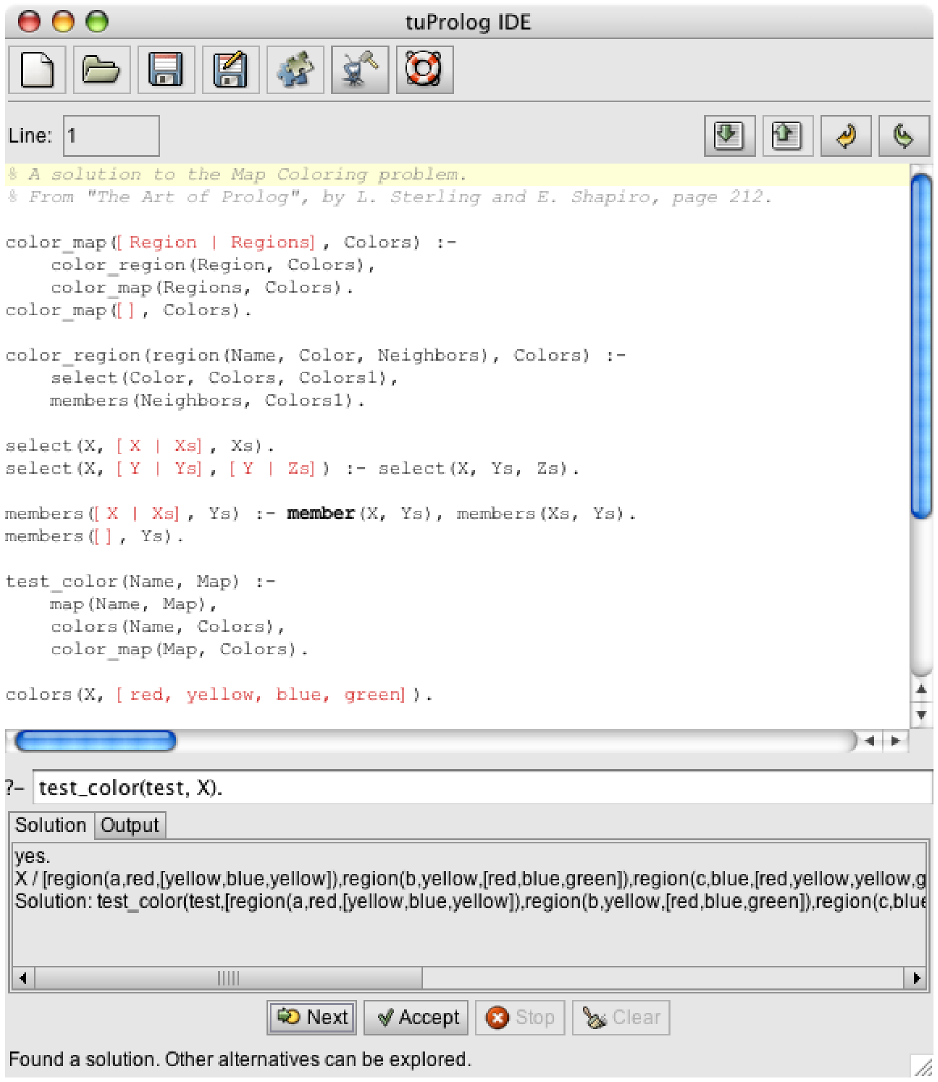
```
% A solution to the Map Coloring problem.
% From "The Art of Prolog", by L. Sterling and E. Shapiro, page 212.

color_map([ Region | Regions] , Colors) :-
    color_region(Region, Colors),
    color_map(Regions, Colors).
color_map([ ] , Colors).

color_region(region(Name, Color, Neighbors), Colors) :-
    select(Color, Colors, Colors1),
    members(Neighbors, Colors1).

select(X, [ X | Xs] , Xs).
select(X, [ Y | Ys] , [ Y | Zs] ) :- select(X, Ys, Zs).

members([ X | Xs] , Ys) :- member(X, Ys), members(Xs, Ys).
members([ ] , Ys).

test_color(Name, Map) :-
    map(Name, Map),
    colors(Name, Colors),
    color_map(Map, Colors).

colors(X, [ red, yellow, blue, green] ).
```

```
?- test_color(test, X).
```

```
yes.
X / [region(a,red,[yellow,blue,yellow]),region(b,yellow,[red,blue,red]),region(c,blue,[red,yellow,yellow,red,
Solution: test_color(test,[region(a,red,[yellow,blue,yellow]),region(b,yellow,[red,blue,red]),region(c,blue,[r
```

```
Found a solution. Other alternatives can be explored.
```

Figure 1.4: The user issued a query `test_color(test, X).` and the first solution is displayed.



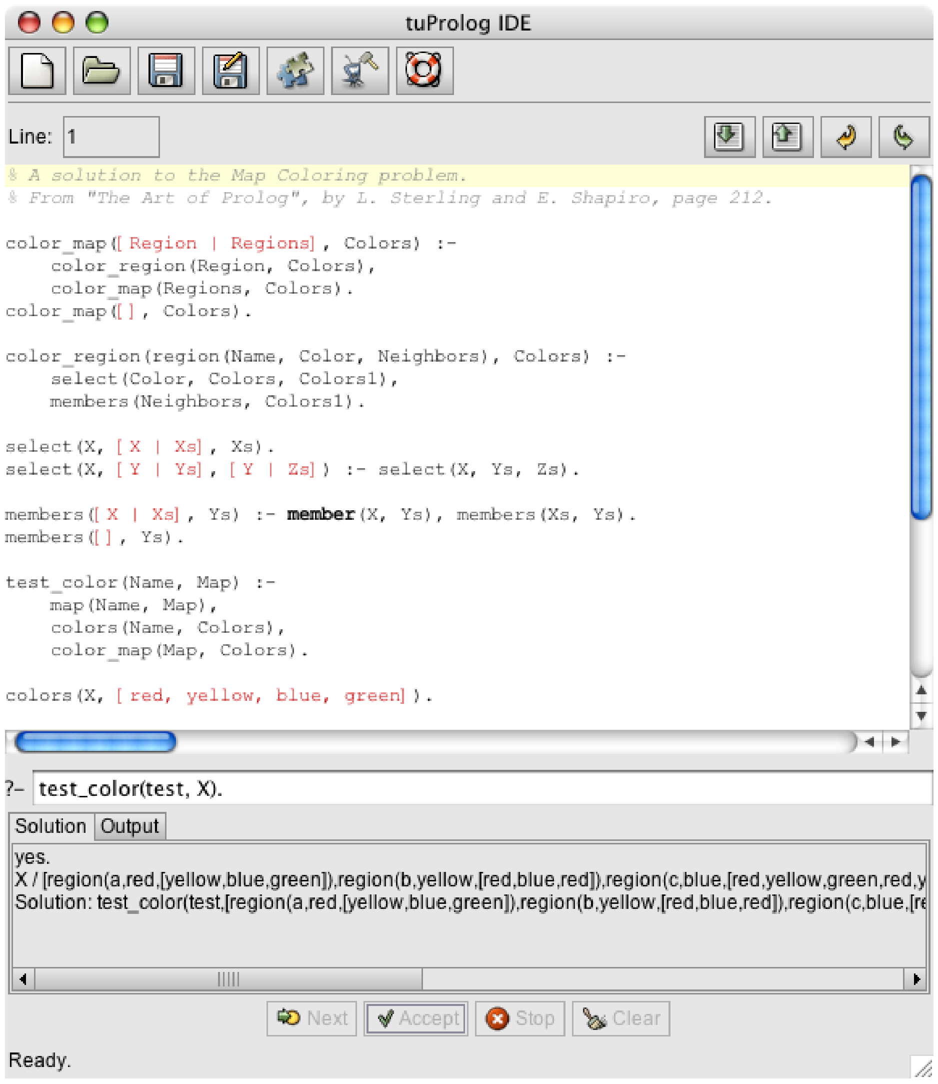
```
% A solution to the Map Coloring problem.
% From "The Art of Prolog", by L. Sterling and E. Shapiro, page 212.

color_map([ Region | Regions] , Colors) :-
    color_region(Region, Colors),
    color_map(Regions, Colors).
color_map([ ] , Colors).

color_region(region(Name, Color, Neighbors), Colors) :-
    select(Color, Colors, Colors1),
    members(Neighbors, Colors1).

select(X, [ X | Xs] , Xs).
select(X, [ Y | Ys] , [ Y | Zs] ) :- select(X, Ys, Zs).

members([ X | Xs] , Ys) :- member(X, Ys), members(Xs, Ys).
members([ ] , Ys).

test_color(Name, Map) :-
    map(Name, Map),
    colors(Name, Colors),
    color_map(Map, Colors).

colors(X, [ red, yellow, blue, green] ).
```

```
?- test_color(test, X).
```

```
yes.
X / [region(a,red,[yellow,blue,yellow]),region(b,yellow,[red,blue,green]),region(c,blue,[red,yellow,yellow,g
Solution: test_color(test,[region(a,red,[yellow,blue,yellow]),region(b,yellow,[red,blue,green]),region(c,blue
```

Found a solution. Other alternatives can be explored.

Figure 1.5: The user issued a Next command and got another solution.

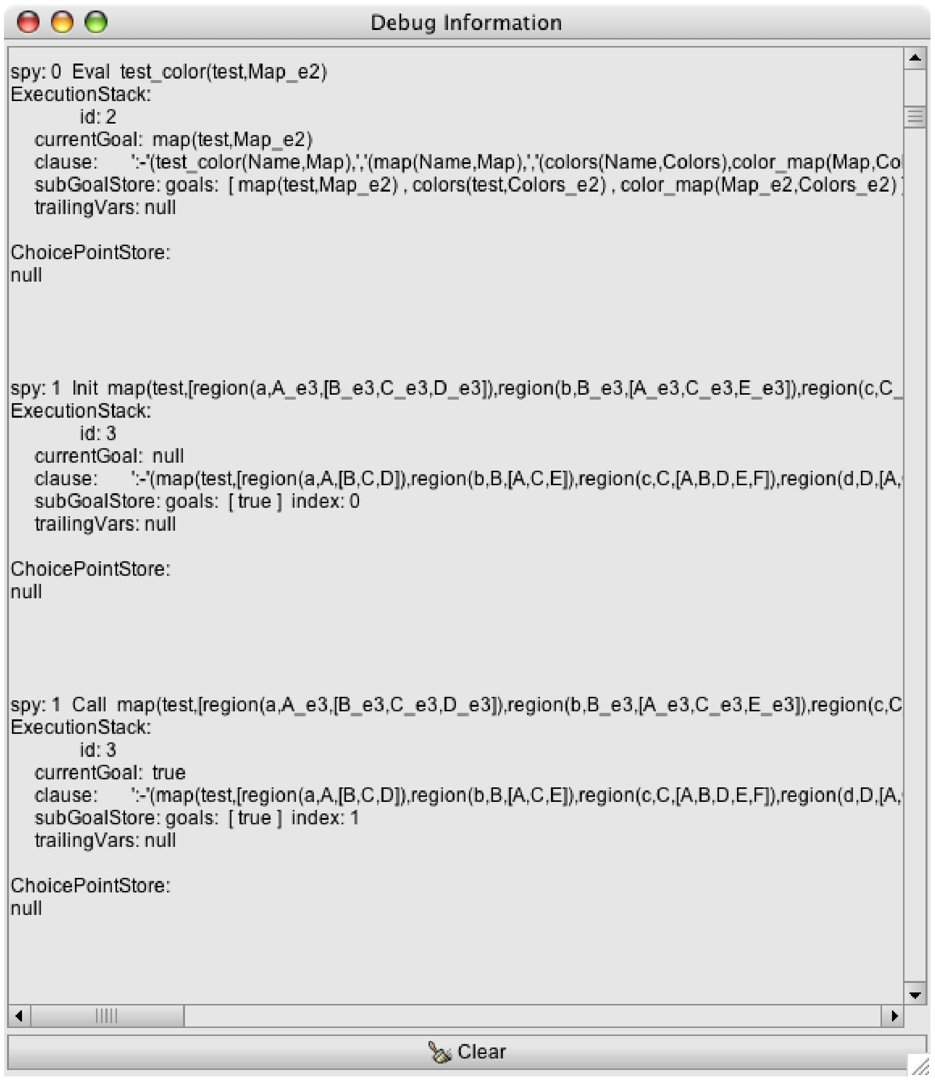


```
% A solution to the Map Coloring problem.
% From "The Art of Prolog", by L. Sterling and E. Shapiro, page 212.

color_map([ Region | Regions] , Colors) :-
    color_region(Region, Colors),
    color_map(Regions, Colors).
color_map([ ] , Colors).

color_region(region(Name, Color, Neighbors), Colors) :-
    select(Color, Colors, Colors1),
    members(Neighbors, Colors1).

select(X, [ X | Xs] , Xs).
select(X, [ Y | Ys] , [ Y | Zs] ) :- select(X, Ys, Zs).

members([ X | Xs] , Ys) :- member(X, Ys), members(Xs, Ys).
members([ ] , Ys).

test_color(Name, Map) :-
    map(Name, Map),
    colors(Name, Colors),
    color_map(Map, Colors).

colors(X, [ red, yellow, blue, green] ).
```

```
?- test_color(test, X).
```

```
yes.
X / [region(a,red,[yellow,blue,green]),region(b,yellow,[red,blue,red]),region(c,blue,[red,yellow,green,red,y
Solution: test_color(test,[region(a,red,[yellow,blue,green]),region(b,yellow,[red,blue,red]),region(c,blue,[re
```

Figure 1.6: The user accepted the third solution by pressing the Accept button.


Debug Information

```
spy: 0  Eval  test_color(test,Map_e2)
ExecutionStack:
           id: 2
    currentGoal:  map(test,Map_e2)
    clause:      ':-'(test_color(Name,Map),','(map(Name,Map),','(colors(Name,Colors),color_map(Map,Co
    subGoalStore: goals:  [ map(test,Map_e2) , colors(test,Colors_e2) , color_map(Map_e2,Colors_e2)
    trailingVars: null

ChoicePointStore:
null



spy: 1  Init  map(test,[region(a,A_e3,[B_e3,C_e3,D_e3]),region(b,B_e3,[A_e3,C_e3,E_e3]),region(c,C_
ExecutionStack:
           id: 3
    currentGoal:  null
    clause:      ':-'(map(test,[region(a,A,[B,C,D]),region(b,B,[A,C,E]),region(c,C,[A,B,D,E,F]),region(d,D,[A,
    subGoalStore: goals:  [ true ]  index: 0
    trailingVars: null

ChoicePointStore:
null



spy: 1  Call  map(test,[region(a,A_e3,[B_e3,C_e3,D_e3]),region(b,B_e3,[A_e3,C_e3,E_e3]),region(c,C
ExecutionStack:
           id: 3
    currentGoal:  true
    clause:      ':-'(map(test,[region(a,A,[B,C,D]),region(b,B,[A,C,E]),region(c,C,[A,B,D,E,F]),region(d,D,[A,
    subGoalStore: goals:  [ true ]  index: 1
    trailingVars: null

ChoicePointStore:
null
```

Figure 1.7: Debug Information View after the execution of a goal.

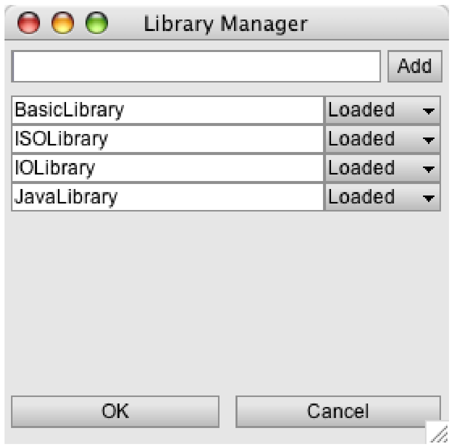


Figure 1.8: The Library Manager dialog.

current status. The user can add a library to the Library Manager simply by providing the fully qualified name of the library's class in the textfield on the top of the dialog, then pressing the Add button: the added library will be displayed with an initial Unload status. The user can further select the status of each library in the list, and commit changes to the tuProlog engine by pressing the OK button, or dismiss the dialog by pressing the Cancel button.

The library manager is also capable of updating itself accordingly to the events of libraries load and unload fired by the tuProlog engine. Such events are triggered by the use of the `load_library/1` and `unload_library/1` predicates or directives in query issued or theories feeded to the engine. So, if an user asks to solve the goal `load_library('TestLibrary'), test(X).`, for example, the manager would immediately reflect the change occurred in the engine's libraries pool, adding a new entry if `TestLibrary` had not been previously loaded or, if necessary, changing the library's entry status to show the result of the loading action.

Both the action of adding a library to the manager and the action of loading a library into the engine can fail. If, for example, the classname provided does not identify a tuProlog library (i.e. it identifies a class not extending the `alice.tuprolog.Library` class) or the identified class does not exist, an appropriate message will appear in the status bar at the bottom of the dialog. When adding or loading a library, please remember that every class needed by that library must be in the classpath in order to have the library correctly added to the manager's list or loaded into the engine.