

# **RATSY**

## **Requirements Analysis Tool with Synthesis**

Version 2.1

### **Authors**

Roderick Bloem, Roberto Cavada, Alessandro Cimatti, Karin Greimel,  
Georg Hofferek, Robert Koenighofer, Alessandro Mariotti, Ingo Pill,  
Marco Pensallorto, Marco Roveri, Viktor Schuppan, Richard Seeber,  
Simone Semprini, Andrei Tchaltsev, and Martin Weiglhofer

## Notices

For information, contact RATS<sub>Y</sub> ([ratsy@list.fbk.eu](mailto:ratsy@list.fbk.eu)).

The first version of this tool (RAT) has been developed within the PROSYD European project, contract number 507219 (<http://www.prosyd.org>). The current version (RATS<sub>Y</sub>) has been created within the COCONUT European project, contract number 217069 (<http://www.coconut-project.eu>), within the DIAMOND European project, contract number 248613 (<http://www.fp7-diamond.eu/>), and within the Provincia Autonoma di Trento project EMTELOS.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright 2005, 2009, 2010 FBK-irst and Graz University of Technology. All rights reserved.

# Contents

Contents .....	iii
Table of Figures .....	iv
List of Tables .....	vi
1 RATS Y Users Manual.....	1
1.1 Running RATS Y .....	1
1.2 Property Assurance in RATS Y .....	3
The Main Window .....	4
Traces and their management.....	6
An Example .....	9
1.3 Property Simulation in RATS Y .....	12
The Main Window .....	14
The Analysis Window.....	15
An example .....	16
1.4 Property Realizability and Synthesis in RATS Y .....	20
Realizability Problem .....	21
Specifying a Realizability Problem.....	22
The Main Window .....	22
Synthesis.....	24
The Automaton Editor .....	26
1.5 Simulating and Debugging Specifications using Games .....	30
How to play a Game .....	30
The Game Log Window .....	32
Integration with the Automaton Editor .....	32
Specifying Design Intent.....	33
Additional Features for the Normal Game .....	34
Additional Features for the Countergame .....	34
Example.....	35
2 RATS Y Architecture.....	41
2.1 Architecture and Implementation Notes .....	41
2.2 Architectural Patterns .....	42
The Model-View-Controller pattern.....	43
The Observer pattern .....	44
2.3 Software Structure.....	44
Tools Stubs.....	45
A vertical view over the Software Structure.....	46
3 RATS Y Installation and Distribution.....	49
3.1 Installing the Binary Distribution .....	49
3.2 Installing the Source Distribution .....	49
3.3 Running MARDUK .....	50
3.4 Licensing .....	50
4 References.....	53

## Table of Figures

Figure 1 - RATS Y- Main window. ....	2
Figure 2 - RATS Y- New project wizard. ....	2
Figure 3 - RATS Y- New project wizard, project data. ....	3
Figure 4 - Property Assurance main window. ....	4
Figure 5 - Creating signals, requirements. ....	5
Figure 6 - Creating possibilities and assertions. ....	6
Figure 7 - Verification panels.....	7
Figure 8 - An example of trace visualization. ....	7
Figure 9 - An example of trace visualization. ....	8
Figure 10 - Editing a category.....	9
Figure 11 - Editing a trace.....	9
Figure 12 - Counter - initial specification. ....	10
Figure 13 - Counter - checking an assertion.....	11
Figure 14 - Counter - fixing the specification.....	12
Figure 15 - Counter - checking a possibility. ....	13
Figure 16 - Counter - traces of the session.....	13
Figure 17 - Property Simulation Main Window.....	14
Figure 18 - Property Simulation Evaluation Analysis Window.....	16
Figure 19 - Create a project for Property Simulation. ....	17
Figure 20 - Property Simulation Start Window.....	17
Figure 21 - Witness for property $G(r \mapsto F(a))$ . ....	18
Figure 22 - Analysis of trace for property $G(r \mapsto F(a))$ .....	18
Figure 23 - Ask for a request on signal r. ....	19
Figure 24 - Witness with request for property $G(r \mapsto F(a))$ . ....	19
Figure 25 - Witness for property $G(r \mapsto F(a)) \&\& F(r)$ . ....	20
Figure 26 - Witness for property $(G(r \mapsto F(a))) \&\& (F(r))$ .....	20
Figure 27 - Shaping the trace.....	21
Figure 28 - Witness for shaped trace request.....	21
Figure 29 - Specification of an environment signal in RATS Y. ....	22
Figure 30 - Specification of a system guarantee property in RATS Y.....	23
Figure 31 - The Realizability window in RATS Y. ....	23
Figure 32 - The Realizability window in RATS Y. ....	24
Figure 33 - Create a new automaton. ....	27
Figure 34 - Edit the properties of a state in the automaton. ....	27

Figure 35 - Specify the transition condition for an edge. ....	28
Figure 36 - The main window of the automaton editor. ....	29
Figure 37 - The Game window in RATSY.....	31
Figure 38 - The specification used for Game demo.....	36
Figure 39 - A possible simulation run. ....	36
Figure 40 - The specified design intent. ....	37
Figure 41 - The new specification containing the desired behavior. ....	37
Figure 42 - The countergame containing the countertrace. ....	38
Figure 43 - The state of the play in an automaton. ....	39
Figure 44 - RATSY- Software parts and collocation .....	42
Figure 45 - RATSY- Software Structure .....	44
Figure 46 - RATSY- Hierarchy of main software entities .....	46

# List of Tables





# 1 RATS Y Users Manual

The tool RATS Y fulfills the need for a proper technological support to formal methods in the setting of requirements analysis and synthesis by providing its users with the integration of four sets of functionalities: Property Simulation, Property Assurance, Property Realizability and Synthesis, and Property Debugging using Games. In this section we show how to interact with RATS Y in order to accomplish the tasks related to these four methodologies.

All the examples in the following sections are written in the Verilog flavor of PSL as from [12], the language supported by the verification engines VIS and NUSMV.

Some of the screenshots in this manual (especially in the sections about Property Assurance and Property Simulation) were taken from a previous version of the tool (RAT). Thus, they might look slightly different than what you will see in the current version (RATS Y). This should, however, not affect understanding of the respective sections.

---

## 1.1 Running RATS Y

RATS Y can be executed from the command line by the following command

<b>ratsy</b> - <i>Launches the python interpreter to execute the RATS Y program</i>	Command
---	---------

```
ratsy [-h|--help] [-v|--version]
      [-f <FILE.rat> | --project = <FILE.rat>]
```

Command Options:

-h	Prints the command usage.
-v	Prints the program version.
-f <FILE.rat>	Loads the given project file

Figure 1 shows the start-up screen-shot of RATS Y when the tool is launched without any project as argument.

The unit of interaction with RATS Y is the *project*, i.e. a collection of formal properties and results of verification checks. The relevance of the role of a project, as an object with a state that can be saved and reloaded is clear as far as Property Assurance and Property Realizability are regarded: the user that builds formal specifications and inspects their quality, must have the possibility to work in different sessions and of saving the results of the work performed from session to

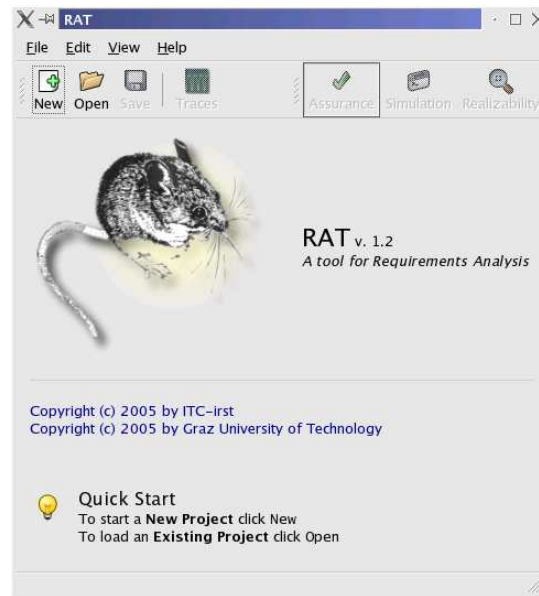


Figure 1: RATSY- Main window.

session. With Property Simulation, such a feature could seem less relevant, but the value of having the possibility of saving simulation sessions (i.e. the properties simulated and the connected traces) shows clearly if we think of long time consuming work sessions and of the importance of having a quick reference to their results.

Through the menu **File** or the command **New** in the tool bar it is possible to access the wizard for the creation of new projects, shown in Figure 2, select the kind of project, and specify the details of the project entering the data in the fields shown in Figure 3.



Figure 2: RATSY- New project wizard.

As a result of the integration of Property Simulation, Assurance and Realizability into RATSY (rather than simply juxtaposing them), it is possible to shift between these three kinds of projects at any time, and to load properties, for example, from Property Assurance into Property Simulation or Property Realizability. A project hence sums up all the history of a design development process, from the initial explorations of properties prototypes, to the definition of a set of requirements, from the inspection of requirements adherence to the intended meaning, to the possi-



Figure 3: RATSY- New project wizard, project data.

ble use of simulation to perform a fine grained inspection of properties coming from Property Assurance, and to checking the interplay between controlled and uncontrolled signals and their requirements with Realizability.

Once a project has been created, the user can proceed as described in Sections 1.2, 1.3 and 1.4.

---

## 1.2 Property Assurance in RATSY

RATSY enacts the Property Assurance Methodology (see [2] Section 2.2) by supporting the users in Property Assurance related tasks; RATSY provides a proper framework for managing set of properties, a user-friendly interface towards verification engines, and a proper framework for managing the results of Property Assurance proof obligations. In this section we describe how to interact with the tool by following a typical use case, which encompasses the following steps:

- editing of a project;
  - editing of signals
  - editing of requirements
  - editing of possibilities
  - editing of assertions
- verification
  - activation of the checks
  - management of traces

In the setting of Property Assurance, *Projects* are the entities that correspond to the ensemble of a specification together with the results obtained by the connected proof obligations. The building blocks of a specification in the Property Assurance Methodology are *requirements*, *possibilities* and *assertion*, all of which are properties formally expressed on a set of atomic symbols called *signals*. Following the

methodology, given a specification, some proof obligation need to be discharged; in [2] Section 2.2 it has been shown how these proof obligations can be mapped onto SAT technology: the tool provides an interface towards this technology and communicates the results of the performed verification checks by means of extended waveforms called *traces* that show the evolution of the values of signals in possible models of the system under specification.

## The Main Window

RATSY main window when in Property Assurance mode is shown in Figure 4. In the upper part of the body of the window there are the tables for the management of signals and requirements; in the middle the are the tabbed tables for the management of possibilities and assertions (on the left), and the control panel for the verification tasks (on the right); the bottom of the window is occupied by a text box showing the output of the verification activity.

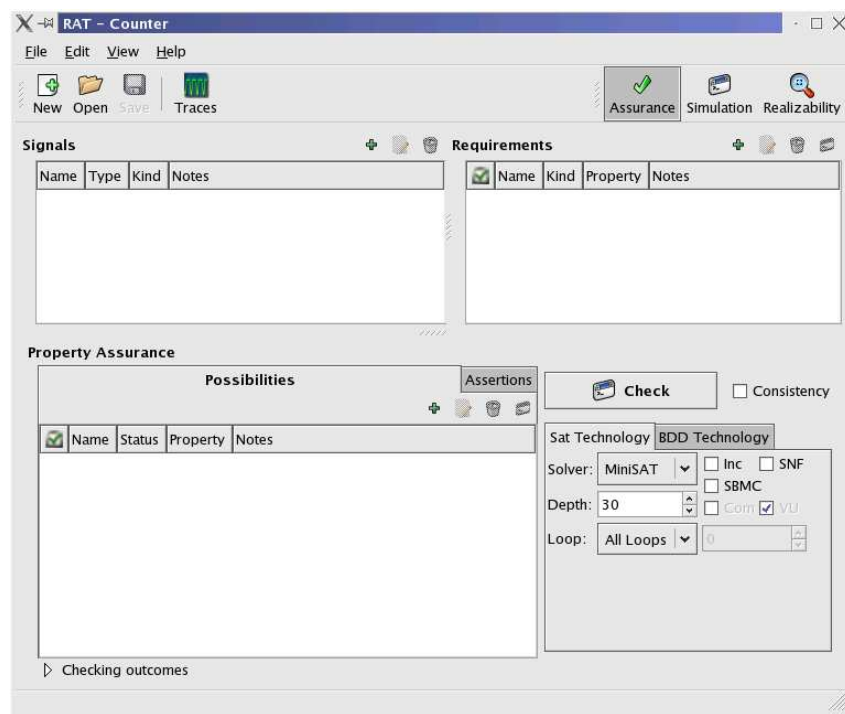


Figure 4: Property Assurance main window.

**Adding and modifying elements of a project.** The activities of adding, editing and removing items from the sets of signals, requirements, possibilities and assertions follow the same pattern regardless the class the items belong to. The screen-shots in Figure 5 and 6 show the windows for creating a new signal, a new requirement, a new possibility and a new assertion respectively, all of which are accessible by clicking on the first one among the buttons on the top right of the table of the proper class.

Note that in Property Realizability signals are distinguished of being System or Environment. Similarly, requirements are distinguished of being Assumption or Guarantee. For Property Assurance and Property Simulation these distinctions are of no importance and therefore ignored.

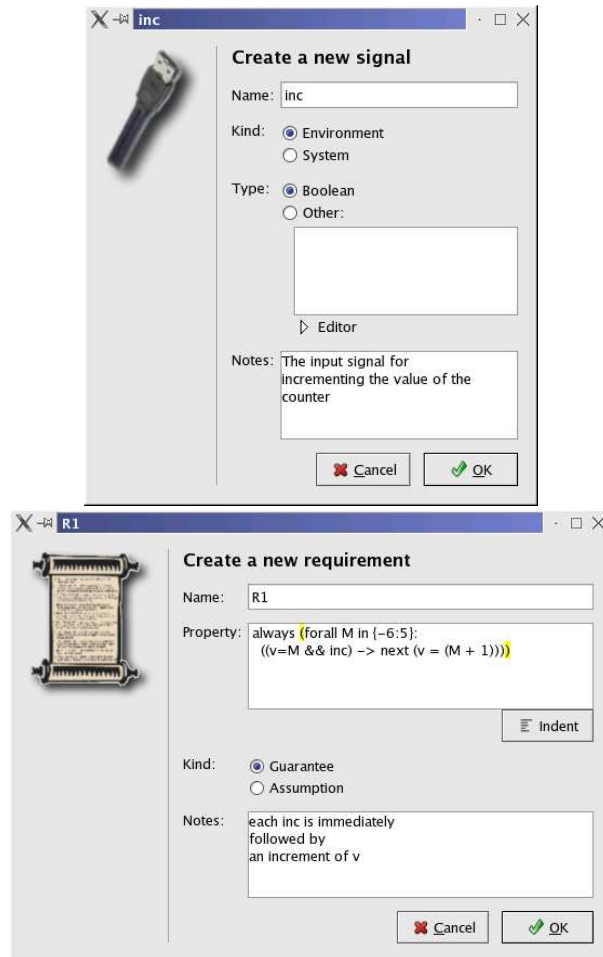


Figure 5: Creating signals, requirements.

Once an item is created, it is shown in the table of its class and it is possible to modify or to delete it by clicking on the proper button on the table of the class of the item. A window similar to the one used for creation is used for editing, and a warning window will ask for the user's confirmation before deleting an item. Multiple selection is allowed (Ctrl keyboard button pressed when left-clicking with the mouse on the desired items) and hence is possible to open the editing windows of several items at one time, or to delete more than one item at one time. Multi-row editing and parenthesis highlighting are provided to ease the input of properties and to make more effective their visualization. Notice that, all the tasks that can be performed on signals, requirements, possibilities, assertion, traces and categories are accessible also through pop-up menus that shows when the user right-click with the mouse on an item; the pop-up menus offer also selection facilities like “select all”, “deselect all” and “invert selection”.

Since, as pointed out in [2] Section 2, it may be of great use to simulate a property when the results of a Property Assurance check are not of ease comprehension,



Figure 6: Creating possibilities and assertions.

the user is provided with the possibility of loading an item that belongs to requirements, possibilities or assertions into Property Simulation mode; this can be accomplished by selecting the desired items and clicking on the last one among the four buttons on the top right corner of the proper table, or by selecting the voice Load into Simulation from the pop-up menu accessible by right clicking on the selected items. The logical conjunction of the selected items is copied in the Property text box in the Property Simulation mode (See Section 1.3).

**Verification** The verification tabbed panel, on the middle right of the window, provides the user with control on the execution of the verification engine used to perform Property Assurance related checks. The two tabs, shown in Figure 7, allow to choose among SAT-based BMC techniques or BDD-based MC techniques, and to set the respective options. As far as SAT-based BMC is regarded, it is possible to choose which SAT solver to use, whether incremental techniques should be used, the depth of the BMC problem generated, and the value for the loop back. With regard to BDD-based MC, the user can define the partition method, whether using Cone of Influence techniques, and which kind of dynamic reordering should be used, if any. For more details on the meaning of these options, the user can refer to the user manual of NUSMV [8].

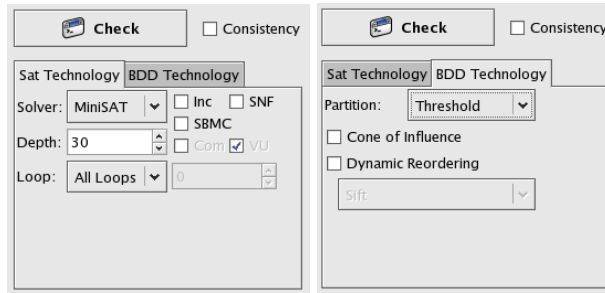


Figure 7: Verification panels.

## Traces and their management

The results of verification checks are shown as traces, which are shown as new tabs beside the Output tab as depicted in Figure 8.

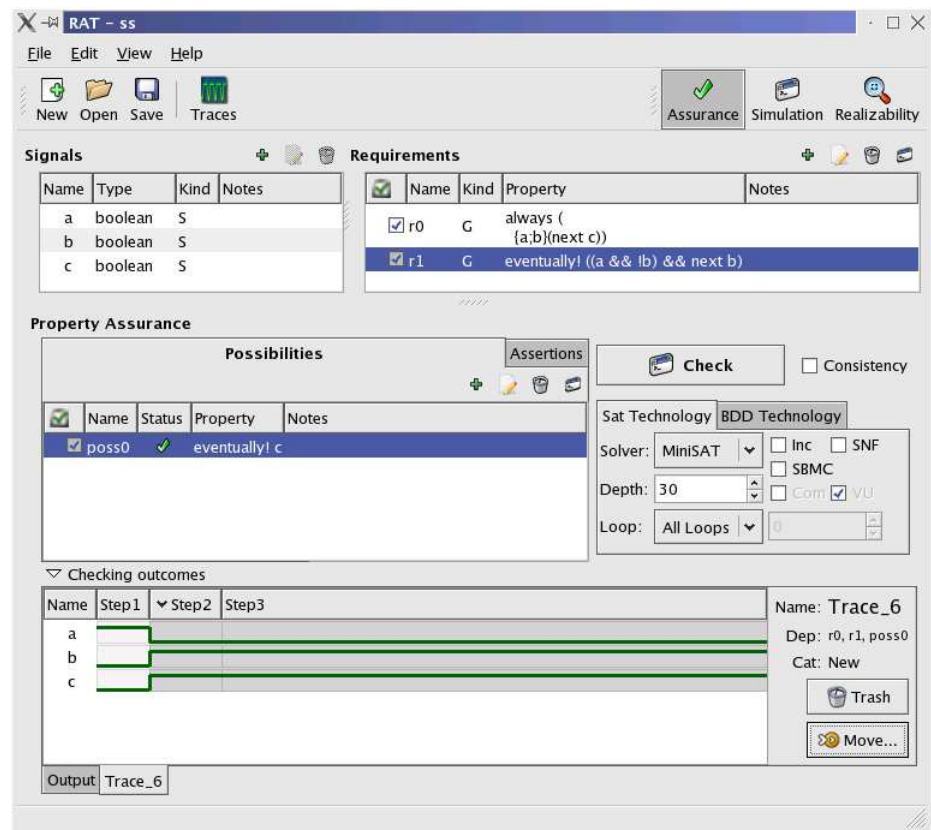


Figure 8: An example of trace visualization.

Each trace has a name and is connected to the requirements and the possibilities/assertions it has been generated from, i.e. those that were selected to perform the check of which the trace is the result. These data allow to track the dependencies among the traces and the other elements of the project; for example, knowing which requirements a trace depends on allows the system to signal it as out of date or no longer meaningful if some changes have been performed to one of the requirements the trace depends on.

In Figure 8, the trace shown is composed by an initial step followed by an infinite repetition of the second step, i.e. a loop. Loops are signaled by a little black

arrow close to the name of the step they start from. Color of steps changes to help depicting the finite prefix and the infinite loop in traces, light gray for the former, dark gray for the latter.

To ease their management and to reflect the typical use case of Property Assurance, traces are organized in different *categories* among which the following system categories are provided:

**New:** the category where traces generated in the current session are stored by default;

**Default:** the category where up to date traces that have been generated in previous sessions are stored;

**Out of date:** the category where out of date traces are stored (a trace is out of date when some element in its dependencies have been deleted or modified);

**Trash:** the category of traces the user scheduled for deletion.

A simple way of managing traces with respect to categories is provided by the buttons Trash and Move on the right of each trace in the main window, as shown in Figure 8.

Clicking on the button Traces in the tool-bar, it is possible to access the window of the *trace manager*, as shown in Figure 9, which allows the user to manage traces by editing the associated data, moving them from a category to another category, deleting them, creating new categories and editing the data connected to categories.

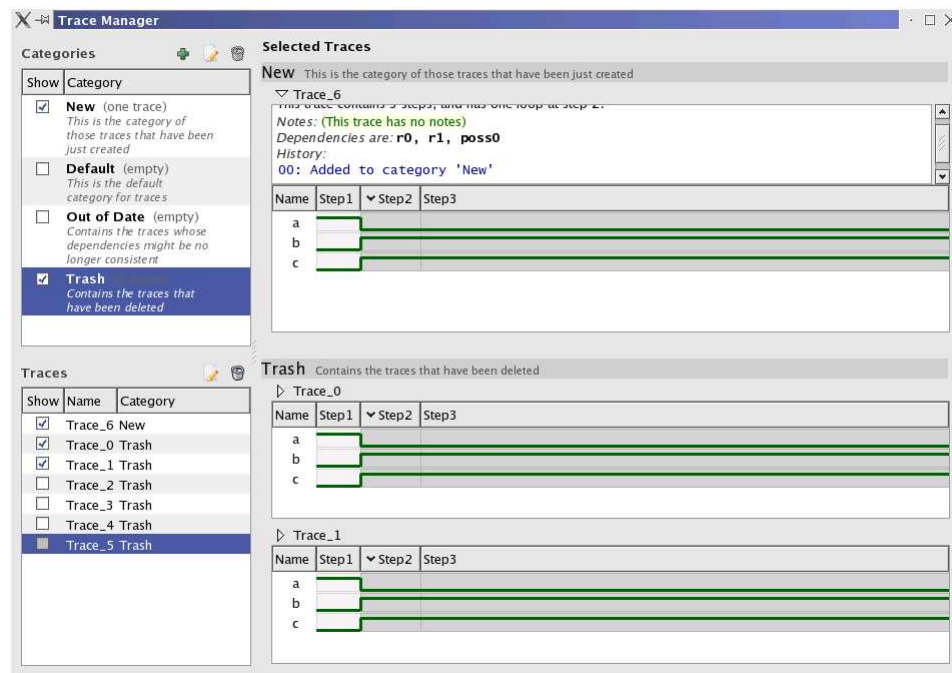


Figure 9: An example of trace visualization.

At the top left corner of the trace manager window the list of categories is shown, where each category has a name and a Description; it is possible to select more than one category and, on selection, the contained traces are shown on the right part of the window grouped under the name of the category they belong to. In the left

bottom corner of the window there is the list of the names of the traces contained in the selected categories, by selecting or de-selecting names it is possible to show or hide traces in the right part. As shown, each trace is visualized together with its complete data that comprise a brief description, the notes entered by the user, the list of dependences and the history (when the trace was generated, etc.). Categories and traces tables on the left part of the window, allow the users to edit, delete or add items, in Figure 10 and Figure 11 the editing dialog for categories and traces are shown.

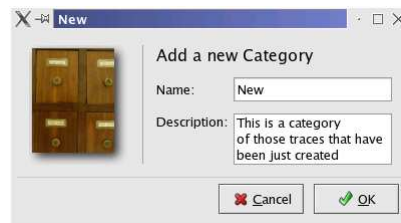


Figure 10: Editing a category.

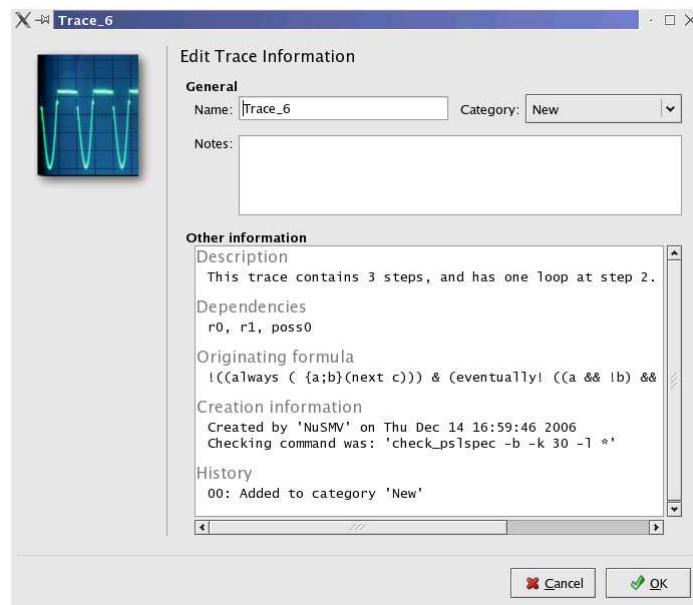


Figure 11: Editing a trace.

## An Example

In this section we work out a simple but meaningful example that covers the most relevant Property Assurance features of RATSY, and link together in a cohesive view the usage information given in the previous section.

The example we are going to tackle is the specification of a bounded counter (an instantiation of what described in [2] Section 2.2); a first naïve specification could be the one shown in Figure 12.

The specification is based on the following signals:

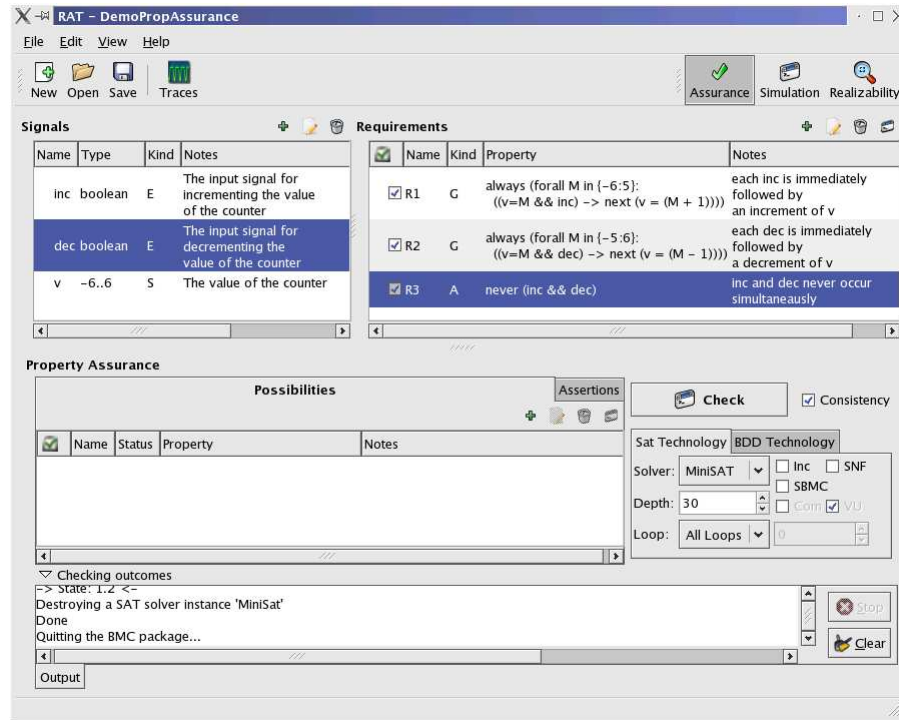


Figure 12: Counter - initial specification.

- inc:** the signal that models the issuing of increment operations
- dec:** the signal that models the issuing of decrement operations
- v:** the signal (integer valued) that models the value of the counter

this signals are shown in the Signals table together with their type and notes.

The Requirements table collects three requirements that constitute an initial specification of the functional behavior of the counter, and of the assumptions on the environment

- R1:** prescribes that any increment operation is immediately followed by a unit increment in the value of the counter
- R2:** prescribes that any decrement operation is immediately followed by a unit decrement in the value of the counter
- R3:** states that increment and decrement operations must not occur simultaneously (this is a constraint on the environment)

Once this initial specification is entered by the user, it is possible to proceed and check it for consistency, i.e. checking that the requirements are not mutually contradictory. This can be achieved by selecting all the requirements, by ticking the check box Consistency check, and by clicking on the Check button in the control panel at the top. Figure 12 shown the result of this check is positive: the output from the verification engine, shown in the tab Output, reports that the run of the engine has completed successfully and no warning message is issued by RATSY. As shown in the control panel, this check has been performed using SAT technology with a depth of the problem equal to 30, and checking for all possible loop-backs.

Now that we have an initial consistent specification, we can start analyzing it and check if it describes exactly the behavior we have in mind.

The first step can be that of checking that the value of our counter is always coherent with the inputs received. In particular, we want to be sure that if no operation is issued, the value of the counter does not change, whatever the value is; this is the meaning of assertion A1 shown in the Assertions table in Figure 13.

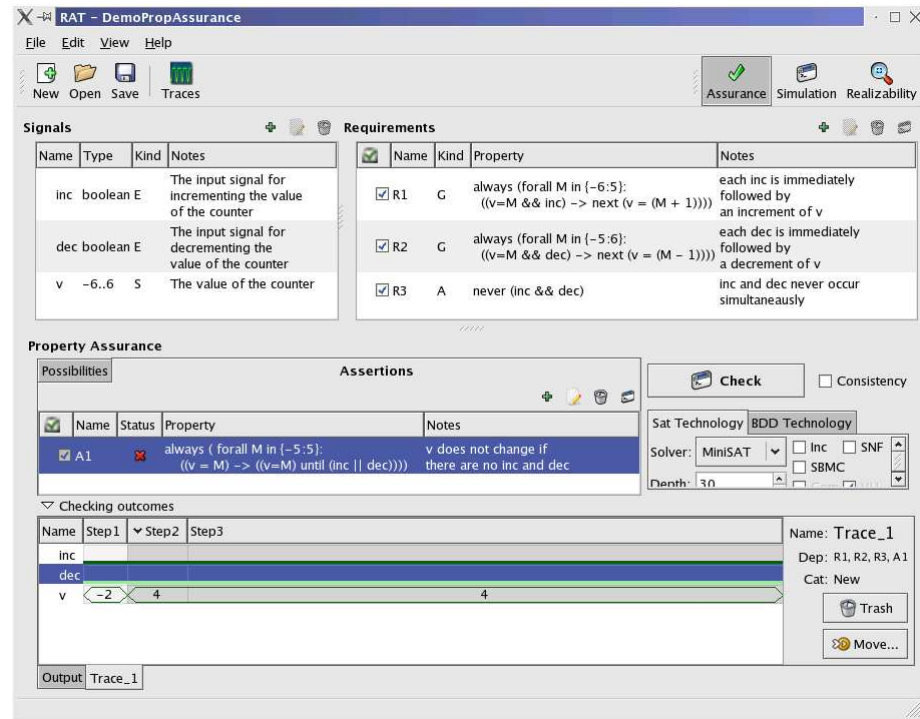


Figure 13: Counter - checking an assertion.

Once A1 has been entered, we can check it against all the requirements and get the result shown in Figure 13: the assertion is signaled as *failed* by a red bullet next to its name in the Assertions table, and a trace showing a counterexample to A1 is created and shown at the bottom of the main window. Note that a summary of the information related to the trace is provided close to the trace itself. By examining the trace, we notice that the counterexample shown has an initial *step* in which the value of the counter is -2 and no operation is issued, and a second step in which the value of the counter is changed to 4. Note that the last state is actually the first and only one of an infinite loop, as signaled by the little black arrow close to the name of the step in the header of the trace. A review of the requirements reveals that actually nothing is said about the evolution of signal v when no operation is issued, and this leads us to the definition of a new requirements that fills this hole

**R4:** prescribes that if no operation is issued the value of the counter remains unchanged

Figure 14 illustrates the new state of the specification and shows that if R4 is added, the check for A1 passes, as signaled by the green bullet in the Assertions table. Note that in this case the check has been performed using BDD technology with

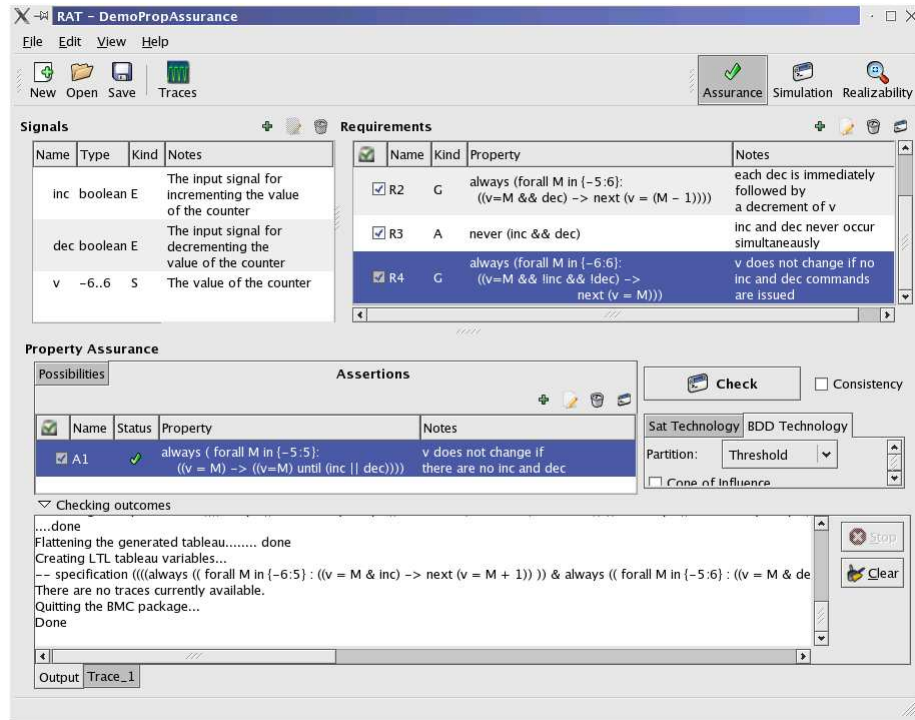


Figure 14: Counter - fixing the specification.

the Sift dynamic reordering method. In this case no trace is shown because no counterexamples has been found.

Once the check for A1 is passed, we gained more confidence on how the counter reacts to the stimuli of the environment. Now we can check that the system exhibits desired behaviors, i.e. that it is possible that something happens, even if not mandatory. For example, we may want to check that it is actually the case that the value of the counter may change, this means looking for a scenario in which the system evolves reacting to the stimuli of the environment in such a way to modify the initial value of the counter. This check can be performed by the possibility P1 shown in Figure 15.

The possibility is signaled as *passed* in the Possibilities table, and a trace corresponding to a witness of the desired system behavior is shown; the trace exhibits a five step loop in which initially v is 1 and two consecutive inc operations are issued (the value of v changes accordingly) and then two dec operations are issued making the value of v going back to 1 in the fifth step.

The result of a work session is a specification, a set of possibilities, a set of assertions and a set of traces corresponding to the results of the checks performed. Figure 16 shows the trace manager window with the traces generated during this session (actually other traces are shown that we do not described but that have been generated within this section).

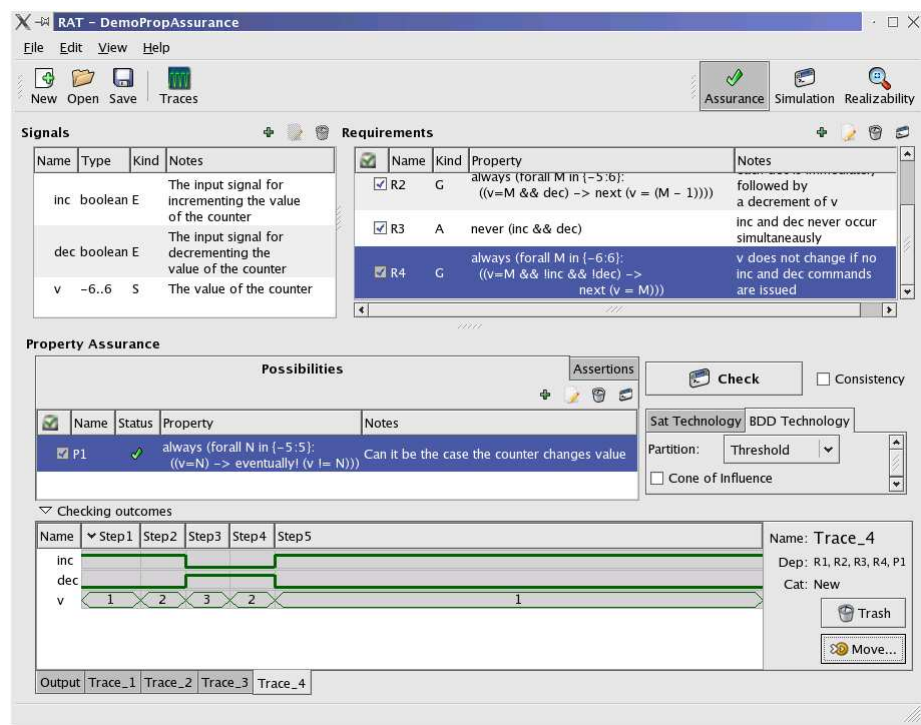


Figure 15: Counter - checking a possibility.

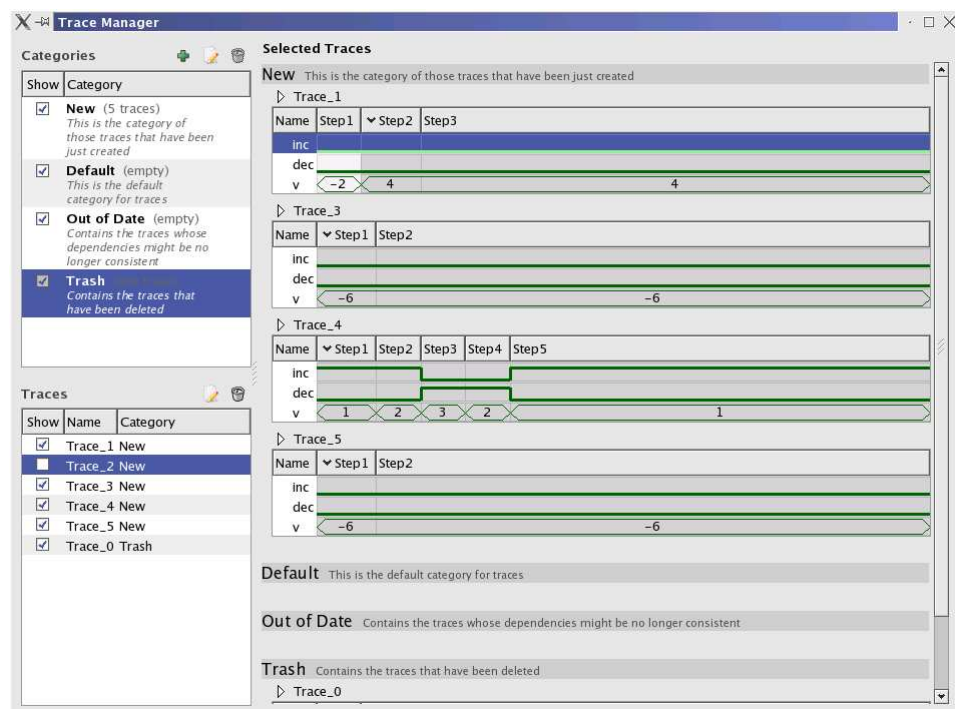


Figure 16: Counter - traces of the session.

---

## 1.3 Property Simulation in RATSY

This section illustrates the RATSY Property Simulation features. Some general GUI features will be introduced, followed by explanations of the main and analysis windows and an example scenario for a simple standard property.

### The Main Window

When enacting Property Simulation in RATSY you will see the RATSY main window to change to Property Simulation mode as illustrated in Figure 17. Please note that the user is able to switch the mode at any time using the switch controls in the upper right of the main window.

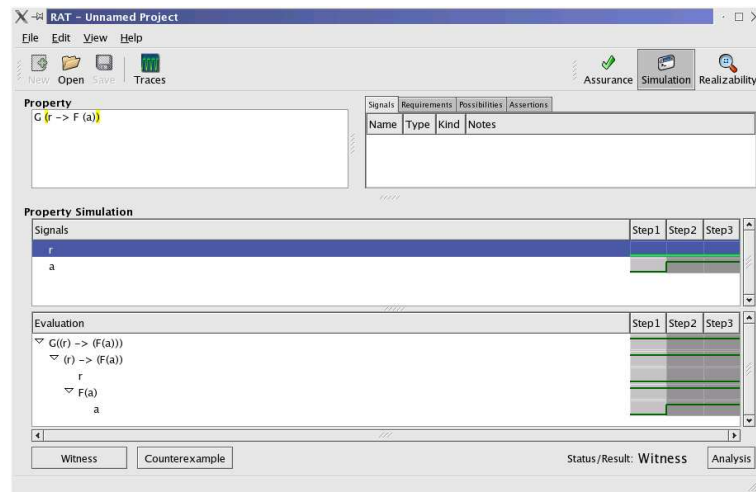


Figure 17: Property Simulation Main Window.

In the figure you see the three main sections of the Property Simulation interface. On the upper left you can see a multi-row text entry window where you can enter your property. The various lines are combined to a single property, thus you may split your property to several lines for a better overview.

The middle section of the Property Simulation window consists of two widgets showing waveforms. The upper one illustrates the derived example behavior using waveforms. The different waveforms illustrate the signal values for every time step in the trace. The whole trace is determined by the finite part as prefix completed by an infinite repetition of the infinite parts. The background color indicates whether the value is in the finite or infinite part of the trace. Light grey corresponds to the finite part and dark grey to the infinite part. You may select a single signal to highlight its waveform, there is no further impact of such a selection. The trace/signal view offers the possibility to request features for the next trace. A click on the right button of your mouse on a step of the trace produces a pop-up window offering the following requests:

- **Insert timestep:** Another time-step is entered just before the one you have clicked on. The default value is ‘Do not care’, which means that you don’t have any preference for the value in the next trace.
- **Remove timestep:** A given time-step is removed in the next trace.
- **Fix value to False:** In the next trace this value shall be false.
- **Fix value to True:** In the next trace this value shall be true.
- **Set to ‘Do not care’:** You do not care about the signals value at this time step in the next trace. This option can be used to unset required values.

When you establish requests you will notice that the color of the trace for this signal and time step changes to red. Red parts in the trace show that these parts are requested to be fixed to the current values for the next trace request. You’ll also notice that the status Value at the bottom changes to “Outdated” and the waveform color of the formula evaluation changes to black. This means that the tree-view for the Formula/Property evaluation does not correspond to the trace anymore.

The tree-view for the Formula/Property evaluation beneath the Trace/Signal view is not editable, so you cannot shape the waveform here. It illustrates and correlates the single parts of the property to the trace. For each time-step of the trace the property and all its sub-formulae are evaluated to true or false, visualized by waveforms organized in a tree. The tree structure is derived from the property to illustrate the dependencies between the parts of the property. Use the tree-view to make sure that the formula has been parsed the way you expected. Relating the waveforms to each other shows how the different parts of the property interact with each other interpreted on the trace.

The last part of the Property Simulation main window is the control and status bar located at the bottom. It includes the following contents:

- **Witness Button:** Pressing this button you can ask RATSY to derive a trace living up to the property and the feature requests you may have stated.
- **Counterexample Button:** With a click on this button you can ask RATSY to provide a trace contradicting the property or possible feature requests.
- **Status:** At this location you can always see what RATSY is up to when doing a computation and the status of the trace and evaluation when idle. Examples are *Witness*, *Counterexample*, *VIS Error*, ....
- **Analysis Button** A click on this button raises another second analysis window offering coverage information and controls as discussed in the very next section.

## The Analysis Window

The analysis window completes the information and controls of the main window. For each sub-formula of the property the window contains coverage statistics and

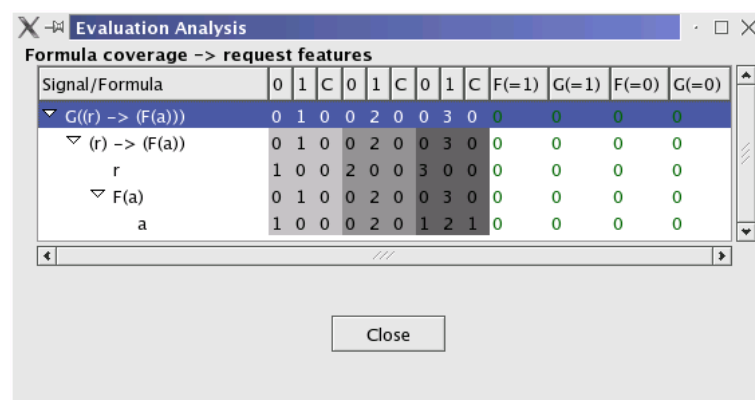
offers controls to request for the next trace that this part should evaluate globally or finally to true or false.

The coverage statistics tell how often a properties part evaluates to true and false, and how often this evaluation change during the evaluation of the trace. These statistics are derived for the finite and infinite parts of the trace, complemented by numbers for the entire trace including possible changes at the interconnection of the trace and the transition from the last state to the first state of the infinite part.

The graphical concept uses a tree-view for organization of the visualization and offers a 'close' button at the bottom to close the window. The tree-view shows the coverage statistics for each part of the property and the controls to request features. The first column contains the name of the part, followed by nine columns to illustrate the coverage information. For each part there are columns labeled '0', '1', and 'C', corresponding to the numbers for false ('0'), true ('1') and evaluation result changes ('C'). The three sections for the finite, infinite parts, and the whole trace are distinguished by the used background colors. The sections for the finite and infinite parts use the same colors used for the waveforms; light grey and dark grey. The section for the whole trace uses a very dark grey.

Additional four columns offer the option to request features for the next trace. You can request a sub-formula to evaluate a property eventually to true (' $F(=1)$ '), globally to true (' $G(=1)$ '), finally to false (' $F(=0)$ '), or (' $G(=0)$ '). A green zero for a request indicates that there is no request for the next trace, whereas a red one indicates a desired request. Pressing the right mouse button on a value produces a pop-up window enabling to set or unset a request.

Considering the tree structure and the coverage information can be of great help in exploring the behavior of a property. Considering the example of a property requiring an request to be acknowledged the coverage information may show that there is no request happening (columns labeled '1' show zero values for request) for a vacuous trace. So by setting the request to be eventually true you can ask for a more interesting trace for example. When a part of the property doesn't evaluate to a specific value at any time you may ask for an illustration of what happens if it does by seating the corresponding request.



The screenshot shows a window titled "Evaluation Analysis" with a tab "Formula coverage -> request features". It contains a tree view of formula coverage and a table of request features.

Signal/Formula	0	1	C	0	1	C	0	1	C	F(=1)	G(=1)	F(=0)	G(=0)
▽ G((r) -> (F(a)))	0	1	0	0	2	0	0	3	0	0	0	0	0
▽ (r) -> (F(a))	0	1	0	0	2	0	0	3	0	0	0	0	0
r	1	0	0	2	0	0	3	0	0	0	0	0	0
▽ F(a)	0	1	0	0	2	0	0	3	0	0	0	0	0
a	1	0	0	0	2	0	1	2	1	0	0	0	0

At the bottom of the window is a "Close" button.

Figure 18: Property Simulation Evaluation Analysis Window.

## An example

This section illustrates RATSYS Property Simulation functionality with a simple example. For this example scenario we will consider the informal property that a request should be eventually acknowledged .

First we have to start a new project. This is done by calling `rat` and clicking the “New” button at the top of the window. As for this example we decide to do Property Simulation only we can skip the step of entering project details at this stage; Property Simulation extracts the information it needs for its computations directly from the property itself. With a click on the finish button (Figure 19) we are presented with the main window of Property Simulation (Figure 20). Please note that if you would like to perform Property Simulation in an existing requirements engineering project for a device under construction, you can switch to Property Simulation by clicking the control button at the top right of the main window.



Figure 19: Create a project for Property Simulation.

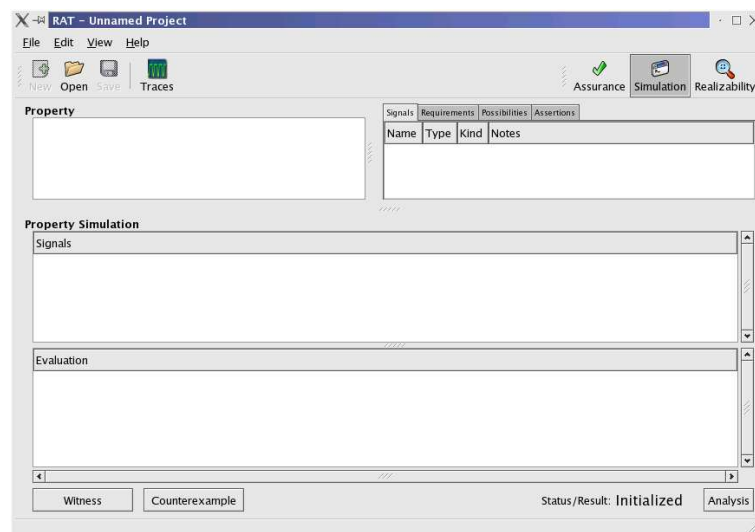


Figure 20: Property Simulation Start Window.

Our first guess on PSL syntax for our informal property is  $G(r \mapsto F(a))$ .  $G$  (“Globally”) is the short form of the PSL operator “always”, and  $F$  (“Eventually, Finally”)

is the short form of the “eventually!” operator. We enter that property into the entry widget of the Property Simulation main window and press the “Witness” button to ask for an example trace fulfilling and illustrating the property. We’re presented with the trace illustrated in Figure 21.

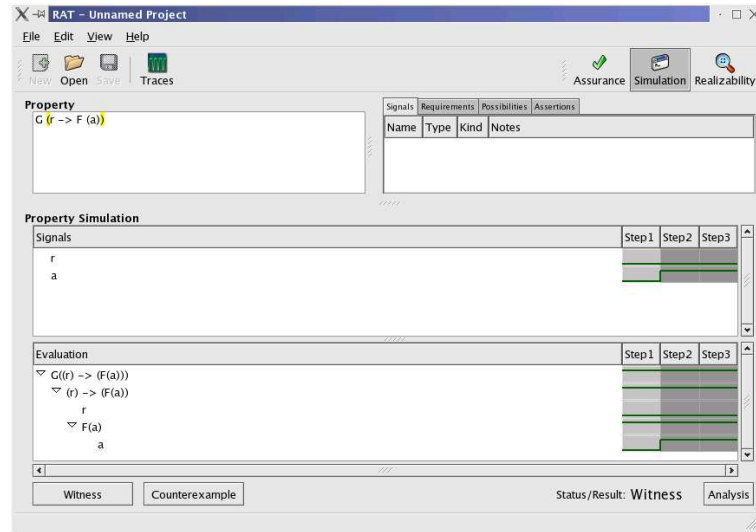


Figure 21: Witness for property  $G(r \mapsto F(a))$ .

The trace is vacuous because there is no request, but actually there are acknowledges. We see that the property does neither need a request to happen, nor that there is a request for an acknowledge to occur. Although the example is very simple and we can obtain that information by judging and interpreting the waveforms we now press the analysis button to show the coverage information illustrated by Figure 22.

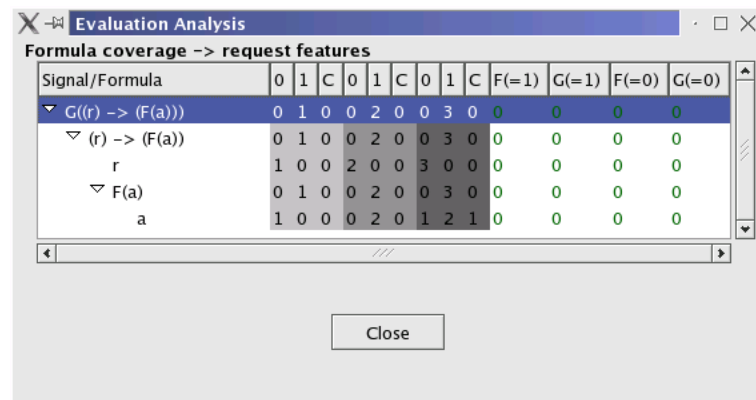


Figure 22: Analysis of trace for property  $G(r \mapsto F(a))$ .

A check of the analysis reassures our preliminary conclusions. To gain a more interesting trace we request a request to eventually happen as illustrated in Figure 23. We keep the analysis window opened and ask for a new witness by pressing the corresponding button in the main window.

We are presented with the trace illustrated in Figure 24. As we are satisfied with the trace and want a request to happen for future examples we change our property

Signal/Formula	0	1	C	0	1	C	0	1	C	F(=1)	G(=1)	F(=0)	G(=0)
▽ G((r → (F(a)))	0	1	0	0	2	0	0	3	0	0	0	0	0
▽ (r → (F(a)))	0	1	0	0	2	0	0	3	0	0	0	0	0
r	1	0	0	2	0	0	3	0	0	1	0	0	0
▽ F(a)	0	1	0	0	2	0	0	3	0	0	0	0	0
a	1	0	0	0	2	0	1	2	1	0	0	0	0

Figure 23: Ask for a request on signal r.

to  $G(r \mapsto F(a)) \& \& F(r)$ . By asking for a new witness we want to recheck this change. Please note that the requests are reset for every trace; so you might not include a forgotten request forever resulting in the miss of interesting behaviors during property exploration.

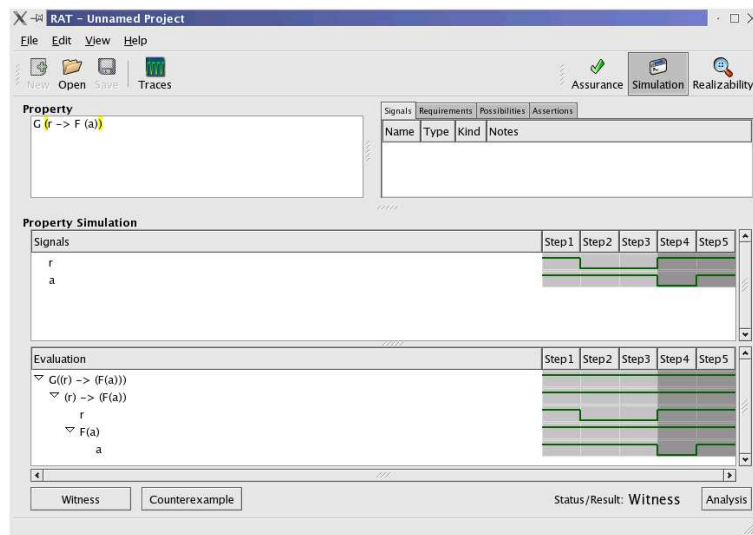


Figure 24: Witness with request for property  $G(r \mapsto F(a))$ .

The derived trace illustrated in Figure 25 however, unveils that we have got something wrong, as the tree structure does not fit our intention. By the investigation of the tree structure we uncover that we have forgotten two brackets. We have to put the  $G()$  part of the property into brackets, otherwise the *logical and* binds the  $F(r)$  to the implication part and not to the globally part. We add additional brackets to the property to gain  $(G(r \mapsto F(a))) \& \& (F(r))$ . By asking for a new witness we recheck the property and are satisfied with the presented trace and evaluation (Figure 26).

Now we want to check if a single of the two acknowledges conforms to the property. Again this might be obvious for our example, but it might not be obvious for a more complex one. Thus we shape the trace by editing the waveform. We fix the values of signal  $r$  to the values of the trace and signal  $a$  to true for time-step one and false for the remaining time-steps (Figure 27).

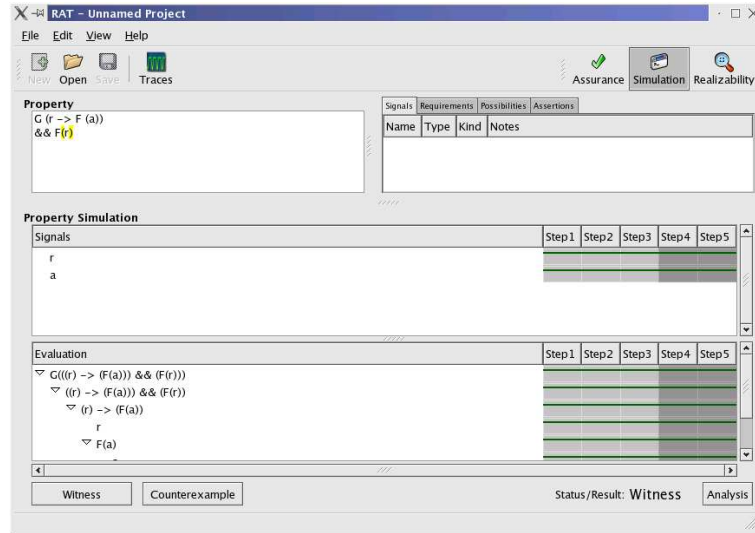


Figure 25: Witness for property  $G(r \mapsto F(a)) \&\& F(r)$ .

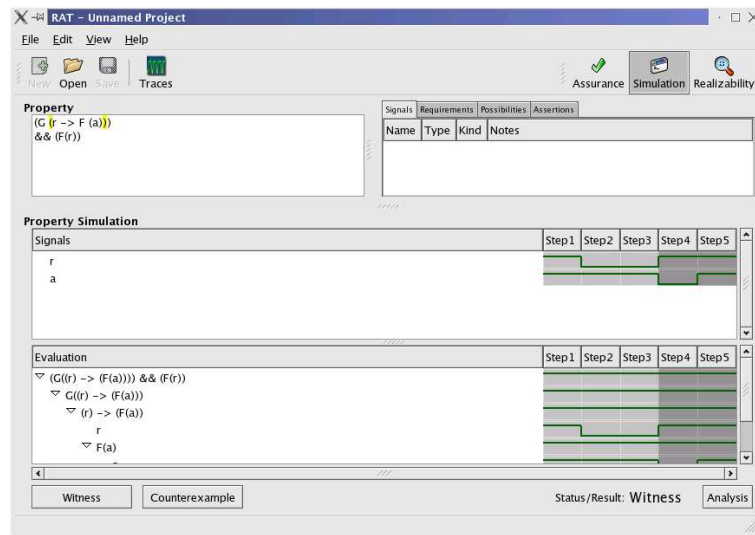


Figure 26: Witness for property  $(G(r \mapsto F(a))) \&\& (F(r))$ .

Asking for a new witness produces a trace illustrating that our requests are satisfiable (Figure 28).

We have used all elements of the Property Simulation interface so far, and now it is up to you to explore the property and the potential of Property Simulation on your own. To give you some initial direction we would like to suggest to enhance the property to allow an acknowledge only on a request, or to limit the length of an acknowledge to one time-step.

## 1.4 Property Realizability and Synthesis in RATSY

This section illustrates the RATSY Property Realizability and Synthesis features.

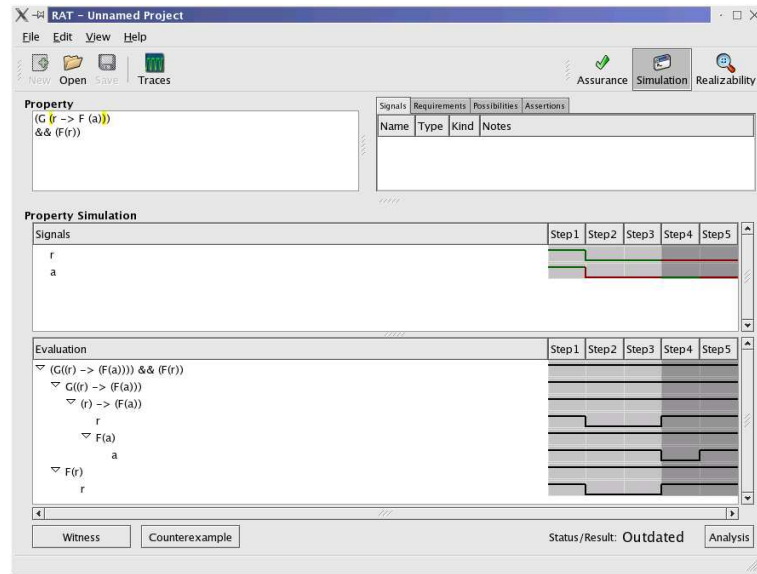


Figure 27: Shaping the trace.

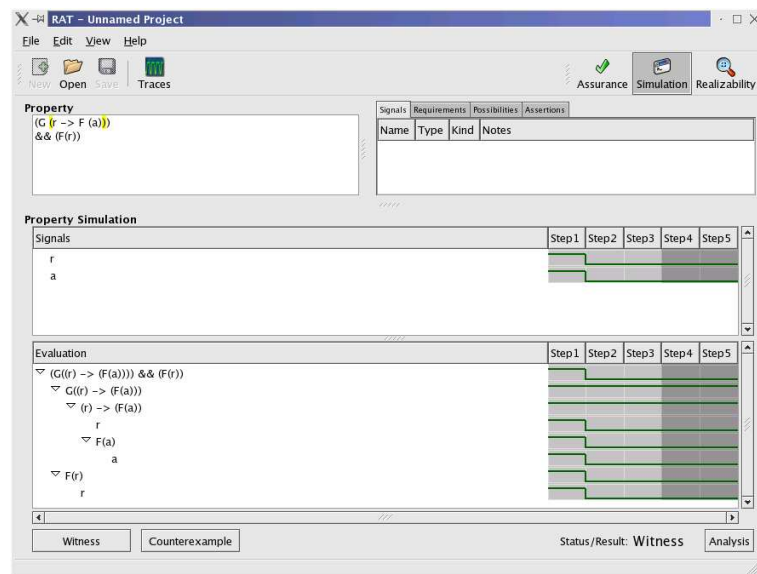


Figure 28: Witness for shaped trace request.

For using the Realizability feature the enhanced version of NUSMV [5] is required. For using the Synthesis feature the command line tool MARDUK is required. See Section 3 for details.

## Realizability Problem

Informally, the Property Realizability problem can be described as follows. All signals are divided into two disjoint sets – uncontrolled (environment) signals and controlled (system) signals. Similarly, every requirement belongs to one of two sets – the assumptions and the guarantees. At every time step the environment variables are set to some unknown-beforehand values and then the system decides

values for its variables. Assuming that the assumptions hold the task of the system is to satisfy the guarantees. If the system is able to do that for every possible behavior of the environment the specification is Realizable. Otherwise the specification is Unrealizable. For a detailed definition of the Realizability problem see [5].

## Specifying a Realizability Problem

As was told in Section 1.2 the distinction of signals in System and Environment as well as the distinction of requirements in Assumption and Guarantee is important only for Property Realizability. Thus now, a user has to specify explicitly whether a signal is an environment signal or a system signal. For example, Figure 29 shows the wizard to specify an environment signal `inc` of type boolean. Similarly, a

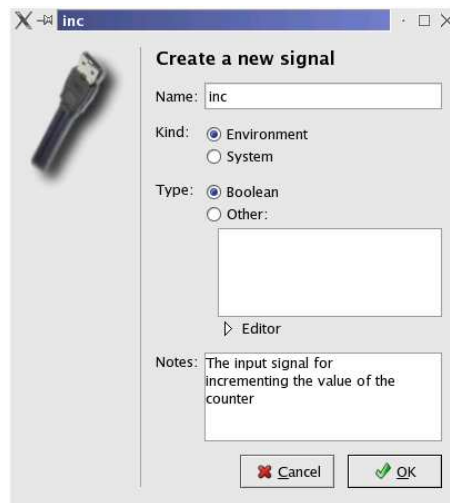


Figure 29: Specification of an environment signal in RATSY.

requirement describes an assumption on the behavior of the environment, or a guarantee on the behavior of the system. For instance, Figure 30 show the RATSY wizard to specify the system guarantee `always(forall M in {-6:5}: ((v=M && inc) -> next(v=(M+1))))`.

## The Main Window

Once all the signals and all the requirements have been inserted in the RATSY project, it is possible to move to the Realizability window where the button that performs the check of realizability for the selected properties can be pressed to start the check for realizability. Figure 31 shows the Realizability window with an example of realizability problem.

The Check Realizability button on the right in the Realizability window of RATSY activates the realizability checks. The result of the check is shown in the left text area. In this particular example the specification is unrealizable because

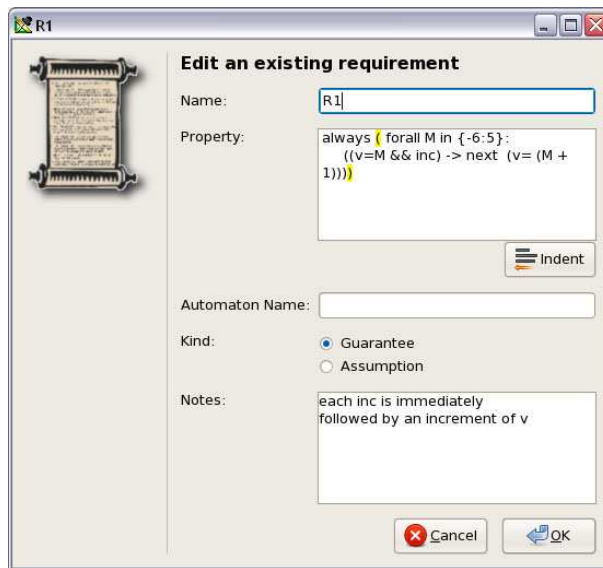


Figure 30: Specification of a system guarantee property in RATS.

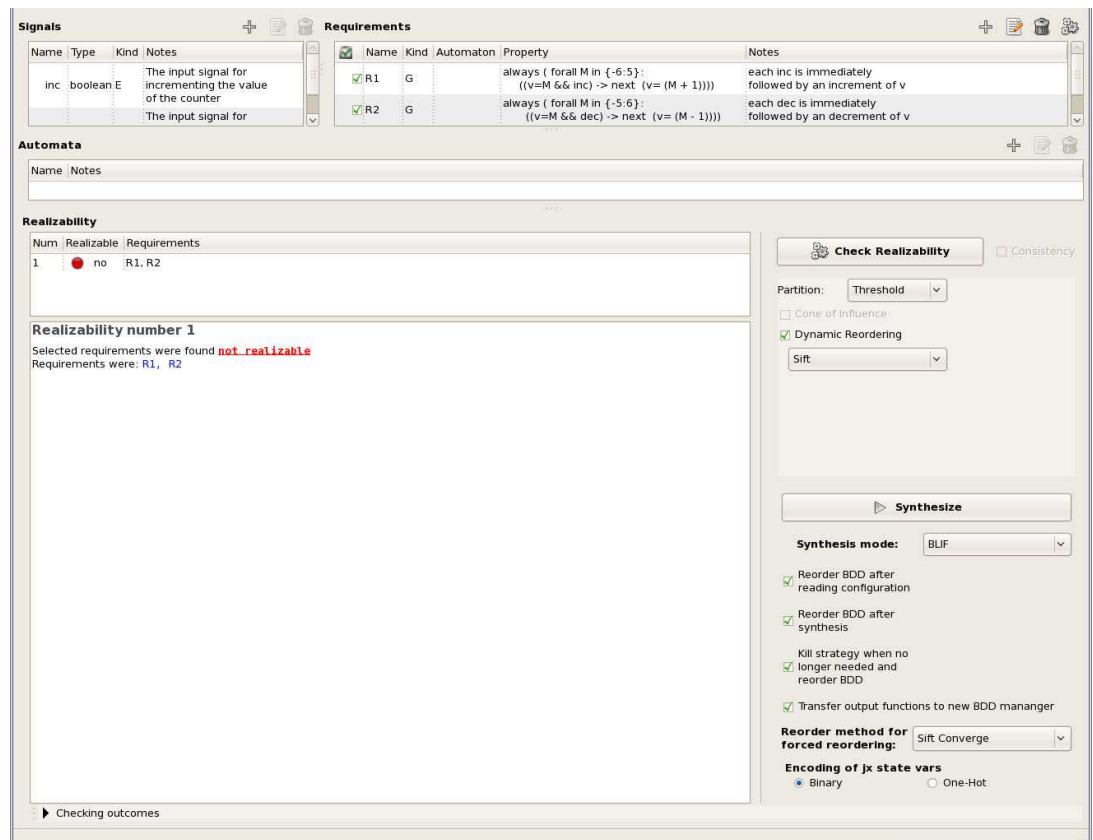


Figure 31: The Realizability window in RATS.

the system may force the violation of the guarantee requirements by setting both signals `inc` and `dec` up.<sup>1</sup> To avoid such behavior we can add an assumption requirement `never(inc && dec)`. With this assumption the specification becomes

<sup>1</sup>The cause of unrealizability may not always be so obvious. See Section 1.5 to learn how to debug an unrealizable specification.

realizable (Figure 32).

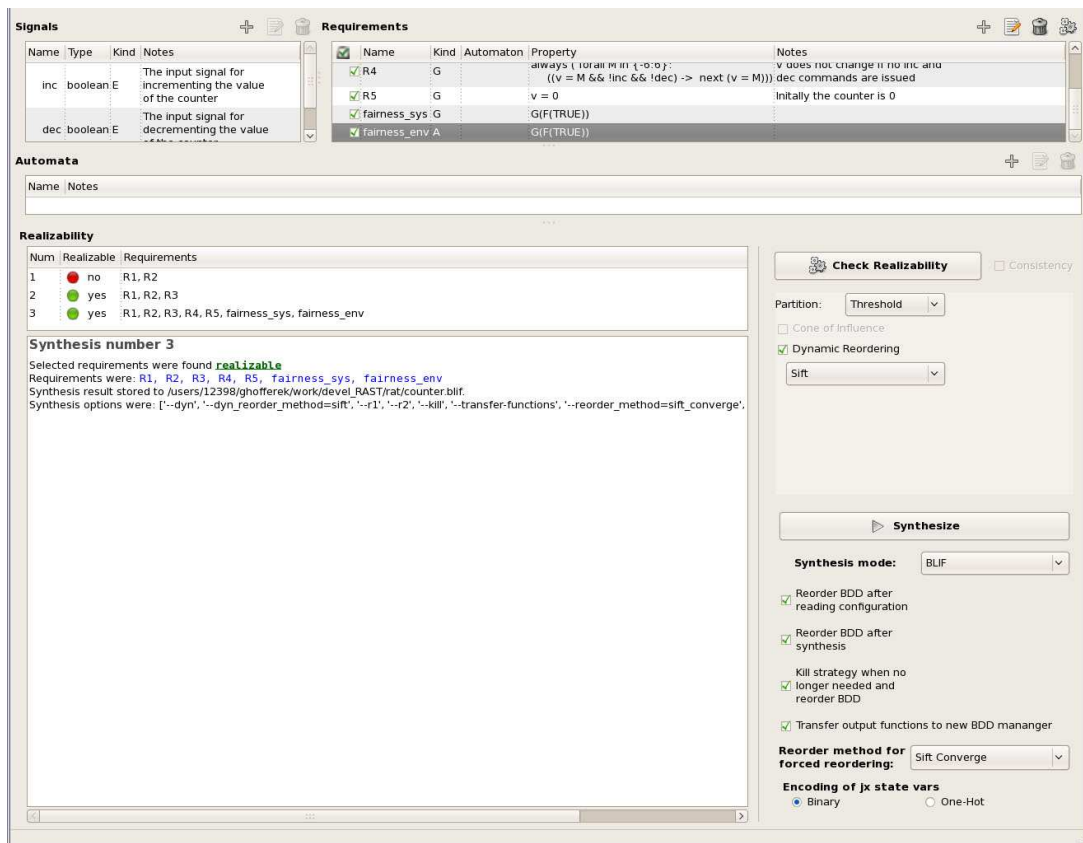


Figure 32: The Realizability window in RATSY.

A set of assumptions and guarantees is internally converted into an equivalent NUSMV game structure, and depending on the generated game structure the corresponding check algorithms are invoked (with the help of the enhanced version of NUSMV [5]). The generated game structure is printed in the log tab, as to allow the user to inspect it. Note that, such a game structure may have fresh variables introduced during conversion. If the tool is not able to convert a RATSY specification into a NUSMV game structure an error message with the subexpression causing the problem is printed out.

## Synthesis

For realizable specifications an implementation can be automatically synthesized. Synthesis works according to the algorithm presented in [10]. To perform synthesis, RATSY relies on the command line tool MARDUK, which it calls as an external process. Since synthesis can take a very long time (for larger specifications), it might be preferable to invoke the MARDUK tool directly from the command line, in order to have it run in background and independent of the graphical user interface. MARDUK is able to read and process RATSY project files. Also, the command line tool provides some advanced and experimental features and options,

which are not available via the graphical user interface of RATSY. If you want to start synthesis directly from inside RATSY, click the Synthesize button and select a file to store the synthesis results. The following options for controlling the synthesis process are available on the lower right-hand side of the window. The synthesis process will also respect the options for dynamic reordering, which can be set right below the Check Realizability button.<sup>2</sup> Before starting the synthesis process, the desired mode should be selected. At present, the following modes are available.

1. **COFACTOR:** A cofactor-based approach, presented in [3], and the BDD-Restrict operation of the CUDD package is used to compute output functions from the strategy.
2. **IRRSOP:** This mode is based on the Minato-Morreale algorithm for computing *irredundant sums-of-products* [7], combined with a cache of already implemented subfunctions, to find deterministic output functions.
3. **FACTOR:** This mode is a generalization of the IRRSOP mode. Instead of single literals, arbitrary Boolean functions are used in each recursive step. This mode is experimental!
4. **OLD:** This mode is a legacy mode from a previous release. It is basically the same as the COFACTOR mode, except that the output file is directly dumped from the CUDD package. This might save a little computation time, but it limits the output format to BLIF.

After setting the mode, one of three different output languages can be selected: BLIF (Berkeley Logic Interchange Format), Verilog, and HIF (HDL Intermediate Format)<sup>3</sup>. Below the mode and language selection box, some more options about BDDs are available. The first checkbox lets you enforce a reordering of the BDDs after reading the specification and creating BDDs for all assumptions and guarantees. It is recommended to leave this option turned on, as it usually shortens synthesis time. The next option enforces a reordering after the strategy has been synthesized. The third option will delete the strategy BDD and also trigger a reordering, once the output functions have been determined. It is recommended to turn the latter two option on in BLIF mode, and turn them off in the IRRSOP modes. In IRRSOP mode, output functions are not represented as BDDs any more, so reordering will not improve synthesis results, but just take time. The next option enforces a transfer of all BDDs that represent output functions to a new DD manager, before they are dumped into the output. Experience showed that turning this option on slightly improves synthesis results, at very little extra time. This option is only available in BLIF mode. The reorder method for forced reordering may differ from the one used for dynamic reordering. Experience shows that using one of the “converging” methods is preferable. Finally, it is possible to choose between two different encodings of the *jx* state variables. These variables store which fairness condition the system is going to fulfill next (cf. [10] for details). The default value is Binary.

---

<sup>2</sup>It is highly recommended to use dynamic reordering, as it will greatly reduce memory and CPU time usage.

<sup>3</sup>See <http://hifsuite.edalab.it/> for details.

**Caveats** There are two very common scenarios which can cause the synthesis process to terminate abnormally. Due to limitations of the implementation the current version of RATSy does not specifically report the causes of abnormal termination in these cases. Thus, if RATSy reports that the synthesis process terminated abnormally, you should check whether one of the two following scenarios applies to your project. First, RATSy can only synthesize specifications in *Generalized Reactivity(1)* format. That means that (at least) one assumption and (at least) one guarantee must be a fairness/liveness constraint. If your specification violates this restriction you will see a message similar to the following at the end of the Checking Outcomes window. Error: The given specification is not a 'GenReactivity' specification. The game type is 'AvoidDeadlock'. If you see such a message you should augment your specification with a guarantee/assumption of the form  $G(F(\text{TRUE}))$ , to make it a GR(1) specification.

Second, if one of your requirements causes a parse error, the synthesis process will also terminate abnormally. Unfortunately the synthesis process can not give information about which of the requirements is malformed. The error message in the Checking Outcomes window will look like this: ERROR! Encountered an exception! Error: could not parse the input file! If you click the Check Realizability button without changing your specification, the realizability check will report in more detail which of your requirements is malformed. These information will be displayed in the Checking Outcomes window.

## The Automaton Editor

Specifications for reactive systems are often easily expressible as a set of deterministic and complete Büchi word automata, where the edges correspond to safety constraints, and the accepting states correspond to fairness/liveness constraints. RATSy provides a graphical tool to create and edit such automata. The automata are automatically converted into PSL formulas, which can then be used as requirements. The following example illustrates the use of the automaton editor. Think of a very simple arbiter, with just one request line (`req`) and one acknowledge line (`ack`). We want to model a property that captures the fact that every request should eventually be acknowledged. We will do so by means of a simple Büchi automaton with 2 states.<sup>4</sup> First click the plus sign above the automata list to add a new automaton. In the dialog window that opens (Fig. 33), specify a name for this automaton. Then click the `Edit` button to actually create/edit the automaton. In the main automaton editor window, click the buttons `New State` and then click on an empty spot in the editor pane to add a state to the automaton. Double-click the state to edit its properties (Fig. 34). You can specify a name for the state, whether or not it is the initial state of the automaton, and whether or not this state is one of the accepting states. Name the state `s0`, and make it initial and accepting. Create a second state, which should not be accepting, and label it `s1`. You will notice that the new states have a dangling edge labeled `true`. Dangling edges lead to an im-

---

<sup>4</sup>This example is included in the RATSy distribution. The corresponding project file is `examples/demo/DemoAutomaton.rat`.

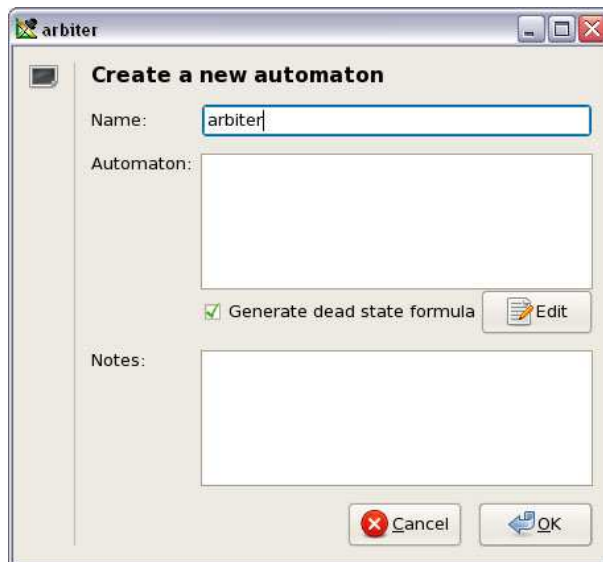


Figure 33: Create a new automaton.

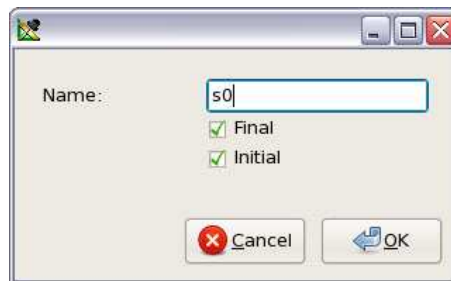


Figure 34: Edit the properties of a state in the automaton.

plicit, non-accepting dead state, which has a self-loop labeled `true`. For reasons of clarity this dead state is not drawn explicitly. The editor keeps the automaton deterministic and complete at all times. Since we have not specified any transitions yet, all transitions (hence the label `true`) lead to the dead state. In order to specify transitions of our own, we must first add signal names. Right-click in the list of signal names in the lower-right part of the editor window and select **Create**. Specify a name for the signal and click **Ok**. The signal names that you specify here will be used to create the PSL formula representing the automaton. Thus, make sure that you only use names of signals that you created in your main project. Otherwise the resulting formulas will not work. After you have created signals named `req` and `ack`, you are ready to add the edges to the automaton. First, we want to create an edge from `s0` to `s1`. Thus, first click the **New Edge** button, then click on `s0`, and finally on `s1`. A new edge from `s0` to `s1` will be displayed. Notice that the new edge is labeled `true`, and that the dangling edge of `s0` has disappeared, to keep the automaton deterministic and complete. We will label the newly created edge later, after we have created all the edges we need. Next, create an edge back from `s1` to `s0`. Note that you can add way points to edges. After you clicked `s1`, click on an empty spot in the editor pane, before you click `s0`. The new edge will pass through the point you clicked. You can of course move the way point at any later time. You can also add new way points to an edge, by first selecting it with a left-click and then clicking the middle mouse button somewhere on the edge.

Way points can also be deleted by selecting them (left-click) and using the Delete button on the right-hand side. Next, add loop edges to both  $s_0$  and  $s_1$ . Doing so is straight forward. Just click New Edge, and then click twice on the state you want to have a loop edge.

After you have created all the states and edges, we are going to specify the transition condition for the edges. The idea is that state  $s_0$  is the state in which there are no outstanding requests. Thus, it is an accepting state and should be visited infinitely often. On the other hand,  $s_1$  is the state in which there is a request which has not yet been acknowledged. So the transition from  $s_0$  to  $s_1$  should be taken whenever  $req=1$  and  $ack=0$ . Double-click the edge to edit its properties (cf. Fig. 35). In the Minterm field enter 10, meaning  $req=1$  and  $ack=0$ .<sup>5</sup> If there is no



Figure 35: Specify the transition condition for an edge.

request ( $req=0$ ), we don't care about the value of  $ack$ , and stay in  $s_0$ . Also, if a request is immediately answered ( $req=1, ack=1$ ), we stay in  $s_0$ . These two cases correspond to the minterms 0- and 11. Enter them for the loop edge on  $s_0$ , one minterm per line. For the state  $s_1$ , we want to stay there as long as we do not answer the outstanding request. Thus, set the minterm for the loop edge to -0. You will notice that the edge from  $s_1$  to  $s_0$  is automatically updated from  $true$  to  $ack$ , because the automaton is always kept deterministic. Thus, we are already finished. Your automaton should now look like in Fig. 36. Click Ok to close the editor. You will see the formula that has been automatically generated in the remaining dialog window. There are two ways in which the formula can be generated, which differ in the way in which they handle the implicit dead state that has been mentioned before. If the checkbox Generate dead state formula is ticked, the implicit dead state will be treated just like any other state. It will be encoded using state variables, it will be a non-accepting state, and it will have a self-loop labeled  $true$ . If this checkbox is not ticked, the dead state will not be treated as a real state of its own. No state encoding will be assigned to it. Instead, whenever an edge which would lead to the dead state is traversed, a special signal  $dead$  is asserted. However, the formula contains also a conjunct stating  $G(dead=0)$ . Note that both types of formulas define the same  $\omega$ -regular language. Choose whatever suits your needs or your liking better, but don't forget to create the signal  $dead$  in your main project if you decide to use the latter case. Finally, click Ok again to save the automaton.

On a side note: The labels on the edges can also be moved. To do so, first select the edge with a left-click. Then press and hold the left mouse button on an arbitrary position along the edge. If you now move the mouse to the right (while still holding

<sup>5</sup>The order of signals for specifying minterms corresponds to the order on the lower-right side of the automaton editor window.

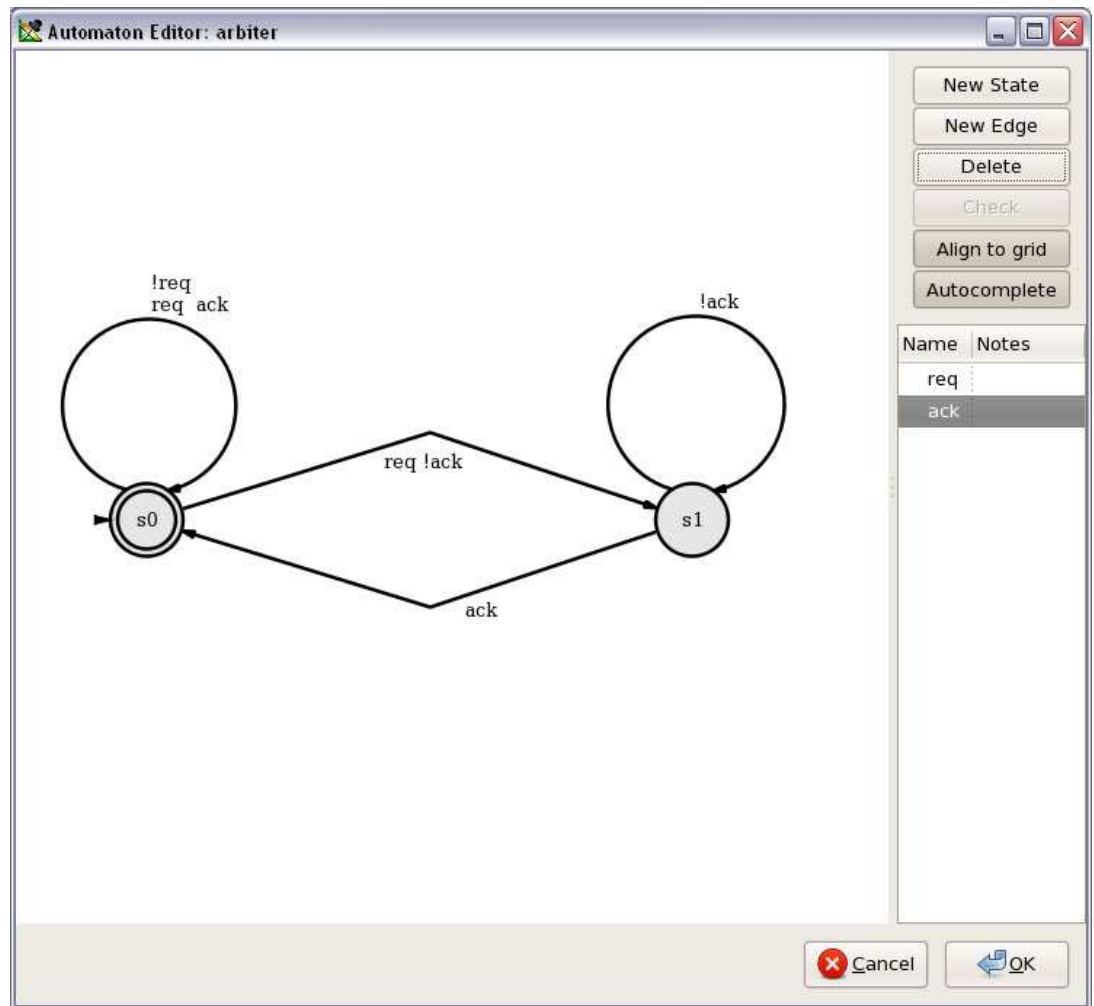


Figure 36: The main window of the automaton editor.

down the left mouse button), the label of the edge will be moved along the edge, towards the target state. If you move the mouse to the left, the label will move towards the source state. If necessary, you can also zoom the editor pane at any time, by using the scroll wheel of your mouse.

Once the automaton has been saved, you can use it as a requirement. To do so, type the name of the automaton in the corresponding field in the requirements dialog (cf. Fig. 30). After you typed in the automaton name, the formula corresponding to the automaton will be automatically inserted into the property field. Note that in this case you cannot manually edit the formula. With the radio buttons below, you can choose whether the automaton represents a guarantee or an assumption. Click `Ok` to save the requirement. Note that if you do changes to the automaton at a later time, the PSL formula in the requirement will be automatically updated to match the latest version. Furthermore you may use template-parameters to reuse the same automaton several times. Parameters have the form `%{name}` and may be used in the signal names of the automaton. Parameter names may only consist of letters, numbers and underscores and must start with a letter or an underscore. If you use an automaton with parameters as a requirement, an additional field will appear where you can assign values to the parameters. Note that you still have to create the

signals, that are finally used, in your main project, including those used to encode the current state of the automaton. To facilitate the creation of these signals a refresh button is provided above the signal list which adds all missing signals used by automata. The signals created in this way will be marked as automatically generated, which means that if their names change or they become obsolete, for example because the requirement using them was deleted, you can easily update them by clicking refresh again. You may edit an automatic signal as well, for example to change its type, but in this case it loses its automatic status and has to be maintained manually.

---

## 1.5 Simulating and Debugging Specifications using Games

The game part of RATSY provides three main features. It allows you

1. to play a normal game in order to test the specified system,
2. to play a counter game in order to understand why a certain specification is unrealizable, and
3. to specify desired behavior if undesired behavior was observed during a play.

In the normal game, you are in the role of the environment while the tool is in the role of the system. In every time step, you first choose values for the inputs. Then the system responds with outputs that conform to the specification. In order to find such outputs, a winning strategy for the system is synthesized.

In the counter game, you are in the role of the system while the tool is in the role of the environment. In every time step, the tool first provides input values. You are then asked to choose the values of the outputs in such a way that the specification is fulfilled. You win if you manage to fulfill the specification. You lose otherwise. The tool uses a counter strategy to find problematic inputs, i.e., inputs for which no behavior of the system can fulfill the specification. Hence, you will lose for sure. However, while losing, you will understand where the specification is too restrictive to be realizable. This knowledge can then be used to correct the specification in order to obtain a realizable specification. More information on debugging unrealizable specifications with counter games can be found in [6].

As within Property Realizability, signals are distinguished of being under the control of the system or the environment. Furthermore, requirements can be assumptions or guarantees. The specification requires the system to fulfill all guarantees if all assumptions are fulfilled by the environment. Similar to the synthesis feature of RATSY, the game part requires the specification to be in the *Generalized Reactivity(1)* format, i.e., at least one assumption and at least one guarantee must be a fairness/liveness constraint. Otherwise an according error message is printed when trying to start a game.

## How to play a Game

Figure 37 shows the Game window in action. Initially, the trace views labeled with

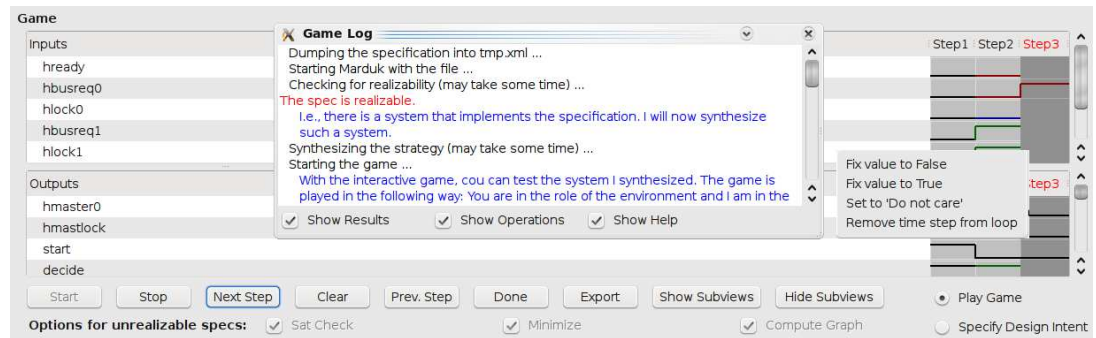


Figure 37: The Game window in RATSY.

“Inputs” and “Outputs” are empty. The button `Start` starts the game. First, the tool checks the specification for realizability. If it is realizable, a normal game is started, otherwise a countergame is started.

In either case, the current time step of the game is marked with red letters. You are only allowed to modify signal values in the current time step. Signal values can be modified by right clicking onto the according position in the trace. A pop-up menu appears that allows to set the value to 0, 1, or “don’t care” (see Figure 37). In the normal game, you are only allowed to modify input signals. In the countergame, you can only modify the output signals.

Different waveform colors are used to mark different origins of signal values.

- Black is used if the signal value is the only possibility fulfilling the safety requirements.
- Red is used for user selections.
- Blue is used if the signal value is a consequence of some user selection for other signals.
- Green is used if the signal value was chosen completely arbitrarily.

During the play the tool enforces that all safety requirements are met. When your choice violates some safety requirement, an error message is printed.

After setting all signals to their desired values in the current time step, click `Next Step` and the next time step can be edited. By clicking `Clear`, all user selections for the current time step are cleared again, i.e., set to “don’t care”. If some signal values are still “don’t care” when the `Next Step` button is clicked, these signals will be chosen arbitrarily by the tool. With the button `Prev. Step`, the previous time step of the play can be edited again. This is useful when you want to change some selection in some previous time step. Note, that the user selections for the current time step are lost when going back to the previous step.

You can put time steps into the infinite loop or put them back into the finite part by right clicking onto any signal in that time step and choosing the according menu item. Only the last time step before the loop can be put into the loop and only the

first time step of the loop can be removed from the loop. This restriction avoids that you end up with more than one infinite loop or that the loop is not located at the end of the trace. Time steps within the finite part are marked with light gray background. Time steps with dark gray background belong to the infinite loop.

You can finish a play by clicking the Done button. This causes the tool to analyze the play in order to find out the winner. Furthermore, explanations to this verdict are printed to make you accept that you have indeed lost the play. When you finally click the Stop button, the play engine is reset and a new game can be started by a subsequently clicking Start. Changes of the specification that are made during a play do not affect the play. One has to click Stop followed by Start to start a new game using the modified specification.

Game traces can be exported by clicking the button Export. You can choose between three formats: jpeg, png and vcd (Value Change Dump). Exporting the game traces as jpeg or png improves over a simple screenshot in that no part of the trace is hidden due to scroll bars. Game traces exported as vcd can be opened by most waveform viewers. However, the colors in the trace as well as the position of the infinite loop are lost when exporting traces as vcd. This is due to a lack of support of such elements in the Value Change Dump format. Note, that there is (currently) no way to save the current state of a play. In particular, exported game traces cannot be loaded again to continue a play.

There are two sub-windows related to the Game window: the Game Log window and the Automata window. Both can be shown or hidden with the buttons Show Subviews and Hide Subviews, respectively. These sub-windows are described in the next sections.

## The Game Log Window

The Game Log window is also shown in Figure 37. It contains three types of log messages:

- Results: Written in red, they contain the main results obtained by the tool during the play.
- Operations: Such messages show what the tool is currently doing. They are written in black.
- Help Messages: These messages guide you through a game. They are written in blue.

All types of messages can be enabled or disabled with the corresponding checkboxes. Information will be stored in the background, even while a particular message type is disabled. When re-enabling it later, the messages will be displayed as if the message type was never disabled.

## Integration with the Automaton Editor

The following features are only available if the specification contains automata constructed with the automaton editor (see Section 1.4).

Figure 43 shows an example for the Automata sub-window of the game. The names of all automata of the specification are shown on the left side (in case of Figure 43 there is only one). One such name can be selected, and this selected automaton is shown on the right side. The current state of the play is marked with yellow in the automaton. Also, all edges that are still possible with the current user selection in the Game window are marked yellow. If you want to traverse a certain (yellow) edge of the automaton in the game, you can simply select this edge with a mouse click. The restrictions imposed by traversing this edge are then added as additional user selections to the signals that are under control of the user in the game. User selections obtained by selecting edges can be cleared again by clicking the Clear button in the Game window or by setting the corresponding signal values to “don’t care”.

Not only the current state of the play but arbitrary time steps can be displayed in the automata. Simply right click onto any signal in the desired time step and selects the menu item Show step in automata.

## Specifying Design Intent

When you observe undesired behavior of the system while playing a normal game, you can switch into the Specify Design Intent - mode by selecting the corresponding radio button in the Game window (on the bottom of the right-hand side in Figure 37). The game trace can be used as a starting point to specify the desired behavior of the system. You can change the value of any signal (inputs and outputs) in any time step to 0, 1, or “don’t care”. This is done by right clicking onto the signal in that time step and selecting the corresponding menu item. It is also possible to set a certain signal in all time steps to a certain value by right clicking onto the signal name in the trace. The waveform color black is used for signal values that came from the game. The waveform is colored in red if the signal value was changed by you. The finite part and the infinite loop of the trace can be edited in the same way as in the game. Unlike in the game, new time steps can be inserted and existing time steps can be removed from the trace, again by right clicking onto the desired position and choosing the corresponding menu item.

In the end, the input trace and the output trace should represent the desired behavior in the following way: If the behavior of the environment matches the specified input trace, then the behavior of the system must match the specified output trace. Finally, click Done and an automaton that accepts only the desired behavior is created automatically and added to the table of automata in the project. It can be added as an additional guarantee to the specification as described in Section 1.4. This eliminates the undesired behavior originally observed during the play. Clicking Stop clears all data from the traces.

Once the Specify Design Intent - mode is activated, you cannot return to the Game mode again. You have to click Stop followed by Start to start a new game. Specifying design intent is only possible from a normal game but not from a counter game.

## Additional Features for the Normal Game

There is a special waveform `jx` in the output trace that does not correspond to an output signal. It contains the memory content of the strategy according to [10]. This memory content is the index of the fairness constraint of the system that will be fulfilled next. You can simply ignore this row if you are not familiar with the work of [10].

## Additional Features for the Counter game

Following [6], the counter game is integrated with additional features that make it easier for you to find out why the specification is unrealizable.

**SAT-check:** Unrealizable specifications are checked for satisfiability first. The result is written to the Game Log window. If a specification turns out to be unsatisfiable, you do not have to play a game in order to understand the problem. You can also use Property Simulation to learn why no trace can fulfill the specification. This may lead to simpler explanations. However, as unsatisfiability is just a special case of unrealizability, the counter game can also be used to explain unsatisfiability. If undesired, the SAT-check can be deactivated with the corresponding checkbox in the Game window.

**Minimization:** All output signals and guarantees that are irrelevant for the unrealizability problem are removed from the specification before a counterstrategy is computed. A guarantee is irrelevant if you cannot fulfill the specification even if you would not have to fulfill this guarantee. A signal is irrelevant if you cannot fulfill the specification even if you could choose the value of the signal completely arbitrarily in every time step without any consequences for other signals. Irrelevant guarantees and signals are not included in the game. This helps you focus on the actual problem. The irrelevant guarantees are deactivated in the table of requirements of the project. Which signals are irrelevant can be seen from the Game Log window. Minimization can be deactivated with the corresponding checkbox in the Game window.

**Countertraces:** A *counterstrategy* is a strategy for the environment to find problematic inputs, i.e., inputs for which no behavior of the system can fulfill the specification. The inputs suggested by the counterstrategy depend on the outputs previously chosen by the user. On the other hand, a *countertrace* is a fixed trace

of inputs for which no behavior of the system can fulfill the specification. It is independent of the moves of the system and thus easier to understand. RATSY heuristically searches for a countertrace. If it could find one, this countertrace is used instead of the counterstrategy in the countergame. The complete countertrace is shown right from the beginning of the play, so you know in advance how the environment will behave. This makes it easier for you to localize the problem in the specification.

**Summarizing Graph:** A graph is computed that summarizes all plays that are possible when the environment adheres to the counterstrategy (or the countertrace). Its vertices correspond to states in the game, edges correspond to state transitions which are possible in the game. This graph can be seen as a “cheat sheet” for the interactive game. It allows you to see how the environment will react to your outputs. Thus, you may discard some choices a priori. This reduces the number of plays necessary to understand the cause of unrealizability.

The graph is written in two version into the files `game_data/graph.dot` and `game_data/graph_with_signals.dot`. The latter contains the signal values that make a certain state transition possible, the former does not. Pictures of the graph can be produced with the DOT program by typing for example:

```
dot -Tpdf ./graph_with_signals.dot -o ./graph_with_signals.pdf
```

in a shell opened in the directory `game_data`. Detailed information to the graphs is written into the file `game_data/graph.info`. This file contains the signal values corresponding to the different vertices of the graph.

The current state of the play in the graph is displayed in the waveform labeled with `state` in `graph` of the input trace. For larger specifications, the graph tends to become huge. Huge graphs are no real help for the user, so their computation is aborted if they exceed 100 vertices. With the checkbox in the Game window, graph computation can be disabled completely.

**Special Waveforms:** The input trace in the Game window contains some waveforms that do not represent input signals. The special waveform `state` in `graph` contains the current state of the play in the graph as already explained in the previous paragraph. The waveforms labeled with `ix` and `jx` show the memory content of the counterstrategy as defined in [6]. The value of `ix` gives the index of the fairness constraint of the environment that will be fulfilled next. The value of `jx` gives the index of the fairness constraint of the system which the environment tries to evade. Thus, you can concentrate on fulfilling this fairness constraint only. The environment can change this index a finite number of times. The maximal number of changes of `jx` is contained in the waveform `jx changes`. All these values are addressed to advanced users, they can also be ignored.

## Example

This section illustrates on a concrete example how the game features can be used. We will use the specification depicted in Figure 38.<sup>6</sup> It defines a simple arbiter for some resource shared by two entities. With the inputs `r0` and `r1`, access to the resource can be requested by entity 0 and entity 1, respectively. With the outputs `g0` and `g1`, the resource is granted to the entities. The output `error` signals an error. Forget about `startup_failed` for a moment. All signals are initialized to 0 (`env init` and `sys init`). There is a guarantee that enforces that the resource is not granted to both entities at the same time (`sys tran 0`). There is a guarantee that ensures that no grant is given in case of an error (`sys tran 1`). Finally, there are guarantees that state that every request must be granted eventually (`sys fair 0` and `sys fair 1`). The assumption `env fair` is added so that NUSMV identifies the specification as a *Generalized Reactivity(1)* specification and not as a Büchi game specification. (Remember that games can only be played on *Generalized Reactivity(1)* specifications).

Signals				Requirements				
Name	Type	Kind	Notes	Name	Kind	Automaton	Property	Notes
r0	boolean	E	resource request by entity 0	env init	A		r0=0 && r1=0 && startup_failed=0	initial condition
r1	boolean	E	resource request by entity 1	sys init	G		g0=0 && g1=0 && error=0	initial condition
startup_failed	boolean	E	environment could not start up	sys tran 0	G		never(g0=1 && g1=1)	never more than one grant
g0	boolean	S	resource granted to entity 0	sys tran 1	G		always(error=1 -> (g0=0 && g1=0))	no grant on error
g1	boolean	S	resource granted to entity 1	sys fair 0	G		always(eventually!(r0=0    g0=1))	eventually a grant for entity 0
error	boolean	S	something went wrong	sys fair 1	G		always(eventually!(r1=0    g1=1))	eventually a grant for entity 1
				env fair	A		always(eventually!(TRUE))	no environment assumption

Figure 38: The specification used for Game demo.

When the button **Start** is clicked, the tool first checks if the specification is realizable. This specification is indeed realizable, so the tool starts a normal game. You can define values for the inputs and the tool responds with outputs that conform to the specification. A possible simulation run is depicted in Figure 39.



Figure 39: A possible simulation run.

Suppose now that you are not satisfied with the behavior of the system during simulation. Suppose that the original, informal design intent was that the output signal `error` has to be set indefinitely if `startup_failed` is always set after the first time step. This behavior cannot be observed in Figure 39. You can now switch into the **Specify Design Intent** - mode in order to use the simulation trace as starting point for the definition of the desired behavior.

<sup>6</sup>This example is included in the RATSY distribution. The corresponding project file is `examples/demo/DemoGame1.rat`.

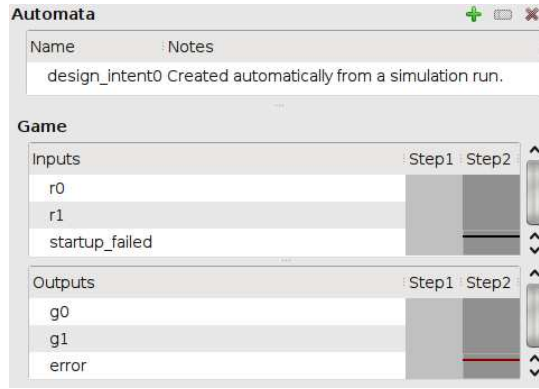


Figure 40: The specified design intent.

Figure 40 shows the result of the specification of the desired behavior. When `startup_failed = 1` right after the initial state until infinity, then so must be the output signal `error`.

Click the Done button, and an automaton is created automatically which accepts only the desired behavior. Add the automaton to the specification and obtain the specification depicted in Figure 41.<sup>7</sup>

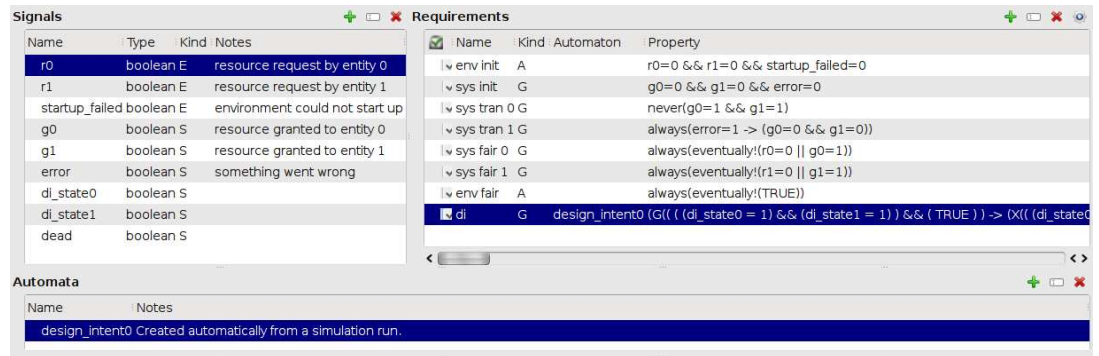


Figure 41: The new specification containing the desired behavior.

The play engine is reset with the button Stop and a new game with the enhanced specification can be started by clicking Start again. The tool finds out that the enhanced specification is unrealizable, so it starts a counter game in order to illustrate the cause of unrealizability. It first minimizes the specification. The tool says that the specification is still unrealizable if the system can choose the value of `g1` completely arbitrarily in every time step. It furthermore states that the specification is still unrealizable if the system does not have to fulfill `sys fair 1` (eventually a grant for entity 1) and `sys tran 0` (never more than one grant). The counter game is played on this simplified specification, where `g1`, `sys fair 1` and `sys tran 0` have been removed.

Next, the tool computes a counterstrategy and attempts to obtain a countertrace from it. Our heuristic is able to find such a countertrace. This countertrace is used in the counter game as depicted in Figure 42. It sets `startup_failed = 1` and `r0`

<sup>7</sup>This example is included in the RATSY distribution. The corresponding project file is `examples/demo/DemoGame2.rat`.

$= 1$  forever. Due to our design intent, error must be raised. Due to `sys tran 1` (no grant on error), no grant can be given. Additionally,  $r0 = 1$  forever, so the guarantee `sys fair 0` cannot be fulfilled. This explanation is also given by the tool in the Game Log of Figure 42.

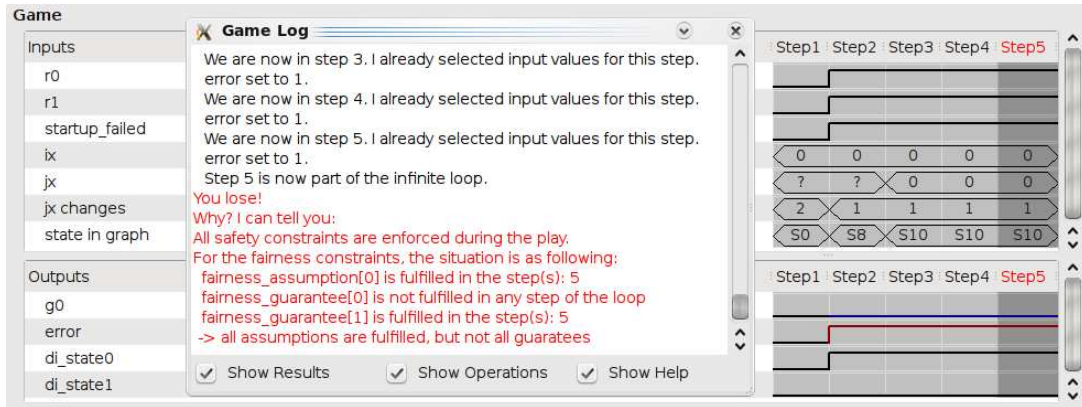


Figure 42: The countergame containing the countertrace.

Figure 43 shows the Automata window in Step 2 of the play. It contains only the automaton representing the design intent which we specified earlier. In Step 2, we are in the state V2. We have that `startup_failed = 1`, so we can only set `error = 1` in order to stay in V2, which is accepting. Setting `error = 0` would bring us to the state R2. This step is non-accepting and it cannot be left any more if not the environment sets `startup_failed = 0` (which it does not, following its countertrace). Clicking on one of the yellow edges in the Automata window makes `error = 1` or `error = 0` in the Game window.

The countergame helps you to understand the conflict between the specified design intent and the rest of the specification, that is, why the enhanced specification is unrealizable. The elimination of the problem is up to you, as there are typically various solution. You could allow grants to be given on error, you could restrict the fairness guarantees `sys fair 0` and `sys fair 1` to cases where `error = 0`, you could add an assumption that forbids that `startup_failed = 1` forever, etc.

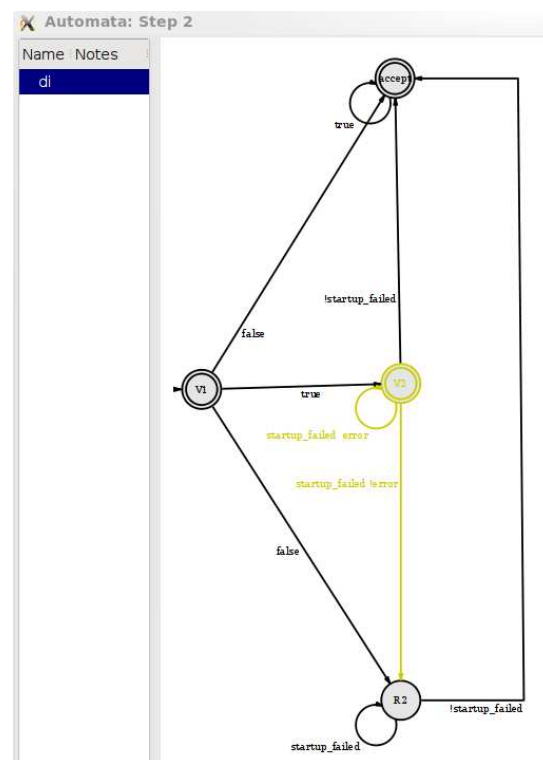


Figure 43: The state of the play in an automaton.



## 2 RATS Y Architecture

In the following the design and implementation of RATS Y will be discussed. The general information about RATS Y implementation and run time environment will be described in Section 2.1. Section 2.2 explains architectural patterns used during RATS Y development. The hierarchy of the RATS Y software is described in Section 2.3.

---

### 2.1 Architecture and Implementation Notes

RATS Y is a stand-alone multi-platform application that runs in one process. Even if multi-threading is used to run external verification engines, the GUI part fits into a single main thread.

RATS Y has been fully developed with the *Python* object-oriented programming language, and the GUI part relies on the *PyGTK* graphical toolkit to draw itself to the screen, and to handle the interaction with the user.

The coding followed a few standards "de facto". Classes, methods and functions names follow *PyGTK*'s convention (see <http://www.pygtk.org>), that derives from the *GTK*'s one (see <http://www.gtk.org>). Style and indentation are strictly *Python* compliant. Packages and filenames are java style, but slightly less restrictive: e.g. a file `foo_and_foo.py` contains definition of class `FooAndFoo`, but may contain the definitions of other classes if convenient.

RATS Y uses external tools to check properties for Property Assurance, Simulation, Realizability, and Synthesis. In particular currently it relies on the NUSMV and VIS model checkers that are written in Posix C language. Furthermore it uses the MARDUK tool, written in Python, which in turn uses some functions from NUSMV via a wrapper. The tools are called and used by RATS Y as external processes, and are kept separated from RATS Y by an abstraction layer called *Stub* that exports a standard interface. The MARDUK tool is partially tighter integrated with RATS Y, since both are written in Python. Especially the Game features rely on this integration.

RATS Y is based on several other software entities, that affect its software architecture. The picture in Figure 44 shows the main set of layered software entities which RATS Y relies on. The layers depict the dependencies among the entities, as higher parts depend on lower parts.

At the top is positioned the RATS Y Application, gray shaded to make it clearly distinguishable from the other parts.

The single parts are described in the following from the bottom to the top.

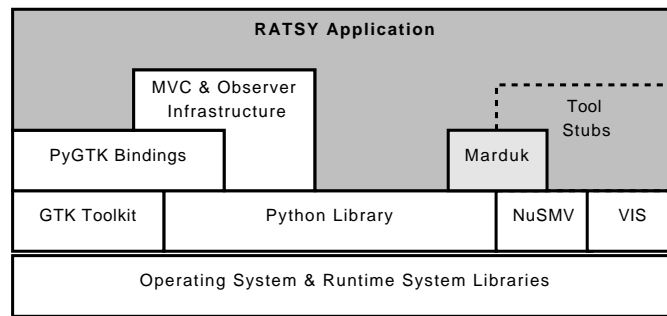


Figure 44: RATS Y- Software parts and collocation

**Operating System & Runtime System Libraries** Those depend on the specific architecture implemented on the host computer. Currently RATS Y has been tested under *GNU/Linux* with a 2.4 and 2.6 kernel.

**GTK Toolkit** GTK is a set of libraries that provide a pretty platform independent support for drawing and handling graphical widgets like windows, buttons, text entries, fonts, etc. See <http://www.gtk.org> for further information about GTK and its components.

**Python Library** This is a general multi-platform runtime environment provided by the *Python* environment. It provides a large set of features and data structures to be used from any *Python*-based application. It also provides a portable abstraction layer over the underlying Operating System, making the application platform independent. See <http://www.python.org> for further information.

**NuSMV and VIS** These are the Model Checkers RATS Y is currently based on.

**MARDUK** is a command-line *Python* program for synthesis and specification debugging.

**PyGTK Bindings** This is a *Python* binding that allows *Python* programs to use the GTK Toolkit. See at <http://www.pygtk.org> for further information.

**MVC & Observer Infrastructure** This is a *Python* package that helps to design and develop GUI applications. It implements the *Model-View-Controller* and the *Observer* patterns developed specifically for *PyGTK*.

**RATS Y Application** This is the set of *Python* packages that implement the RATS Y application. The underlying layers make RATS Y platform independent, and the internal sub-part *Tool Stubs* insulates RATS Y even from the model checkers.

## 2.2 Architectural Patterns

RATS Y has a pretty complex structure, as it currently fits in six packages, about 68 modules and 21400 lines of *Python* code (including comments, excluding blank

lines). RATS<sub>Y</sub> is characterized by strongly interconnected features, and by the need of horizontal communication among independent parts. Furthermore, it provides many different independent views over the same objects, and those views are often potentially editable by the user. Whenever one of those view is changed by the user or by RATS<sub>Y</sub> itself, all the other should react accordingly.

To reduce the structural complexity, to keep a clean design, and to minimize the development and maintenance costs, two architectural patterns were considered: The Model View Controller (*MVC*) and the *Observer* patterns, see [4].

## The Model-View-Controller pattern

*MVC* is an architectural pattern that forces the designer to break up the application being designed among three main parts: a Model, a View and Controller. The traditional implementation of this pattern reflects the normal data flow of non-GUI applications: data input, data processing, and result presentation. Historically, the *MVC* pattern is an attempt to map this natural data flow to the GUI design. In fact, it associates the data input to the Controller, the data processing to the Model, and the result presentation to the View.

In RATS<sub>Y</sub> this pattern is implemented in the *MVC and Observer Infrastructure*. This implementation wanted to be different from the traditional one, as it is specific for the underlying graphical toolkit (*PyGTK*) and language (*Python*) to exploit their peculiarities and features. In particular, a part of the traditional View's features have been moved to the Controller, and the model has been made not aware of the existence of any Controller or View. In combination with the *Observer* pattern (see next section), this allows for a real separation of the application logic from the presentation layer.

**Model** Contains the logic of the program, intended as data and data manipulation routines. Models can communicate with other models (especially with models that they contain), but do not know the other parts of the *MVC* pattern, namely the Controller and the View. This limitation guarantees the insulation between the application logic and presentation.

**View** Contains the presentation layer. The View constituted by a set of graphical widgets organized as a forest (typically a single tree). A single *widget* is one atomic GUI element, like a button, a text label, a window, etc. Often widgets are containers for other widgets, hence widgets are organized in trees, where vertices represents the containment relations. As for the models, views do not know the models they are connected to, as the connection is delegated to the controllers. This is another variation with respect to the original *MVC* pattern, as this implementation is intended to fit better with the *PyGTK* toolkit.

**Controller** Contains the actions that must be carried out when a view event requires the interaction with the model's logic. The Controller is always connected to a single Model, and to a single View, making a sort of link among

these two separated parts of the pattern. If a Controller can be connected to one Model, the same model can connect more controllers at a given time.

## The Observer pattern

The *Observer* pattern connects the application logic to the presentation layer, by allowing the latter to be notified when the former changes.

The *Observer* pattern is often used together with the *MVC* pattern, and to a certain extent it may be considered as complementary, as it handles the data flow from the model to the view, whereas in the *MVC* pattern the communication goes generally from the View to the Model through the Controller.

This communication is carried out without making the model even know the existence of the view, by using observable properties within the model, and by defining observers over those properties. The observers will be notified of any changing that occur to the observable properties.

In RATS<sub>Y</sub> the *MVC and Observer Infrastructure* provides an implementation for both the patterns. In particular, any Model can contain observable properties, and any Controller is by default an Observer for the Model it is connected to.

---

## 2.3 Software Structure

The software structure of RATS<sub>Y</sub> is strongly affected by the patterns it is based on, and by the other software entities it relies on, that have been already shown in Figure 44.

The main part of RATS<sub>Y</sub> is represented by its core, fully based on the *MVC & Observer Infrastructure*. At the core sides, there exist services and resources, that are available transversally to the core. Figure 45 provides more details about the core and the provided services.

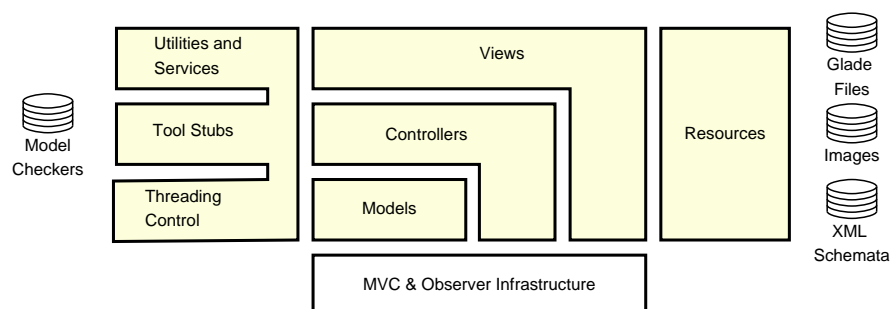


Figure 45: RATS<sub>Y</sub>- Software Structure

At the leftmost side of Figure 45 are depicted the most important services that are available to models, controllers and views. These services do not fit well with the *MVC* and *Observer* patterns as they do not have any associated view, or any user interaction.

**Utilities and Services** Contains general utilities, globally accessible data, etc.

**Tool Stubs** Stubs are those entities that isolate RATS Y from the external Model Checkers. Stubs export an interface known to RATS Y, and each model checker has an associated stub. The result is that RATS Y can call a model checker careless of the specific Model Checker it is actually calling.

**Threading Control** Provides fine-grained portable control over threads. This service is used for example in stubs invocation, for running the model checkers in background, for controlling the associated process, and for capturing its output.

At the rightmost side of Figure 45 are depicted those resources that are exclusively used by the RATS Y Views. Noticeable resources are:

**Glade Files** As already mentioned, a Views is a forest of widgets. The widgets can be build and connected each other by hand, or by using programming tools like *glade* (see <http://glade.gnome.org>). This tool can be used to visually design a forest of widgets representing the view's widgets. With very few limitations, this tool can be used then to set the properties of all widgets, and to associate action to be carried out when a certain events occur (signals). For example a widget like a button can be associated with a function name to be called when clicked. The result of this creation and setting process is a glade file, that can be loaded at runtime by the *MVC and Observer Infrastructure* that provides the needed support for Views creation based on glade files, and to connect the associated Controllers that provide the implementation of signals actions.

**Images** Contains icons, and other images to be shown by the views.

## Tools Stubs

As already mentioned, the interaction with the model checkers like NUSMV and VIS is managed by a *Stub*, a software entity that provides platform and Operating System independent support for running generic external model checkers. The execution of a model checker is restricted to a stand-alone thread that controls the model checker within a *session*. The session is monitored, and can be stopped at any time if the underlying Operating System supports process interruption. Also, the stub provides access to the session I/O, allowing to capture the model checker standard output and error, and to control its standard input.

A stub execution is a sequence of events:

1. The stub is initialized.

2. A session is initialized.
3. The session is prepared (setting of session options).
4. The session is run.
5. Session results are processed.
6. The session is de-initialized.
7. The stub is de-initialized.

The phases from 2 to 6 may be possibly repeated indefinitely.

A generic stub might control a model checker in any way, either in batch mode, in interactive mode or through its library. In RATS<sub>Y</sub> the stubs that control both NUSMV and VIS use the model checkers in batch mode, launching their respective executable files. This is achieved by specializing the generic stub classes, by implementing some interfaces and overloading some class methods that handles the execution of a single session in batch mode.

## A vertical view over the Software Structure

The RATS<sub>Y</sub> software structure has been split horizontally by using the *MVC and Observer Infrastructure*. There exists also a vertical splitting that breaks the software structure up through a hierarchy of software entities.

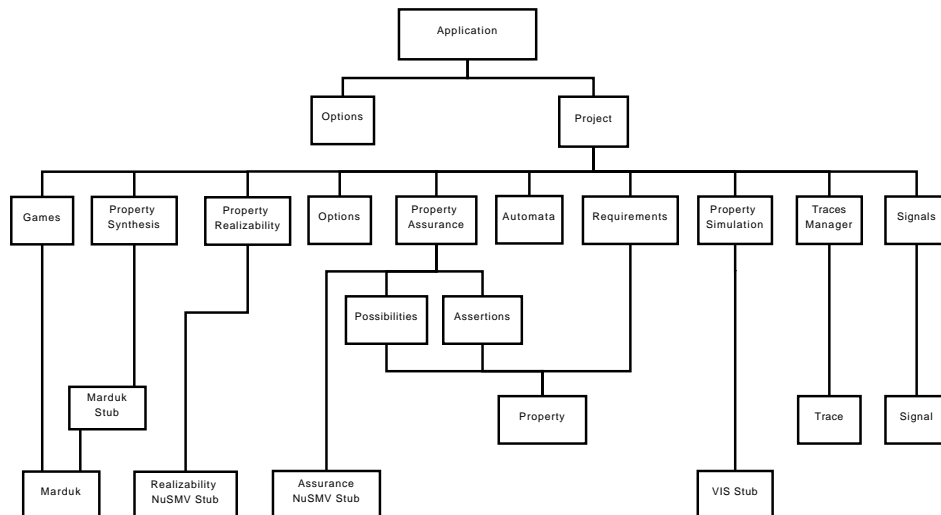


Figure 46: RATS<sub>Y</sub>- Hierarchy of main software entities

Figure 46 depicts the hierarchy of the main software entities that occur within RATS<sub>Y</sub>. Each of the boxes represents a software entity, and each vertex of the hierarchy tree is a containment relation, where cardinality is not expressed. That means for example that an Application contains one (or more) software entities to represent a Project and the Options of the Application.

The way each software entity is implemented depends on the entity's role. Those entities that need to be shown, will follow the *MVC* pattern, and will be mapped

down to three object-oriented classes (or to a triple of a limited set of classes) to associate to each entity a Model, a View and a Controller. For example, the entity application's Options has a model to hold the options, and a couple View/Controller to present the options to the user, and to allow the user to modify the options. Those entities that instead do not need to be shown (e.g. the stubs), will be mapped directly down to one class, or to a set of classes.

In the following the software entities depicted in Figure 46 are detailed.

**Application** The application is the top-level entity. When the RATSY executable file is run, a triple Model, View and Controller of this entity will be instantiated and connected each other, and RATSY will finally enter in the main event loop to handle user interaction and events.

**Application Options** This entity is a container for application's options. For example tools paths, and other general purpose options should be localized within this entity. At the moment this entity is empty, and there is not an associated View for it.

**Project** This entity represents a RATSY project. The project's model contains most of the application logic, meaning that most of the application's models are contained within this model. The view is embedded within the application's main window whenever a project is created, and it is constituted by a large number of sub-views corresponding to the contained entities.

**Project Options** This entity is a container for the project's options. Similarly to the Application Options entity, this entity is currently empty, and there is no associated view.

**Signals** This entity contains the set of signals used by Property Assurance, Realizability, Synthesis, and the Games.

**Requirements** This entity contains the set of requirements used by Property Assurance, Realizability, Synthesis, and the Games.

**Automata** This entity contains the set of automata. Automata can be instantiated to Requirements.

**Property Assurance** This is the entity for Property Assurance. Its view is shown when the Property Assurance feature is selected at the application level.

**Property Simulation** This is the entity for Property Simulation. Its view is shown when the Property Simulation feature is selected at the application level.

**Property Realizability & Synthesis** This is the entity for Property Realizability & Synthesis. Its view is shown when the Property Realizability & Synthesis feature is selected at the application level.

**Games** This is the entity for playing games. Its view is shown when the Game feature is selected at application level. This entity is quite interweaved with MARDUK, and hence, accesses MARDUK directly and not via a stub.

**Traces Manager** This entity handles the set of traces that have been generated in the project. Also, this entity organizes the set of traces within a set of categories that traces belong to.

**Assurance NUSMV Stub** The Property Assurance NUSMV stub handles the interaction of RATSY with the NUSMV model checker when Property Assurance is run. This entity has no associated View and Controller, and it is implemented by a single class. This class is the specialization of a more generic classes hierarchy that provides support for implementing specific tool stubs.

**Realizability NUSMV Stub** The Property Realizability NUSMV stub handles the interaction of RATSY with the enhanced version of NUSMV [5] when Property Realizability is run. This entity has no associated View and Controller, and it is implemented by a single class. Similarly to the Property Assurance NUSMV Stub already available in RATSY, this class is the specialization of a more generic classes hierarchy that provides support for implementing specific tool stubs.

**MARDUK** The MARDUK tool handles synthesis as well as strategy computation for the Game features.

**MARDUK Stub** Like the NUSMV Stubs entities, but specific for MARDUK.

**Possibilities** Contained within the Property Assurance entity, this entity represents the set of possibilities for Property Assurance.

**Assertions** Contained within the Property Assurance entity, this entity represents the set of assertions for Property Assurance.

**Signal** This entity represents a single signal. The model contains information about the signal, like the name and type information. The view is shown when the user wants to create or edit a signal.

**VIS Stub** Like the NUSMV Stubs entities, but specific for the VIS model checker.

**Trace** A trace is the result of model checking, and can represent either a witness or a counter-example. In RATSY there exist several views over a trace, as they can occur within the main application window, and within the Trace Manager window. In general a trace can be shown as a graphical waveform, with some associated information like the category it belongs to, the number of steps, the loop information, etc.

**Property** This entity represents a single property, like a requirement or a possibility. The model contains information about the property, like the name and formula. The view is shown when the user wants to create or edit a property. There exists a dependency between a property and those traces that were generated from it. Whenever a property's formula is changed, the corresponding traces will be invalidated.

More information about RATSY implementation details can be obtained in [2].

# 3 RATS Y Installation and Distribution

This section gives information on installation and distribution related issues. RATS Y can be downloaded in the form of binaries for 32-bit and 64-bit Linux systems, and also as a source tree.

---

## 3.1 Installing the Binary Distribution

To start RATS Y from the binary distribution, simply extract the downloaded archive into any directory and start the script `ratsy/ratsy`. For more convenience, you can add the `ratsy` folder to your `PATH` environment variable.

The archive contains binaries of all external tools such as the model checkers VIS and NUSMV. You do not need to download and install them separately. The one exception is the tool LILY, which is needed to perform realizability checks on full LTL specifications (not only on specifications given in *Generalized Reactivity(1)* format). If you do not need this feature, then you do not have to install LILY. If you do, simply download LILY<sup>8</sup>, extract the archive, patch it with `NuSMV-game/nugat/contrib/Lily-1.0.2.patch`, and include it into your `PATH` and `PERL5LIB` environment variables.

---

## 3.2 Installing the Source Distribution

To build RATS Y and all the external tools such as the model checkers VIS and NUSMV from source, simply extract the downloaded source archive into any directory and execute the `build.sh` script in the top-level directory. Follow the instructions of this script. As for the binary distribution, if you need support for full LTL realizability checking, you have to install LILY (see Section 3.1). When the build process has finished, the script `ratsy/ratsy` starts up RATS Y.

Known issues:

---

<sup>8</sup>[http://www.iaik.tugraz.at/content/research/design\\_verification/lily/](http://www.iaik.tugraz.at/content/research/design_verification/lily/)

- The NuSMV wrapper does not compile with Swig Version 1.3.39 or above installed. The reason is that Swig changed interface names (see <http://www.swig.org/Release/CHANGES> at date 2008-12-04) without backward compatibility. As a workaround you could
  - use the binary distribution,
  - downgrade Swig to Version 1.3.38 or below, or
  - use the patched<sup>9</sup> file `NuSMV-game/NuSMVWrap/dd.ini.swig.1.3.39`.

---

### 3.3 Running MARDUK

No matter whether you use the binary distribution or the source distribution, in order to run the MARDUK tool stand-alone, go to the `marduk/src` folder and launch the file `marduk.py` with your Python interpreter. Make sure to set your environment variable `LD_LIBRARY_PATH` such that it also includes the directory `NuSMV-game/NuSMVWrap/nusmv/club`. This is necessary for MARDUK to find and use the NUSMV wrapper. Run `python marduk.py -h` to display a help message, detailing the options and arguments of MARDUK. In order to test whether the installation was successful, you can run the script `marduk/src/test_marduk.sh`.

---

### 3.4 Licensing

RATSY and MARDUK are distributed under GNU LESSER GENERAL PUBLIC LICENSE Version 2.1, February 1999 (LGPL) with the copyright held by Graz University of Technology and FBK-irst. See `ratsy/License` for a copy of this license.

NUSMV (<http://nusmv.fbk.eu>) is distributed under the same license with the copyright held by FBK-irst only. See `NuSMV-game/nusmv/README` for details. Since the same licence applies, NuSMV sources are included in the source distribution of RATSY for convenience.

VIS is available under a different (and even less restricted) licence. See <http://vlsi.colorado.edu/~vis/> for details. The VIS sources are not included in the source distribution of RATSY. For convenience, the build script of RATSY automatically downloads the VIS sources, however.

Note that the license for RATSY, VIS sources and NUSMV sources allows for commercial use (currently the use of VIS and NUSMV takes place in commercial settings).

---

<sup>9</sup>[https://swig.svn.sourceforge.net/svnroot/swig/trunk/Tools/pyname\\_patch.py](https://swig.svn.sourceforge.net/svnroot/swig/trunk/Tools/pyname_patch.py)

The development of the first version of this tool (RAT) has been supported in part by the European Union under contract 507219 (PROSYD)<sup>10</sup>. The current version (RATSY) has been supported by the European Union under contract 217069 (COCONUT)<sup>11</sup> and 248613 (DIAMOND)<sup>12</sup>, as well as by the Provincia Autonoma di Trento (project EMTELOS).

---

<sup>10</sup><http://www.prosyd.org/>

<sup>11</sup><http://www.coconut-project.eu/>

<sup>12</sup><http://www.fp7-diamond.eu/>



# 4 References

- [1] R. Bloem, R. Cavada, A. Cimatti, I. Pill, M. Roveri, S. Semprini, and A. Tchaltev. RAT: A tool for formal analysis of requirements. In *Demo Session of the 17<sup>th</sup> European Conference on Artificial Intelligence*, Riva del Garda, Italy, 2006.
- [2] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and property assurance tool, November 2005. Prosyd Deliverable D1.2/4-5.
- [3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware form PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007. Electronic Notes in Theoretical Computer Science <http://www.entcs.org/>.
- [4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley & Sons Ltd., West Sussex, England, 1996.
- [5] A. Cimatti, M. Roveri, and A. Tchaltev. Manual for property realizability tool, December 2006. Prosyd Deliverable D1.2/8.
- [6] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Formal Methods in Computer-Aided Design (FMCAD'09)*, 2009. To appear.
- [7] S. Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):156–170, 2001.
- [8] NuSMV home page. <http://nusmv.fbk.eu/>.
- [9] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In Ellen Sentovich, editor, *Design Automation Conference (DAC)*, pages 821–826. ACM, 2006.
- [10] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.
- [11] PROperty based SYstem Design PROSYD. <http://www.prosyd.org/>, 2006.
- [12] Accellera, Property Specification Language - Reference Manual - Version 1.01. [http://www.eda.org/vfv/docs/psl\\_lrm-1.01.pdf](http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf), April 2003.
- [13] RAT — Requirements Analysis Tool. <http://rat.fbk.eu/>.