RAT

Requirements Analysis Tool

Version 1.2

Authors
Roderick Bloem, Roberto Cavada,
Alessandro Cimatti, Ingo Pill,
Marco Roveri, Simone Semprini and
Andrei Tchaltsev

Notices

For information, contact RAT (rat@fbk.eu).

This tool has been partially developed within the PROSYD European project, contract number 507219. (http://www.prosyd.org)

Contents

Table of Figures

List of Tables

1 RAT Users Manual

The tool RAT fulfill the need for a proper technological support to formal methods in the setting of requirements analysis by providing its users with the integration of three sets of functionalities that enact the Property Simulation, Property Assurance and Property Realizability methodologies. In this section we show how to interact with RAT in order to accomplish the tasks related to these three methodologies.

All the examples in the following sections are written in the Verilog flavor of PSL as from [8], the language supported by the verification engines VIS and NUSMV.

1.1 Running RAT

RAT can be execute from the command line by the following command

rat - *Launches the python interpreter to execute* RAT *program* — Command

```
rat [-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 RAT when the tool is launched without any project as argument.

The unit of interaction with RAT 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 inspect their quality, must have the possibility to work in different sessions and of saving the results of the work performed from session to 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.

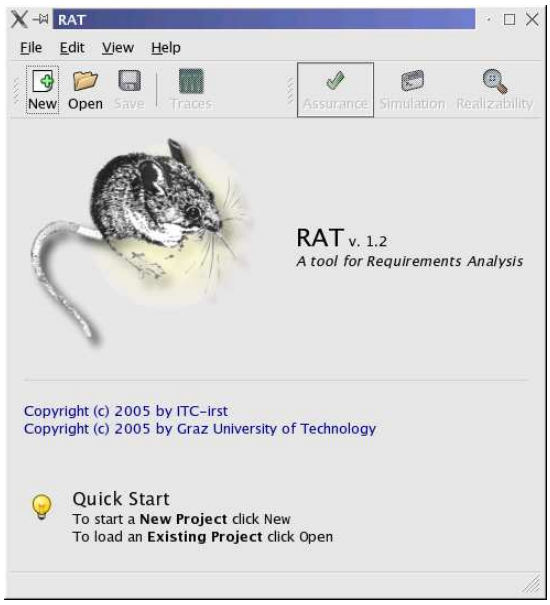

Figure 1: RAT- Main window.

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: RAT- New project wizard.

As a result of the integration Property Simulation, Assurance and Realizability into RAT (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 possible 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.

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

1.2 Property Assurance in RAT

RAT enacts the Property Assurance Methodology (see [2] Section 2.2) by supporting the users in Property Assurance related tasks; RAT 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

RAT 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.

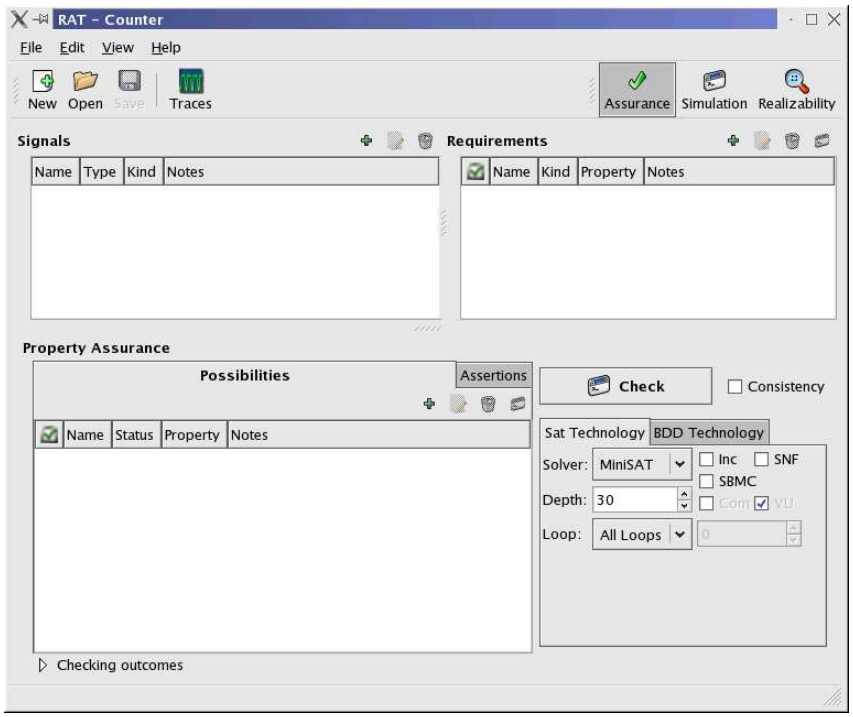

Figure 4: Property Assurance main window.

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.

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

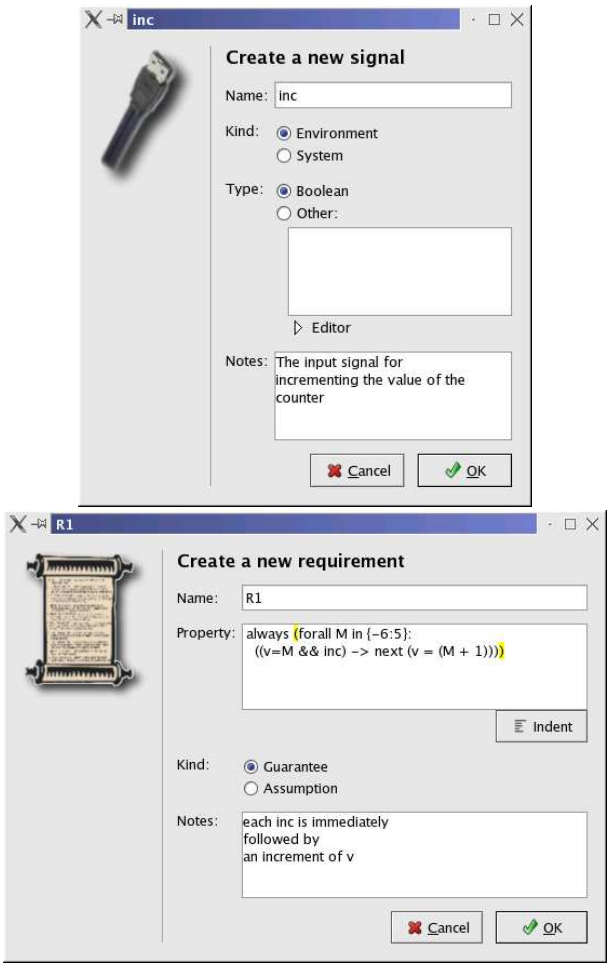

Figure 5: Creating signals, requirements.

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, 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

Figure 6: Creating possibilities and assertions.

perform Property Assurance related checks. The two tabs, shown in Figure 7, allow to chose 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 [5].

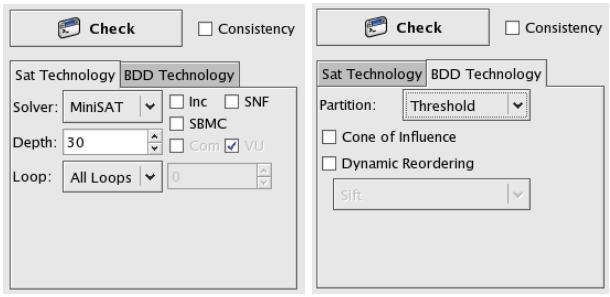

Figure 7: Verification panels.

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.

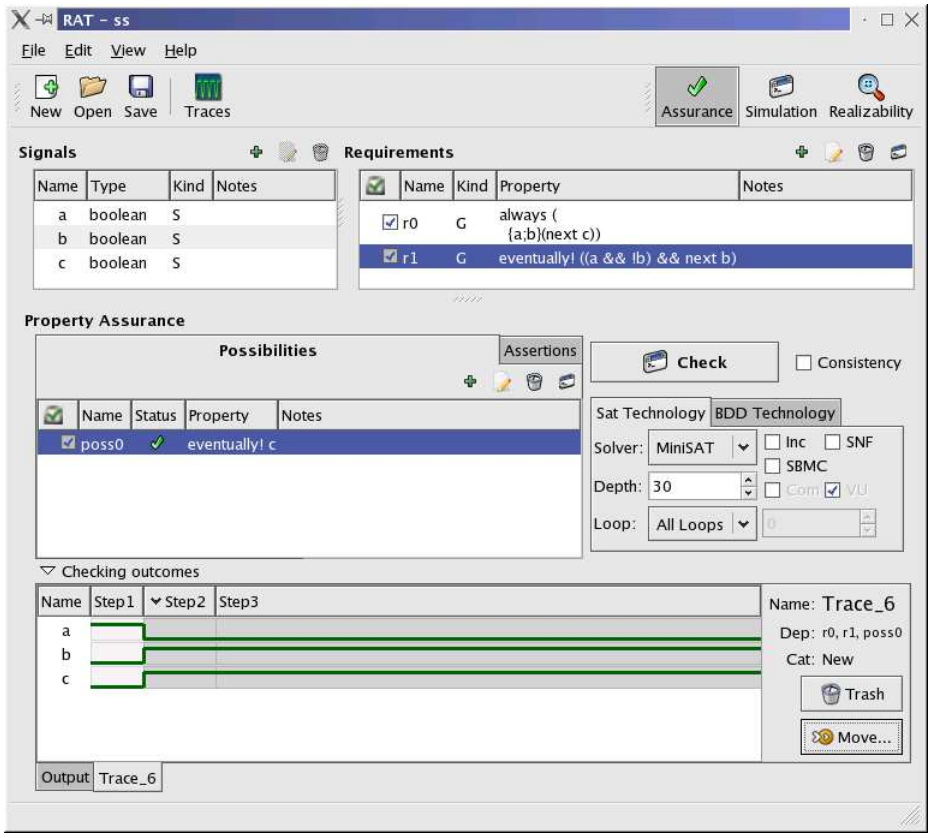

Figure 8: An example of trace visualization.

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.

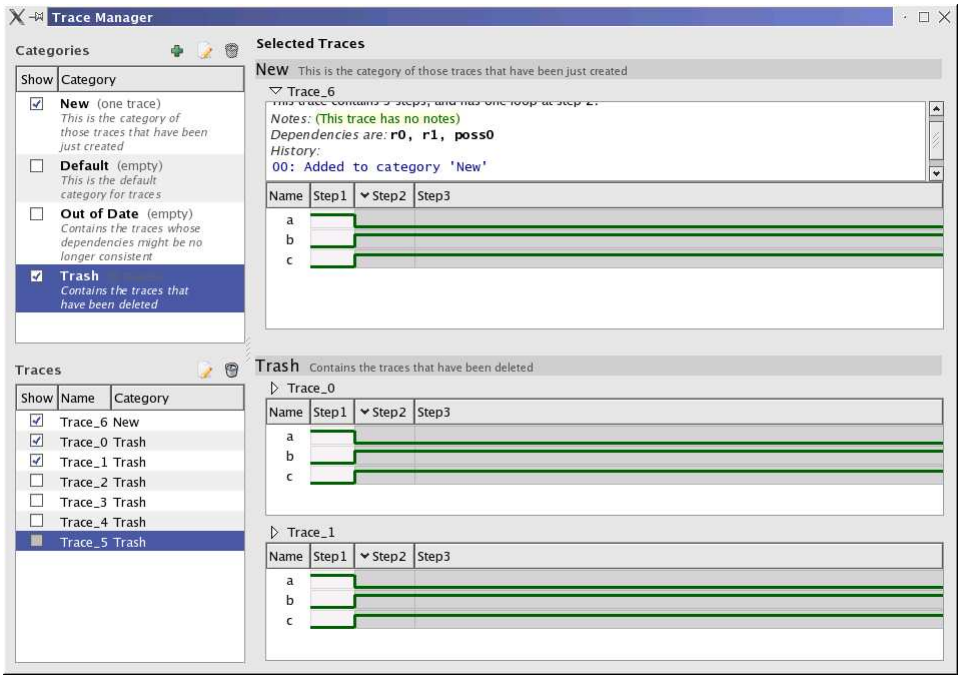

Figure 9: An example of trace visualization.

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.

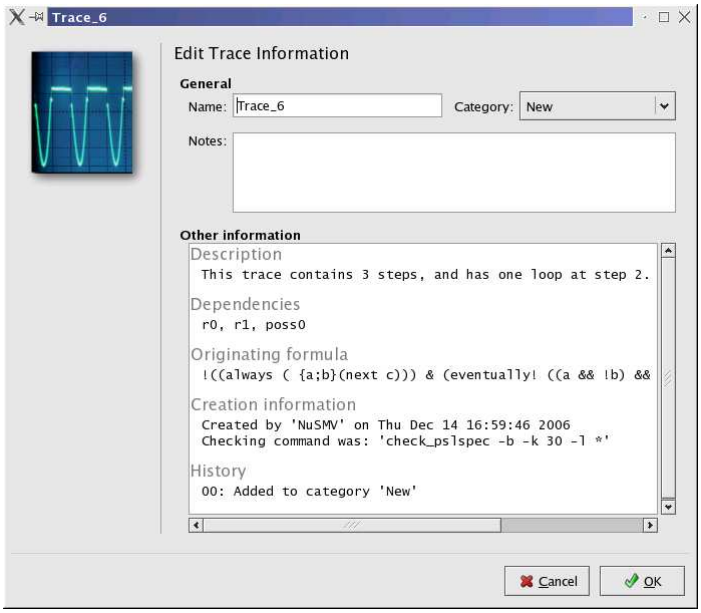

Figure 11: Editing a trace.

An Example

In this section we work out a simple but meaningful example that covers the most relevant Property Assurance features of RAT, 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 naivë specification could be the one shown in Figure 12.

The specification is based on the following signals:

`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

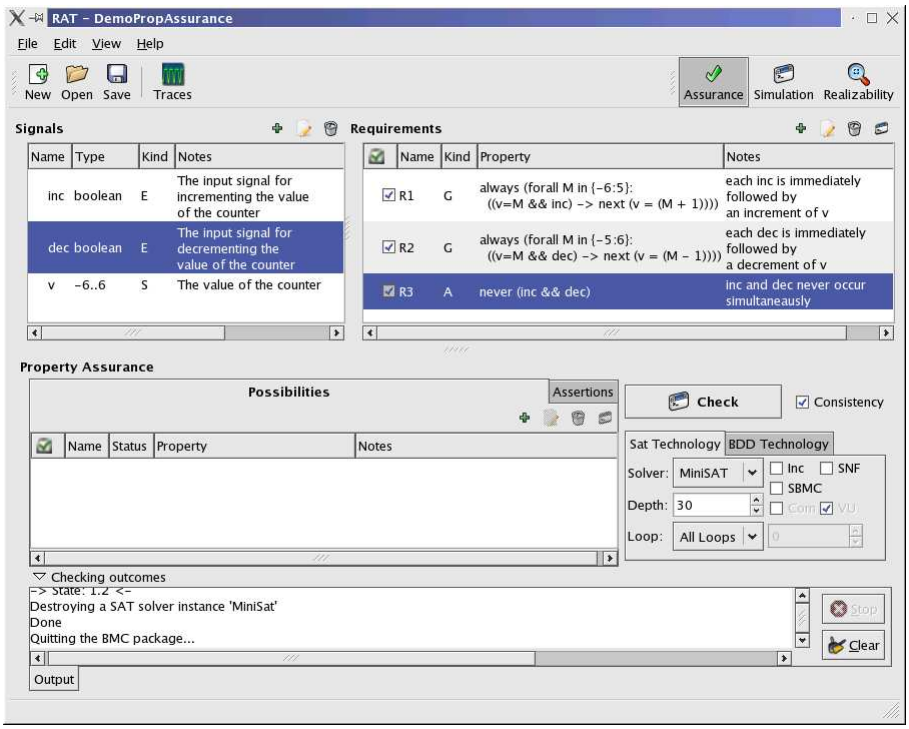

Figure 12: Counter - initial specification.

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 RAT. 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.

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

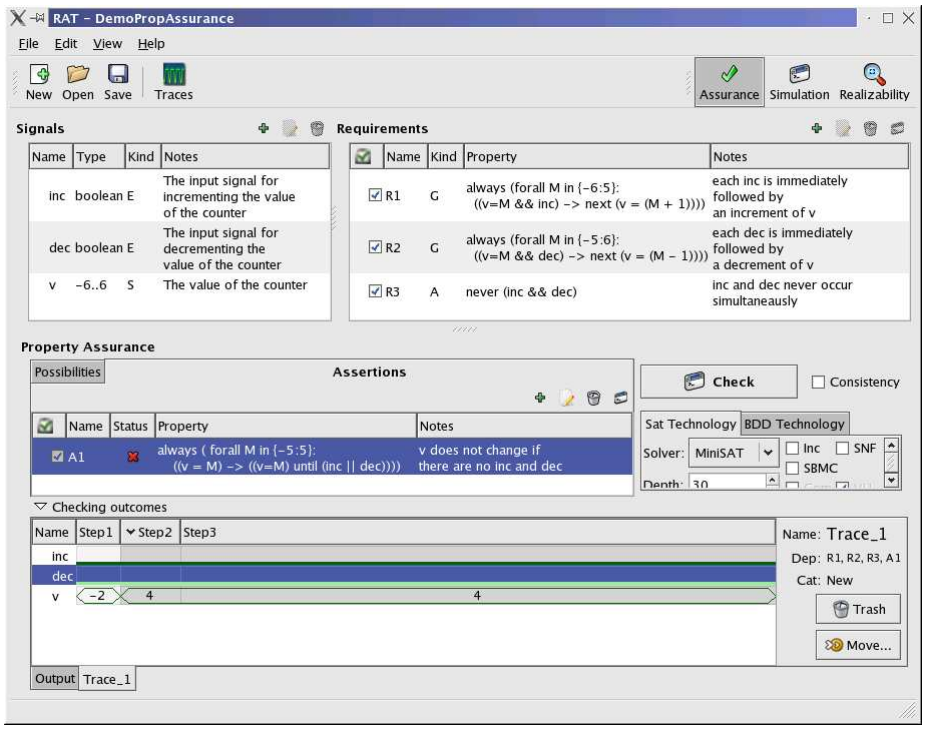

Figure 13: Counter - checking an assertion.

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 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

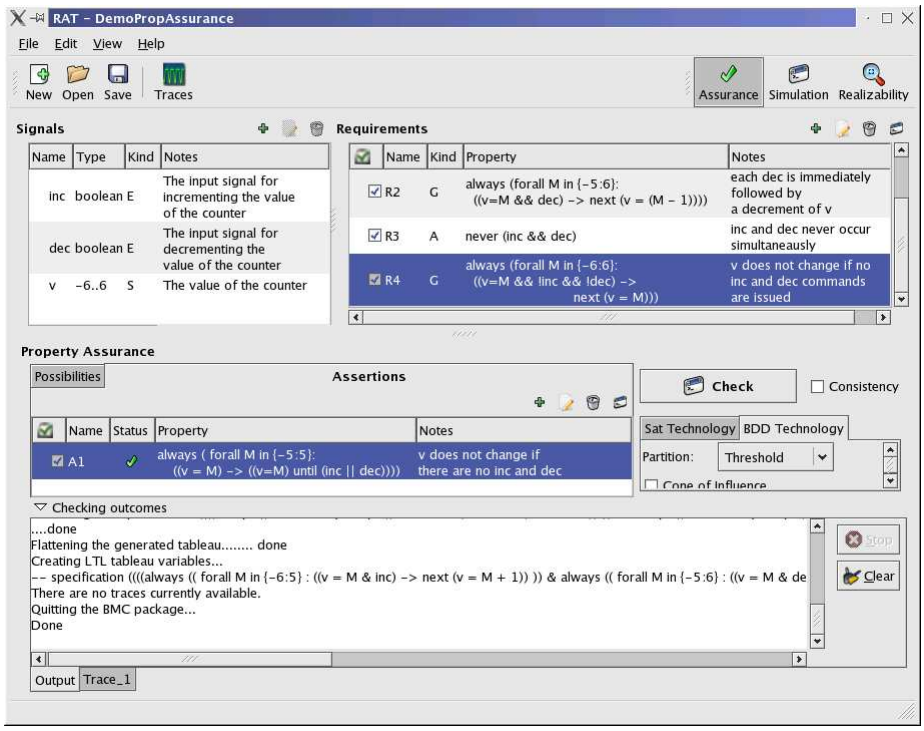

Figure 14: Counter - fixing the specification.

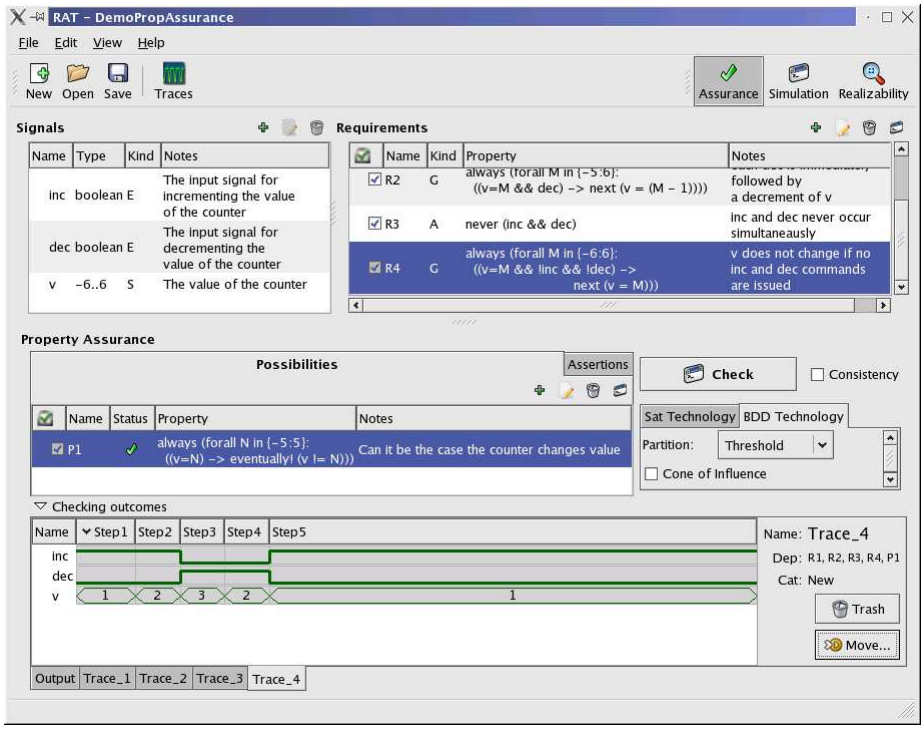

Figure 15: Counter - checking a possibility.

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).

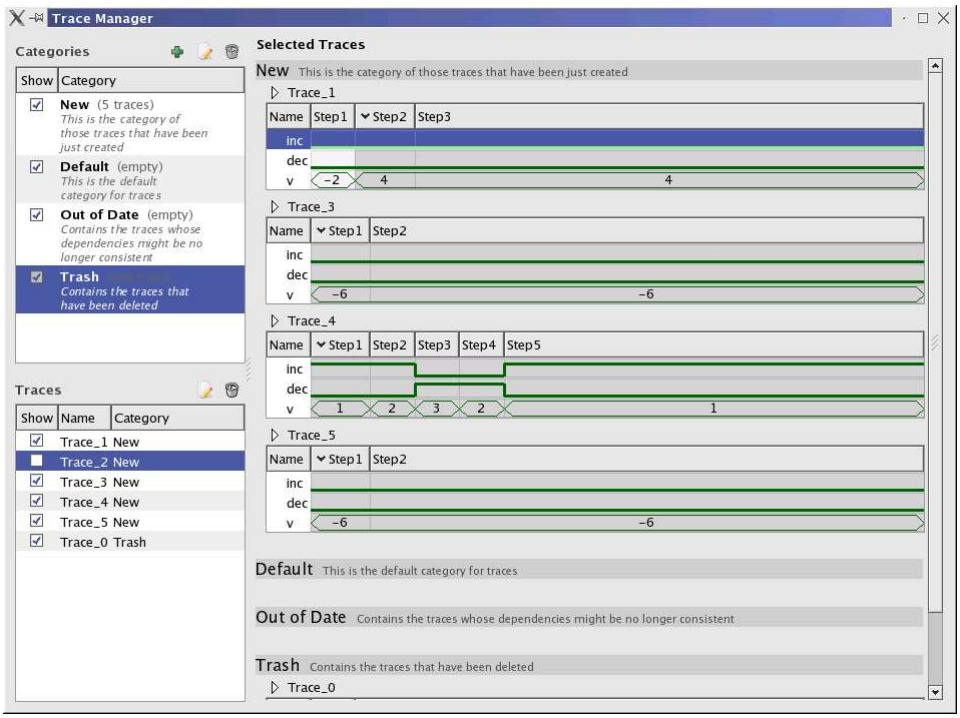

Figure 16: Counter - traces of the session.

1.3 Property Simulation in RAT

Note: Property Simulation is not supported within the COMPASS Project. However, for completeness of this methodological section Property Simulation is described here for the sake of readability.

This section illustrates the RAT 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 RAT you will see the RAT 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.

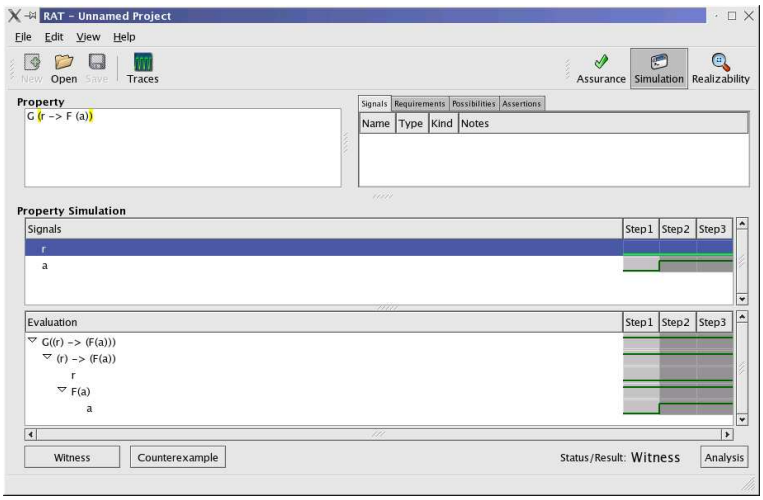

Figure 17: Property Simulation Main Window.

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 RAT 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 RAT to provide a trace contradicting the property or possible feature requests.
- **Status:** At this location you can always see what RAT 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.

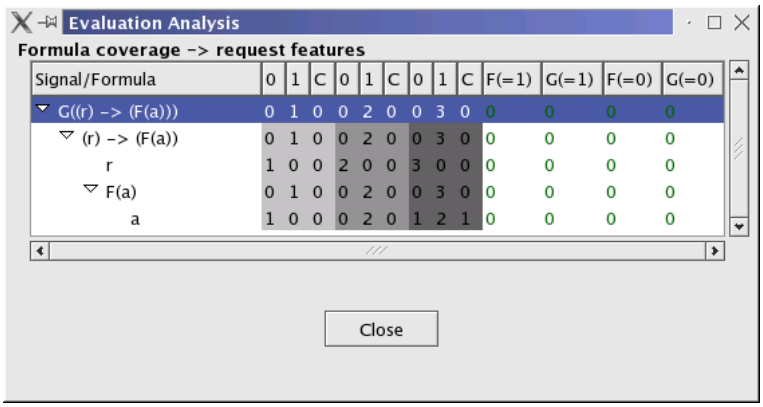

Figure 18: Property Simulation Evaluation Analysis Window.

An example

This section illustrates RAT 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.

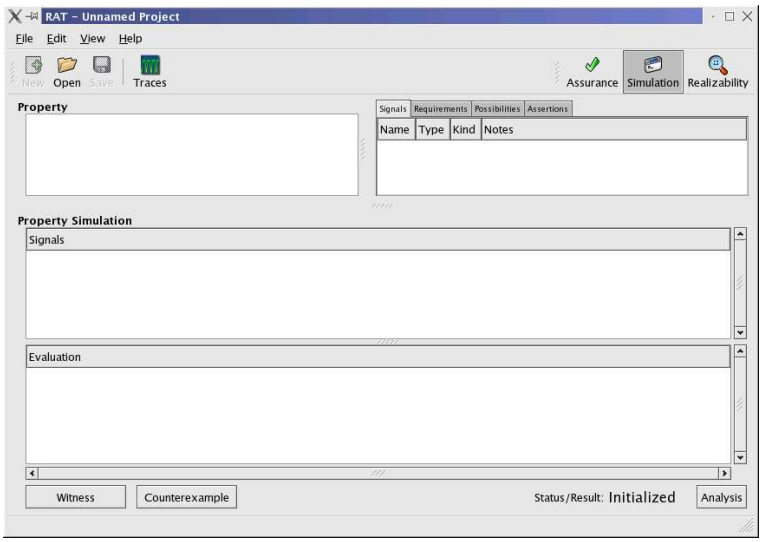

Figure 20: Property Simulation Start Window.

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.

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

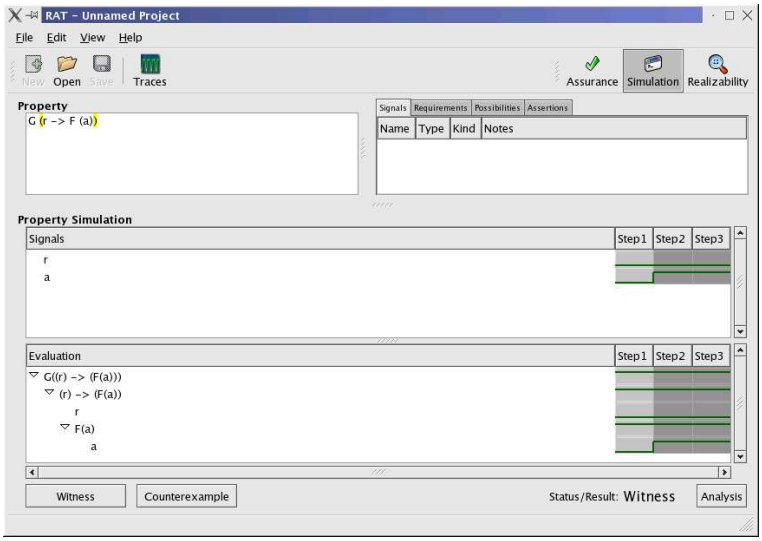

Figure 21: Witness for property $G(r \mapsto F(a))$.

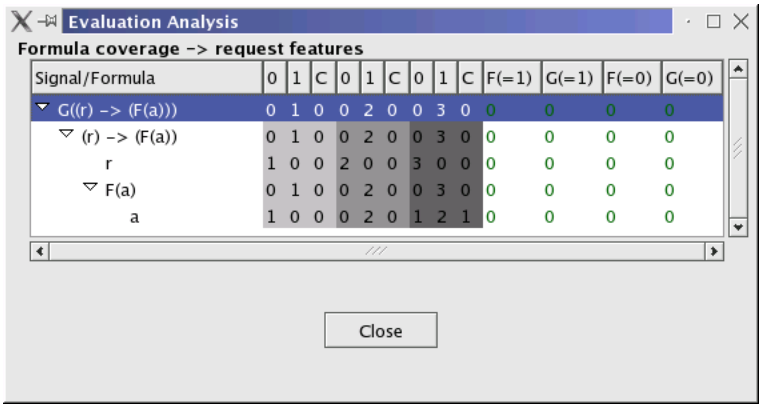

Figure 22: Analysis of trace for property $G(r \mapsto F(a))$.

we now press the analysis button to show the coverage information illustrated by Figure 22.

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 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.

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 brack-

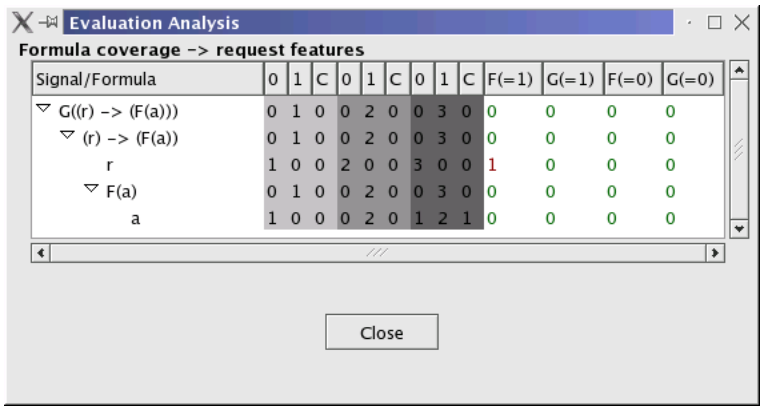

Figure 23: Ask for a request on signal r.

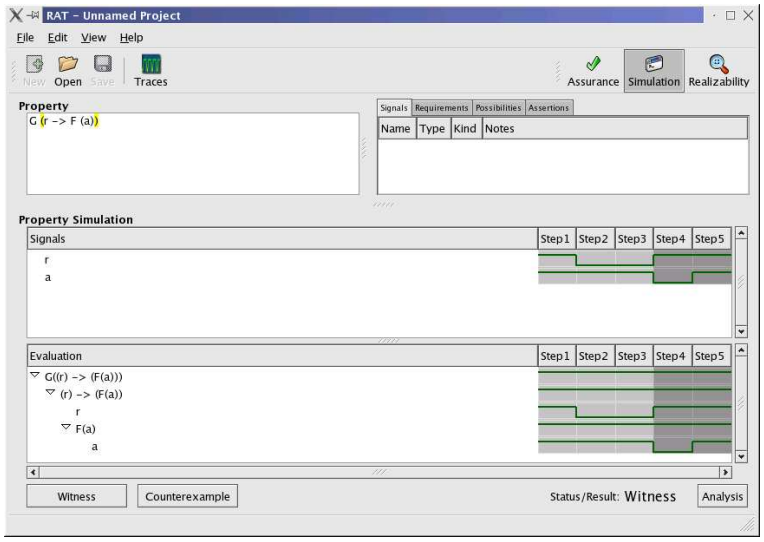

Figure 24: Witness with request for property $G(r \mapsto F(a))$.

ets 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).

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.

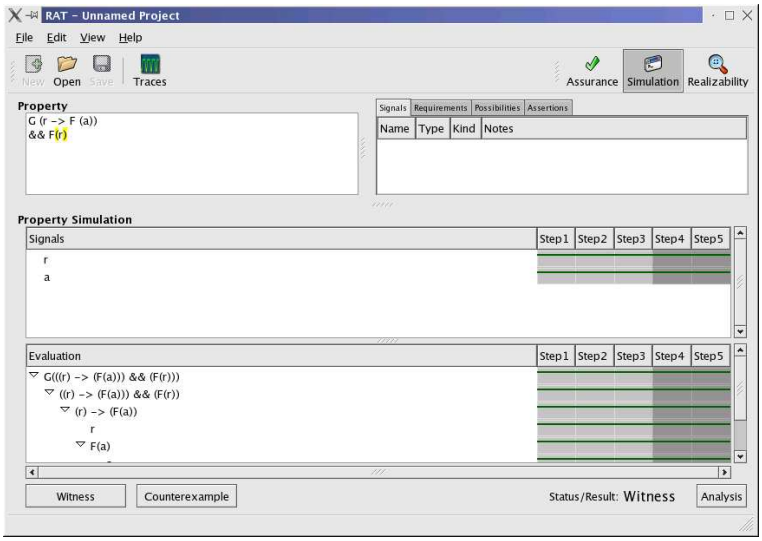

Figure 25: Witness for property $G(r \mapsto F(a))\&\&F(r)$.

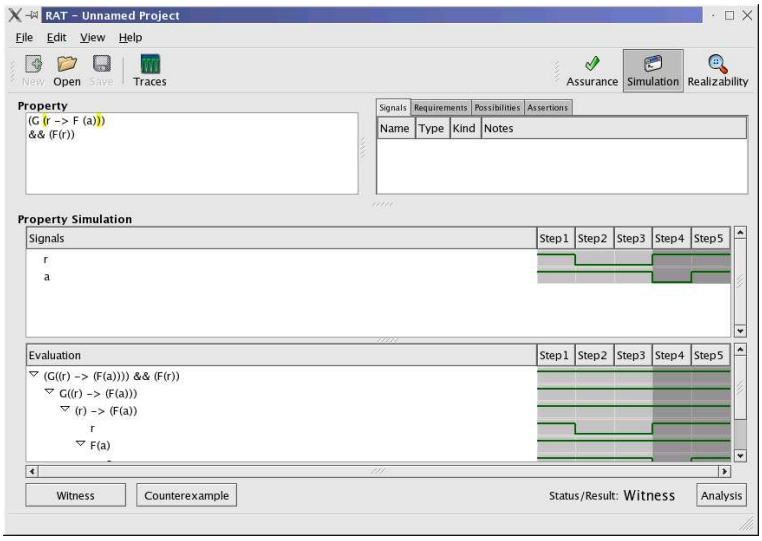

Figure 26: Witness for property $(G(r \mapsto F(a)))\&\&(F(r))$.

1.4 Property Realizability in RAT

Note: Like for Property Simulation, Realizability is not supported within the COMPASS Project. However, for completeness of this methodological section Realizability is described here for the sake of readability.

This section illustrates the RAT Property Realizability features.

For using Realizability feature the enhanced version of NUSMV [4] is required. See Section **??** for details.

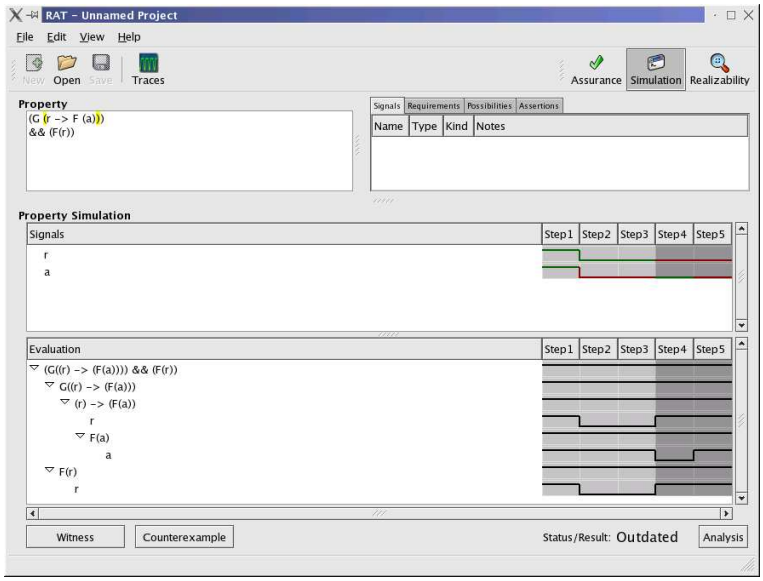

Figure 27: Shaping the trace.

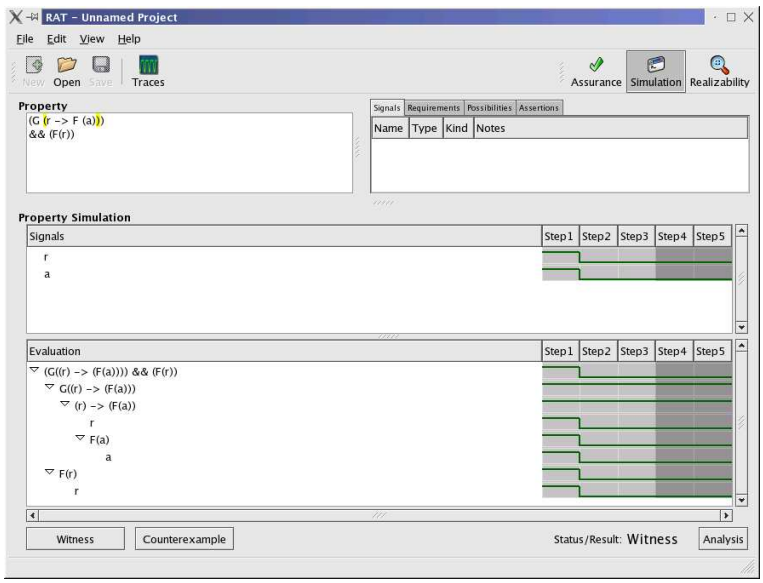

Figure 28: Witness for shaped trace request.

Realizability Problem

Informally, Property Realizability problem can be described as follows. All signals are divided into two disjoint sets – uncontrolled (environment) signals and the controlled (system) signals. Similarly, every requirement belongs to one of two sets – the assumptions and the guarantees. At every step of a play at first the environment variables are set to some unknown-beforehand values and then 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 the detailed definition of the Realizability problem see [4].

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 have to specify explicitly whether a signal is an environment signal or a system signal. For example, Figure 30 shows the wizard to specify an environment signal `inc` of type boolean. Similarly, a

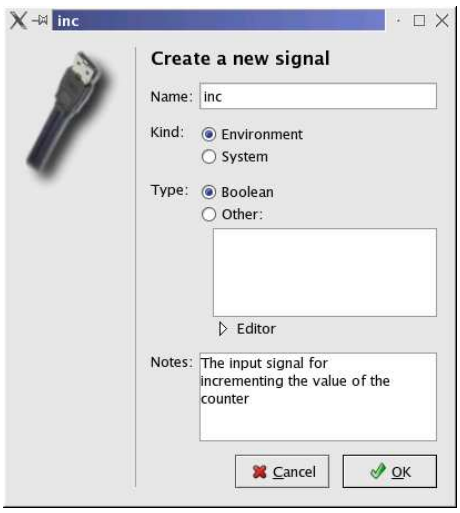

Figure 29: Creating signals, requirements.

Figure 30: Specification of an environment signal in RAT.

requirement describes an assumption on the behavior of the environment, or a guarantee on the behavior of the system. For instance, Figure 31 show the RAT wizard to specify the system guarantee `always(forall M in {-6:5}: ((v=M && inc) -> next(v=(M+1))))`.

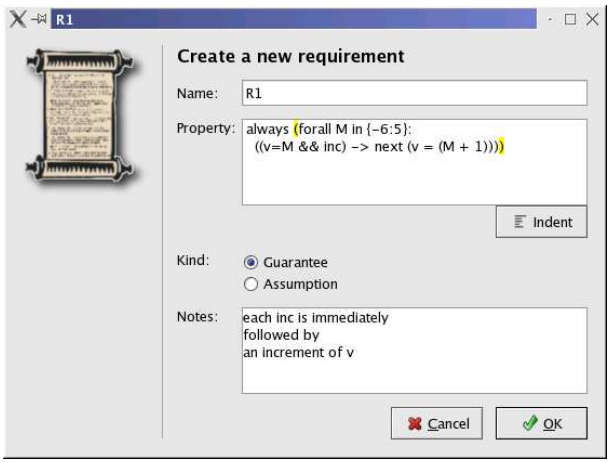

Figure 31: Specification of a system guarantee property in RAT.

The Main Window

Once all the signals and all the requirements have been inserted in the RAT project, it is possible to move to the Realizability window from where the button that performs the check of realizability for the selected properties can be pressed as to start the check for realizability. Figure 32 shows the Realizability window with an example of realizability problem.

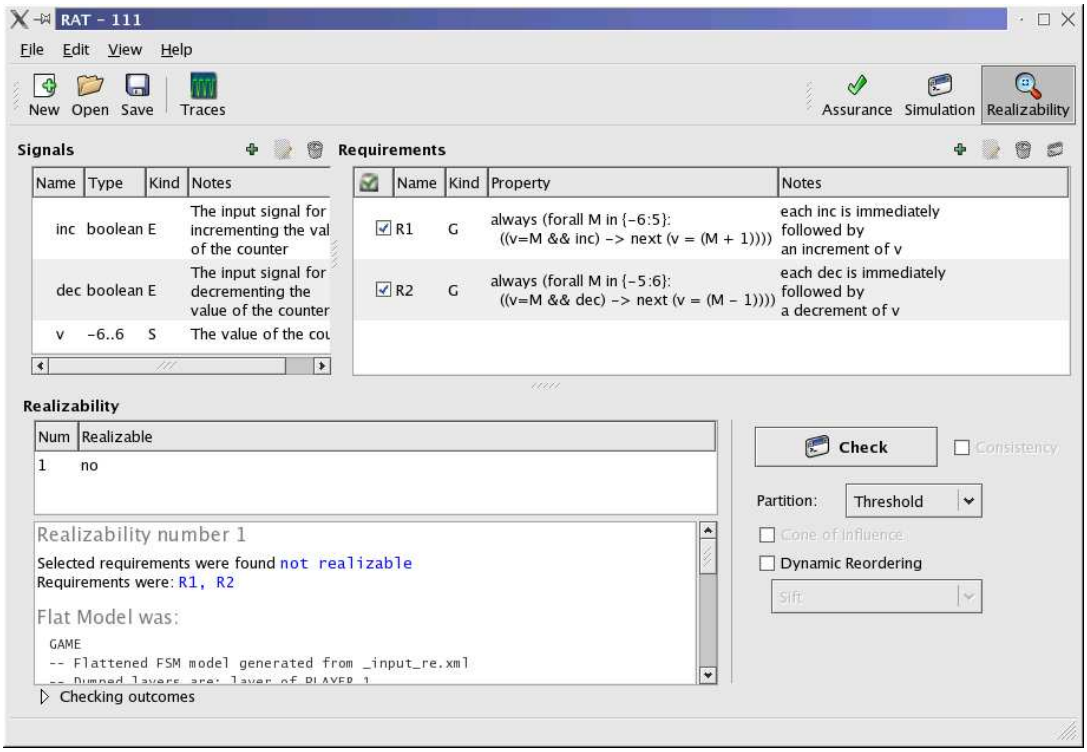

Figure 32: The Realizability window in RAT.

The `Check` button on the right in the Realizability window of RAT activates the realizability checks. The result of the check is showed in the left text area. In this particular example the specification is unrealizable because the system may force the violation of the guarantee requirements by setting both signals `inc` and `dec` up.[1] To avoid such behavior we can add an assumption requirement `never(inc && dec)`. With this assumption the specification becomes realizable (Figure 33).

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 [4]). 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 RAT specification into a NuSMV game structure an error message with the subexpression causing the problem is printed out.

[1] At the moment no debugging information is printed out. Though the enhanced version of NuSMV [4] in many cases is able to construct a strategy for the system as well as for the environment. In future such support may be added to RAT.

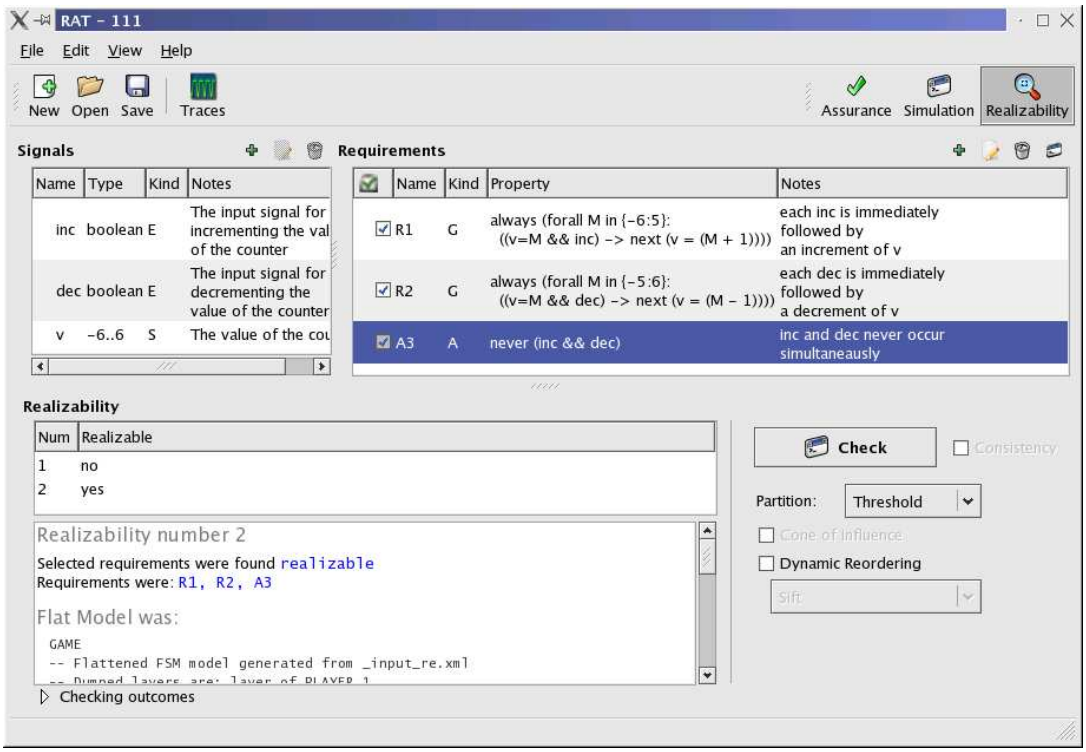

Figure 33: The Realizability window in RAT.

2 RAT Architecture

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

2.1 Architecture and Implementation Notes

RAT 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.

RAT 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 contains the definitions of other classes if convenient.

RAT uses external tools to check properties for Property Assurance, Simulation and Realizability. In particular currently it relies on the NUSMV and VIS model checkers that are written in Posix C language. The tools are called and used by RAT as external processes, and are kept separated from RAT by an abstraction layer called *Stub* that exports a standard interface.

RAT is based on several other software entities, that affect its software architecture. The picture in Figure 34 shows the main set of layered software entities which RAT relies on. The layers depict the dependencies among the entities, as higher parts depend on lower parts.

At the top is positioned the RAT 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.

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

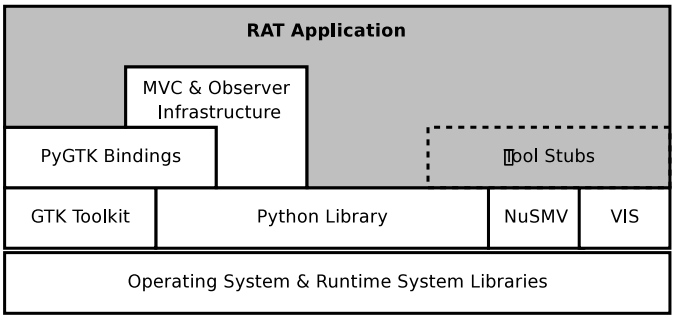

Figure 34: RAT- Software parts and collocation

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 RAT is currently based on.

***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*.

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

2.2 Architectural Patterns

RAT has a pretty complex structure, as it currently fits in seven packages, about 65 modules and 12300 lines of *Python* code including comments. RAT 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 RAT 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 [3].

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 RAT 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 ofter 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 RAT 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 RAT 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 34.

The main part of RAT 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 35 provides more details about the core and the provided services.

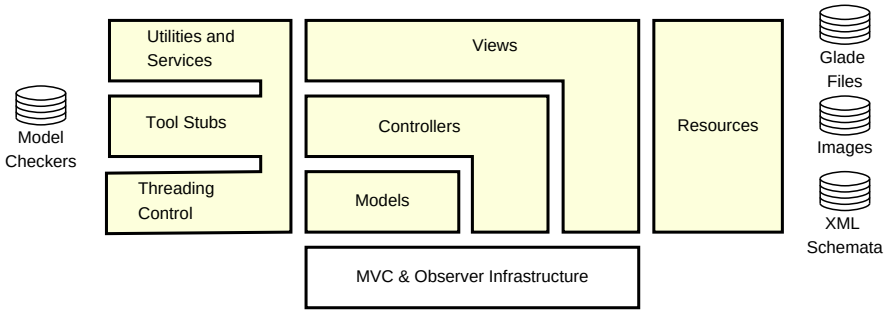

Figure 35: RAT- Software Structure

At the leftmost side of Figure 35 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 RAT from the external Model Checkers. Stubs export an interface known to RAT, and each model checker has an associated stub. The result is that RAT 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 35 are depicted those resources that are exclusively used by the RAT 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 RAT 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 RAT 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.

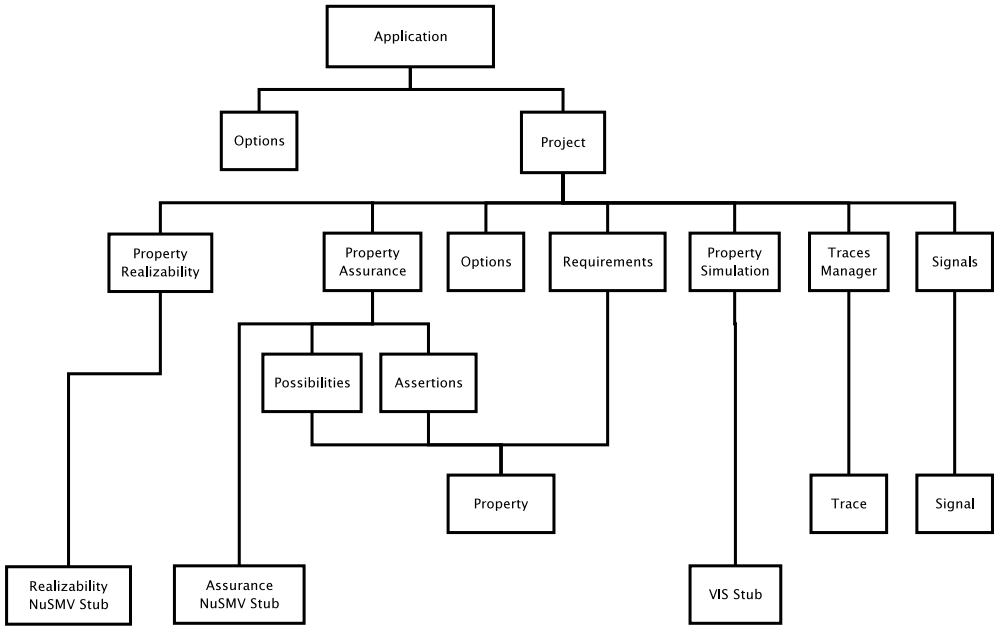

Figure 36: RAT- Hierarchy of main software entities

Figure 36 depicts the hierarchy of the main software entities that occur within RAT. 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 36 are detailed.

Application The application is the top-level entity. When the RAT executable file is run, a triple Model, View and Controller of this entity will be instantiated and connected each other, and RAT 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. A the moment this entity is empty, and there is not an associated View for it.

Project This entity represents a RAT 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 and Realizability.

Requirements This entity contains the set of requirements used by Property Assurance and Realizability.

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

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 RAT with the NuSMV model checker when Property Assurance is run. This entities 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 RAT with the enhanced version of NuSMV [4] when Property Realizability is run. This entities has no associated View and Controller, and it is implemented by a single class. Similarly to the Property Assurance NuSMV Stub already available in RAT, this class is the specialization of a more generic classes hierarchy that provides support for implementing specific tool stubs.

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 represent 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 RAT there exist several view 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 represent 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 exist a dependency between a property and those traces there were generated from it. Whenever a property's formula is changed, the corresponding traces will be invalidated.

More information about RAT implementation details can be obtain in [2].

3 References

[1] R. Bloem, R. Cavada, A. Cimatti, I. Pill, M. Roveri, S. Semprini, and A. Tchaltsev. RAT: A tool for formal analysis of requirements. In *Demo Session of the 17th 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 Delivarable D1.2/4-5.

[3] 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.

[4] A. Cimatti, M. Roveri, and A. Tchaltsev. Manual for property realizability tool, December 2006. Prosyd Delivarable D1.2/8.

[5] NuSMV home page. `http://nusmv.irst.itc.it/`.

[6] 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.

[7] PROperty based SYstem Design PROSYD. `http://www.prosyd.org/`, 2006.

[8] Accellera, Property Specification Language - Reference Manual - Version 1.01. `http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf`, April 2003.

[9] RAT — Requirements Analysis Tool. `http://rat.itc.it/`.