FP6-IST-507219

PROSYD

Property-Based System Design

Instrument: Specific Targeted Research Project
Thematic Priority: Information Society Technologies

Property Simulation

(Deliverable D1.2/1)

Due date of deliverable: 1 May 2005 (reissue)
Actual Delivery date: 1 May 2005

Start date of project: 01.01.2004 Duration: 3 years

Organisation name of lead contractor for this deliverable: Graz University of Technology

Revision: 2.0

Project co-funded by the European Commission within the Sixth Framework Programme (2000-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact roderick.bloem@ist.tu-graz.ac.at.

This document is intended to fulfill the obligations of the PROSYD project concerning delivery D1.2/1, described in contract number 507219.

Table of Revisions

Version	Date	Description & Reason	By
0.1	June 2004	Creation	Frank, Moulin, Pill
1.0RC1	July 2004	Revision	Jobstman n, Pill
1.0RC2	August 2004	Incorporated comments from Klaus Winkelmann (IFX) Sitvanit Ruah, Mark Moulin (IBM) Marco Roveri (ITC-irst)	Pill
1.0RC3	September 2004	Revision	Bloem
1.0	September 2004	Submission to Commission	
1.1	November 2004	Adressing comments from October review meeting. New style, sections for research that was previously omitted.	Bloem
1.2	November 2004	Technical work: Büchi Automata	Bloem
1.3	November 2004	Technical approach: separated normal form	Roveri, Semprini
1.4	November 2004	Inserted description of effort	Bloem
1.5	December 2004	Description of pathfinder	Sterin
1.6	December 2004	Added implementation description	Bloem
1.7	March 2005	Incorporated comments from written EU review report	Bloem
1.8	April 2005	incorporate comments from management	Bloem
1.9	April 2005	Incorporated comments from management	Roveri, Semprini
1.10	April 2005	Final pass for quality	Bloem
1.11	April 2005	formatting	Bloem
2.0	April 28, 2005	Final approval by project coordinator	Eisner

Authors

Ingo Pill,

Barbara Jobstmann

Roderick Bloem

Roy Frank

Mark Moulin

Baruch Sterin

Marco Roveri

Simone Semprini

Executive Summary

Properties in PSL may be hard to understand. What is currently provided to help designers in developing a clear understanding of the language is not enough. The material is either not comprehensive (like tutorials and examples handbooks), or it

is not suitable for professionals that do not have a proper education in formal methods (like the formal semantics in the PSL Language Reference Manual).

When a design fails a property, it may be because the design is incorrect, or because the property is incorrect. Worse, due to a misunderstanding of the semantics, the property could actually express something different from what the designer has in mind, and in this case, the property's passing is meaningless. In order to minimize such cases, it is essential that the meaning of a property is absolutely clear to the user.

The main factor inhibiting the understanding of properties is that a specification in PSL is currently not executable. This work aims to alleviate that drawback by proposing an automated method of deriving example traces from properties. Two concerns are of importance:

1. A designer needs to be able to explore the property by proposing constraints on the traces and asking the tool if a trace exists that satisfies the constraints and the property.
2. A representative overview of the behavior of the property must be given.

We propose a tool that helps a designer explore a property, show the theory underlying the functioning of the tool, and provide a mockup of the tool to illustrate the interaction with the tool. We discuss some alternatives for presenting the output of the tool and discuss their relative merit.

The research is expected to benefit both novices and expert verification engineers, the former for the exploration of simple properties, and the latter for the understanding of the interaction between properties in a larger set of interacting properties.

The concept of property simulation is novel to this paper. There are as yet no interactive tools to help a designer understand a property. We build on existing work in temporal logics, and extend it with novel approaches to visualize properties, to capture user constraints, to find properties that are interesting, and to make sure that all possible behavior allowed by the property is captured in simulation. We propose two novel technical approaches for property simulation based on Büchi automata and Separated Normal Form, respectively. The main advantage of Separated Normal Form is its efficiency, but Büchi automata are amenable to coverage techniques. We are studying methods to combine the two approaches.

Purpose

The purpose of this document is to report on the research in property simulation within the PROSYD project. It presents a novel technique for the understanding of properties, and discusses its technical background. It will serve as a basis for the implementation within the PROSYD project and may serve as a basis for implementations by third parties.

Intended Audience

Chapters 1 through 3 are aimed at designers and verification engineers. They explain the working of the tool. Chapters 4 and on describe the workings of the tool and are aimed at researchers and programmers looking to implement property simulation.

Background

The semantics of PSL are described in the language reference manual, and a handbook of examples was delivered as Deliverable 1.1/3.

Other research in specifying properties includes the use of graphical methods for specification and the use of templates. These methods have not found wide acceptance as they do not allow for easy specification of complex properties, and because they hide the complexity in the interaction of simple properties. These methods are described in more detail in Section 5 .

Contents

Table of Figures

Table of Tables

Glossary

Büchi Automaton A finite automaton on infinite words (omega-automaton). A Büchi automaton has the same structure as a standard finite-state automaton, but for it acceptance condition. The acceptance condition is a set of set of states. An infinite word is accepted if it has a run that visits a state in each of these sets infinitely often. Defined in Chapter 4 . See also [23].

Emerson-Lei algorithm An algorithm for detection of accepting cycles in a Büchi automaton [18]. Described as a mixed least/greatest fixpoint computation, it uses a modified breadth-first search that makes it efficient to implement in symbolic approaches. Commonly used in model checkers.

Expansion rule A rule that splits a property in to two parts: an obligation for the current time step and an obligation for the next time step.

Hintikka Structure A structure showing the why a linear-time logic formula is or is not satisfied by showing the truth value of every subformula at every point of time.

Interesting trace A trace that satisfies the formula, but does not do so vacuously.

Linear Time Logic A logic in which structures are infinite words. In contrast, for a branching-time logic, the structure is a tree. Also: one specific linear-time logic also called LTL, introduces in [24]. PSL, with the exception of the optional branching extension, is a linear time logic based on LTL and regular expressions.

Negated Normal Form A normal form for logic formulas that prescribes negation to occur only on literals, i.e. Boolean symbols may be negated, but other than that, no negations are allowed. Any property can be put into negation normal form by the use of rules similar to DeMorgan's.

Separated Normal Form A translation of linear temporal logic into propositional logic based on the introduction of new propositional variables for the subformulas. Defined in Chapter 4 , see also [25].

Symbolic algorithms A class of graph algorithms that work with symbolic representations of the characteristic functions of a set of nodes. The basic operations are the computation of the set of all successors or predecessors of a set of nodes, and therefore (modified) breadth-first searches are efficiently implementable in a symbolic setting.

Vacuous satisfaction A trace satisfies a formula vacuously if it satisfies it, but at least one of the subformulas is irrelevant. I.e., the formula remains true when the subformula is replaced by either false or true.

Witness, counterexample A trace showing the truth (in the case of a witness) or falseness (in case of a counterexample) of a property on a given design.

1 Introduction

Motivation

Writing a correct property is hard. The work on property assurance, for example, is motivated by the fact that a passing property does not necessarily mean that the design under test is correct – the property may be wrong. The worst-case scenario is where a bug in a design goes undetected because of an incorrectly specified property.

Understanding a property is not trivial. The formal semantics are helpful in understanding a property, but are likely too complex for the average user. Pertinent examples come in handy, and so does experience, but neither can be counted on. For properties to be easy to understand, they must be executable. The task of this deliverable is to show how they are best made executable.

Property Simulation

Property simulation is a novel technique that allows a designer to explore the set of traces that is allowed by the property as well as the set of traces that is not allowed. Intuitively, this is related to hardware simulation, but properties target infinite behavior and are nondeterministic. (A well-known theoretical result prevents determinization of properties on infinite behaviors.) The exploration takes place by the user's specifying parts of the trace and the tool completing the trace to satisfy the property. The tool presents the user with a set of interesting and varied traces.

Property simulation focuses on properties per se, it does not take a design into consideration, as the interaction between the design and the property is not likely to aid the understanding of the property, and is the focus of verification.

To this end, the user will be presented with a waveform trace representing a certain behavior. The designer may change certain parts of the waveform, define "don't care" and "fixed" segments, and ask the tool to extend the waveform to either adhere or to violate the property. We propose two methods to help the user understand why a trace satisfies or violates a property.

1. The user is presented with a waveform stating which sub-formulas are satisfied when, or
2. The user is presented with a finite-state Büchi automaton showing how the trace is recognized or rejected

An exact description of the reasons for satisfaction or violation will help the user understand the property and will help look for corner cases in the behavior described.

An option for comparing two formulas will assist the user in exploring the distinctions and particularities of different formulas. A comparison is provided by offering traces which are valid for one property but not for the other.

Audience and Place in Design Flow

Property simulation is foremost a learning aid for understanding properties, targeted at designers with limited knowledge in temporal specifications. A basic understanding of the syntax is assumed. As a second aim, it helps advanced engineers to understand complex properties. It may also be used to explore complex counterexamples found during verification.

We assume that users with little experience with temporal logic will profit from this technology substantially, both through learning by doing and through the possibility to explore and validate single aspects of properties. It is important to note that there are no tools that allow a user to try different variants of properties or to explore subtle aspects of the properties and to receive immediate feedback.

For the intermediate to expert user, Property Simulation offers a means to deal with a complex set of formulas by simulating its behavior. It will help assess the interaction between formulas and the effect of minor changes.

Property Simulation can be used throughout the specification process to improve understanding of the expressiveness of the specification language as well as the exact semantics of the specifications. Specification is performed both before the implementation is started, and during implementation and verification. The tool would be useful in both stages.

Background

With the exception of Chapters 2 and 3 , this document presupposes a basic knowledge of PSL. The linear fragment of PSL is based on linear-time logic (LTL) and regular expressions. For the purpose of simplicity of illustration, we will limit ourselves to LTL properties in this document, but the techniques are easily generalized to full PSL. (The use of the full linear fragment of PSL is the subject of Deliverable 3.2/4.) We do not consider the optional branching extension: it is not commonly used and is inherently incompatible with the concept of traces.

We offer a quick overview of the operators in the linear fragment to refresh the reader's mind.

Table 1 Linear operators

Future Tense		Past Tense	
Operator	Meaning	Operator	Meaning
X	Next state	Y	Previous state (initially false)
		Z	Previous state (initially true)
U	Until	S	Since
F	Eventually	O	Once
R	Release	T	Trigger
G	Globally, always	H	Historically

Flow of Paper

The flow of the paper is as follows. First, we describe the basic approach of property simulation. We describe the visualization of a trace and the explanation of the truth value of the property for that trace. We also describe how the user can specify constraints on the traces shown. In Chapter 3 , we discuss extensions to this basic idea. We discuss alternative methods for visualization and for specifying constraints. Then, we discuss the concept of interesting and uninteresting traces, and the importance of separating them. Finally, we discuss how to cover a large amount of possible behavior of a property with few traces, and we discuss how to visualize the difference between related properties.

In Section 4 , we discuss the technical approach of finding traces using either Büchi automata or Separated Normal Form. We discuss how Büchi automata can be used to compute coverage measures and to provide a better sampling of the behavior of the property. Separated Normal Form, however, is likely to be more efficient. We also discuss the structure of the implementation and porting issues. Finally, we discuss related work.

2 A Property Simulation Tool

In this section we discuss the novel idea of the basic tool using a typical scenario. We discuss explanation of a property using subformulas and contrast this approach to visualization using automata. The next section describes extensions to the basic idea, including an alternative visualization method and research concerning coverage of different types of behavior of the property.

The tool will have a graphical user interface that contains waveforms, the property, and information illustrating why the property is true or false for the given waveform. The user will be able to fix signals to certain values for certain time steps and the tool will try to extend these signals to a valid or an invalid trace at the user's request.

See Figure 1 for an example user interface. The tool will be able to handle sets of properties, but we illustrate its function using the single property `G(r → Fa)`. This property expresses the functionality of a bus controller with one input signal `r` for request and an output signal `a` for acknowledge. The property states that whenever (globally, `G`) a request comes in, eventually (`F`) an acknowledge is produced[1].

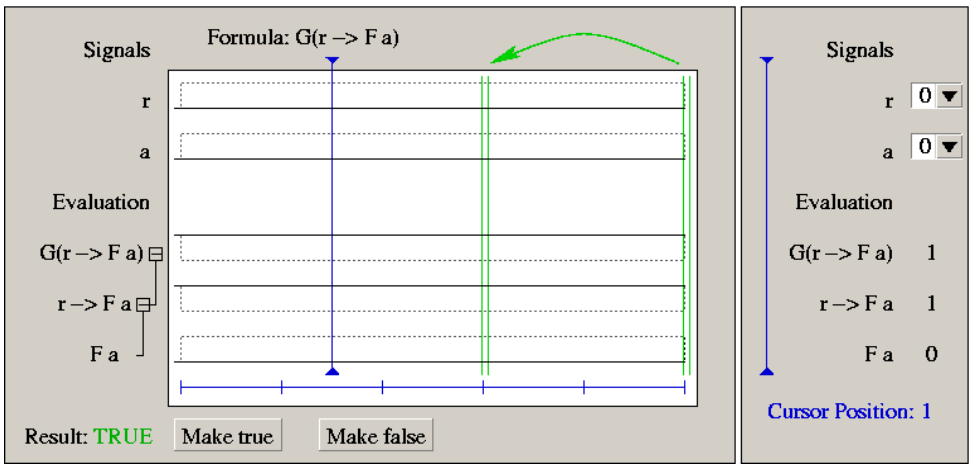

Figure 1. The user interface. The waveform starts on the left moves to the right and then loops between the pairs of double green lines

[1] The meaning of the `F` operator allows a request to be acknowledged in the same clock tick as it is issued.

The figure shows a waveform in which the signals `r` for request, and `a` for acknowledge are always low. The property is satisfied by this waveform, `G(r → Fa)` is initially true, which is indicated on the bottom left of the figure. The part to the left of the first vertical green double line is executed once, after which the part between the two double lines is executed ad infinitum. The vertical blue line indicates a cursor. The signal status related to the cursor position is presented in the window to the right, which allows for editing of the signals.

Visualization Using Sub-formulas

There is more than one way to illustrate the evaluation of a formula. The method chosen for Figure 1 illustrates the satisfaction of the formula by showing waveforms for all sub-formulas using a *Hintikka structure* [10]. Hintikka structures are well known but have not been previously used in tools to increase the understandability of a property.

We can see that `G(r → Fa)` is initially true, because the sub-formula `r → Fa` is always true. This results from the fact that `r` is always low, which means that the value of Fa is irrelevant. For increased readability, the user can fold away uninteresting sub-formulas (using the tree structure on the left of the figure).

Constraining the Trace

The waveform shown in Figure 1 is not particularly interesting: The reason that every request is acknowledged is that there are no requests.

In order to find a more interesting example, the user may set `r` high for the interval (1..2) to simulate a request. This is illustrated in Figure 2, in which the thick line shows the user's constraints. The rest of the trace is not altered and thus there is a request that is not acknowledges, and the property is now false. (Note that FALSE is displayed in the bottom left corner)

The fact that the property is false is explained by noting that signal a remains low forever, and therefore `r → Fa` is low in the interval (1..2). This implies that `G(r → Fa)` is false in the interval (0..2). Note that the property becomes true at time 2, as from then there are no further requests. However, the property does not hold for the trace in its entirety.

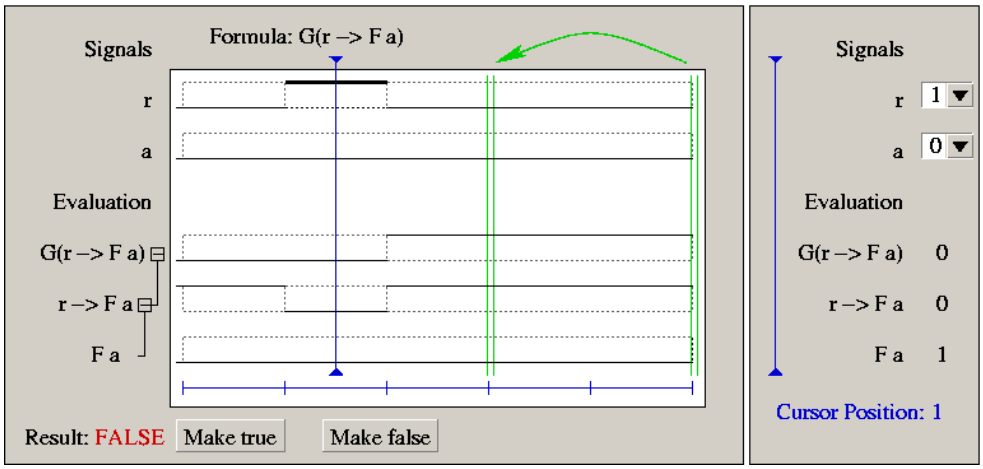

Figure 2 A similar trace with a request.

Requesting a True Trace

By clicking the “make true” button, the user can request the tool to change the traces to adhere to the property, while the user-specified signals are left unchanged. A possible trace provided by the tool is displayed in Figure 3.

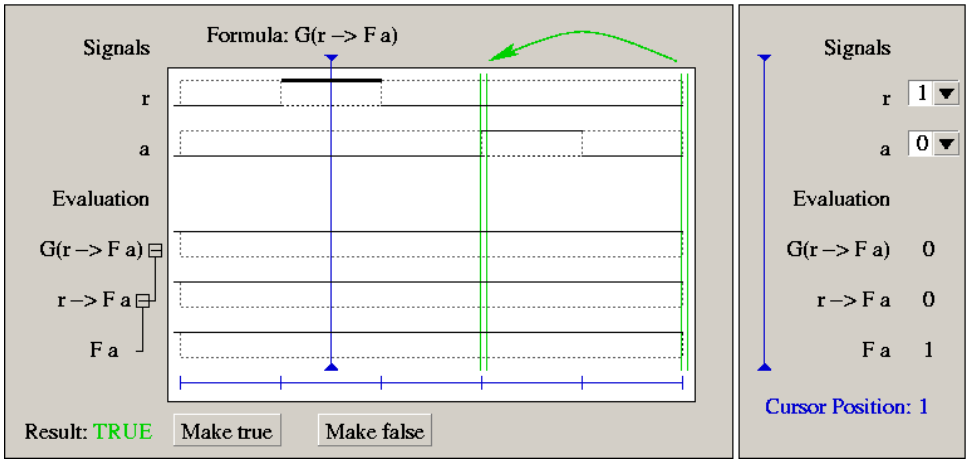

Figure 3. The corrected trace

In the case portrayed in Figure 3, the tool has responded by raising the acknowledge signal in the infinite portion of the trace, which means that acknowledge becomes high at time stamps 3, 5, 7, etc. This satisfies the property, since the request at time stamp 1 is answered.

The result shown in Figure 3 points out that the specification is probably incomplete: The user may want to refine the property to include the constraint that `a` shall not be high twice without an intermediate request. Thus, the tool may help the user discover that the specification is incomplete.

This interaction between the user and the tool allows the user to try corner cases in the behavior of the property and to see whether her intuitive understanding matches the actual meaning of the property. It requires that the user understand which aspects of the behavior maybe interesting. In a subsequent section, we will discuss how to help the designer at this task.

The approach using subformulas and waveforms is appealing to a designer as it closely mirrors the semantics of a formula and designers can be assumed to be familiar with waveform presentations.

3 Extensions

Having described the basic idea of property simulation, we discuss how to make the tool even more useful. First, we discuss the quality of visualization and an alternative visualization method using automata. Then we turn to the problem of finding the right traces. We discuss the concept of interesting properties and argue that it distinguishes two important classes of traces. Then, we introduce the concept of coverage of a property with the aim of finding a small set of traces that describes a large amount of possible behavior of the property. Finally, we discuss ways to visualize the difference between two properties.

Quality of Visualization

Visualization Using Automata

We now discuss an alternative approach to explaining why a property holds or does not hold.

A formula can be translated to a nondeterministic Büchi automaton [15]. Instead of showing the evaluation of the sub-formulas as waveforms, we can show the current positions in the automaton related to the cursor position within the signal traces. Büchi automata are well known, but there are no previous attempts at using them to explain why a property is satisfied.

Figure 4 shows the example of an automaton corresponding to the property `G(r → Fa)`. An automaton functions as a monitor that receives the signals from the waveforms as input. Thus, there is an infinite run of the automaton corresponding to any trace. The automaton has good states, shown with double circles, and bad states, shown with single circles. The automaton accepts a waveform, signifying that the property is true, if there is a run corresponding to the waveform that does not get stuck in a set of bad states. (See Section 4 for a definition of Büchi automata.)

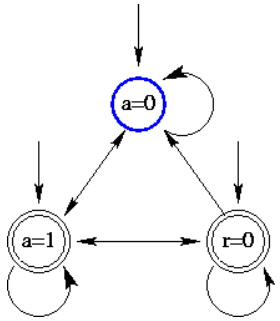

Figure 4. An automaton for G(r → F a)

If we consider the automaton in Figure 4 and the cursor position in the examples of Figures 2 or 3, the automaton is in the top state labeled with `a=0` (shown in blue). If the automaton is run on the waveforms of Figure 2 it will stay in this state, as signal `a` remains low, meaning that the property fails.

If the automaton is run on the waveforms of Figure 3, it will move to the state labeled `a=1` and then it can move to the state labeled `r=0` and stay in this state. Thus, the property passes, since the state `r=0` is a good one. Note that the automaton is non-deterministic: It contains another, non-accepting run for Figure 3 in which it returns to the top state. A trace is accepted if there exists at least one accepting run.

Non-determinism of Büchi automata poses a problem to the usability of this approach to visualization. Unfortunately, the usual subset construction can not be used for Büchi automata: there are properties that can be expressed by a nondeterministic but not by a deterministic automaton [23]. We can deal with nondeterminism by showing a set of possible states that the automaton can be in.

A drawback of the use of automata is that designers are not familiar with Büchi automata. Apart from that, the automata may become quite large. There are two ways in which we alleviate the problem of large formulas:

- The automaton is be separated into strongly connected components, and only the component in which the run currently is shown.
- For the frequent case of conjunctions we use one automaton per conjunct to keep the single automata small, and visualize all these automata with their actual states.

An advantage of visualization using automata is that they give a global overview of the property and allow the user to discern which completions are possible for the beginning of a trace. Visualization using waveforms and subformulas gives less information about possible future traces.

Optimizing Understandability

The approach we propose minimizes the effort required to understand the truth value of a formula. In fact, the truth value of LTL formulas has a simple definition in terms of its subformulas and its truth value in the next step. These dependencies, expressed as *expansion rules* are

- `f U g  = g ∨ (f ∧ X f U g)`
- `G f  = f ∧ X G f`
- `F f  = f ∨ XF f`

- f R g = g ∧ (f ∨ X f R g)

The expansion f U g = g ∨ (f ∧ X f U g), for example, means that the until formula f U g can be satisfied in one of two ways: either its second subformula is true now, or its first subformula is true now, and f U g is satisfied in the next state as well.

These rules form the basis for the simplicity of the presentation. In order to understand the valuation of any formula, the user has to understand only its valuation in the next step and the valuation of its subformulas. This information is presented closely together, and the relation between these items is shown clearly in the diagram presented: The truth value of a subformula is very easily understood by opening the display for the recursive subformulas and cursor helps the user focus on one particular time frame. Thus, the valuation of a formula, which inherently depends on the trace as a whole, can be reduced to the understanding of a very localized part of the diagram shown.

Consider the example shown in Figure 5. Suppose the user expects the formula G(r → F a) to be false for the given trace, and is looking for an explanation why it is true. The user's intuition is reasonable, as the second request apparently is not acknowledged, but the user oversees that the F operator allows an event to take place immediately. In Subfigure (a), the user sees that the formula is true, and she opens the subformula r → Fa to inspect why this is the case (See Subfigure (b)). The truth value of r → Fa at step 2 of is surprising to her: she had expected it to be false. Hence, the user opens the subformula Fa and notices that it, too, is true at step 2 (Subfigure c). Finally, the user opens the subformula a and here her conceptual mistake becomes clear: Fa is true if a is true. At any point, the user considers only one time step, and at most two subformulas at a time.

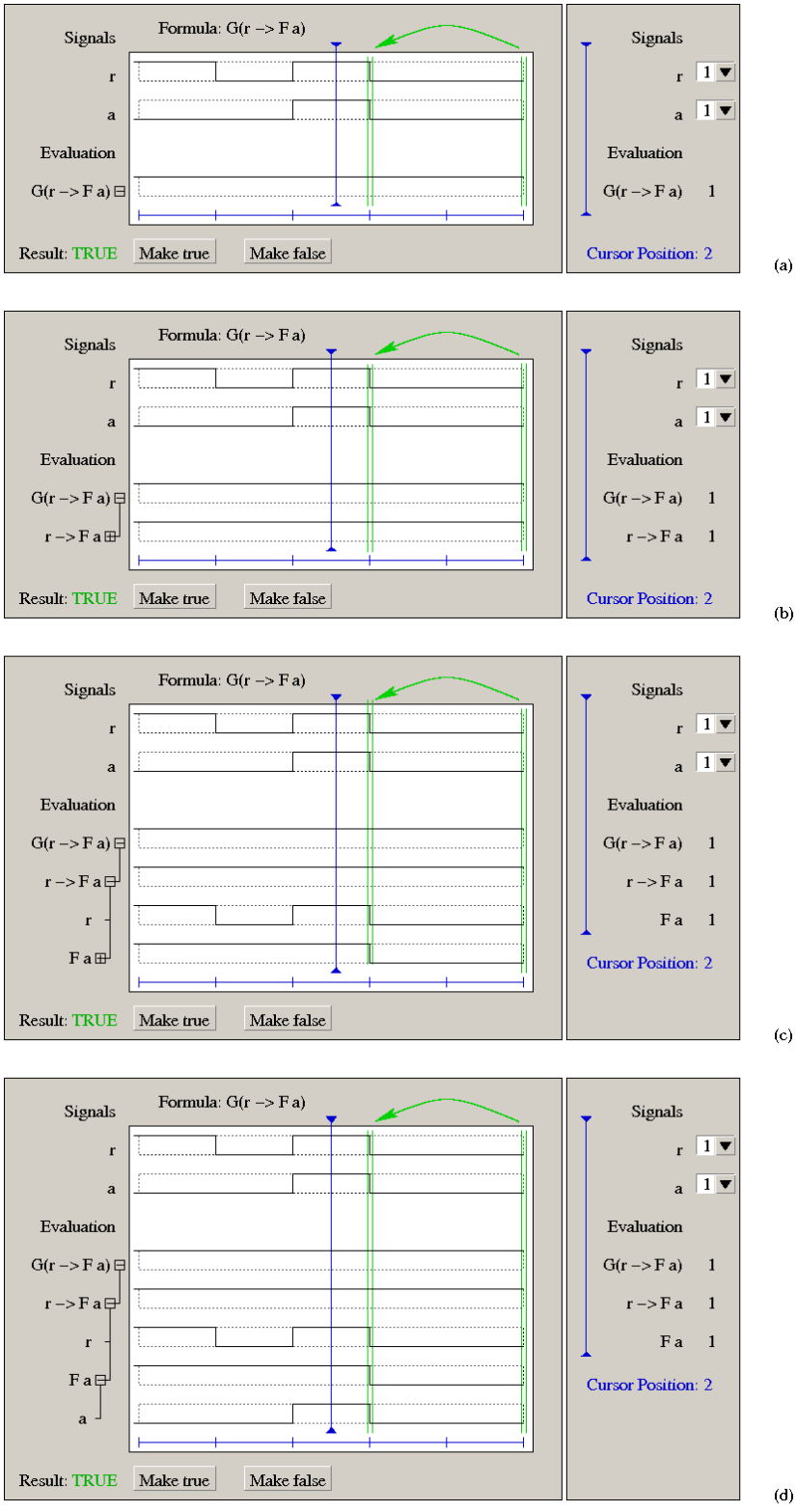

Figure 5 Locality of Explanation

Finding Representative Properties

Interesting Traces

Remember that the trace in Figure 1 was uninteresting: the fact that there were no requests trivially implied that all requests are eventually acknowledged. The tool should be able to treat such traces differently from the more standard examples..

The concept of an *interesting trace* was introduced by Beer et al. in [12], and addressed by Kupferman and Vardi in [13] in a more technical setting. A trace is uninteresting when it satisfies an implication *vacuously*, that is, there is a sub-formula that does not influence the truth of the formula. If a property is satisfies vacuously, we have a pathological case. The concept was originally introduced to find cases in which a design is found to pass a property in a pathological way. This indicates that the user may need to write more properties. In this setting, we use interesting traces to distinguish different kinds of satisfaction and to present them separately. This aspect is novel.

The trace shown in Figure 1 is an example of this definition: The property is `G(r` $\rightarrow$ `a)`, and the trace has `r` false throughout. That means that the property is true, and its truth value is not dependent on the sub-formula `a`. Thus, since `a` can be everything, we are not properly exercising the property.

The trace in Figure 3, this is interesting: the values of both sub-formulas are important, and therefore the trace is interesting.

Note that uninteresting traces describe corner case behavior that may be unexpected and is therefore important. Therefore, interesting and uninteresting traces should be presented separately and should be clearly marked, but neither should be dropped entirely.

Coverage

It is impossible to present the user with all traces satisfying (or violating) a property, as this number is usually infinite. The user should, however, be able to obtain a good idea of the different kinds of behavior allowed by the property, present using few traces. In a way, the traces need to show good *coverage* of the property. The concept of coverage of a property is not known from the literature, although it is related to the concept of interesting traces discussed in the last section. We propose to base coverage on the Büchi automaton for the property. We describe the approach in more detail in the subsection titled Coverage in Section 4 .

Comparing Properties

Errors often originate when a designer changes a property, since a change may lead to unexpected side effects. There is no previous research in the exploration of the difference between two properties, but it is important to have.

Suppose that the original property is φ and the new property is ψ, and suppose, without loss of generality, that the properties differ only in the subformula ξ. We extract traces that satisfy φ but not ψ by finding traces satisfying $\psi \wedge \neg\varphi$ and vice-versa. We then explore the behavior of the difference in the same way we do for properties.

The presentation of the subformulas is adapted for the purpose: the subformulas that are unique to φ or ψ are shown separately. Subformulas that do not have ξ as a (recursive) subformula are shown only once, and subformulas having ξ as a subformula are shown with two traces in different colors for φ and ψ. In this way, the user has a compact display with optimal representation of the difference between the two formulas.

For instance, consider the properties $\varphi = \mathtt{p} \rightarrow \mathtt{Fq}$ and $\psi = \mathtt{p} \rightarrow \mathtt{XFq}$. In this case, we show

- The traces for the subformulas `p`, `q` and `Fq` only once
- A trace for the formula `XFq`
- The traces for `p` $\rightarrow$ `Fq` and `p` $\rightarrow$ `XFq` are shown overlapping in two colors, so that the difference is immediately apparent. The same goes for the formulas `p` $\rightarrow$ `Fq` and p $\rightarrow$ `XFq`.

4 Technical Approach

This section discusses the technical approach to property simulation. There are two problems to be tackled

1. How to extract examples from a specification
2. How to select examples that adhere to the user's wishes.

These problems can be handled in two ways, either by building Büchi automata for the property and for the user's selections, or by using Separated Normal Form. Separated normal form has the advantage of being small in size. Traces for separated normal form are obtained using a SAT solver and can thus be computed quickly. The disadvantage is that there is no way of excluding the existence of a trace: we can only check the presence of a trace of any given length. Büchi automata are larger, but offer the option of excluding the existence of traces. Its state/transition based nature also allows coverage measures to be computed.

We conclude this section with a description of the structure of the implementation.

Büchi Automata

This section describes how to extract traces that satisfy both the formula and the requirements of the user. It is divided in two parts:

1. the construction of a Büchi automaton representing the combination of the formula and the user requirements and
2. the extraction of a trace

Büchi automata are well known, the trace extraction procedure is existing work. The conversion of the user specifications into a trace is novel, and so is the work on coverage.

Büchi Automata

We assume that a set AP of atomic propositions is given. The atomic propositions are the basic elements from which the formulas are built and can be derived from the formula. They are typically the signals in the design under test. We write $S = 2^{AP}$ for the alphabet. An element of S includes the signals that are 1 and excludes those that are 0; it is a minterm over AP.

A *generalized Büchi Automaton* [14], henceforth just Büchi automaton, over AP is a tuple (Q, q_0, d, L, F), where

- Q is a finite set of states,
- d: $Q \times Q$ is the transition relation,

- L: $Q \rightarrow 2^S$ is the labeling function, and
- $F \subseteq S$ is the set of acceptance conditions.

A *run* of an automaton A is a sequence $r \in Q^\omega$ such that $r_0 = q_0$, and for all i, $(r_i, r_{i+1}) \in d$. Let inf(r) be the set of states that appears in r infinitely often. A run is *accepting* if for every $f \in F$ we have that $inf(r) \cap f \neq \emptyset$. A word $w \in S^\omega$ is *accepted* if there is an accepting run r such that for all i, $w_i \subseteq r_i$. The *language* of A, L(A), is the set of all accepted words. The language of the automaton is non-empty iff there is a *fair cycle* in the graph (Q,d): a cycle that does not contain any states with the empty label and intersects all acceptance conditions.

Note that every state is labeled with sets of possible letters from the alphabet, instead of a single letter. Let AP be the set of atomic propositions in our formula, and let S be the power set of AP. A member of S is a minterm over our atomic propositions. Note that for every state of the *design* some labels are true and others are false, i.e., we can associate a minterm with every state of the design. We want to label the states of the *automaton* with cubes over AP, which are sets of minterms or subsets of S.

We use labels over 2^S for greater compactness. Consider Figure 4. It has a state labeled a=1. This state corresponds to the minterms ar and $a\bar{r}$. If we had not allowed for cubes on the states, we would have needed two states. This is especially important in the general case where we have many atomic propositions, but a state typically only constrains a few [15], [16]. Standard automata theory remains applicable for the major part.

Also note that the labels are on the states, and not on the arcs. These two representations are equivalent, but we prefer the former for technical reasons.

The *product* of two automata $A_1 = (Q_1, q_{01}, d_1, L_1, F_1)$, and $A_2 = (Q_2, q_{02}, d_2, L_2, F_2)$ is defined as $A_1 \times A_2 = (Q_1 \times Q_2, (q_{01}, q_{02}), d_{12}, L_{12}, F_{12})$, where

- $d_{12} = \{((q_1, q_2),(q_1', q_2')) \mid (q_1, q_1') \in d_1 \text{ and } (q_2, q_2') \in d_2\}$,
- $L_{12}((q_1, q_2)) = L_1(q_1) \cap L_2(q_2)$, and
- $F_{12} = \{f_1 \times Q_2 \mid f_1 \in F_1\} \cup \{Q_1 \times f_2 \mid f_2 \in F_2\}$.

We have that the language of the product of two automata is the intersection of their languages: $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$.

For generalized Büchi automata, it is relatively easy to form the product, as the acceptance conditions do not need to be combined. Note that the resulting automaton may have some states with the label false, which means that no accepting run can visit such a state.

The translation of LTL to automata is discussed in [15]; the translation of PSL into automata is subject of Deliverable 3.2/4.

Automaton for User Specification

Deriving a Büchi automaton from a user-specification is done as follows. Suppose that the user has specified a prefix of length n and a cycle of length m. Suppose in step k, the user has specified that the set of signals $P_k \in AP$ is positive and the set of signals $N_k \in AP$ is negative. (The two sets are disjoint and the set of signals $AP \setminus (P_k \cup N_k)$ is left unspecified by the user.) We construct a Büchi automaton U = (Q, q_0, d, L, F) with

- Q = [1..n+m],
- $q_0 = 1$,

- $d = \{ (i, i+1) \mid 1 \leq i < m+n \} \cup \{(n+m,n+1)\}$,
- $L(k) = \{ s \subseteq S \mid P_k \subseteq s \text{ and } N_k \cap s = \emptyset \}$, and
- $F = \{\{m\}\}$.

It should be clear that U accepts exactly those words that the user specified.

Given a formula f, we construct the automaton A_f such $L(A_f)$ is the set of words that satisfy f. We also construct the automaton U such that L(U) is the set of words that the user specified. Now, $A_f \times U$ accepts exactly the words that are specified by the user and adhere to the formula. Similarly, we can take $\neg f$ and construct $A_{\neg f} \times U$, which describes the words that adhere to the user specification but violate the property.

In order to ascertain the difference between two properties f and g, we extract traces from the automaton $A_{f \wedge \neg g}$, for traces that are allowed in f but not in g, or from $A_{\neg f \wedge g}$ in order to find traces that are allowed in g but not in f.

Extracting a Trace

We now show how to extract the word in the language of a given Büchi automaton. This process corresponds to counterexample generation as described in [17].

Traces are found using the following process:

1. Remove all states with label false,
2. Use the Emerson-Lei algorithm [18] or an alternative [19] to check for the existence of a reachable fair cycle. If no such cycle exists, the language of the automaton is empty: there are no traces that fulfill the property and the user's constraints.
3. Trace a shortest path from the initial state to a state in the first acceptance condition (from which a fair cycle can be reached), from there to a state in the second acceptance condition, etc.
4. When all acceptance conditions have been finished, try to close a loop to the state in acceptance condition 1.
5. If this is not successful, trace a path to acceptance condition 1, and try to close the loop.
6. Repeat this process until closing the loop is possible

A trace found in $A_f \times U$ or $A_{\neg f} \times U$ satisfies the user requirements and the formula or its negation, respectively.

If the language of the automaton is empty, the user specifications contradict the formula (or its negation), and no traces can be extracted. Otherwise, the algorithm is guaranteed to terminate because the state space is finite.

It has been shown in [17] that finding the shortest counterexample is NP complete.

Coverage

The fact that the Büchi automaton is a finite state machine means that we can apply coverage measures inspired by hardware simulation. *State coverage* [20], for example, measures the fraction of visited states of the automaton. T*ransition coverage* [21] is more precise and measures the fraction transitions of the automaton that have been traversed. *Control event coverage* [22] is also applicable: it would be good to cover as many different signal assignments as

possible. Neither has been applied to this problem before, mainly because property simulation is a new concept.

Each approach faces a tradeoff between short traces and few traces. Both increase the usability of the tool, because less information has to be understood by the designers. A technique such as *tour generation* [20], however, does not appear applicable as it tends to yield very long traces. That is more of a concern for property simulation than for hardware simulation, in which many vectors can be simulated very quickly. In the light of the afore-mentioned NP-completeness result on shortest counterexamples, we conjecture that no efficient algorithm exists that can optimize the quality of the result (which could be measured, for example, in the total number of states in all traces). Heuristics, however, should yield acceptable results. We present heuristics for state coverage, transition coverage and control event coverage:

- For state and transition coverage, pick one uncovered state (or transition) for every run, and constructing a trace that visits that state first. Then, pick another unvisited state reachable from the current one and extend the trace. Finally, close the loop. The number of picks allows us to trade length of trace versus number of traces.
- Similarly, for control event coverage, we pick a signal/value pair that was previously uncovered, pick a state in which that combination is possible, and construct a trace through that state. Note that for control event coverage we have more freedom as each state allows multiple assignments.

The coverage methods presented here guarantee that the traces found cover the different modes of behavior of a property, and make sure that no unexpected behavior is missed.

Separated Normal Form

The interaction with a property simulation tool greatly benefits from the availability of efficient techniques for checking the satisfiability of a set of formulas, in order to have prompt responses to the user input, allowing for quick cycles of specification, checking and visualization. With the aim of depicting an efficient and effective engine for a property simulation tool, we propose a novel approach to bounded model checking of PSL formulas that makes it possible to provide users with examples from a specification and select examples that adhere to users' needs. This approach is based on a new encoding of PSL into propositional logic inspired by Separated Normal Form (SNF) for LTL [25]. SNF is not new, but the approach we propose and the resulting encoding exploit the idea underlying SNF in a novel way and allow us to leverage the benefits of bounded model checking [27]. This technique has been developed as a core engine for satisfiability checking as a main result of the work for the property assurance related activities [29], but can be successfully applied also to the setting of property simulation.

Given a formula that we want to check for satisfiability, the main idea underlying the SNF reduction is to rewrite the formula as a set of sub-formulas and to introduce variables (subsequently referred to as 'SNF variables') to take into account the truth value of the sub-formulas. The evolution of SNF variables is constrained by rules that can be seen as defining a transition relation of an observer automaton. The encoding can be enhanced further by considering that, in the bounded case, eventualities can be expressed with a fix-point construction. Our

approach generalizes the construction of Frisch et al. [26], which shows significant improvements over the original construction presented in [27].

In the next sections the Separated Normal Form for PSL is introduced, and it is discussed how to generate an encoding for bounded model checking of PSL; moreover, it is shown how our approach can fit into property simulation needs.

For a more detailed presentation of this approach, including also some relevant optimizations, and for data on the experimental evaluation that has been performed refer to [28] and [29].

Separated Normal Form for PS

The Separated Normal Form (SNF) [30] is a clause-like normal form for temporal logic, based on the Separation Theorem. A formula in SNF has the general form

$$\mathbf{G}\left(\bigwedge_i (P_i \rightarrow F_i)\right)$$

where each implication $P_i \rightarrow F_i$, also referred to as a *rule*, relates some past time formula P_i to some future time formula F_i. Each rule has one of the following forms:

$$\mathbf{start} \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{X} \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{F} \bigvee_j l_j$$

where l_i, l_j are literals (i.e. either atomic propositions or negations of atomic propositions), and **start** is a property that identifies the initial state, In fact, **start** is an abbreviation for $\mathbf{Z}\bot$. The formula $\mathbf{Z}\gamma$, with γ a generic LTL formula, holds in a state if there exist a previous state that γ or if there is no a previous state. The latter is the only way of satisfying $\mathbf{Z}\bot$ given that $\bot$ cannot be satisfied by any state. In the following, the rules are referred to as start, invariant, next, and eventuality rules, respectively. Every LTL formula can be mapped onto a formula in SNF which is equisatisfiable [25], i.e., if the original formula can be satisfied by a given model, then its SNF translation can be satisfied by the same model augmented with the SNF variables derived from the conversion, and vice versa. Note that the translation process has been defined for full PLTL in [29],where a full description of the semantics of past time operators is provided; since PLTL is LTL augmented with past time operators, tour approach offers a full coverage of the LTL subset of PSL that is based only on pure future operators.

With respect to [25], we generalize the form of the rules to permit general

propositional formulas in place of $\bigwedge_i l_i$ and $\bigvee_j l_j$. A further difference is that we adopt non-strict semantics for time operators, so that all temporal operators other than **X**, **Y** and **Z** take into account the present time instant. In order to reduce to SNF a generic LTL formula γ, we define a transformation that manipulates sets of formulas. We start from the singleton set $\mathbf{start} \rightarrow \text{NNF}(\gamma)$,where $\text{NNF}(\gamma)$ is the negated normal form of γ, that intuitively states that γ has to hold in the initial state of any satisfying structure. Then, the conversion is carried out by the function $\text{SNF}(\cdot)$, which takes as input a set of formulas, and applies some transformation to a member of the set. The function is applied repeatedly until a set of rules is obtained. Intuitively, the transformations are devoted to eliminating occurrences of "complex" temporal operators by reducing them to more basic ones (i.e. **X** and **F**). To this end, each transformation can introduce new SNF variables, one for each temporal sub-formula being eliminated. In order to highlight their intuitive meaning, SNF variables are denoted as underlined temporal formulas (e.g. $\underline{\mathbf{XG}\varphi}$).

The transformations defining $\mathrm{SNF}(\cdot)$ are shown Figure 6 and Figure 7. We write Γ for the subset of formulas which are not affected by the transformation, φ and ψ for LTL formulas in NNF, and f and g for propositional formulas. In the rule being transformed, φ is the sub-formula that is not affected. We also write $\psi(\mathbf{G}\, f)$ to say that $\mathbf{G}\, f$ occurs in ψ, while $\psi(g)$ stands for the formula obtained by substituting every occurrence of $\mathbf{G}\, f$ with g in ψ. The same notation is used for the other temporal operators. The first four transformations in Figure 6, $\mathrm{SNF}_{[\mathrm{X}]}$, $\mathrm{SNF}_{[\mathrm{F}]}$, $\mathrm{SNF}_{[\mathrm{Y}]}$ and $\mathrm{SNF}_{[\mathrm{Z}]}$ are used to rename sub-formulas. The others have an intuitive interpretation, based on the fix-point characterizations of temporal operators. Consider the simple case of a $\mathbf{G}\, f$ formula: the corresponding set of rules is $\{\mathbf{start} \rightarrow f \wedge \underline{\mathbf{XG}\, f}, \underline{\mathbf{XG}\, f} \rightarrow \mathbf{X}f \wedge \underline{\mathbf{XG}\, f}\}$. The intuitive interpretation for the SNF variable $\underline{\mathbf{XG}\, f}$ is that $\mathbf{G}\, f$ holds in the next state. Similarly, consider the rule $\mathbf{O}f \rightarrow g$: the corresponding set of rules is $\{f \vee \underline{\mathbf{YO}\, f} \rightarrow g, f \vee \underline{\mathbf{YO}\, f} \rightarrow \mathbf{X}\underline{\mathbf{YO}\, f}\}$. The intuition here is that the SNF variable $\underline{\mathbf{YO}\, f}$ will hold in the next state if f holds in the current state, or it held in some previous state.

It is easy to see that the transformations only introduce SNF variables and **F**, **X**, **Y** and **Z** operators; together, $\mathrm{SNF}_{[\mathrm{Y2X}]}$ and $\mathrm{SNF}_{[\mathrm{Z2X}]}$ replace previous operators with next operators, so that the only remaining operators are **F** and **X**.

The transformations in Figure 6 rely on past operators appearing on the left side of rules and future operators on the right. The transformations $\mathrm{SNF}_{[\mathrm{p2p}]}$ and $\mathrm{SNF}_{[\mathrm{f2f}]}$, reported in Figure 7, are used to move operators onto the appropriate side (we use φ_{P} to denote a LTL formula with at least an occurrence of a past temporal operator applied to a purely propositional formula, and $\varphi_{\neg\mathrm{P}}$ to denote a formula with no such occurrences).

The other transformations in Figure 7 avoid renaming **Y** and **Z** operators in trivial cases.

In order to guarantee the termination of the transformation described above, some syntactic restrictions need to be enforced. The application of $\mathrm{SNF}_{[\mathrm{F}]}$ is forbidden in cases where the **F** operator is the main connective of the conclusion, i.e. when the transformed rule has the form $\psi \rightarrow \mathbf{F}\, g$; similar restrictions apply to $\mathrm{SNF}_{[\mathrm{X}]}$ $\mathrm{SNF}_{[\mathrm{Y}]}$ $\mathrm{SNF}_{[\mathrm{Z}]}$.

Furthermore, transformations $\mathrm{SNF}_{[\mathrm{Y2X}]}$ and $\mathrm{SNF}_{[\mathrm{Z2X}]}$ must not be used while the right hand side is $\neg\mathbf{start}$.

$$\textsc{Snf}_{[\mathbf{X}]}(\{\varphi \to \psi(\mathbf{X}\, f)\} \cup \Gamma) \doteq \left\{ \begin{matrix} \varphi \to \psi(\underline{\mathbf{X}\, f}) \\ \underline{\mathbf{X}\, f} \to \mathbf{X}\, f \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{F}]}(\{\varphi \to \psi(\mathbf{F}\, f)\} \cup \Gamma) \doteq \left\{ \begin{matrix} \varphi \to \psi(\underline{\mathbf{F}\, f}) \\ \underline{\mathbf{F}\, f} \to \mathbf{F}\, f \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Y}]}(\{\psi(\mathbf{Y}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(\underline{\mathbf{Y}\, f}) \to \varphi \\ \mathbf{Y}\, f \to \underline{\mathbf{Y}\, f} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Z}]}(\{\psi(\mathbf{Z}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(\underline{\mathbf{Z}\, f}) \to \varphi \\ \mathbf{Z}\, f \to \underline{\mathbf{Z}\, f} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{G}]}(\{\varphi \to \psi(\mathbf{G}\, f)\} \cup \Gamma) \doteq \left\{ \begin{matrix} \varphi \to \psi(f \wedge \underline{\mathbf{X}(\mathbf{G}\, f)}) \\ \underline{\mathbf{X}(\mathbf{G}\, f)} \to \mathbf{X}\,(f \wedge \underline{\mathbf{X}(\mathbf{G}\, f)}) \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{U}]}(\{\varphi \to \psi(f\, \mathbf{U}\, g)\} \cup \Gamma) \doteq \left\{ \begin{matrix} \varphi \to \psi(g \vee (f \wedge \underline{\mathbf{X}(f\, \mathbf{U}\, g)})) \\ \underline{\mathbf{X}(f\, \mathbf{U}\, g)} \to \mathbf{X}\,(g \vee (f \wedge \underline{\mathbf{X}(f\, \mathbf{U}\, g)})) \\ \varphi \to \mathbf{F}\, g \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{R}]}(\{\varphi \to \psi(f\, \mathbf{R}\, g)\} \cup \Gamma) \doteq \left\{ \begin{matrix} \varphi \to \psi(g \wedge (f \vee \underline{\mathbf{X}(f\, \mathbf{R}\, g)})) \\ \underline{\mathbf{X}(f\, \mathbf{R}\, g)} \to \mathbf{X}\,(g \wedge (f \vee \underline{\mathbf{X}(f\, \mathbf{R}\, g)})) \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{O}]}(\{\psi(\mathbf{O}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(f \vee \underline{\mathbf{Y}(\mathbf{O}\, f)}) \to \varphi \\ \mathbf{Y}\,(f \vee \underline{\mathbf{Y}(\mathbf{O}\, f)}) \to \underline{\mathbf{Y}(\mathbf{O}\, f)} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{H}]}(\{\psi(\mathbf{H}\, f) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(f \wedge \underline{\mathbf{Z}(\mathbf{H}\, f)}) \to \varphi \\ \mathbf{Z}\,(f \wedge \underline{\mathbf{Z}(\mathbf{H}\, f)}) \to \underline{\mathbf{Z}(\mathbf{H}\, f)} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{S}]}(\{\psi(f\, \mathbf{S}\, g) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(g \vee (f \wedge \underline{\mathbf{Z}(f\, \mathbf{S}\, g)})) \to \varphi \\ \mathbf{Z}\,(g \vee (f \wedge \underline{\mathbf{Z}(f\, \mathbf{S}\, g)})) \to \underline{\mathbf{Z}(f\, \mathbf{S}\, g)} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{T}]}(\{\psi(f\, \mathbf{T}\, g) \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \psi(g \wedge (f \vee \underline{\mathbf{Z}(f\, \mathbf{T}\, g)})) \to \varphi \\ \mathbf{Z}\,(g \wedge (f \vee \underline{\mathbf{Z}(f\, \mathbf{T}\, g)})) \to \underline{\mathbf{Z}(f\, \mathbf{T}\, g)} \end{matrix} \right\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Y2X}]}(\{\mathbf{Y}\, f \to \varphi\} \cup \Gamma) \doteq \{f \to \mathbf{X}\, \varphi\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{Z2X}]}(\{\mathbf{Z}\, f \to \varphi\} \cup \Gamma) \doteq \left\{ \begin{matrix} \mathbf{start} \to \varphi \\ f \to \mathbf{X}\, \varphi \end{matrix} \right\} \cup \Gamma$$

Figure 6. Part of the transformation function for SNF.

$$\textsc{Snf}_{[\text{p2p}]}(\{\mathbf{start} \to \varphi_P\} \cup \Gamma) \doteq \{\textsc{Nnf}(\neg\varphi_P) \to \neg\mathbf{start}\} \cup \Gamma$$
$$\textsc{Snf}_{[\text{f2f}]}(\{\psi_{\neg P} \to \neg\mathbf{start}\} \cup \Gamma) \doteq \{\mathbf{start} \to \textsc{Nnf}(\neg\psi_{\neg P})\} \cup \Gamma$$

$$\textsc{Snf}_{[\mathbf{start\,Y}]}(\{\mathbf{start} \to \mathbf{Y}\, \varphi\} \cup \Gamma) \doteq \{\mathbf{start} \to \bot\} \cup \Gamma$$
$$\textsc{Snf}_{[\mathbf{startZ}]}(\{\mathbf{start} \to \mathbf{Z}\, \varphi\} \cup \Gamma) \doteq \{\mathbf{start} \to \top\} \cup \Gamma$$

Figure 7. The transformation functions to deal with combining past and future.

Encoding Bounded Verification into SAT

Bounded Verification

The idea underlying bounded verification is to look for linear structures that can be presented with a number of steps (i.e. transitions) that is fixed a priori. We assume that the number of steps, also called the bound, is denoted k. While completeness may be lost, the exploitation of the bound often enables the use of alternate search techniques.

The idea of Bounded Model Checking [27] is to reduce an existential model checking problem $M \models \varphi$ with bound k to the problem of checking the satisfiability of a propositional formula $|[M \models_k \varphi]|$: this is satisfiable iff there exists a path in M which can be presented with k transitions and satisfies φ. The encoding is structured as a conjunction $\text{PATH}_k \wedge |[\varphi]|_k$, where the (propositional) models of the first conjunct correspond to finitely-expressible paths in M, while the second component encodes the requirements induced by φ. In the following, we assume that $\mathbb{A}$ is the set of atomic propositions occurring in M and in φ. We do not address the construction for PATH_k, which is standard. The case of bounded satisfiability simply reduces to the case of bounded model checking by simply dropping the PATH_k component from the encoding.

The problem of bounded satisfiability for φ is reduced to a propositional satisfiability problem as follows. The language of the propositional theory is defined by introducing, for each atomic proposition p in $\mathbb{A}$, k+1 propositional variables of the form $p(i)$, with i ranging from 0 to k. When the propositional variable $p(i)$ is assigned to true [false, respectively], the intuitive meaning is that p holds [does not hold] in the i-th state of the linear structure. In addition, the language of the propositional theory contains, for each SNF variable associated to $\text{SNF}(\varphi)$, k+1 propositional variables.

Intuitively, with bounded verification, it is possible to encode two different kinds of linear structures for φ: without loops, and with loops. When no loop is required, the propositional model corresponds to a whole class of linear structures sharing the same finite prefix, and which is sufficient to show the satisfiability of the formula φ. Intuitively, this is the case of violations to safety properties, which require that nothing bad ever happens – and it is therefore sufficient to show a finite path leading to a bad situation. Dually, this case can be seen as the satisfaction of a liveness property, for which it is sufficient so show a finite path leading to a good situation.

When a loop is required, the propositional model corresponds to a lasso-shaped linear structure, which is made up of a finite prefix u followed by a portion v repeated infinitely many times.

Intuitively, this is the case of violations to liveness properties, which requires that something good should happen. In this case, the structure reaches a point where only bad states keep repeating. Dually, this case can be seen as the satisfaction of a safety property, for which it is sufficient to show an infinite path where bad things never happen, which amounts to a witness for a Büchi automaton.

While the case of a "finite" prefix requires no additional constraints, in order to find a looping behavior we enforce that the k-th state be equal to some preceding state. In the propositional theory, a loop-back from k to l, with $l < k$, is captured by stating that, for each atomic proposition $p \in \mathbb{A}$, the corresponding propositional variables at k and l are assigned the same truth values, i.e. $_l\text{L}_k = \bigwedge_{p \in \mathbb{A}} (p(l) \leftrightarrow p(k))$.

Encoding the SNF Rules

The problem of *k*-satisfiability for a LTL formula φ is obtained by encoding each rule in SNF(φ) over the *k*+1 time instants, depending on the existence of a loop.

The encoding is structured as follows:

$$\bigwedge_{i=0}^{k} \bigwedge_{\rho \in \mathrm{SNF}(\varphi)} {}_{-}[\![\rho]\!]_k^i \quad \vee \quad \bigvee_{l=0}^{k-1} \left({}_{l}L_k \wedge \bigwedge_{i=0}^{k} \bigwedge_{\rho \in \mathrm{SNF}(\varphi)} {}_{l}[\![\rho]\!]_k^i \right)$$

where $|[]|^i_k$ stands for the encoding operator over a path of *k* steps, at step *i*, with loop-back at *l*. We use $l \in N$ to denote the loop-back point, while *l* = - denotes the absence of a loop.

The rules are encoded as follows:

$$
\begin{aligned}
{}_{l}[\![\mathbf{start} \rightarrow f]\!]_k^i &\doteq \begin{cases} {}_{l}[\![f]\!]_k^i & \text{if } i = 0 \\ \top & \text{otherwise} \end{cases} \\
{}_{l}[\![f \rightarrow g]\!]_k^i &\doteq {}_{l}[\![f]\!]_k^i \rightarrow {}_{l}[\![g]\!]_k^i \\
{}_{l}[\![f \rightarrow \mathbf{X}\, g]\!]_k^i &\doteq \begin{cases} {}_{l}[\![f]\!]_k^i \rightarrow {}_{l}[\![g]\!]_k^{i+1} & \text{if } i < k \\ {}_{l}[\![f]\!]_k^i \rightarrow {}_{l}[\![g]\!]_k^{l+1} & \text{if } i = k \text{ and } l \in \mathbb{N} \\ {}_{l}[\![f]\!]_k^i \rightarrow \bot & \text{if } i = k \text{ and } l = - \end{cases} \\
{}_{l}[\![f \rightarrow \mathbf{F}\, g]\!]_k^i &\doteq \begin{cases} {}_{-}[\![f]\!]_k^i \rightarrow {}_{-}[\![g \vee \underline{\mathbf{XF}g}]\!]_k^i \quad \wedge & \\ \quad {}_{-}[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]_k^i & \text{if } l = - \\ {}_{-}[\![f]\!]_k^i \rightarrow {}_{-}[\![g \vee \underline{\mathbf{XF}g}]\!]_k^{min(i,l)} \quad \wedge & \\ \quad {}_{l}[\![\underline{\mathbf{XF}g} \rightarrow \mathbf{X}(g \vee \underline{\mathbf{XF}g})]\!]_k^i & \text{if } l \in \mathbb{N} \end{cases}
\end{aligned}
$$

Intuitively, the rules are expanded as follows. The start rules express constraints only on the initial situation, and therefore have no effect on the subsequent time points. The invariant rules equally affect all of the time instants. The next rules are encoded in three different ways, depending on *k*, *i*, and *l*. Before the last state, the expansion is independent of *l* and *k*: the premise *f* is codified at state *i*, and the matrix of the conclusion *g* at *i*+1. At the last state, the premise is codified at *k*, while the matrix of the conclusion is either expanded at *l*+1, when a loop exists, or reduces to false, in case of no loop-back.

The expansion of the eventuality rule requires the preliminary creation of an SNF variable, **XF***g*, representing the fact that the eventuality is to be fulfilled at next state. Then, in the case of no loop-back, the expansion performs a renaming, generating an invariant rule, and a next rule describing the dynamics together with the enforcement of the eventuality before the end of the path.

This description expresses the loop optimization obtained in [26] with the introduction of the bound operator.

The loop case is reduced to the case without a loop at *min(i,l)*: this encompasses both the possibility of $i \geq l$, i.e. *i* is in the loop, and of *i*<*l*, i.e. *l* is before the loop.

The expansion of purely propositional formulas is straightforward. Notice however that their conversion may impact the way in which the corresponding CNF is obtained, and therefore on the efficiency of the SAT solver. For the sake of simplicity, we do not address these issues here (see e.g. [32]).

The number of propositional variables in the encoding is $O(|\mathbb{A}| + n) \cdot k$, where n is the number of occurrences of temporal operators in φ. In fact, each transformation introduces one new SNF variable, and each temporal operator can result in the introduction of up to two new variables. The worst case is the **U** operator, which requires the application of $\text{SNF}_{[U]}$, with the encoding for **F** introducing a second variable.

We also notice that the number of rules in $\text{SNF}(\varphi)$ is linear in n: for each occurrence of a temporal operator, SNF applies exactly one transformation, which can in turn require the application of another transformation. The worst case is again associated with the expansion of **U**. The number of rule instances in the above encoding is $O(n \cdot k^2)$, because of the different loop-back points.

Separated Normal Form and Property Simulation

The SNF-based approach can be naturally used to generate traces (witnesses or counterexamples) for the properties supplied by the user. First, the specification obtained conjoining the constraints on the signals and the property (possibly negated) is encoded as described in the previous sections. Then, a propositional problem is produced that can be fed into a SAT solver to produce evidence, if any, of the possibility of satisfying the specification, or to falsify it. Consistently with the aims of Property Simulation, this technology can be used also to allow for constraining the generated traces and let the user better understand the features of the specification under exam. This would allow both specification inspection and deeper understanding, and constraints specification via trace definition instead of properties.

Suppose the user is given a trace that shows the behavior of a number of signals as specified by the properties provided by the user, and suppose the user wants to modify the trace and check if it still represents an evolution of the signals that satisfies the property. Let φ be the specification given by the user and π a trace; here we are not interested on how the trace has been produced, whether by the Property Simulation tool as a witness/counterexample of the property and then modified by the user, or by the user itself as a way of constraining the evolution of signals. In general, the trace has the structure $\pi=\pi_0\pi_1^{\omega}$, meaning by this that it has an initial prefix π_0 possibly followed by a loop on the segment π_1, and contains the constraints the user gave on the values of the signals. In order to check if the trace and the specification are consistent with each other, all that needs to be done is to encode the signals evolution as described in the trace, to conjunct it with the encoding of the specification and check for the satisfiability of the following problem

$$|[\pi]| \wedge {}_l|[\varphi]|_k^k,$$

where k is the length of the trace $|\pi| = |\pi_0| + |\pi_1|$ and l is the length of the loop segment $|\pi_1|$; this formula gives the encoding of the specification φ along a generic path of length k with a loop-back at l, and constrains the path to satisfy the conditions imposed by the trace. The trace defines a conjunction of constraints that impose the values of the signals at all time instants between time zero and time k. The result of this check will be a witness of φ, if one exists, that is a path of length k such that the values of the signals at each state in the path are consistent with the requirements imposed by property φ; moreover, the witness trace will adhere to the constraints given by the user on the signal values and will define proper values for the non care intervals, i.e. the witness is an extension of the signals traces specified from the user. This amount to what is needed in order to answer to a user that asks the property simulation tool to make a property true, analogously, we can easily

map the dual request, that is making a property false, by building the following problem

$|[\pi]| \wedge {}_1|[\neg\varphi]|_k^k$

and submitting it to a SAT solver. The result of this check will be a counterexample for φ that is an extension of the signal traces provided by the user that falsifies the property.

Implementation

We consider the implementation issues of the development platform, use of libraries, licensing, portability, and the technical approach. This section covers the implementation strategy intended for the proposed tool. A mockup of the tool has been implemented following this strategy. The mockup supports the graphical interaction with the user without implementing the underlying technology for producing traces. It is mainly intended to show users what the tool will be able to do, and to make sure that implementation is feasible.

Development platform: To minimize future decisions on distribution platforms we decided to take special precautions to provide as much flexibility and portability. We have selected C as a standard language available on almost every platform. The gnu compiler collection is available for free on most platforms, and C allows for easy integration with legacy code and for fast executables.

Libraries: Since the tool has a graphical user interface, the choice of a portable graphics library is important. GTK+ is a graphical toolkit written in C available for many languages (C, C++, Java, Perl, Python, and many more) as well as multiple platforms (Windows, Linux, Unix, and MacOS). GTK+ is provided under the Lesser Gnu Public License (LGLP), which allows the creation of open source as well as proprietary projects without licensing fees. Thus, there are no licensing impediments to integrate property simulation in a commercial tool. GTK+ fulfills our implementation needs and is likely to be supported in the future, as it is used in well-known projects such as Mozilla, Mozilla Firefox, Mozilla Thunderbird, Netscape, and the graphical tool the Gimp.

Licensing: From licensing point of view, licenses of products used during development as well as distribution licenses are of interest. To gain as much flexibility in the distribution license issue we've taken precaution to restrict this decision as less as possible with the use of development tools and libraries. The used language C is available for free along with free compilers such as the GNU Tools. The use of GTK+ as graphical toolkit does not imply a decision for open source or proprietary release either nor the need to pay any fee. Thus the current development platform is neutral in this concern.

As part of this deliverable, we have implemented a mockup on an x86 Linux platform using the Gentoo Linux distribution running a 2.6 series kernel, GTK+ 2.0 and the GNU C-compiler gcc version 3.3.2. The prototype does not implement the functionality to calculate the waveforms, but features handwritten examples with pre-calculated waveforms. The mockup enables us to iteratively get feedback relating to the newly integrated features at various design steps. The mockup is available on the private PROSYD website for evaluation by project partners.

API and Structure

The structure of the property simulation tool is shown in Figure 8. The tool contains of two main parts, the GUI driver and the Translator. The GUI driver interfaces the tool with the GUI library. It is responsible for displaying traces, and for reading the user requirements. The Translator is responsible for the interface to the model checker. It builds a Büchi automaton or an SNF formula from the specification and the user requirements, and calls the model checker to obtain a trace. All calls to the model checker go through two API functions

`trace *getBATrace(buechiAutomaton *b, property *p)`, and

`trace *getSNFTrace(SNF *s)`.

The exact format of the data structures for the automaton, the property and the trace will be described in Deliverable 1.2/4. This very simple API has enough power to fulfill the needs described above. In the case of `getBATrace`, the Büchi automaton describes the combination of the specification and the user requirements and the property is used to describe requirements needed for coverage, such as 'state q in the automaton has to be reached.' In the case of `getSNFTrace`, the specification, the user requirements and the coverage criteria are all included in the SNF.

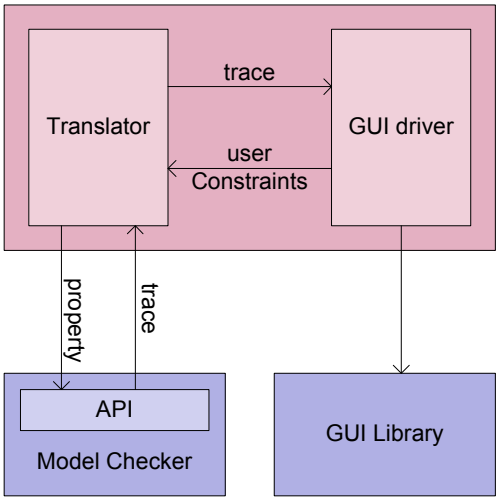

Figure 8. Structure of the tool

The API is easy to integrate in existing verification tools as Büchi automata and propositional formulas are basic data structures in these tools already. Implementing the API will involve a translation of the property simulation tool's data structures into the verification tool's data structures and a call to a function in the model checker that generates the trace.

Table 2 shows the implementation time plan and status.

Table 2. Implementation Time Table

Description	Date	Status
Mockup	August 2004	Done
Display of Traces	November 2004	Done
User Input	January 2005	Done
Interface to Model Checker	July 2005	In progress
Coverage measures	September 2005	Not started
Rework interface	September 2005	Not started

5 Related Work

Classification and Input of Properties

Properties have been classified in safety and liveness properties. Intuitively, safety properties state that something bad will never happen and liveness properties say that something good eventually happens. A syntactic classification is given in [6], and Alpern and Schneider give a topological characterization in [7]. The latter paper also shows that any property is the conjunction of a safety and a liveness property. A more detailed classification, based on the Borel hierarchy is given in [8]. These classifications are mostly helpful to experts in the logic. It is normally not immediately clear in which class a property belongs, so the classification is not helpful as a learning aid for novices.

The issue of specifying properties graphically has been addressed in [1], [2], [3], and [4]. This approach has not been widely applied, because it is hard to specify more complex properties graphically. Furthermore, the exact semantics may still not be clear from the graphical specification. This work is therefore complementary to ours.

Dwyer et al. [5] describe the use of standard templates to write properties. Because there are few templates, the effort needed to understanding them is limited. FormalCheck [9] takes a similar approach. Although the templates are simple, complex properties can be built by combining templates. Thus, the complexity is in the interaction between the templates. Property simulation may be useful for the approaches of [5] and [9] as well, but that is beyond the scope of this work.

Path Finder

PathFinder [11] is used to explore designs rather than properties. It generates interesting traces in the design according to simplistic scenarios.

PathFinder provides a means for the designer to explore, debug, and gain insight into the behavior of the design at a very early stage of the implementation – even before their design is complete. In the usage paradigm enabled by PathFinder, which is called Design Exploration, the design engineer specifies a behavior of interest and the tool then finds and graphically demonstrates a set of execution traces compliant with the specified behavior, if any exist. When presented with each such execution sequence, the designer is essentially furnished with an insight into the design behavior, and specifically with an example of a concrete scenario in which the behavior of interest occurs. This scenario can then be closely inspected, refined, or abandoned in favor of another scenario.

Technically, PathFinder works by translating scenarios specified by the designer into safety properties, and then challenging an underlying model checker with proving the negation of those properties. If the property presented to the model checker turns out to be false, the counter example is a trace demonstrating the scenario requested by the designed. Thus, with PathFinder designers can harness the power of static analysis without being subjected to the learning curve involved with formal specification and verification.

Thus, although the exploration aspect of PathFinder is similar to that of Property Simulation, its application area and its technical content are different.

Property Assurance

Property assurance (Prosyd deliverables 1.2/2 and 1.2/5) is aimed at making sure that the designer has specified the right set of properties. In contrast, property simulation is aimed at learning to use property specification, and at understanding a complicated set of properties.

The concept of *interesting traces* [12], [13], which was discussed in the last section, is a typical application property-assurance.

Property simulation is of significant value to designers using the iterative refinement method proposed for the property assurance task. In this method, a system is specified by a set of properties. Property simulation makes this specification executable and thus far easier to understand. Likewise the user of the property simulation tool is helped by the information provided by the property simulation tool. Therefore, the tools are developed in close cooperation and will have a consistent user interface.

The two approaches of assurance and simulation are complementary, and are both needed to assure a high-quality set of properties.

Dynamic and Static Property Checking

Dynamic and static property checkers are offered by several commercial parties and academic institutions.

Dynamic property checkers target the verification of PSL properties in a simulation setting. The generate monitors for properties that are either run with the simulator, or off-line on a simulator trace.

Static verifiers prove the adherence of a design to a property through formal means.

The usage of both techniques will be made simpler helped by our property simulation tool, as it helps the user be more certain that the stated properties are indeed correct. The waveform visualization techniques employed in simulators and dynamic property checking tools are similar to the ones we use.

6 References

Graphical Specification and Classification

[1] N. Amla, E. A. Emerson, R. P. Kurshan and K. S. Namjoshi, RTDT: A front-end for efficient model checking of synchronous timing diagrams, *Computer Aided Verification (CAV'01),* 2001.

[2] N. Amla, E. A. Emerson, R. P. Kurshan and Kedar S. Namjoshi, Model checking of synchronous timing diagrams, Formal M*ethods in Computer Aided Design (FMCAD'00),* pp. 283-298, Springer, 2000.

[3] U. Brockmeyer and G. Wittich. Tamagotchis need not die – Verification of STATEMATE designs. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, 1998

[4] W. Damm, B. Josko, and R. Schlörr, Specification and verification of VHDL-based system-level hardware designs. In *Specification and Validation Methods*, Oxford University Press, 1994.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, Property Specification Patterns for Finite-State Verification, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pp. 7-15, ACM Press, 1998.

[6] A. P. Sistla, Characterization of safety and liveness properties in temporal logic, *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pp. 39-48, 1985.

[7] B. Alpern and F. B. Schneider, Defining Liveness, *Information Processing Letters* 21, pp. 181-185, 1985.

[8] Z. Manna and A. Pnueli, A hierarchy of temporal properties, *Symposium on Principles of Distributed Computing (PDC'89)*, pp. 377-410, 1989

[9] R. P. Kurshan, *FormalCheck User's Manual,* Cadence Design Inc., 1998.

Hintikka Structures

[10] P. Wolper, Constructing Automata from Temporal Logic Formulas: A Tutorial, In Lectures on Formal Methods and Performance Analysis, LNCS 2090, pp. 261-277, Springer, 2001.

Design Exploration

[11] S. Ben-David, A. Gringauze, B. Sterin, Y. Wolfsthal. Pathfinder: A tool for design exploration, Computer-Aided Verification (CAV'02), pp 510-514, Springer, 2002.

Interesting Witnesses and Counterexamples

[12] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh. Efficient detection of vacuity in ACTL Formulas. Computer-Aided Verification (CAV'97), pp. 279-290, Springer, 1997.

[13] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. Correct Hardware Design and Verification Methods (CHARME'99), pp. 82-96, Springer, 1999.

Languages and Automata

[14] J. R. Büchi, On a decision method in restricted second order arithmetic. *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science,*pp. 1-11, Stanford University Press, 1962.

[15] M. Y. Vardi and P. Wolper, An Automata-Theoretic Approach to Automatic Program Verification, *Proceedings of the First Symposium on Logic in Computer Science*, pp. 322-331, 1986.

[16] E. M. Clarke, O. Grumberg, K. Hamaguchi, Another look at LTL model checking, *Formal Methods in System Design* 10, pp. 47-71, 1997.

[17] E. M. Clarke, O. Grumberg, K. McMillan, and X. Zhao, Efficient generation of counterexamples and witnesses in symbolic model checking, *Design Automation Conference (DAC'95),* 427-432, 1995.

[18] E. A. Emerson and C.-L. Lei, *Efficient model checking in fragments of the propositional mu-calculus,* Proceedings of the First Annual Symposium of Logic in Computer Science, pp. 267-278, 1986.

[19] K. Ravi, R. Bloem and F. Somenzi, A comparative study of symbolic algorithms for the computation of fair cycles, in *Formal Methods in Computer Aided Design (FMCAD'00),* pp. 143-160, Springer-Verlag, 2000

[20] R. Ho, C. Yang, M. Horowitz, D. Dill, Architecture validation for processors, In *Proceedings of the 22nd Annual Symposium on Computer Architecture,* pp. 404-413, 1995

[21] Y. Hoskote, D. Moundanos, and J. Abrahams, Automatic extraction of the control flow machine and applications to evaluating coverage of verification vectors, In *Proceedings of ICDD,* pp. 532-537, 1995.

[22] R. Ho and M. Horowitz, Validation coverage analysis for complex digital designs. In International Conference on Computer Aided, pp. 146-151, 1996

[23] W. Thomas, Automata on infinite objects, In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science,* pp. 133-191, 1990

[24] A. Pnueli, The temporal logic of programs, *IEEE Symposium on Foundations of Computer Science,*pp. 46-57, 1977.

Separated Normal Form

[25] M. Fisher and P. Nöel. Transformation and synthesis in METATEM Part I: Propositional METATEM. Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, February 1992.

[26] A. Frisch and D. Sheridan, T. Walsh. A Fixpoint Based Encoding for Bounded Model Checking. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design; 4th International Conference, FMCAD 2002*, volume

2517 of *Lecture Notes in Computer Science*, pages 238-254, Portland, OR, USA, November 2002. Springer-Verlag.

[27] A. Biere and A. Cimatti and E.~M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In W.R. Cleaveland, editors, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193-207. Springer-Verlag, July 1999.

[28] A. Cimatti and M. Roveri, D. Sheridan. Bounded Verification of Past LTL. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, 2004. To appear.

[29] Novel Techniques for Property Assurance. PROSYD deliverable 1.2/2.

[30] M. Fisher. A Resolution Method for Temporal Logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 1991.

[31] D. Gabbay. The Declarative Past and Imperative Future. In H. Barringer, editor, *Proceedings of the Colloquium on Temporal Logic and Specifications*, volume 398 of *Lecture Notes in Computer Science*, pages 409-448, 1989. Springer-Verlag.

[32] D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report. APES-82-2002, APES Research Group, March 2004.

A Justification of Effort

This section justifies the efforts for Deliverable D1.2/1, as requested by the reviewers in the review report dated 22 February 2005.

The total effort on Deliverable D1.2/1 consisted of 16 person months. The effort was scheduled as follows

Partner	Planned Effort	Actual Effort
IBM	3	1
Graz University of Technology	9	9
ITC-irst	6	6
Total	18	16

The original report did not contain the following information

1. The technical approach
2. The fact that a mockup tool existed.

Both facts should have been included, as they were part of the reported effort, but were left out, as the authors focused the document on the designers that were to use the tool. It was thought that designers would have been unduly burdened with such information.

Split up of the Effort

The effort at **IBM** consisted of:

- Studying the property simulation state of art
- Making a survey among the design community at Haifa Development Lab and Haifa Research Lab.
  - on feasibility of property simulation
  - review of PathFinder technology, as a possible candidate for property simulation prototype
- reviewing the reports

The effort at **ITC-irst** consisted of depicting the right technology to support a property simulation tool, and implementing it; in particular:

- thinking of the features of such a technology:
- compliance with the needs highlighted by the Property Simulation inquiry
- efficiency: in order to provide the user with quick specify-visualize cycles, the technology must allow for extracting results quickly;
- effectiveness: in order to allow the user to trace back reasons of satisfaction/unsatisfaction of a specification, the technology must provide a means to identify the constituents of a specification and connect them to the evolution of signals in a trace: the introduction of SNF variable is a possible way to achieve this connection;
- effectiveness: in order to allow the user to express naturally the properties of interest, the technology must give support to full LTL with both past and temporal operators
- implementing the technology:
- survey of existing technologies to deal with full LTL
- theoretical development of the SNF-based approach as the one that best fit with the requirements
- extension of NuSMV tool to support the SNF-based approach
- experimental evaluation to assess both functional and performance-related parameters

Moreover, ITC-irst collaborated to the writing and reviewing of the report.

At **Graz University of Technology**, one first-year PhD student has been working on property simulation. Additionally, one faculty member has invested statutory effort, which is not listed here. The effort consisted of

- Attaining a good idea of what the ideas and wishes for the deliverable were at the industrial partners (in particular IBM, ST UK, and Infineon),
- Ascertaining the feasibility of the wishes,
- Researching the state of the art in property specification and automata construction,
- Researching the state of the art on techniques similar to property simulation,
- Constructing a visual use scenario, elicitation of comments from partners and adaptation of the proposal to their wishes (this proved particularly time intensive, as we initially believed that we would use synthesis-like technology, which is theoretically very complicated),
- Development of the concept of coverage, and application of vacuity,
- Implementation of a mockup of the tool,
- Devising a technical approach, and
- Cowriting the report