Formal Analysis of Hardware Requirements[*]

I. Pill♣, S. Semprini◇, R. Cavada◇, M. Roveri◇, R. Bloem♣, A. Cimatti◇
♣ Graz University of Technology, 8010 Graz, Austria {ipill, rbloem}@ist.tugraz.at
◇ ITC-irst, 38100 Povo, Italy, {semprini, cavada, roveri, cimatti}@itc.it

ABSTRACT

Formal languages are increasingly used to describe the functional requirements (specifications) of circuits. These requirements are used as a means to communicate design intent and as basis for verification. In both settings it is of utmost importance that the specifications are of high quality. However, formal requirements are seldom the object of validation, even though they can be hard to understand and interactions between them can be subtle. In this paper we present techniques and guidelines to explore and assure the quality of a formal specification. 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 *assertions*, which must be satisfied, and a set of *possibilities*, which must not be contradicted. The proposed techniques support the user in the iterative development and refinement of high-quality specifications.

Categories and Subject Descriptors:
B.7.2 Hardware: Design Aids

General Terms: Design, Verification.

Keywords: Requirements Analysis, Hardware Design, Specification, Property Assurance, Property Simulation.

1. INTRODUCTION

Assertion-based verification is becoming increasingly important. In this setting, formally stated requirements are used to check the correctness of a design using both simulation and static verification. The advent of expressive and convenient specification languages such as PSL [1] and ForSpec [2] strengthens this trend.

Seldom, however, are requirements the *object* of quality control. This is somewhat surprising, since in practice requirements quality turns out to be one of the deciding factors for the success of a design project: industrial data shows that about 50 percent of product defects originate in flawed requirements and that around 80 percent of rework effort can be traced back to requirement defects [17].

In this paper, we address the crucial problem of supporting the designer in writing high quality (functional) requirements of circuits. (In the following we use requirements and specification as synonyms.) We note that such requirements analysis is not the activity of producing an implementation satisfying given properties. Rather, the focus is on enhancing the quality of the requirements before the design phase.

We assume that the requirements are expressed in a formal language such as PSL. The use of a formal language is a first and substantial step towards high quality specifications, as it makes subtle questions explicit that otherwise might be hidden in the ambiguity of natural language. However, a formal notation is obviously not enough to ensure the quality of the specification. In this paper we describe techniques, methodological guidelines, and a tool, RAT, for the formal analysis of functional requirements. Our approach draws from two complementary techniques: *property simulation* and *property assurance*.

Property simulation allows the designer to interactively explore the behaviors associated with the requirements: We construct a set of traces that satisfies the requirements. (Or, alternatively, a set of traces that violates it.) The designer can peruse the traces, change them by adding constraints, and review coverage information.

Property assurance offers a complementary, more global perspective, and enables the designer to analyze the strictness of the specification as a whole. By means of property assurance, it is possible to check if the requirements are *strict enough* to rule out unwanted behavior and if they are *not too strict* to allow for certain desirable behavior. User specified *assertions* must hold uniformly, while *possibilities* define behavioral patterns which have to be feasible.

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

RAT implements and integrates the proposed approach and provides a convenient graphical user interface. A designer can use RAT to develop and manage the specification, to archive requirements analysis results, and to simulate requirements behaviors for user defined scenarios.

[*]Supported by the European Commission (507219-PROSYD).

Previous work includes fine grained approaches that deal with the problem of understanding how well a set of properties covers a given design (coverage analysis, e.g., [12]), or help to distinguish "interesting" ways to satisfy a property from "uninteresting" ones (vacuity analysis, e.g., [4]). Higher level, methodological solutions propose techniques to check system descriptions against user-defined properties. In [10], 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; [13] and [9] contain similar proposals for the SpecTRM and RSML^{-e} languages, respectively. In [3] 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 analysed. 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.

This paper is structured as follows. In Section 2 we present our requirements analysis approach. A use case can be found in Section 3. Section 4 covers the techniques used to tackle the requirements analysis tasks. In Section 5 we describe the RAT tool. Finally in Section 6 we draw some conclusions.

2. REQUIREMENTS ANALYSIS

In this section, we describe how property simulation and assurance can help the requirements engineering process. The designer can use property simulation to examine concrete traces and to construct concrete scenarios. With property assurance, the designer can state more general demands on the requirements and can check those exhaustively.

2.1 Property Simulation

Property simulation provides the designer with an interactive method to understand the semantics of formal requirements by exploring their behavior one trace at a time.

Property simulation presents the designer with a set of traces that are representative of the requirements. The designer can select a trace, *constrain* it by fixing the value of any signal for any given time step, and then check whether the altered trace still adheres to the requirements. If not, the designer can ask for a trace that is correct and adheres to the constraints.

An explanation of the derived traces is provided in the form of the syntax tree of the property plus the truth values of each subformula at every step. The explanation makes the property easy to understand. In temporal logics such as LTL [14], 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.

It turns out to be useful to extend the control of the designer by allowing her 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 allows the designer to ascertain how well the requirements are exercised by the given traces.

The set of signals can be split into inputs and outputs. Based on this classification, the designer can perform a "what-if" analysis by setting inputs and asking to be presented with corresponding outputs. Dually, a "how-can" analysis can be performed by setting output signals and asking how, if at all, these outputs can be achieved.

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 the behavior of the property and allows her to ask concrete questions about it.

2.2 Property Assurance

Property assurance provides the designer with a very general means to assess whether she has written the right set of properties.

First of all, property assurance makes sure that there are no inconsistencies in the requirements: the properties must not contradict each other.

Second, the designer can provide two sets of properties: The *assertions* Φ_A, that must be guaranteed by the requirements, and the *possibilities* Φ_P each of which describes a behavior that must be included in the set of behaviors allowed.

The assertions are used to check whether the requirements are strict enough, that is, whether all unwanted behaviors are ruled out. The possibilities are used to check if the requirements are permissive enough, that is, whether they allow for all desirable behaviors. This extends the consistency check from checking for the existence of some trace to checking for the existence of a specific set of traces.

This approach is very similar to the application of model checking to validate a design against the specification. The main difference here is that the role of the design is played by the requirements, and we write properties to ensure that the formal requirements really capture the intended meaning.

To improve the specification, property assurance provides the designer with additional information based on the results of the performed checks. If an assertion does not pass, we show a behavior that is compatible with the specification but that violates the assertion, like in model checking. The behavior can then be used to correct the specification. If a possibility is satisfied by the specification, the designer is shown a behavior compatible with the specification and the possibility. If a possibility contradicts the specification, we show the designer a subset of the specification that is responsible of ruling out that behavior. All traces are accompanied by an explanation as described in Section 2.1.

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 behaviors, and she can either perform property simulation or write additional assertions or possibilities.

2.3 Guidelines for Requirements Analysis

Figure 1 suggests one way in which property simulation and property assurance may interact.

The designer comes up with an initial approximation of requirements Γ, assertions Φ_A, and possibilities Φ_P. The

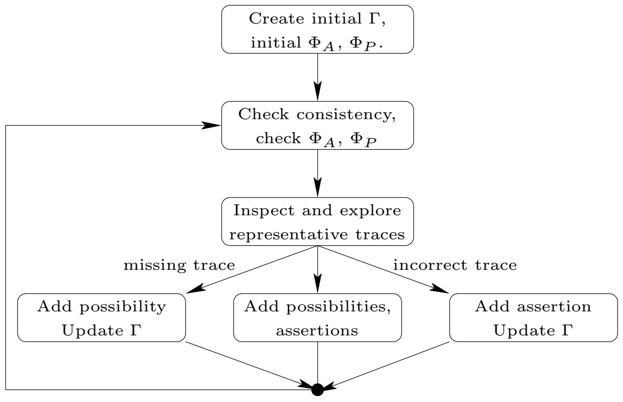

Create initial Γ, initial Φ_A, Φ_P.
Check consistency, check Φ_A, Φ_P
Inspect and explore representative traces
missing trace
incorrect trace
Add possibility Update Γ
Add possibilities, assertions
Add assertion Update Γ

Figure 1: Guidelines for requirements analysis.

process starts by checking whether the requirements are consistent, whether they allow for all possibilities in Φ_P, and whether they do not contradict any assertion in Φ_A. If there are any problems, the designer is presented with diagnosis information, as explained in the previous sections, and consequently refines Γ, Φ_A, and Φ_P to fix any problems encountered.

Subsequently, the designer is given the possibility of exploring a set of traces that is representative of the specification. 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_P. Likewise, if an unwanted trace is present, the requirements are corrected and an assertion is added to Φ_A to rule out all similar ones. In this way, whenever the designer changes the requirements, it is possible to automatically perform an exhaustive regression check by verifying that all assertions and possibilities are preserved. We will see an example of this in the next section.

Finally, the designer may decide to add extra assertions and possibilities. After any change, the requirements are again verified for consistency and for adherence to Φ_A and Φ_P.

3. USE CASE

The following example shows how to deal with the specification of an arbiter adopting our approach, and illustrates how problems with the specification are found and tackled.

The arbiter's signals consist of two input request lines r_1 and r_2, and outputs g_1 and g_2 for corresponding grants. We start with a very simple specification, where Γ consists of the following properties expressed in an LTL-like syntax:

$R1$: *forall i in* $\{1,2\}$: $G\ (r_i\ \rightarrow\ F\ g_i)$
$R2$: $G\ \neg(g_1 \wedge g_2)$

Requirement $R1$ states that for each request line, any request must be eventually answered with a grant. Requirement $R2$ 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, before adding them to Γ.) A first consistency check shows that Γ is consistent.

Initially, the set of possibilities Φ_P and the set of assertions Φ_A are empty. We start by adding assertion $A1$ to check that grants do not come too close together.

$A1$: *forall i in* $\{1,2\}$: $G\ ((g_i \wedge X\ \neg r_i)\ \rightarrow\ X\ \neg g_i)$

Assertion $A1$ 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. Checking Γ against $A1$ we are presented with the following counterexample:

```
r1 :  0 0 0 (0 ....)
g1 :  0 0 0 (0 ....)
r2 :  0 0 0 (0 ....)
g2 :  1 1 1 (1 ....)
```

The trace shows that we forgot to rule out initial spurious grants. Thus, we add a new requirement $R3$ stating that there must be no grants until there is a request.

$R3$: *forall i in* $\{1,2\}$: $(\neg g_i\ U\ r_i)$

Another check against $A1$ produces the following counterexample showing that our specification still violates the assertion. (As the arbiter is symmetric, in the following we will focus on one request/grant signal pair only.)

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

With $R3$ we eliminated initial spurious grants, but we forgot to rule out those following a grant. We further constrain the specification adding a new requirement $R4$ stating that whenever there is a grant, no more grants can be issued by the arbiter until there is a further request.

$R4$: *forall i in* $\{1,2\}$: $G\ (g_i\ \rightarrow\ X\ (\neg g_i\ U\ r_i))$

Once we have checked $\{R1, \ldots, R4\}$ for consistency, and got a positive answer, we re-check against $A1$ and we are assured that the flaw has been solved.

To gain confidence in the current specification's semantics, we simulate Γ to obtain a set of compliant traces. The derived set includes the following traces.

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

We are not provided with a trace featuring a single request with a single grant. Since we would like to examine the specification's behavior for such a simple case, we construct the following trace as stimulus and we simulate Γ for compliant traces.

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

The simulator unexpectedly tells us that the provided stimulus is not featured by the specification. When we consider the unfolded sub-formula evaluation of a counterexample as depicted in Figure 2, we find out that $G\ (g_1\ \rightarrow\ X\ (\neg g_1\ U\ r_1))$ is not true initially because $(\neg g_1\ U\ r_1)$ is false in the subsequent step.

Indeed, the strong until as used in $R3$ and $R4$ requires that r_1 must eventually appear, thus 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. Replacing the strong until in $R3$ and $R4$ with a weak until solves the encountered error:

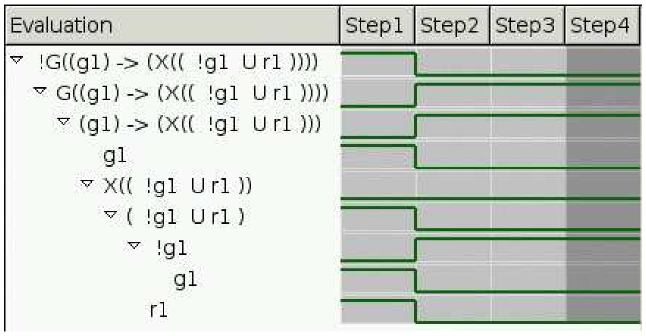

Evaluation
Step1
Step2
Step3
Step4
!G((g1) -> (X((!g1 U r1))))
G((g1) -> (X((!g1 U r1))))
(g1) -> (X((!g1 U r1)))
g1
X((!g1 U r1))
(!g1 U r1)
!g1
g1
r1

Figure 2: Formula evaluation: the trace shown consists of a finite stem from step 1 to 3 and an infinite loop on step 4.

R3 : *forall i in* $\{1, 2\}$: $(\neg g_i\ W\ r_i)$
R4 : *forall i in* $\{1, 2\}$: $G\ (g_i\ \rightarrow\ X\ (\neg g_i\ W\ r_i))$

To avoid similar errors in the future we generalize the trace with the following formula, and we add it to our set of possibilities Φ_P.

P1 : *forall i in* $\{1, 2\}$: $F\ G\ \neg r_i$

Subsequently, we re-check our specification for compliance to Φ_A and Φ_P.

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 two requests are answered by a single grant, and we simulate the specification for this case.

r_1 : `1 1 (0 ...)`
g_1 : `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 requirement assumption *R5*. Our implementation handles assumptions on the environment, which are not part of the specification, like requirements. This assumption restricts the environment, so that it may not issue further requests until the pending one has been answered.

R5 : *forall i in* $\{1, 2\}$: $G\ (r_i\ \rightarrow\ (\neg r_i\ U\ g_i))$

To make sure that the previous scenario is not possible anymore, we add the assertion *A2* to Φ_A. This assertion states that an unacknowledged request must never be followed by another request without a grant in between.

A2 : *forall i in* $\{1, 2\}$: $G\ \neg((r_i \wedge \neg g_i) \wedge X\ (\neg g_i\ U\ r_i))$

Since we have changed the specification, we re-check it for consistency, and we check it against all assertions, including *A2*, and all possibilities. This check passes.

Finally, we have derived a specification for our arbiter, consisting of *R1*, ..., *R5*, that meets our design intent. We found and addressed several flaws in the specification by investigating the specifications semantics, we developed assertions and possibilities to check the specification against, and we elicited a constraint on the environment behavior by means of a special requirement on the inputs. The quality of the specification has been assured by both formal checks and the exploration of the specification's behavior.

4. TECHNICAL ASPECTS

Automata-Based and Bounded Model Checking

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

In the automata-based approach [16], 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 [7].

Bounded Model Checking (BMC) [5] 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 path of length k. A finite path can represent infinite behavior in the following sense: either it is finite and it represents all its infinite extensions, or it is infinite since 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 are encoded as a propositional satisfiability problem and given as input to a SAT solver. The parameters k and l are modified until we either find a witness (the problem is satisfiable) or reach a sufficiently high value of k to guarantee completeness.

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. 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 behavioral aspects.

This approach offers high flexibility in specifying stimuli: anything expressible by an automaton is supported. The ability to state abstract requests, e.g., that something has to happen sometime, strengthens our simulator's usability.

Representative Traces

Using simulation, one must consider a variety of representative scenarios to gain a good overview of a property's behavioral aspects.

We construct a set of traces that visits all reachable fair states of the Büchi automaton for the specification. This set provides a good overview of the property's behavior while still yielding a reasonable number of traces. To compute the set for a non-deterministic Büchi Automaton, we first remove unreachable states and states with no path to a fair cycle (a cycle containing an accepting state). Then, we greedily construct paths containing unvisited states and fair cycles until all states are covered.

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. (See, e.g. [4]). For example, a trace for an arbiter in which no requests are issued does not properly exercise the property $G(r \rightarrow F\, g)$, as the subformula $F\, g$ is irrelevant. Such traces represent extreme behavioral aspects of a property. Thus, it is important to draw the designer's attention to such traces, so she can decide whether they should have been ruled out or not.

For each subformula of a property, we show the user, for both the finite and infinite parts of the traces, how many times the subformula becomes true or false as well as the number of times its value changes. 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.

Considering subformulae dependencies of the shown trace, the designer can judge which subformulae are responsible for the current evaluation and which ones do not affect it. This assists the user in the design of further simulation stimuli.

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 $\varphi \in \Phi_P$ is possible by computing whether $SAT(\Gamma \wedge \varphi)$ holds. An assertion $\varphi \in \Phi_A$ holds if $\Gamma \rightarrow \varphi$ is valid, i.e., if all possible behaviors are compatible with φ. (Dually, we can check that $SAT(\Gamma \wedge \neg\varphi)$ does not hold.)

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 assertion we model check $\Gamma \rightarrow \varphi$, while if it is a possibility 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 a possibility check, if the language is empty there is no behavior compatible with both the specification and the possibility. 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 possibility. (Consistency can be handled in the same way.) Dually, in the case of assertions, if the language is empty there are no behaviors compatible with the specification that contradict the assertion. If the language is not empty, a trace of the automaton is the counterexample compatible with the specification and falsifying the assertion.

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

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 [5] have been developed (e.g., [11]). BMC-based verification is efficient for checking possibility 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 assertion properties. If no counterexamples are found up to a reasonable k, then we can proceed with the more expensive BDD-based approach.

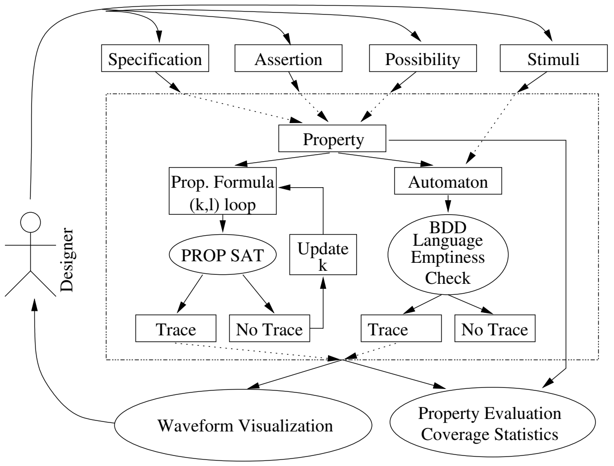
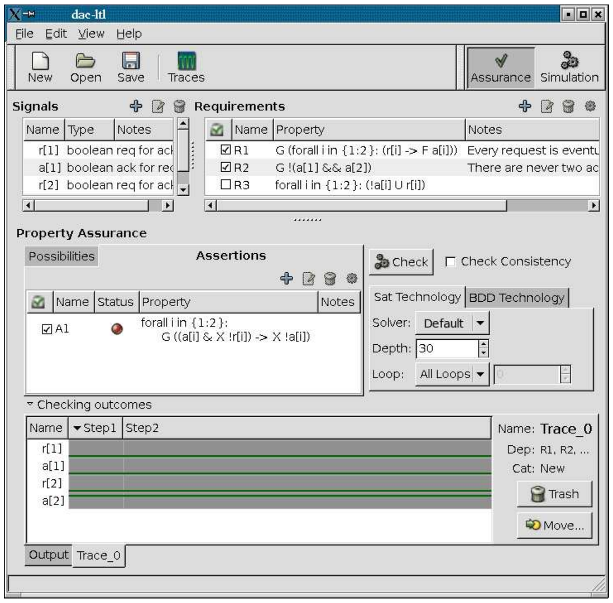


Figure 3: RAT architecture.



Signals

Name	Type	Notes
r[1]	boolean	req for ack
a[1]	boolean	ack for rec
r[2]	boolean	req for ack

Requirements

	Name	Property	Notes
☑	R1	G (forall i in {1:2}: (r[i] -> F a[i]))	Every request is eventu
☑	R2	G !(a[1] && a[2])	There are never two ac
☐	R3	forall i in {1:2}: (!a[i] U r[i])	

Property Assurance

Possibilities Assertions

	Name	Status	Property	Notes
☑	A1	●	forall i in {1:2}: G ((a[i] & X !r[i]) -> X !a[i])	

Check ☐ Check Consistency

Sat Technology BDD Technology

Solver: Default

Depth: 30

Loop: All Loops

Checking outcomes

Name Step1 Step2

r[1]
a[1]
r[2]
a[2]

Name: Trace_0
Dep: R1, R2, ...
Cat: New

Trash Move...

Output Trace_0

Figure 4: A screen-shot of the RAT tool.

Notice that the BDD approach does not perform well on large specifications because of the size of the underlying automaton. (See [8] 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 a possibility $\varphi \in \Phi_P$ single out a (hopefully small) subset of Γ that is inconsistent with φ.

5. TOOL

The concepts presented in the previous sections were used in the design and development of RAT. The high level architecture of RAT is depicted in Figure 3, while Figure 4 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 assertions and possibilities (property assurance).

The GUI provides a user-friendly interface to the underlying engines. For example, in order to check assertions the combination of requirements and assertions is submitted to a BMC engine. The 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 projects*, 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, assertions, and possibilities 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 [6, 15] to provide the proposed functionality. However, our design allows for an easy integration of other verification engines to support further languages or verification algorithms.

RAT can be obtained from its webhome *http://rat.itc.it*.

6. CONCLUSIONS

In this paper, we addressed the problem of supporting a designer in writing high-quality specifications. This is a significant problem since requirements are hard to get right and many bugs can be traced back to flaws in the requirements themselves. In contrast to design verification, where the object of analysis is the design, we propose techniques that are tailored to the analysis of the specification.

We provide the designer with a set of representative traces that she can use to query the tool about the behavior allowed by the requirements. Thus, the designer is presented with tangible and concrete information about the considered requirements.

We also allow the designer to write sets of possibilities and assertions, providing a general way to explore and assure the global behavior of the requirements.

We integrated these techniques in the RAT tool, which allows the designer to manage the requirements, assertions, possibilities, and traces by means of a graphical user interface. The tool is accompanied by a set of guidelines to perform requirements analysis.

We are currently performing an industrial case study with RAT. Initial feedback is encouraging.

7. REFERENCES

[1] Accellera. Property specification language — reference manual, version 1.1, June 2004.

[2] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 296–311. Springer-Verlag, April 2002.

[3] 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.

[4] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, July 1999.

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

[7] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. pages 427–432, San Francisco, CA, June 1995.

[8] 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.

[9] 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.

[10] 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.

[11] T. Junttila K. Heljanko and T. Latvala. Incremental and complete bounded model checking for full PLTL. In K. Etessami and S. K. Rajamani, editors, *17^{th} International Conference CAV 2005*, number 3576 in LNCS, pages 98–111. Springer-Verlag, 2005.

[12] 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.

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

[14] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[15] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

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

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