Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Formal Analysis of Requirements in Temporal Logics*

R. Bloem[1], R. Cavada[2], A. Cimatti[2] I. Pill[1], M. Roveri[2], A. Tchaltsev[2]

[1] Graz University of Technology
[2] Fondazione Bruno Kessler

The date of receipt and acceptance will be inserted by the editor

Abstract. Formal languages are increasingly used to describe the functional requirements of circuits. These requirements are used as a means to communicate design intent and as basis for verification and synthesis. In each of these settings, it is very important that the requirements describe the design intent precisely. Although formal requirements can be hard to understand and subtle, they are seldom the object of verification. In this paper we present techniques, guidelines, and a tool to explore formal specifications and to assure their quality. We define a technique to interactively explore the semantics of a specification by *simulating* its behavior for user-defined scenarios. Furthermore, we define techniques to automatically *check specifications* against a set of user-provided universal properties, which must be satisfied by the specification, and a set of existential properties, which must not be contradicted. Using our requirements analysis tool RAT, a designer can also investigate the *realizability* of a specification. The proposed techniques support the user in the iterative development and refinement of high-quality specifications. RAT was used successfully in several industrial projects.

1 Introduction

Formal properties are increasingly used to describe the functional requirements of electronic designs. Formal requirements are used both for verification and as a means to describe the specifications of a system before it is built.

The use of a formal language to state the requirements (the specification) is a first and substantial step towards a high quality specification, as it makes subtle questions explicit that otherwise might be hidden in the ambiguity of natural language. It also allows us to support designers with automatic tools.

A formal notation is obviously not enough to ensure the quality of a specification. The observation that the specification is often incorrect or incomplete motivates the work in coverage and vacuity. (See below.) Still, there is relatively little published work on the systematic quality control of formal specifications. This is somewhat surprising, as industrial data show that about 50 percent of product defects originate in flawed requirements and that about 80 percent of rework effort can be traced back to requirement defects [Wie01].

In this paper, we present several techniques to assist the user in writing high quality requirements. Our *Requirements Analysis Tool* RAT supports the user with an iterative work flow in the crucial task of writing high quality formal requirements.

We note that such requirements analysis and engineering is not the activity of producing an implementation satisfying given properties. Rather, the focus is on the requirements themselves. This is especially important when the properties are written before the design phase, as they probably should be. In that case, the implementation will depend crucially on the correctness and completeness of the specification. This holds even stronger in case systems are automatically constructed from the specification, a scenarios that is becoming increasingly realistic [KV05, PPS06, JB06, BGJ+07a].

We propose a Requirements Analysis process based on three techniques.

Property Simulation. offers a convenient way to explore the semantics of specifications. Similar to simulating a design, the user may provide stimuli to see how a property "reacts" in given scenarios. Property simulation allows the designer to interactively explore the behaviors associated with the requirements. She

* This work was supported by the European Commission under Contracts 507219 (PROSYD) and FP7-2007-IST-1-217069 (COCONUT)

may construct a set of traces that satisfies the requirements, or, alternatively, a set of traces that violates it. The designer can peruse these traces, change them by adding constraints and then observe changes in their evaluation, and review coverage information. In this way she can explore the semantics and assure their correctness.

Property Assurance. provides a means to formally verify properties themselves. By means of property assurance, it is possible to check that the requirements are *strict enough* to rule out unwanted behavior and that they are *not too strict* to allow for certain desirable behavior. User specified *universal* properties must hold uniformly, while *existential* properties define desired corner-case behavior.

The use of properties to validate the requirements is a powerful method that enables a formal analysis of the specification. Universal and existential properties do not have to be complicated to be useful. Since they are compared against the global set of requirements, even extremely simple universal and existential properties may stimulate the whole set of requirements and can pinpoint problems due to complex interactions between independently specified functionalities.

Property Realizability. allows the user to check whether the specification can be implemented. The signals are partitioned into *input* signals controlled by the environment, and *output* signals controlled by the system. Likewise, the properties are partitioned into *assumptions* on the behavior of the environment and *guarantee* that the system must obey. The realizability check verifies whether there is an implementation of the system that behaves according to the specification for any provided input sequence. (This is a stronger requirement that satisfiability.)

Our requirements analysis tool RAT implements and integrates the proposed techniques and provides a convenient graphical user interface for the suggested requirements analysis process. RAT supports the linear temporal part of PSL [EF06] and its subset LTL [Pnu77]. A designer can use RAT to develop and manage the specification, to archive requirements analysis results, to simulate requirements behaviors for user defined scenarios, and check the specification for realizability.

The quality of the specification has been a concern in work on coverage analysis and vacuity. In coverage analysis [KGG99, KV03, HC06], one checks how complete a specification is, that is, whether or not it allows for alternative, presumably incorrect, behavior. Initially, the question was whether a specification adequately covers a design, but the question can also be asked of the specification in isolation [Cla07, GKD07], by checking to which extent the specification constrains allowed behavior. In general, however, a series of specifications may be written, each a refinement of the other. Especially at the higher level, specifications are abstractions that allow for more than one behavior and thus do not exhibit full coverage. Our approach complements coverage analysis by allowing the user to decide what is and what is not important about the specification. It should be noted that this requires more manual effort than coverage analysis, which is largely automatic.

Vacuity detection [BBDER01, KV03] checks the quality of a specification by making sure that properties do not pass "vacuously." For instance, the property that every request is eventually acknowledged is satisfied by a system in which requests never occur. The user should be warned in such cases, as the behavior of the system is probably not what she expected. Vacuity is traditionally seen as a property of the specification and the design in combination. However, more recent work also considers the notion of vacuity that is inherent to the specification [CS07, FKSV08]. We consider this work complementary to ours. Again, our approach is more general, but also requires more effort.

Higher level, methodological solutions propose techniques to check system descriptions against user-defined properties. In [HABJ05], a tool is presented that allows the users to define a system using the Software Cost Reduction model and to perform sanity checks and state or transition invariant checks on the specification; [LRH98] and [HWT03] contain similar proposals for the SpecTRM and RSML^{-e} languages, respectively. In [BBDG$^+$02] the design exploration through model checking was introduced, which enables the user to explore a design by generating interesting traces. All these approaches are in the setting of design verification, where the quality of a design is analyzed. In contrast, our approach focuses on the requirements themselves and addresses the early stages in a design cycle, those before an implementation or a design is present.

The remainder of this part is structured as follows. The next section gives the necessary definitions. In sections 3.1, 3.2, and 3.3 we depict property simulation, property assurance, and property realizability. In Section 3.4 we propose our integrated requirements analysis approach. A use case can be found in Section 4. In Section 5 we cover the technical details of our approach. In Section 6 we describe the RAT tool, and report industrial feedback on its use from several projects. A conclusion on our current work and a perspective on future work can be found in Section 8

This paper is based on [PSC$^+$06, BCP$^+$07].

2 Linear Temporal Logic

Although RAT supports the industrial and user-friendly Property Specification Language PSL [EF06], we will limit ourselves to Linear Temporal Logic (LTL) in this paper. LTL was introduced in [Pnu77].

We adopt the positive normal form (a.k.a. negation normal form) for the definition of LTL. The set of atomic propositions A of LTL corresponds to the set of inputs

and outputs of the design. LTL formulae in positive normal form are defined using the temporal operators X (*next time*), U (*until*), and R (*releases*), as follows:

- **true**, **false**, the atomic propositions, and their negations are formulae;
- if φ and ψ are formulae, then so are $\varphi \vee \psi$, $\varphi \wedge \psi$, $\mathsf{X}\,\varphi$, $\varphi\,\mathsf{U}\,\psi$ and $\varphi\,\mathsf{R}\,\psi$.

We define a few abbreviations: $\mathsf{F}\,\varphi$ abbreviates $\mathbf{true}\,\mathsf{U}\,\varphi$, $\mathsf{G}\,\varphi$ abbreviates $\mathbf{false}\,\mathsf{R}\,\varphi$, and $\varphi\,\mathsf{W}\,\psi$ abbreviates $(\varphi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\varphi$. The Boolean connectives $\rightarrow$ and $\leftrightarrow$ are also defined as abbreviations in the usual way.

We define the set of states to be $S = 2^A$. A *state* is a set $s \subseteq A$ such that the atomic propositions in s are true, those not in s are false. The semantics of LTL are defined with respect to an infinite path $\pi \in S^\omega$. We denote by π^i the suffix of π starting at s_i. The satisfaction of an LTL formula along path π is defined as follows.

$$
\begin{array}{lll}
\pi \models \mathsf{true} & & \\
\pi \not\models \mathbf{false} & & \\
\pi \models p & \text{iff} & p \in s_0 \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi \vee \psi & \text{iff} & \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models \varphi \wedge \psi & \text{iff} & \pi \models \varphi \text{ and } \pi \models \psi \\
\pi \models \mathsf{X}\,\varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \varphi\,\mathsf{U}\,\psi & \text{iff} & \exists i \geq 0 : \pi^i \models \psi \wedge \forall 0 \leq j < i : \pi^j \models \varphi \\
\pi \models \varphi\,\mathsf{R}\,\psi & \text{iff} & \forall i \geq 0 : \pi^i \models \psi \vee \exists 0 \leq j < i : \pi^j \models \varphi
\end{array}
$$

A formula is *satisfiable* iff there is an infinite trace π such that $\pi \models \varphi$. It is *valid* iff all traces satisfy φ. A formula is *realizable* if there is a Mealy machine M such that all possible infinite executions of M satisfy φ. (We will not define Mealy machines formally.) Satisfiability and validity of LTL are PSPACE-complete, realizability is 2EXP-complete [Var96].

3 Requirements Analysis

3.1 Property Simulation

Property simulation allows the user to exercise the behavior of a single property or a complete specification. Thus, the user can make sure that she understands the precise meaning of the specification and the interaction of the properties.

Property simulation allows the user to exercise the specification much like an executable implementation. Simulation is a well-known and widely used concept for the exploration of a system's behavior. To conduct a simulation, the designer provides a design with input stimuli and observes the resulting behavior on the system's outputs. By investigating simulation results for various input scenarios, the designer can evaluate and verify the behavior of the design. Property simulation transfers this concept to the exploration of formal property semantics. In this setting, the user has more freedom. For instance, unlike a hardware simulation, property simulation does not require the input stimuli to be complete. This means that a designer may specify an incomplete input vector, and then ask the simulator to provide either a complete input output trace that is either *valid* (the specification is fulfilled) or *contradicting* (the specification is violated). The designer can further *constrain* a derived trace by fixing signal values for certain time steps, and then ask the simulator for a valid or contradicting trace.

Based on a classification of the signals into inputs and outputs, the designer can perform a "what-if" analysis by defining input values and asking for corresponding outputs. Dually, a "how-can" analysis can be performed by setting (some of) the output signals and asking how, if at all, these outputs can be achieved.

An explanation of the derived trace is provided as a set of waveforms: We show the value of each subformula of the specification at every step of the trace. In temporal logics such as LTL, the truth value of a subformula at a given time step depends only on the truth value of its subformulae and on its truth value in the next time step. Thus, understanding the evaluation of a property is reduced to understanding very local relations.

We also allow the designer to require that a signal or subformula is set to 1 (or 0) either throughout the whole trace, or at least once. We also provide the designer with quantitative information on the activity of signals and subformulae, stating how often they become 1 (or 0) and how often they change their values. This provides a form of coverage information, allowing the designer to ascertain how well the requirements are exercised.

In a way, property simulation makes the requirements executable and simulates them in much the same way that a hardware design is simulated. Thus the use of property simulation is very intuitive: traces are presented as waveforms and fixing signals corresponds to constraining a simulation, both concepts that a designer is accustomed to. Our method gives the designer concrete examples of a property's behavior and allows her to ask concrete questions about it.

3.2 Property Assurance

Property assurance is a complementary technique to property simulation that provides the designer with a more general means to assess whether she has written the right set of properties.

The basis for property assurance are three sets of properties:

- Γ: The set of *requirements* describing the system behavior, possibly including some assumptions on the environmental behavior.
- Φ_U: A set of *universal* properties that must be guaranteed by Γ.
- Φ_E: A set of *existential* properties, each of which defines corner case behavior that must be allowed by Γ.

The two sets Φ_U and Φ_E consist of golden properties to be checked on the requirements. Any behavior allowed by the requirements must fulfill all properties in Φ_U. Using Φ_U, the user can verify whether the requirements are underconstrained with respect to desired system invariants. To check for overconstraining, the user may add desired corner case behavior to the set of existential properties Φ_E. She can check then whether Γ allows specific behavior such that for each property in Φ_E there is at least one trace satisfying the considered property. Another important aspect of the requirements is consistency: the properties in Γ may not contradict each other. That is, there has to be at least one behavior consistent with Γ.

Property assurance assists the designer in tackling the following three questions:

1. Is the set of requirements Γ consistent?
2. Does the satisfaction of Γ imply the satisfaction of all properties in Φ_U?
3. Is it possible to satisfy Γ and any single property in Φ_E?

It is obvious that the last two questions check over- and underconstraining of the requirements. The formulations are closer to our technical implementation: we check the implication $\Gamma \rightarrow \Phi_U$ to verify universal properties Φ_U, and for any $\varphi \in \Phi_E$ we check that it has a model that is also a model for Γ.

Our approach is similar to the application of model checking to validate a design against a specification. The main difference is that the specification takes the place of the model, and we write a further set of properties to ensure that the formal requirements really capture the intended meaning.

To assist the designer in improving the specification, property assurance provides her with additional information based on the results of the performed checks. As in model checking, when an universal property does not pass, we show a trace that is compatible with the specification but violates the property. This behavior can then be used as starting point to correct the specification. If an existential property is satisfied by the specification, the designer is presented with a trace compatible with both the specification and the property. On the other hand, if an existential property contradicts the specification, we show the designer a subset of the specification properties that is responsible for the violation.

Inconsistent specifications can be dealt with similarly by providing the designer with a minimal inconsistent subset of the requirements. If the requirements are mutually consistent, the designer is presented with a set of representatives traces, and she can either perform property simulation, write additional universal and existential properties, or refine the requirements.

3.3 Property Realizability

Property realizability checks whether the specification is realizable, i.e., whether there is an implementation (a circuit or a program) that fulfills the specification for any possible behavior of the environment. The basis for property realizability are the following:

- $\mathcal{E}, \mathcal{S}$: Where $\mathcal{E}$ is the set of *input* signals controlled by the environment, while $\mathcal{S}$ is the set of *output* signals controlled by the system;
- Γ: The set of *requirements*, expressed in LTL over atomic propositions on $\mathcal{E} \cup \mathcal{S}$ (written $\Gamma(\mathcal{E}, \mathcal{S})$), describing the system behavior and possibly also including some assumptions on the behavior of the environment.

The *realizability problem* for a specification Γ consists of checking whether there exists a *program* such that its behavior satisfies Γ [PR89]. Specifications for which such a program exists are called *realizable* or *implementable*. Dually, specifications for which such a program does not exist are called *not realizable* or *unrealizable*.

The realizability problem can be formalized as a two player game between the system we are going to realize and the environment. At every step of the game, first the environment produces an input, and second the system produces an output. The system wins if it can guarantee that the resulting infinite trace fulfills the specification. Thus, checking for realizability amounts to checking for the existence of a winning strategy for the system in the corresponding game.

The specification Γ is consists of two separate sets – a set of *assumptions* A and a set of *guarantees* G, thus a specification is given as a tuple $\Gamma = \langle A, G \rangle$.

If a specification is unrealizable, a frequent debugging aid for the developer is a counter-strategy for the environment [Sti95, TA99, BSL04, BCD+07]. While a counter-strategy can help a developer to understand how the environment can prevent the system from fulfilling its obligations, the developer has to figure out which parts of the specification are responsible for unrealizability by herself. Moreover, while counterexamples are considered a very valuable way of feedback in verification [CV03], a counter-strategy can be much more complex than a single path and, therefore, more difficult to understand.

In property realizability the user is not only presented with a yes or no answer. We also present debugging information in the case $\langle A, G \rangle$ turns out to be either unrealizable or realizable. She can "zoom into" the specification by being presented as debugging information fragments of the specification that are by themselves (un)realizable, in order to facilitate the understanding of the problem. Thus, the designer can be presented with a specification $\langle A', G' \rangle$ as an *explanation* for a specification $\Gamma = \langle A, G \rangle$ where A', G' are subsets of A, G.

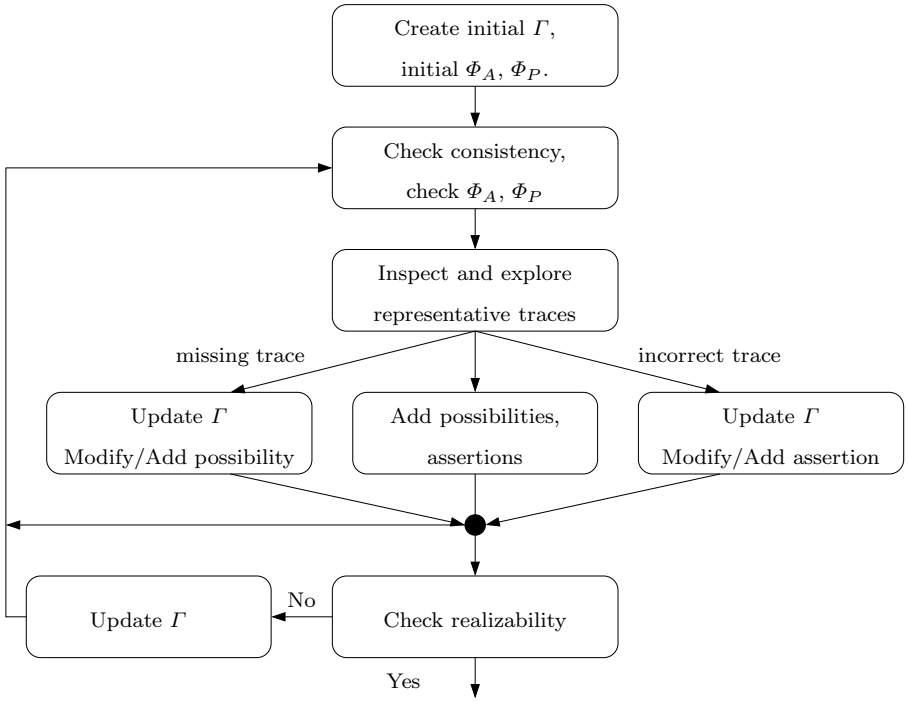

Fig. 1. Guidelines for requirements analysis.

3.4 Methodology

In this section, we describe how property simulation, assurance, and realizability can be used together to assist a designer in the requirements engineering process. Figure 1 suggests one way in which property simulation and property assurance may interact. We describe the flow here and present an example in the next section.

The designer comes up with initial approximations of the requirements Γ, universal properties Φ_U, and existential properties Φ_E. We propose an iterative approach, checking whether the requirements are consistent, whether they allow for all existential properties stated in Φ_E, and whether they do not contradict any universal property in Φ_U. For any problem, the designer is presented with diagnostic information, as explained in the previous sections, and consequently refines Γ, Φ_U, and Φ_E to fix it.

At any point the designer may use property simulation to explore the semantics of properties to isolate the origins of any problem encountered, and add or refine Γ, Φ_E, and Φ_U. By visual inspection of the traces and by checking the correctness of user-defined variants, the designer makes sure that all desired traces are viable and that no undesirable ones are present. The explanations provided help the designer understand why certain traces satisfy the requirements and others do not.

Whenever the designer finds a trace that should be allowed, but is erroneously excluded, she first corrects the requirements. Then she generalizes the trace and adds it to Φ_E. Likewise, if an unwanted trace is present, the requirements are corrected and an universal property is added to Φ_U to rule out all similar traces. In this way, whenever the designer changes the requirements, it is possible to automatically perform an exhaustive regression check by verifying that all universal and existential properties are preserved.

Subsequently, the designer may decide to add extra universal and existential properties. After any change, the requirements are again verified for consistency and for adherence to Φ_U and Φ_E.

Finally the designer checks whether the requirements are realizable. If the requirements are unrealizable, the designer revises the specification.

Prior to writing the initial approximations, a novice may use property simulation to experiment with the language. By examining the semantics of temporal operators and gaining confidence by writing and examining example properties, she gains experience and confidence she can draw on when writing an actual specification.

4 Use Case

The following example illustrates the approach using the development of the specification for an arbiter. The arbiter's signals consist of the input signals r_1 and r_2 for requests and the output signals g_1 and g_2 for the corresponding grants. Thus, we have $\mathcal{E} = \{r_1, r_2\}$ and $\mathcal{S} = \{g_1, g_2\}$.

In the following, we take the liberty of writing $\forall i \in \{1, 2\} : \varphi(i)$ to denote $\varphi(1) \wedge \varphi(2)$. We start with a very

simple specification, where Γ consists of the following properties.

$G1: \forall i \in \{1,2\}: \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i)$
$G2: \mathsf{G}\neg(g_1 \wedge g_2)$

Requirement *G1* states that for each request line, any request must be eventually answered with a grant. Requirement *G2* states that there must not be simultaneous grants on g_1 and g_2. For more complex properties, it could be useful to simulate their behavior first, before adding them to Γ. A first check shows that Γ is consistent.

Initially, the sets Φ_E and Φ_U, of existential and universal properties respectively, are empty. We start by adding the universal property *U1* to check whether grants do not come too close together.

$U1: \forall i \in \{1,2\}: \mathsf{G}((g_i \wedge \mathsf{X}\neg r_i) \rightarrow \mathsf{X}\neg g_i)$

Universal property *U1* states that if a grant is given in one step and no request is issued in the next step, then no grant can be given in the next step either. Checking Γ against *U1* we are presented with the following counterexample. (All traces shown are lasso shaped: they consist of a finite prefix and a suffix which is repeated ad infinitum.)

r_1: 0 0 0 (0 ....)
g_1: 0 0 0 (0 ....)
r_2: 0 0 0 (0 ....)
g_2: 1 1 1 (1 ....)

The trace shows that we forgot to rule out initial spurious grants. Thus, we add a new requirement *G3* stating that there must be no grants until there is a request.

$G3: \forall i \in \{1,2\}: (\neg g_i \,\mathsf{U}\, r_i)$

Another check against *U1* produces the following counterexample showing that our specification still violates the universal property. (As the arbiter is symmetric, in the following we will focus on one request/grant signal pair only.)

r_1: 1 0 0 (0 ....)
g_1: 1 1 0 (0 ....)

With *G3* we eliminated initial spurious grants, but we forgot to rule out those following a grant. We further constrain the specification adding a new requirement *G4* stating that whenever there is a grant, no more grants can be issued by the arbiter until there is a further request.

$G4: \forall i \in \{1,2\}: \mathsf{G}(g_i \rightarrow \mathsf{X}(\neg g_i \,\mathsf{U}\, r_i))$

Once we have checked $\{G1, \ldots, G4\}$ for consistency, and got a positive answer, we re-check against *U1* and we are assured that the flaw has been solved.

To become more confident of the current specification's semantics, we simulate Γ to obtain a set of compliant traces. The derived set includes the following two traces.

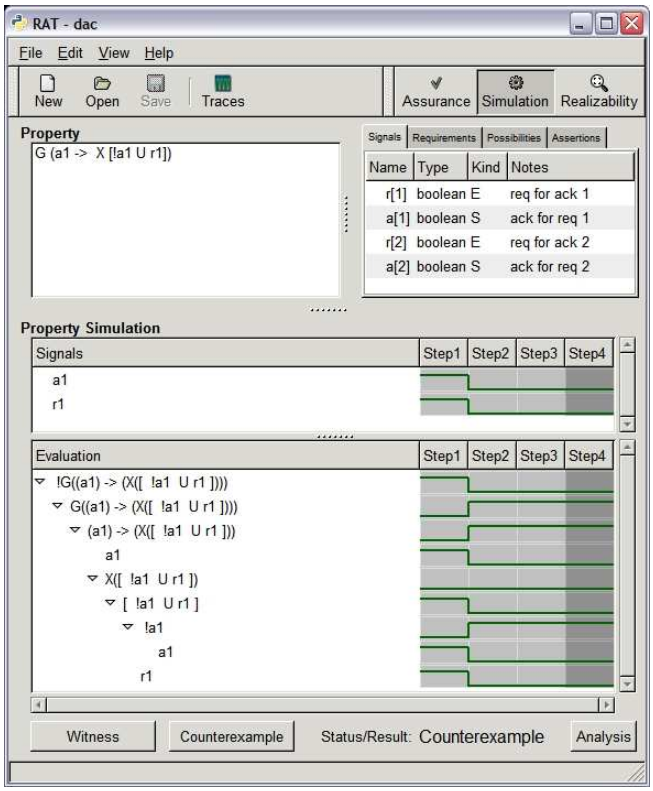

Fig. 2. Formula evaluation example.

r_1: 0 0 (0 ...)
g_1: 0 0 (0 ...)

r_1: 1 1 (1 ...) ....
g_1: 1 1 (1 ...) ....

The tool does not provided us with a trace consisting of a single request with a single grant. Since we would like to examine the specification's behavior for that simple case, we construct the following trace as stimulus and we simulate Γ for compliant traces.

r_1: 1 0 (0 ...)
g_1: 1 0 (0 ...)

The simulator unexpectedly tells us that the provided stimulus does not satisfy the specification. When we consider the unfolded sub-formula evaluation of a counterexample as depicted in Figure 2 (the trace consists of a finite stem from step 1 to 3 and an infinite loop of step 4), we find out that $\mathsf{G}(g_1 \rightarrow \mathsf{X}(\neg \,\mathsf{U}\, r_1))$ is not true initially because $(\neg g_1 \,\mathsf{U}\, r_1)$ is false in the subsequent step.

Indeed, the strong until as used in *G3* and *G4* requires that r_1 must eventually appear, which contradicts our intent. The weak until, W, differs from the strong until, U, in the fact that it does not require the second operand to become true, but is satisfied also if the first operand stays true for the whole trace. As this behavior matches our design intent, replacing the strong until in *G3* and *G4* with a weak until solves the encountered error:

$G3: \forall i \in \{1,2\}: (\neg g_i \,\mathsf{W}\, r_i)$
$G4: \forall i \in \{1,2\}: \mathsf{G}(g_i \rightarrow \mathsf{X}(\neg g_i \,\mathsf{W}\, r_i))$

To avoid similar errors in the future we generalize the trace to the following formula, which we add it to our set of existential properties Φ_E. The meaning of *E1* is that r_i is low infinitely often.

$$E1 : \forall i \in \{1,2\} : \mathsf{F}\,\mathsf{G}\,\neg r_i$$

Subsequently, we re-check our specification for compliance to Φ_U and Φ_E.

Now, as the specification implements our basic idea, we can ask more subtle questions about the interaction of requests and grants. For instance, is it true that requests and grants are always balanced? To derive an answer, we define a scenario in which there are two requests answered by a single grant, and we simulate the specification for featured behavior for this case.

```
r1: 1 1 (0 ...)
g1: 0 1 (0 ...)
```

The simulation succeeds and shows us that multiple requests can be answered with a single grant.

To rule out scenarios in which the environment is too demanding and issues too many requests, we add a new assumption requirement *A5*. This requirement restricts the environment, so that it may not issue further requests until the pending one has been answered.

$$A5 : \forall i \in \{1,2\} : \mathsf{G}(r_i \rightarrow (\neg r_i \,\mathsf{U}\, g_i))$$

To make sure that the previous scenario is not possible anymore, we add a universal property *U2* to Φ_U. This property states that an unacknowledged request must never be followed by another request without a grant in between.

$$U2 : \forall i \in \{1,2\} : \mathsf{G}\,\neg((r_i \wedge \neg g_i) \wedge \mathsf{X}(\neg g_i \,\mathsf{U}\, r_i))$$

Since we have changed the specification, we re-check it for consistency, and we check it against all existential and universal properties, including *U2*. This check passes.

Finally, we have derived a specification for our arbiter, consisting of $\{G1, \ldots, G4, A5\}$, that meets our design intent. We found and addressed several flaws in the specification by investigating the specifications semantics, we developed universal and existential properties to check the specification against, and we elicited a constraint on the environment behavior by means of a special requirement on the inputs. The obtained specification's quality has been assured by both formal checks and the exploration of the specification's behavior. The specification is also realizable.

At this point the customers asked for a change in the requirements, to take into account timing constraints. Thus, we further restrict *G1* by adding *G1.1* and *G1.2*.

$$G1.1 : \mathsf{G}(r_1 \rightarrow g_1)$$
$$G1.2 : \mathsf{G}(r_2 \rightarrow (g_2 \vee \mathsf{X}\, g_2))$$

G1.1 requires that request r_1 has to be granted immediately. *G1.1* requires that request r_2 has to be granted immediately or in the next step.

Next, we check satisfiability of the new specification $\{G1, \ldots, G4, G1.1, G1.2, A5\}$, whether all the existential properties are satisfied, and whether all the universal properties holds. All these checks pass without problems. However, the check for realizability no longer passes. Indeed, if both requests arrive simultaneously and are followed by an r_1, then there is no possibility to grant g_2 either at the time r_2 is requested nor at the immediately successive step.

The specification can be made realizable, by modifying *G1.1*:

$$G1.1 : \mathsf{G}(r_1 \rightarrow (g_1 \vee \mathsf{X}\, g_1))$$
$$G1.2 : \mathsf{G}(r_2 \rightarrow (g_2 \vee \mathsf{X}\, g_2))\,.$$

As our requirements are realizable, we obtained again a realizable high quality specification.

5 Technical Aspects

5.1 Automata-Based and Bounded Model Checking

Our approach relies on automata-based and bounded model checking techniques.

In the automata-based approach [VW94], typically BDD-based, we derive an automaton for the property and we check its language for emptiness. If the language is empty, there is no behavior satisfying the property. On the other hand, accepting traces in the automaton correspond to examples of behavior that is consistent with the property. These traces, consisting of a finite stem and a loop that is repeated infinitely often, can be obtained by counterexample construction, a standard feature of any model checker [CGMZ95].

Bounded Model Checking (BMC) [BCCZ99] searches for *bounded witnesses* (or *bounded counterexamples*) for a temporal property. A bounded witness (counterexample) is an infinite path on which the property holds (does not hold), which can be represented by a finite witness of length k. A finite witness can represent infinite path in the following sense: either the finite witness represents all its infinite extensions, or it consists of a finite stem and a loop repeated infinitely often from state k back to state l. (In the latter case we speak of (k,l)-path.) In BMC all possible (k,l)-paths of a specification Γ are encoded as a propositional satisfiability problem $SAT(\Gamma)$ and given as input to a SAT solver. The parameters k and l are modified until we either find a witness (the formula is satisfiable) or reach a sufficiently high value of k to guarantee completeness.

Automata derived for properties in languages like PSL might be alternating [BDBF$^+$05], which would cause an exponential blowup in the state space when translating it to a nondeterministic version needed for many model checking engines [MH84, CGP99]. Direct translations are also possible [BCPR07, CRST06a].

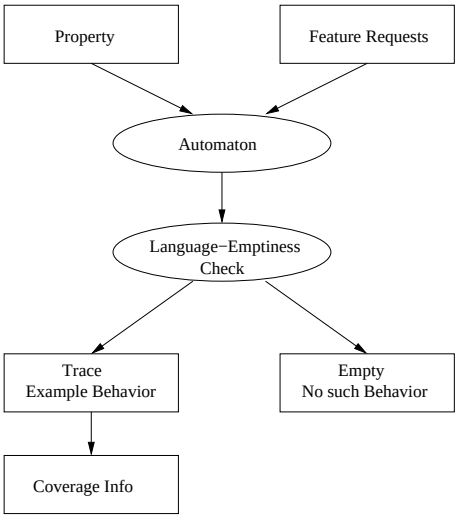

Fig. 3. Property simulation methodology searching for a witness.

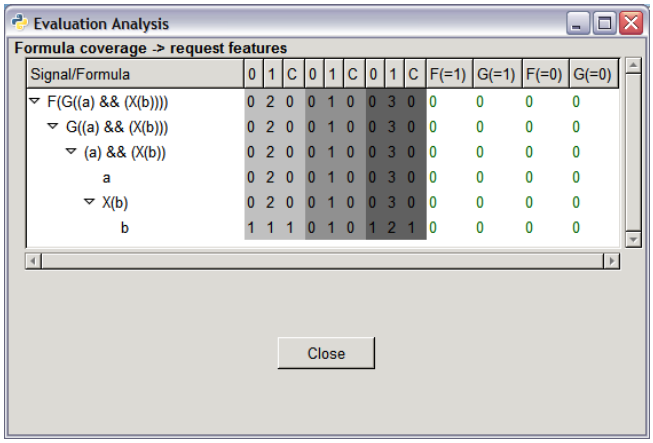

Fig. 4. RAT's coverage analysis window.

5.2 Property Simulation with User Constraints

Our implementation of property simulation relies on automata-based model checking techniques: we derive two automata, one for the property and one for the simulation stimuli, and we perform a language emptiness check on their product (see Figure 3). If the language is empty, there is no property behavior compatible with the stimuli. On the other hand, accepting traces in the automaton correspond to examples of the desired behavior.

More formally, suppose we are looking for a trace that satisfies both the stimuli and the property. We derive an automaton A_S for the simulation stimuli and an automaton A_φ for φ. We then construct an automaton A_P for the product of A_S and A_φ [CGP99]. An accepting run of A_P corresponds to a witness of φ that satisfies the stimuli.

To explore the differences between complying and contradicting behavior to a formula the user can also search for behavior that satisfies the stimuli but contradicts the property. To construct such a trace, we derive an automaton $A_{\neg\varphi}$ for the negated property $\neg\varphi$ and then we construct an automaton A_P for the product of A_S and $A_{\neg\varphi}$. An accepting run of A_P corresponds to a trace that satisfies the simulation stimuli but contradicts the property.

This approach extends simulation stimuli beyond fixing signal values at certain time steps to the support of more general user constraints such as "signal c shall be true for at least one time step in the future". Technically we support any simulation stimuli expressible via automata.

5.3 Representative Traces

Using simulation, one must consider a variety of representative scenarios to gain a good overview of a property's behavioral aspects. Coverage and vacuity analysis concepts, e.g. [KGG99, BBDER01], play a significant role in this context. To understand a property, a user must explore traces allowed by a property, with special attention to the "extreme behaviors". When exploring the traces of a property, coverage can be of great help to check whether behaviors have been unnoticed so far.

A trace is called *uninteresting for a certain subformula* φ if the trace fulfills the specification even if φ is replaced by an arbitrary formula [BBDER01]. Consider the formula $\varphi = \mathsf{G}(r \rightarrow \mathsf{F}\, g)$. First, traces is which no request occur satisfy φ vacuously because they are uninteresting for the subformula $\mathsf{F}\, g$. Second, traces in which grants occur infinitely often are uninteresting for the subformula r. Third, traces that consists of finitely many requests and grants are interesting for all subformulae. We argue that all three classes of traces should be represented in the feedback to the users, as each of them highlights a different aspect of the property.

We furthermore show the user how many times each subformula is true or false as well as the number of times its truth value changes. We do this both for the finite stem and for the loop. (See Figure 4.) These figures give an indication of the coverage provided by the traces: how many and which aspects of a property are addressed by the traces. Subformulae or behavioral patterns not exercised by current simulations may suggest future simulation stimuli.

5.4 Property Assurance

Property assurance relies on validity/satisfiability of the logic used for the specification. We can check consistency of Γ by computing whether $SAT(\Gamma)$ holds, i.e., whether

there exists at least one behavior that satisfies Γ. We check whether an existential property $\varphi \in \Phi_P$ is supported by computing whether $SAT(\Gamma \wedge \varphi)$ holds. A universal property $\varphi \in \Phi_A$ holds if $\Gamma \rightarrow \varphi$ is valid, i.e., if all possible behaviors are compatible with φ. We check this by computing $\neg SAT(\Gamma \wedge \neg\varphi)$.

Note that all the property assurance problems can be seen as standard model checking problems with completely unconstrained models. For instance, if φ is an universal property we model check $\Gamma \rightarrow \varphi$, while if it is an existential property we model check $\neg(\Gamma \wedge \varphi)$. The model checking problem can be tackled using automata-based model checking techniques or BMC.

In the automata-based approach we derive an automaton for the property assurance problem, and we check its language for emptiness. In the case of an existential property check, if the language is empty there is no behavior compatible with both the specification and the property. If, on the other hand, the language is not empty a trace of the automaton is a witness of a behavior compatible with the specification and with the existential property. (Consistency can be handled in the same way.) Dually, in the case of universal properties, if the language is empty there are no behaviors compatible with the specification that contradict the property. If the language is not empty, a trace of the automaton is the counterexample compatible with the specification and falsifying the universal property.

SAT-based bounded model checking can be used for checking an existential property by looking for a (k, l)-path satisfying $\Gamma \wedge \varphi$. Dually, for checking an universal property, if we find a (k, l)-path satisfying $\Gamma \wedge \neg\varphi$, we have a counter-example compatible with the specification and violating the property.

The automata-based and BMC-based model checking techniques complement each other. SAT-based BMC model checking is usually faster than automata-based BDD model checking in finding a bounded witness, and several optimization to the original BMC encoding [BCCZ99] have been developed (see e.g. [KHL05] for a discussion). Extensions to make the approach not only able to disprove a property, but also to prove the property holds have been developed (e.g. [KHL05]). BMC-based verification is efficient for checking existential properties and for checking consistency. A valid scenario for a property, if one exists, can typically be produced in a few seconds. BMC-based verification is also good for preliminary verification of universal properties. If no counterexamples are found up to a reasonable k, then we can proceed with the more expensive BDD-based approach. Notice that the BDD approach does not perform well on large specifications because of the size of the underlying automaton. (See [FLM+04] for further details.)

As a final remark, property assurance problems inherently differ from model checking problems, where the computation bottleneck originates from the model rather than from the property. In addition, the notion of diagnostic information is completely different: rather than a counterexample trace, we can show the designer information in terms of the requirements, and, for instance, in the case of an existential property $\varphi \in \Phi_P$ single out a (hopefully small) subset of Γ that is inconsistent with φ.

5.5 Realizability of Properties

The realizability problem can be formalized as a two player game among the system we are going to realize and the environment: the system plays against the environment in such a way that at every step of a game the environment moves and then the system try to move by producing behaviors compatible with the specification. The system wins if it produces a correct behaviors regardless of the behaviors of the environment. In this framework, checking for realizability amounts to check for the existence of a winning strategy for the system in the corresponding game. This is tackled by generating from the specification a deterministic Rabin automaton using the Safra construction [Saf88]. This automaton is interpreted as a two player game among the system and environment and it is traversed as to find a witness of the non emptiness of the language of the automaton (which corresponds to a correct implementation of the given specification) [PR89].

The high complexity established in [PR89], and the intricacy of Safra's determinization construction, have caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to implement it. However there are several classes of properties restricted to particular subsets of LTL, which can be synthesized efficiently with polynomial algorithms. One of the most recent and advanced results is achieved in [PPS06] where for the class of *Generalized Reactivity*(1) specifications (from now on referred to as GR(1) specification) is presented a (symbolic) polynomial algorithm for extracting a program from a GR(1) specification.

A GR(1) specification has the form $\langle A, G\rangle = (\{\varphi_I^{\mathcal{E}}, \varphi_R^{\mathcal{E}}, \varphi_\psi^{\mathcal{E}}\}, \{\varphi_I^{\mathcal{S}}, \varphi_R^{\mathcal{S}}, \varphi_\psi^{\mathcal{S}}\})$. For $\alpha \in \{\mathcal{E}, \mathcal{S}\}$, the tuple $(\varphi_I^\alpha, \varphi_R^\alpha, \varphi_\psi^\alpha)$ is such that each formula has the following form:

- φ_I^α - a formula of the form $\bigwedge_i I_i$ where every I_i is a propositional formula over signals ($\varphi_I^{\mathcal{E}}$ is over $\mathcal{E}$ and $\varphi_I^{\mathcal{S}}$ is over $\mathcal{E} \cup \mathcal{S}$). These formulas represent game *initial conditions* of the environment and system, respectively.
- φ_R^α - temporal formulas of the form $\bigwedge_i R_i$ where every R_i is a propositional formula over signals $\mathcal{E} \cup \mathcal{S}$ and expressions of the form $\mathbf{X}\, v$ where $v \in \mathcal{E}$ if $\alpha = \mathcal{E}$ and $v \in \mathcal{E} \cup \mathcal{S}$ if $\alpha \in \mathcal{S}$. These formulas represent the *transition relations* for the environment and the system, respectively.

– φ_ψ^α - temporal formulas of the form $\bigwedge_i \mathbf{G}\,\mathbf{F}\,A_i$ where A_i is propositional formula over signals $\mathcal{E} \cup \mathcal{S}$. These properties represent the *liveness* or *fairness* condition of the design.

Intuitively the definition of GR(1) specifications causes the environment and the system interact the following way: a play starts by the environment choosing initial values for its signals such that $\varphi_I^{\mathcal{E}}$ is satisfied and the system initializing its signals such that $\varphi_I^{\mathcal{S}}$ holds. Similarly, at every consecutive step of the play at first the environment assigns its signals, trying to satisfy the environment transition relation $\varphi_R^{\mathcal{E}}$, and then the system does the same with its signals and its transition relation $\varphi_R^{\mathcal{S}}$. For an infinite behavior the environment and the system try to satisfy their liveness conditions $\varphi_\psi^{\mathcal{E}}$ and $\varphi_\psi^{\mathcal{S}}$, respectively. A play who first violates its constraints looses. If all constraints are satisfied then the system wins.

The class of GR(1) specifications is sufficiently expressive to provide complete specifications of many designs [PPS06].

The algorithm described in [PPS06], and implemented in [BGJ+07b, JGWB07, CRST08], reduces the realizability problem of a GR(1) specification to the problem of computing a set of *winning states* for a properly constructed specification obtained from a GR(1) specification as described above. We refer the reader to [PPS06] for details of the transformation and of the algorithms for computing such set of winning states W_S. After computing winning state W_S the last step in identifying realizability is to check W_S against initial conditions. The game is winning for the system (and thus the GR(1) specification is realizable) iff formula Ω in 1 is the constant true.

$$\Omega = \forall \mathcal{E}.(\varphi_I^{\mathcal{E}} \rightarrow \exists \mathcal{S}.(\varphi_I^{\mathcal{S}} \wedge W_S)) \qquad (1)$$

6 RAT

The concepts presented in the previous sections were used in the design and development of RAT [BCP+07]. The high level architecture of RAT is depicted in Figure 5, while Figure 6 offers a screen-shot of the graphical user interface (GUI).

With RAT the user can take (a subset of) properties of the specification and simulate their behavior for user-provided stimuli (property simulation) as well as determine consistency or perform checks against user-defined universal and existential properties (property assurance). Finally she can check them for realizability (property realizability).

The GUI provides a user-friendly interface to the underlying engines. For example, in order to check universal properties, the combination of requirements and universal properties is submitted to a BMC engine. The

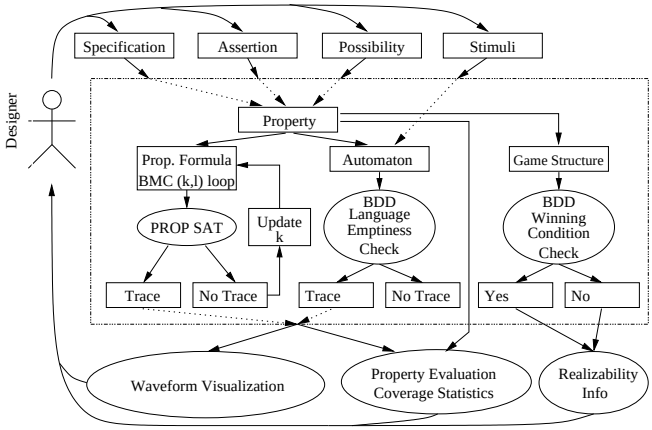

Fig. 5. RAT architecture.

BMC engine increases k, the length of the path, until either a counterexample is found or a user-defined upper bound for k is reached.

Furthermore RAT's GUI offers the designer facilities to manage the specification and acquired analysis results in a *requirements analysis project*, which can be saved for future reference. A project contains the complete status of the specification and analysis activities: it contains the sets of requirements, universal, and existential as well as analysis results and debugging information that is either generated by the tool or provided by the user. For example, traces can be stored for future reference and be associated with additional information, including the set of properties from which they were generated, whether they are witnesses or counterexamples, and optional user notes. This helps the designer in monitoring the current status and analysis coverage of the specification.

RAT relies on extended versions of the NuSMV and VIS model checkers [CCGR99, B+96] to provide the proposed functionality for PSL specifications. However, our design allows for an easy integration of other verification engines to support further languages or verification algorithms.

RAT and its documentation can be obtained from the `http://rat.fbk.eu`. Since its first release on June 2006, we had about 1064 downloads.

7 Feedback from Industry

Several companies experimented with RAT [ABF+06]. The experimenters were IBM Haifa Research Labs, Infineon GmbH and OneSpin Solutions GmBH in Munich, and STMicroelectronics in Agrate, Italy, in Grenoble, France, and in Bristol, UK. The case studies using RAT included:

– Two interconnect protocols - the Bus Protocol (ST France and UK) and the SOC Interconnection Architecture (IBM).

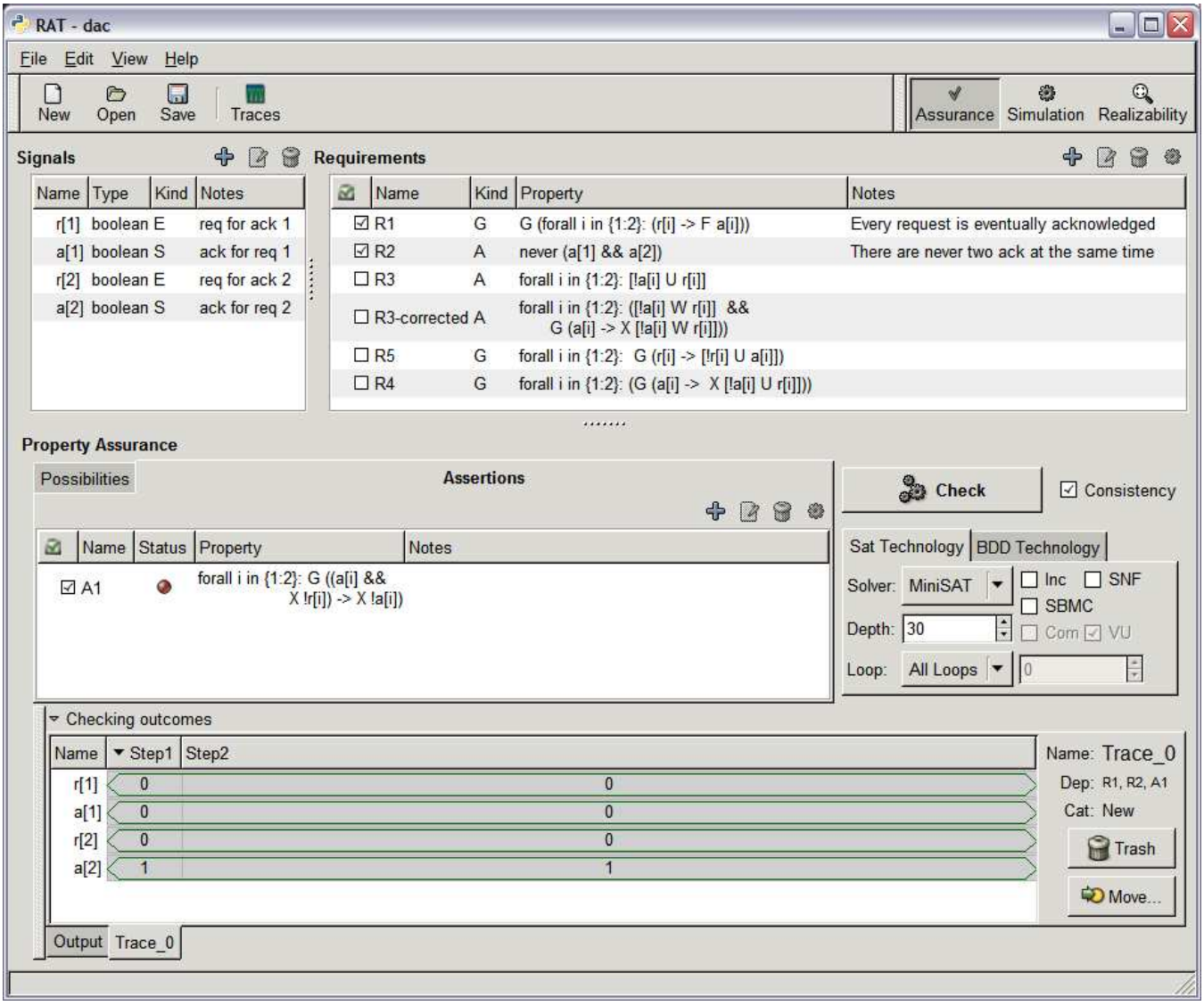

Fig. 6. A screen-shot of the RAT tool.

- Three SOC communication blocks - the Memory Interface (ST France), and the Memory Controller, and the Bridge (ST Italy).
- One specific-purpose IP design - the Transport Front End (ST UK). The Transport Front End is a large IP consisting of several components.

Although the users commented on the immature status of the tool (since improved), the report shows that our technology appeals to designers.

> "We found the concept of property simulation attractive as it allows a developer to debug her/his own PSL code easily, quickly and independently".

Our prototype's robustness was considered

> "High (better than average for a tool this stage of development, only a few minor failures when the tool's capabilities were stressed)".

Several specification bugs were found with property simulation, for instance in the Memory Interface. Property Assurance has also been

> "used effectively in specific cases to prove that one set of properties can be substituted by another"

For the Bus Protocol a property R_1 of a golden reference specification Γ was improved and replaced by R_2. The refinement was verified in two steps: first, considering R_2 as universal property for the old version of the specification Γ. For the second step, R_1 was considered as universal property for checking the new version of the specification $\Gamma' = (\Gamma \setminus \{R_1\}) \cup \{R_2\}$. The designers involved in the SOC Interconnection Architecture case study found that the interface and usability features provided by RAT "make the development process easier and provide an enjoyable development experience".

> "RAT presents traces in a both intuitive and informative way. The trace is clearly divided into a finite head and infinite tail, a full tree of subformulas is available and the analysis window allows advance[d] insight."

They judged property simulation to save both time and work:

> "For about half of the total 33 point-to-point properties written, the original PSL property was found to be correct. For more than 10% of the properties it require[d] more than 4 versions before a correct PSL statement was found. The results show that debugging is necessary, since about half of the properties are wrong on the first draft. The alternative to debugging the properties using RAT ([BCE+05]) is involving others to review them. The evidence is that careful review is necessary as 10% of properties were wrong on the 4th draft. We estimate that a careful review will take 1.5 PM whereas the debugging using RAT took 0.5 PM, thus saved 1PM."

IBM found that

> "The ideas of RAT ([BCE+05]), especially property simulation, have a lot of value to users of PSL and PSL-based IBM tools. Thus, we started to design and develop a feature similar to property simulation in RuleBasePE soon after starting this case study."

According to Cindy Eisner from IBM this feature is already available in newer versions.

8 Conclusions

conclusions and future work.

Regarding property assurance, we plan to integrate within RAT the advanced techniques as described in [BCP+06, CRST06b, CRT07] top perform the check. While to provide the user with useful debugging information we are integrating the techniques described in [CRST07].

As far as property realizability is concerned we would like to integrate in the tool the techniques described in [CRST08]. These techniques aims at extracting debugging information from a (un)realizable specification.

8.0.1 Acknowledgements

The authors would like to thank Simone Semprini for his contributions to earlier versions of RAT and related technologies. Furthermore we would like to thank Gadiel Auerbach, Lyes Benalycherif, Cindy Eisner, Andrea Fedeli, Dana Fisman, Anthony McIsaac, and Klaus Winkelmann for their efforts in the industrial case studies, the detailed feedback, and fruitful discussions.

References

[ABF+06] G. Auerbach, L. Benalycherif, A. Fedeli, D. Fisman, A. McIsaac, and K. Winkelmann. Case studies in property-based requirements specification, November 2006. PROSYD Deliverable D1.4/1, www.prosyd.org.

[B+96] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[BBDER01] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18:141–163, 2001.

[BBDG+02] S. Barner, S. Ben-David, A. Gringauze, B. Sterin, and Y. Wolfsthal. An algorithmic approach to design exploration. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 146–162, London, UK, 2002. Springer-Verlag. LNCS 2391.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[BCD+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for playing games! In *Proc. Computer Aided Verification (CAV'07)*, pages 121–125, 2007.

[BCE+05] 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, www.prosyd.org.

[BCP+06] R. Bloem, A. Cimatti, I. Pill, M. Roveri, and S. Semprini. Symbolic Implementation of Alternating Automata. In O. H. Ibarra and H.-C. Yen, editors, *CIAA*, volume 4094 of *Lecture Notes in Computer Science*, pages 208–218. Springer, 2006.

[BCP+07] R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev. Rat: A tool for the formal analysis of requirements. In *Computer Aided Verification*, pages 263–267, 2007.

[BCPR07] R. Bloem, A. Cimatti, I. Pill, and M. Roveri. Symbolic implementation of alternating automata. *International Journal of Foundations of Computer Science*, 18(4):727–743, 2007.

[BDBF+05] S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah. Automata construction algorithms optimized for PSL, 2005. PROSYD Deliverable D3.2/4, www.prosyd.org.

[BGJ+07a] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[BGJ+07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design Automation and Test in Europe*, 2007. To appear in the proceedings of DATE'07.

[BSL04] Y. Bontemps, P. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inform.*, 62(2):139–169, 2004.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer-Aided Verification (CAV'99)*, pages 495–499. Springer-Verlag, July 1999. LNCS 1633.

[CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 1999. ISBN 0-262-03270-7.

[Cla07] K. Claessen. A coverage analysis for safety property lists. In *Proc. Formal Methods in Computer Aided Design*, pages 139–145, 2007.

[CRST06a] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From psl to nba: a modular symbolic encoding. In *Proc. Formal Methods in Computer Aided Design (FMCAD)*, pages 125–133, 2006.

[CRST06b] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: a Modular Symbolic Encoding. In *FMCAD*, pages 125–133. IEEE Computer Society, 2006.

[CRST07] A. Cimatti, M. Roveri, V. Schuppan, and S. Tonetta. Boolean abstraction for temporal logic satisfiability. In *Proc. Computer Aided Verification*, pages 532–546, 2007.

[CRST08] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic Information for Realizability. In F. Logozzo, D. Peled, and L. D. Zuck, editors, *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2008.

[CRT07] A. Cimatti, M. Roveri, and S. Tonetta. Syntactic Optimizations for PSL Verification. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 505–518. Springer, 2007.

[CS07] H. Chockler and O. Strichman. Easier and more informative vacuity checks. In *Proc. Formal Methods and Models for Codesign*, pages 189–198, 2007.

[CV03] E. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In N. Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Z. Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 208–224. Springer, 2003.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer-Verlag, 2006.

[FKSV08] D. Fisman, O. Kupferman, S. Seinvald, and M. Y. Vardi. A framework for inverent vacuity. In *Proc. Haifa Verification Conference (HVC'08)*, 2008.

[FLM+04] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 9:132–150, 2004.

[GKD07] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *DATE*, pages 1176–1181, 2007.

[HABJ05] C. Heytmeyer, M. Archer, R. Bharawaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science and Engineering*, 20(1):19–35, January 2005.

[HC06] M. Y. Vardi H. Chockler, O. Kupferman. Coverage metrics for temporal logic model checking. *Formal Methods in System Design*, 28:189–212, 2006.

[HWT03] M. P. E. Heimdahl, M. W. Whalen, and J. M. Thompson. NIMBUS: A tool for specification centered development. In *Proc. Requirements Engineering Conference*, page 349. IEEE Computer Society, 2003.

[JB06] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[JGWB07] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification (CAV)*, 2007. To appear in the proceedings of CAV'07.

[KGG99] S. Katz, O. Grumberg, and D. Geist. "Have I written enough properties?" — A method of comparison between specification and implementation. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 280–297, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[KHL05] T. Junttila K. Heljanko and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. K. Rajamani, editors, *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 98–111. Springer-Verlag, 2005. LNCS 3576.

[KV03] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Software Tools for Technology Transfer*, 4:224–233, 2003.

[KV05] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science*, pages 531–542, Pittsburgh, PA, October 2005.

[LRH98] N. Leveson, J. Reese, and M. Heimdahl. SpecTRM: A CAD system for digital automation, 1998.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E. Emerson and K. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[PSC+06] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *DAC '06: Proceedings of the 43rd annual conference on design automation*, pages 821–826, 2006.

[Saf88] S. Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327. IEEE, 1988.

[Sti95] C. Stirling. Local model checking games. In *Proc. Concurrency Theory*, pages 1–11. Springer-Verlag, 1995.

[TA99] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 233–252. Springer, 1999.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata*, pages 238–266, 1996.

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[Wie01] K. E. Wiegers. Inspecting requirements. *StickyMinds Weekly Colum*, July 2001.