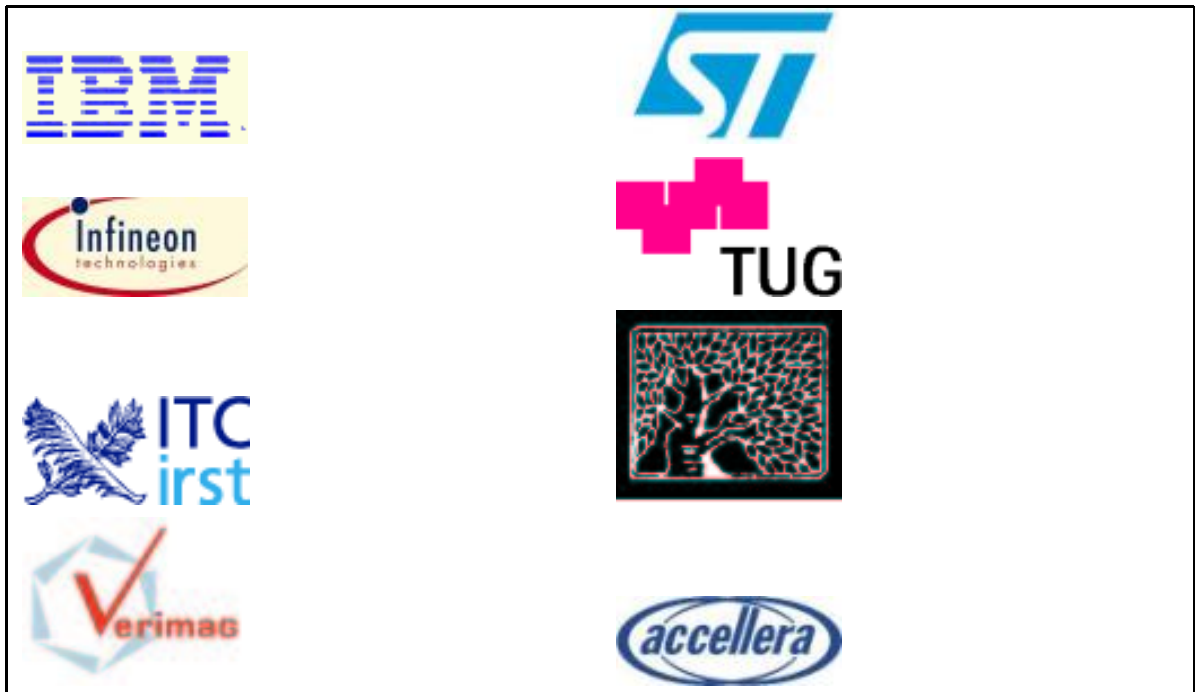


Novel Techniques for Property Assurance

Marco Roveri
ITC-irst

(Deliverable 1.2/2)



Contents

1	Introduction	1
2	Property assurance	2
2.1	A simple example	3
2.2	Property assurance proof obligations	4
2.3	Vacuity proof obligations	5
3	A refinement-based methodology	5
4	Background	7
4.1	Linear Temporal Logic with Past Operators	7
5	Existing technologies	10
5.1	LTL Satisfiability	10
5.2	Vacuity	13
6	Novel techniques	14
6.1	Bounded verification of Past LTL	15
7	Connections with other derivables and future works	23

1 Introduction

Property assurance is the activity of eliciting requirements that really capture the designer intent within the development of systems adopting a property-based approach in such a way that the design process is guided by the set of properties that the systems are called to satisfy. The way this activity is performed is dictated by a methodology that guarantees the correctness of the activity itself and prescribes how to develop a specification out of a set of initial properties: the methodology defines what to do, how to do it and which technologies to use.

Property assurance is not the activity of producing an implementation that satisfies given properties; the focus is on the specification phase, and the step from a specification to an implementation that is correct with respect to the specification is up to other activities (e.g. property synthesis), methodologies and technologies. Sinergies between property assurance and property synthesis do exist. Indeed, integrated in a process together with property synthesis, property assurance may provide a way to ease the delivery of implementations, written for example in VHDL or Verilog, that satisfy by construction a given set of properties written in a property specification language like PSL/Sugar. Since implementations automatically synthesized from a set of properties could be not as efficient and performing as hand written code, the coupling property assurance/property synthesis could not fit the development of performance-driven systems. Other interesting results could stem up from the coupling property assurance/property simulation as far as the understanding of specifications and the training of designer on temporal formalisms are regarded.

The input of property assurance is a set of properties/requirements that describe the behaviour of the system and possibly of the environment. Among the system properties there are those which are assumed to hold and those that must be guaranteed to hold given the assumptions. The output is a requirements specification of a system that satisfies those properties.

In the setting of the PROSYD project, the reference specification formalism is PSL/Sugar, consequently, both the input and the output are given as a collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviors.

People involved in any phase of the design and implementation of a system would greatly benefit from the introduction of a property assurance based methodology; indeed, having a property based approach and a methodology that allows deriving a specification from a set of properties (from the initial set of rather abstract and general assertions about the system and the environment, to a detailed description of its behaviour) would help to better define the interfaces between the various phases of the system development process and would foster a better and smoother integration of formal methods in hardware design.

This document present our work on property assurance and is organized as follow: Section 2 presents property assurance at an abstract level and shows how property assurance problems can be dealt with exploiting SAT theory; Section 3 outlines a methodology that could be used to discipline and direct the activities related to property assurance in a refinement-based style; Section 4.1 provides syntax and semantics for the temporal logic LTL; Section 5 presents a brief survey of the proposals from the literature on LTL satisfiability and

vacuity related topics; Section 6 is devoted to our proposal to deal with the bounded verification/satisfiability of full LTL specifications as a contribution to the suite of techniques designers can exploit to deal with system specification and verification; Section 7 highlights the connections that exists between the work presented in this document and the topics dealt with in other deliverables of the project, and depicts some future research directions.

2 Property assurance

The adoption of a formalism complemented by a proof system is essential to guarantee a high level of confidence on the specifications. A key element for the success of a methodology based on formal methods is the availability of tools that allow verifying automatically the specification. For this reason we foster the introduction of the proper technological support in the methodology since the beginning. In this section we depict the tasks related to property assurance and map them onto the proper technologies.

Suppose we have

- a set Γ of properties (the *requirements*) that describe the behaviour of the system (possibly even of the environment);
- a set Φ_A of properties (the *assertions*) that must be guaranteed by all behaviours of the system as described by Γ ;
- a set Φ_E of properties (the *possibilities*) each of which must be satisfied by at least one behaviour of the system that satisfies Γ .

With Γ the designer constraints the behaviour of the system by assuming the properties in Γ to hold; those properties in turn, must be enough to guarantee the properties collected in Φ_A , meaning by this that in any case the system described by Γ is required to satisfy all the properties in Φ_A . This is to ensure that the system is not under-constrained with respect to a set of desired properties, or, to put in into another way, that the system as we specified it will exhibit all the desired properties. Checking whether all the properties in Φ_A being valid in Γ is not enough. Indeed, there is the risk to over-constraint the system limiting too much the set of its possible behaviors. As a consequence it is necessary to check whether the system as described by Γ can satisfy the properties collected in Φ_E , meaning by this that for each property in Φ_E there must be at least one behavior compatible with Γ that satisfy that property. It has to be noticed that even if the use of Φ_E resemble the use of CTL non-determinism, non-determinism is not actually brought into the specification: in fact we are not sure about its practicality in the requirements specification phase. What we need is just to check the set of the models of our systems for the existence of particular models that satisfy some desirable but not mandatory properties.

In the requirements-assertions-possibilities approach to specification, the designer has to answer the following questions (in the following also referred as proof obligations):

1. Is the set of properties Γ consistent?

2. Is it possible to vacuously satisfy Γ ?
3. Does the satisfaction of Γ imply the satisfaction of any property in Φ_A ?
4. Is it possible to satisfy both Γ and any single property contained in Φ_E ?

Questions, or better proof obligations, (1), (3) and (4) can be mapped on existing proof technologies, in particular satisfiability (SAT) as shown in Section 2.2. The vacuity of a set Γ of formulae is the characteristic of Γ of being trivially satisfied. For example, when checking for the satisfiability of the set $\{G(q \rightarrow Fq)\}$, we say that the set is satisfied vacuously by models in which q is always false. The analysis of the vacuity conditions of a set of temporal formulae is dealt with in Section 2.3.

2.1 A simple example

In this section we give an example of the properties in the different sets of properties we envisaged considering the specification of a simple modulo `MAXCOUNTER` counter, being `MAXCOUNTER` a constant.

Γ could collect the following properties:

```
forall M in -MAXCOUNTER...MAXCOUNTER-1
  always (v=M & inc) -> next v=M+1
```

```
forall M in -MAXCOUNTER+1...MAXCOUNTER
  always (v=M & dec) -> next v=M-1
```

The first assumption states that whenever a `inc` operation is issued, in the next state the value of the counter will be incremented by one with respect to the current state value. The second assumption states an analogous property that relates decrement operations and the value of the counter in the current and next states.

An example of possible assumption on the environment is that increments and decrements are not issued at the same time, as stated by the assumption:

```
never (inc & dec)
```

The set Φ_A would collect assertions like:

```
forall N in 0...MAXCOUNTER
  always {v=0;inc[*N];dec[*N]} |=> v=0
```

stating that starting from a zero counter, after N consecutive `inc` operations and N consecutive `dec` operations, we get again to have a zero counter.

```
forall N in -MAXCOUNTER...MAXCOUNTER
  always (v=N) -> (v=N) until (inc | dec)
```

This property states that the value of the counter remains unchanged if no `inc` or `dec` operations occur.

Since we are dealing with the specification of a counter, an interesting property that we would like to be satisfied by at least a behavior of the counter is that it can be the case that the value of the counter changes, but, on the other hand, we do not want the counter to be forced to change: this is an example of Φ_E .

```
forall M in -MAXCOUNTER+1..MAXCOUNTER-1
  always ((v=M) -> eventually ((v=M+1) | (v=M-1)))
```

This property is satisfied by the models of Γ in which an `inc` or a `dec` is issued in a state where the value of the counter is equal to M (whatever M in the range). It would be impossible to satisfy such a formula, given the Γ we have, if it were a Φ_A , indeed it is not true that all the models of Γ satisfy it; for example those models in which increments and decrements are never issued do not satisfy this property (but are anyway legal models of Γ).

2.2 Property assurance proof obligations

Here we show how to map property assurance proof obligations onto a satisfiability (SAT) problem. In the following we write Γ and Φ_A for the conjunction of all their elements, namely $\bigwedge_{\gamma \in \Gamma} \gamma$ and $\bigwedge_{\phi_A \in \Phi_A} \phi_A$ respectively; this indeed is their intended meaning, since Γ is the set of properties that describe the system and Φ_A is a set of properties that must always be satisfied by the system. Φ_E will be treated differently, since it is asked that any single property in this set is satisfied by at least one behaviour of the systems, and not that all the behaviours must satisfy all the properties. We write $VALID(\Phi)$ to denote the problem of validity of the set of formulae Φ ; $SAT(\Phi)$ to denote the satisfiability problem of the set of formulae Φ . We use the duality $VALID(\Phi) \iff \neg SAT(\neg \Phi)$. In general the problem $SAT(\Phi)$ provides a “yes/no” answer, plus in the case of a “yes” answer a witness, that’s a behavior compatible with the ones allowed by Φ .

Proof obligation (1), that is the problem of checking whether a set of properties is consistent or not can be naturally reduced to the check on whether there exists a model for that set of properties. Thus, it reduces to check $SAT(\Gamma)$.

Proof obligation (3) aims to check whether that the set Γ of requirements is enough to ensure that all the assertions in the set Φ_A are satisfied, that’s check whether all the behaviors of Γ satisfy also all the properties in Φ_A : Γ is the specification of a system and Φ_A is the set of properties that any behaviour of the system is required to exhibit. This check corresponds to verify the validity of the implication $\Gamma \rightarrow \Phi_A$, that in turn can be reduced to the check on the unsatisfiability of the conjunction of Γ and the negated of Φ_A , indeed, what we need to verify is whether there are models of Γ that are not models of all the properties in Φ_A . Thus, proof obligation (3) is reduced to: $\forall \phi_A \in \Phi_A. \neg SAT(\Gamma \wedge \neg \phi_A)$.

Proof obligation (4) aims to check whether the set Γ of requirements is such that it is satisfiable in conjunction with any single possibility contained in the set Φ_E : that is it aims to check whether there is a behavior compatible both to Γ and to $\phi_E \in \Phi_E$. This check can be reduced to a verify on the satisfiability of the conjunction of Γ and ϕ_E for all $\phi_E \in \Phi_E$, indeed, what we need to prove for any formula in Φ_E is whether there exists a

model of Γ that is also a model for that formula. Thus, proof obligation 4 is reduced to: $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$.

2.3 Vacuity proof obligations

In model checking, a specification is vacuously true, if some subformula can be modified without affecting the truth value of the specification. Intuitively, this means that the property expressed in this subformula is satisfied for a trivial reason, and likely not the intended one.

The vacuity concept can be lifted in our framework by identifying which formula in Γ is vacuously true. Similar concepts can be applied to the formulae in Φ_E and in Φ_A .

The pieces of the requirements specification that are vacuously true can be used to strength the requirements specification with conditions that discard the behaviors that staify the specification vacuously. Moreover, whenever a property, be it an assertion or a possibility, does not hold [hold], we can exploit the vacuity techniques to extract a meaningful counter-example [witness] that does not falsify [does satisfy] the property vacuously, thus avoiding spending time trying to interpret useless traces.

A deeper investigation on the kind of technologies involved in the detection and identification of vacuity conditions is still an open point and part of future works, in particular as far as the possibility of exploiting SAT techniques also for vacuity proof obligations is regarded.

3 A refinement-based methodology

To guide the property assurance activity, we propose a methodology based on the concept of refinement. This approach would, indeed, allows deriving the final specification of a system following a multi-step process that starts from an initial coarse grained description of the system. At each step of the process, the designer refines the set of properties of the previous step by adding new properties, or by substituting a subset of the properties of the previous step for a set of new properties that describe the same features as the old ones but at a finer grained level of detail.

To ensure the correctness of such a process, the designer must prove that the set of properties at each step of the refinement process implies the properties of the previous levels; this guarantees that the final specification satisfies the initial properties. Being Γ^i the set of properties at the i^{th} step, this proof obligation can be dealt with as follows: we need to check the validity of the formula $\Gamma^i \rightarrow \Gamma^{i-1}$: $VALID(\Gamma^i \rightarrow \Gamma^{i-1})$. Which in turns reduces to check $\neg SAT(\Gamma^i \wedge \neg \Gamma^{i-1})$

Table 1 describes how the design process evolves accordingly to our methodology, and the proofs obligations at each step. We assume that only the set Γ evolves while Φ_A and Φ_E remain unchanged during the process; a further generalization of the methodology would be straightforward. In the table, for each step of the refinement process, it is shown what the set Γ contains, which properties we need to prove and the proof obligations we must cope with. At the beginning (step 0), we assume a given set Γ , which contains the initial requirements specification, and sets Φ_A and Φ_E . The proof obligations for the initial step are summarized by $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$ and $\forall \phi_A \in \Phi_A. \neg SAT(\Gamma \wedge \neg \phi_A)$. This is sufficient to cover the proof

Step	Specification	Props
0	Γ	$\Phi_A \Phi_E$
Proof obl	$\forall \phi_A \in \Phi_A. \neg SAT(\Gamma \wedge \neg \phi_A)$ $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$	
i	$\Gamma^i = \Gamma^{i-1} \cup \neg vacuity(\Gamma^{i-1}) \cup \Theta$	$\Phi_A \Phi_E$
Proof obl	$\forall \theta \in \Theta. \neg SAT(\Gamma \rightarrow \theta)$ $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$	
i	$\Gamma^i = \Gamma^{i-1} \setminus \{\psi\} \cup \neg vacuity(\Gamma^{i-1} \setminus \{\psi\}) \cup \Theta$	$\Phi_A \Phi_E$
Proof obl	$\psi \in \Gamma^{i-1}$ $\neg SAT((\bigwedge_{\theta \in \Theta} \theta) \wedge \neg \psi)$ $\neg SAT(\psi \rightarrow (\bigwedge_{\theta \in \Theta} \theta))$ $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$	

Table 1: The formalities of the methodology

obligations (1), (3) and (4) from Section 2.2; indeed, by proving $\forall \phi_E \in \Phi_E. SAT(\Gamma \wedge \phi_E)$, we also prove the consistence of Γ by showing that there is at least a model satisfying it (possibly at least as many models as many properties are contained in Φ_E , in any case every single property in Φ_E is guaranteed to be satisfied by a model of Γ).

The vacuity conditions of proof obligation (2) are considered in the transition from a step to the next one where Γ is conjuncted with the negated of its own vacuity conditions (see Table 1).

While we assume that Φ_A and Φ_E do not change during the refinement process, we dictate the conditions that rule the evolution of Γ from the initial requirements specification towards the final details system specification. The intuition is that the designer adds details to the specification by adding new properties Θ . When adding a set of new properties Θ , it is required that the new properties are not already implied by the existing ones (proof obligation $\forall \theta \in \Theta. \neg SAT(\Gamma \rightarrow \theta)$). When replacing a property $\psi \in \Gamma$ for a set of new properties Θ (Table 1, second row dedicated to step i), it is required that the new properties actually implies the old one, but are not implied by it, i.e. it is required that Θ is a real refinement of ψ (proof obligations $\neg SAT((\bigwedge_{\theta \in \Theta} \theta) \wedge \neg \psi)$ and $\neg SAT(\psi \rightarrow (\bigwedge_{\theta \in \Theta} \theta))$). This guarantees that the refinement proof obligation $\neg SAT(\Gamma^i \wedge \neg \Gamma^{i-1})$ is satisfied. In both cases, the satisfaction of Φ_E must be checked, while the satisfaction of the refinement relation together with the proof obligations of step 0 guarantee that at any step i $\Gamma^i \rightarrow \Phi_A$ (it can be verified easily since we have $\Gamma^i \rightarrow \Gamma^{i-1} \rightarrow \dots \rightarrow \Gamma^1 \rightarrow \Gamma \rightarrow \Phi_A$).

This is the presentation of the methodology in its generality, however the schema presented in Table 1 could follow a stricter pattern where two iterative phases can be depicted:

Phase 1 - defining Γ : during the steps of this phase, the designer just adds new formulae to Γ to satisfy the properties in Φ_A and Φ_E ; this phase terminates when Φ_A and Φ_E are satisfied and the designer has gained enough confidence on the quality of the specification.

Phase 2 - refining Γ : during the steps of this phase, the real refinement has place, i.e. the

designer substitutes old formulae in Γ for new ones that better detail the behaviour of the system: this phase terminates when the desired level of refinement has been reached.

These two phases are independent from each other and can be interleaved or serialized. In the first case it is possible to depict a process that links sets of system features to sets of properties; each set of features is specified independently and independently refined. This suggests a compositional and hybrid approach to specifications where different levels of abstraction can coexist. The sequential combination of definition and refinement, on the other hand, corresponds to a more classical approach to specification where first the requirements are identified, and only when they are all supposed to be stable, the designer uses them as input to produce the actual design of the system.

4 Background

As shown in Section 2.2 all the proof obligations, but the vacuity ones, can be mapped directly onto a satisfiability (SAT) problem. Since our reference specification formalism is PSL/Sugar, and since LTL (together with SEREs), constitutes a significant part of PSL/Sugar that is strictly relevant to requirements specification, we restrict to this subset for the time being.¹ In the following we describe the syntax and semantics of full LTL.

4.1 Linear Temporal Logic with Past Operators

We consider PLTL, i.e. the Linear Temporal Logic (LTL) augmented with past operators. The starting point is standard LTL which is a subset of the PSL/Sugar language. The formulae of LTL are constructed from propositional atoms by applying the future temporal operators **X** (next), **F** (future), **G** (globally), **U** (until), and **R** (releases), in addition to the usual Boolean connectives. PLTL extends LTL by introducing the past operators **Y**, **Z**, **O**, **H**, **S**, and **T**, which are the temporal duals of the future operators and allow us to express statements on the past time instants.

The **Y** (for “*Yesterday*”) operator is the temporal dual of **X** and refers to the *previous* time instant. At any non-initial time, $\mathbf{Y}\varphi$ is true if and only if φ holds at the previous time instant. The **Z** operator is similar to the **Y** operator, and it only differs in the way the initial time instant is dealt with: at time zero, $\mathbf{Y}\varphi$ is false, while $\mathbf{Z}\varphi$ is true. The **O** (for “*Once*”) operator is the temporal dual of **F** (sometimes in the future), so $\mathbf{O}\varphi$ is true iff φ is true at some past time instant (including the present time). Likewise, **H** (for “*Historically*”) is the past-time version of **G** (always in the future), so that $\mathbf{H}\varphi$ is true iff φ is always true in the past. The **S** (for “*Since*”) operator is the temporal dual of **U** (until), so that $\varphi\mathbf{S}\psi$ is true iff ψ holds somewhere in the past and φ is true from then up to now. Finally, we have $\varphi\mathbf{T}\psi = \neg(\neg\varphi\mathbf{S}\neg\psi)$ (**T** is called the “*Trigger*” operator), exactly as in the future case we have $\varphi\mathbf{R}\psi = \neg(\neg\varphi\mathbf{U}\neg\psi)$.

¹The discussion with industrial partners and the analysis of the literature in requirements engineering and specification pointed out doubts on the usefulness of CTL specific constructs in this requirements specification phase.

The syntax of PLTL is formally defined as follows:

Definition 1 (Syntax of PLTL) *The grammar for PLTL formulae is*

$$PLTL \ni \varphi, \psi \quad \doteq \quad p \mid \neg\varphi \mid \varphi \circ^{\mathbf{B}} \psi \mid \circ_1^{\mathbf{F}} \varphi \mid \varphi \circ_2^{\mathbf{F}} \psi \mid \circ_1^{\mathbf{P}} \varphi \mid \varphi \circ_2^{\mathbf{P}} \psi$$

where $p \in \mathcal{A}$ and \mathcal{A} is a finite set of atomic propositions, $\circ^{\mathbf{B}} \in \{\wedge, \vee\}$ stands for a Boolean connective, $\circ_1^{\mathbf{F}} \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$ and $\circ_2^{\mathbf{F}} \in \{\mathbf{R}, \mathbf{U}\}$ are future temporal operators (unary and binary, respectively), and $\circ_1^{\mathbf{P}} \in \{\mathbf{Y}, \mathbf{Z}, \mathbf{O}, \mathbf{H}\}$ and $\circ_2^{\mathbf{P}} \in \{\mathbf{T}, \mathbf{S}\}$ are past temporal operators (unary and binary).

In the following, we use φ and ψ to denote PLTL formulae, and p to denote propositions in \mathcal{A} . We write $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi$ for $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

As usual, PLTL formulae are interpreted over (linear) structures, that are basically infinite sequences of assignments to the propositions.

Definition 2 (Semantics of PLTL) *A linear structure π over a finite set of propositions \mathcal{A} is a function $\pi : \mathbb{N} \rightarrow 2^{\mathcal{A}}$.*

Let π be a linear structure over \mathcal{A} , let φ and ψ be PLTL formulae, and let $i, j, k \in \mathbb{N}$. Then φ holds in π at time i , written $(\pi, i) \models \varphi$, is inductively defined in Figure 1. φ is true in π , written $\pi \models \varphi$, iff $(\pi, 0) \models \varphi$.

Although the use of past operators in LTL does not introduce expressive power, it may allow to express temporal properties in an exponentially more succinct manner [16]. On an informal (but very important) level, past operators allow us to formalize properties more naturally. For instance, *if a problem is diagnosed, then a failure must have previously occurred*, can be represented in PLTL as

$$\mathbf{G}(\text{problem} \rightarrow \mathbf{O} \text{failure})$$

that is more natural than its pure-future counterpart $\neg(\neg\text{failure} \mathbf{U} \text{problem})$. Similarly, the property *grants are issued only upon requests* can be easily specified as

$$\mathbf{G}(\text{grant} \rightarrow \mathbf{Y}(\neg\text{grant} \mathbf{S} \text{request}))$$

compared to the corresponding pure-future translation

$$(\text{request} \mathbf{R} \neg\text{grant}) \wedge \mathbf{G}(\text{grant} \rightarrow (\text{request} \vee (\mathbf{X}(\text{request} \mathbf{R} \neg\text{grant}))))).$$

As for the pure future case, any formula in PLTL can be reduced to *Negation Normal Form* (NNF), where negation only occurs in front of atomic propositions. This linear time transformation is obtained by pushing the negation towards the leaves of the syntactic tree of the formula, and exploiting the dualities between conjunction and disjunction, \mathbf{F} and \mathbf{G} , \mathbf{U} and \mathbf{R} , \mathbf{O} and \mathbf{H} , and \mathbf{S} and \mathbf{T} . Notice that, in the case of previous time we have to rely on the two properties $\neg\mathbf{Y}\varphi \equiv \mathbf{Z}\neg\varphi$ and $\neg\mathbf{Z}\varphi \equiv \mathbf{Y}\neg\varphi$, which extend the single future-case rule $\neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$ (we have both $\neg\mathbf{Y}\varphi \not\equiv \mathbf{Y}\neg\varphi$ and $\neg\mathbf{Z}\varphi \not\equiv \mathbf{Z}\neg\varphi$, because of their semantics at the initial time point). We write the transformation to NNF of a formula φ as $\text{NNF}(\varphi)$.

Traditionally, temporal logics are used to express requirements over designs, represented as Kripke structures.

$$\begin{aligned}
(\pi, i) \models p & \quad \text{iff} \quad p \in \pi(i) \\
(\pi, i) \models \neg\varphi & \quad \text{iff} \quad (\pi, i) \not\models \varphi \\
(\pi, i) \models \varphi \vee \psi & \quad \text{iff} \quad (\pi, i) \models \varphi \text{ or } (\pi, i) \models \psi \\
(\pi, i) \models \varphi \wedge \psi & \quad \text{iff} \quad (\pi, i) \models \varphi \text{ and } (\pi, i) \models \psi \\
\\
(\pi, i) \models \mathbf{X}\varphi & \quad \text{iff} \quad (\pi, i+1) \models \varphi \\
(\pi, i) \models \mathbf{F}\varphi & \quad \text{iff} \quad \exists j \geq i. (\pi, j) \models \varphi \\
(\pi, i) \models \mathbf{G}\varphi & \quad \text{iff} \quad \forall j \geq i. (\pi, j) \models \varphi \\
(\pi, i) \models \varphi \mathbf{U}\psi & \quad \text{iff} \quad \exists j \geq i. ((\pi, j) \models \psi \text{ and } \forall k : i \leq k < j. (\pi, k) \models \varphi) \\
(\pi, i) \models \varphi \mathbf{R}\psi & \quad \text{iff} \quad \forall j \geq i. ((\pi, j) \models \psi \text{ or } \exists k : i \leq k < j. (\pi, k) \models \varphi) \\
\\
(\pi, i) \models \mathbf{Y}\varphi & \quad \text{iff} \quad i > 0 \text{ and } (\pi, i-1) \models \varphi \\
(\pi, i) \models \mathbf{Z}\varphi & \quad \text{iff} \quad i = 0 \text{ or } (\pi, i-1) \models \varphi \\
(\pi, i) \models \mathbf{O}\varphi & \quad \text{iff} \quad \exists j \leq i. (\pi, j) \models \varphi \\
(\pi, i) \models \mathbf{H}\varphi & \quad \text{iff} \quad \forall j \leq i. (\pi, j) \models \varphi \\
(\pi, i) \models \varphi \mathbf{S}\psi & \quad \text{iff} \quad \exists j \leq i. ((\pi, j) \models \psi \text{ and } \forall k : j < k \leq i. (\pi, k) \models \varphi) \\
(\pi, i) \models \varphi \mathbf{T}\psi & \quad \text{iff} \quad \forall j \leq i. ((\pi, j) \models \psi \text{ or } \exists k : j < k \leq i. (\pi, k) \models \varphi)
\end{aligned}$$

Figure 1: The semantics of PLTL

Definition 3 A (Boolean) Kripke structure over \mathcal{A} is a tuple $M = \langle S, I, T \rangle$, where $S = 2^{\mathcal{A}}$ is a finite set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a transition relation between states. A path in M is an infinite sequence of states s_0, s_1, \dots such that $s_0 \in I$ and, for all i , $T(s_i, s_{i+1})$. Given a path s_0, s_1, \dots , the corresponding linear structure maps i to s_i , for every i . A formula φ is existentially valid in M ($M \models \mathbf{E}\varphi$) iff it is true in the linear structure associated to some path π in M . Conversely, φ is universally valid in M ($M \models \mathbf{A}\varphi$) iff it is true in every linear structure associated to a path in M .

Clearly, there is a duality between the existential and the universal versions of the model checking problem, i.e. $M \models \mathbf{A}\varphi$ iff $M \not\models \mathbf{E}\neg\varphi$. The universal model checking problem can be intuitively interpreted as checking if all the behaviors in the system represented by M comply with the requirement φ ; the existential version is often interpreted as the problem of finding a witness to a violation of a required property. In the following, we assume that a Kripke structure M is given, and do not distinguish between a path in M and the corresponding linear structure. The satisfiability problem for φ can be seen as a model checking problem $M \models \varphi$, where M is a completely unconstrained Kripke structure of the form $\langle S, S, S \times S \rangle$, with $S = 2^{\mathcal{A}}$, and \mathcal{A} is the set of atomic propositions in φ .

5 Existing technologies

In this Section we present a brief survey of the most relevant approaches to LTL satisfiability and LTL vacuity detection.

5.1 LTL Satisfiability

5.1.1 “Emptiness of linear weak alternating automata” [17]

Authors: S. Mertz and A. Sezgin

Summary: The authors propose to exploit a particular class of alternating automata to check satisfiability and perform model checking of LTL formulae built using only future operators. The proposed approach is based on the fact that LTL formulae can be translated into linear alternating automata (also called weak alternating automata) that have size linear in the length of the formulae, and can be checked for language emptiness efficiently thus avoiding the need of a further translation in Büchi automata, which have size exponential in the length of the formula. Runs of alternating automata can be represented as dags. The rules for the translation from LTL to this class of automata are such that

- each location in the run dag is assigned a rank;
- no rising edges can exist, i.e. edges from a location with rank r to a location with a rank higher than r
- only self loops can exist.

The way the rules are given guarantees that each path in a run dag has non increasing and eventually stationary rank; this allows defining as accepting those dags that have at least one infinite path with limit rank even. As a consequence the non accepting run dags exhibit at least one path ending in a self loop on a location that has odd rank. The emptiness problem is thus reduced to a check on the existence of a finite run dag that exhibits two identical configurations (sets of locations obtained slicing the dag vertically) among which all the self loops at locations with odd rank can be avoided at least once. Iterating the loops of such a path we get an infinite accepting run dag. Since the authors give a bound to the length of the dags that must be searched for, they also suggest the naive decision procedure for emptiness of an automaton: enumerate all the run dags of the needed length and check whether they satisfy the required properties. A symbolic version of this procedure has been implemented by the authors, but in the case of automata with empty language the computation time is too high. The authors propose an alternative implementation based on the analysis of the graph of the reachable configurations of an automaton to check the existence of a nontrivial SCC that satisfies given properties. This implementation is based on a variant of Tarjan’s algorithm to enumerate SCC. The authors have not yet implemented extensions to LTL model checking.

5.1.2 “LTL with past and two-way very-weak alternating automata” [13]

Authors: P. Gastin, D. Oddoux

Summary: The problem addressed is that of an efficient translation from PLTL to Büchi automata; with regard to this, the authors propose a translation from LTL with past operators (PLTL) to Generalized Büchi automata (GBA) that is based on an intermediate translation to two-way very-weak alternating automata (2VWAA). This approach is compared with tableau based methods, both declarative or incremental, that are impractical or inefficient and hard to implement. The translation from LPTL to 2VWAA produces an automaton that has size linear in the length of the PLTL formula. The translation from 2VWAA to GBA produces an automaton that has size exponential in the size of the input automaton. The overall result is that the size of the final automaton is exponential in the length of the PLTL formula but has a exponent linear in the length of the PLTL formula. This result is opposed to other results that yield greater exponents.

5.1.3 “Model-Checking TRIO Specifications in SPIN” [18]

Authors: A. Morzenti, M. Pradella, P. San Pietro and P. Spoletini

Summary: The authors propose a methodology for checking the satisfiability of TRIO formulae based on a translation from TRIO to Promela programs guided by an equivalence between TRIO and two way alternating automata. It is shown as a proper decidable subset of TRIO can be seen as a more concise and syntactically sugared version of PLTL with past operators; the authors develop their translation for this subset of TRIO and map the specifications onto a particular class of two way alternating automata, namely two way modulo counting alternating automata. By using counting automata, the authors can succinctly translate the TRIO operators that, otherwise, should be expanded (e.g. as nested chains of \mathbf{X} s) thus giving place to an explosion in the size of the automaton resulting from the translation. The result of the translation is a Promela program that actually simulates the alternating automata and not the Büchi automata, as a consequence the size of the program is essentially proportional to the size of the formula translated. The input for this process has the form *specification* \rightarrow *properties* where both premises and consequences of the implication are sets of TRIO formulae and no operational component that can generate values for the elements of the alphabet of the specification is given. The result of the translation is a set of Promela processes that accept a language defined over the alphabet of the specification. The Promela processes obtained by a TRIO specification must be coupled with generators that give values for the logical variables. Since a systematic exhaustive enumerations of all possible variable values over time can lead to an explosion in the search state, the authors propose some optimizations based on the modular structure of TRIO specifications and on the analysis of input/output/state relations that hold among variables, clearly avoiding the generation of values for state and output variables. The results obtained by applying this methodology to a case study show that it is possible to deal with versions of the case study that exhibits values for TRIO metric constants that make other approaches unfeasible.

5.1.4 “A decision algorithm for Full Propositional Temporal Logic” [14]

Authors: Y. Kesten, Z. Manna, H. McGuire, A. Pnueli

Summary: This work proposes an efficient algorithm for checking the satisfiability of full propositional temporal logic (LTL with Past operators). The proposed algorithm can be used to check validity as well as satisfiability of such formulae over all models as well as over computations of a finite-state program. The approach is tableau based. The main difference with regard to the existing approaches based on tableau construction is the incremental construction of the tableau. Indeed, the algorithm builds only those *atoms* that are reachable from a possible initial atom satisfying the formula to check. The approach is decomposed in three phases. First, given the formula to check for satisfiability, the initial graph is built. Initially the graph contains as vertexes the set of initial atoms each containing the formula to check, the formula $\neg Y \top$ indicating the initial state, and a generic atom containing no formulae. All the edges connect the initial atoms to the empty atom. In the second phase, the initial graph is augmented. As long as an edge in the graph is unsatisfactory in the future or in the past, a new atom is inserted in order to augment the two vertex atoms considered as to make the edge satisfactory. The unsatisfactory edge is replaced by the edge connecting the old vertex to the new added atom. This augmentation continues until all the edges of the graph are satisfactory in the future and past directions. In the last phase, the graph is analyzed to extract maximal strongly connected components and identifying those where every atom has at least one successor, and where for every formula pUq belonging to an atom, there exists another atom in the strongly connected component that contain q . Optimizations are proposed to the second phase of the approach aiming to eliminate inefficiencies of the basic algorithm. If the procedure finds such a strongly connected component the formula from which the process started is satisfiable, otherwise the formula is unsatisfiable, i.e. the negation of the original formula is valid.

5.1.5 “A Tableau System for Linear-TIME Temporal Logic” [21]

Authors: P. H. Schmitt and J. Goubault-Larrecq

Summary: This work presents a tableau system for a subset of PLTL where only **G** and **F** temporal operators occurs. The approach thus does not cover past temporal operators and cannot deal with formulae where the next temporal operator occur (**X**). The proposed tableau system not only operates on formulae, but it exploits ”semantic” information in the form of signed clauses and linear constraints over time points. A signed clause can be either a time interval, a clause in the classical meaning, or the infinite symbol. The approach proposed starts building an initial tableau, that is then expanded with expansion steps until we have applicable expansion steps, accumulating constraints that are then used to check whether they become unsatisfiable. The formula is thus valid if starting from the initial table applying expansion steps we reach a “closed” tableau. The advantage of the approach is that no backtrack on the choice of the expansion rule has to be performed.

5.2 Vacuity

The following survey presents some works on the topics of vacuity detection and generation of meaningful witnesses for satisfiable formulae.

5.2.1 “Efficient Detection of Vacuity in Temporal Model Checking” [1]

Authors: I. Beer, S. Ben-David, C. Eisner and Y. Rodeh

Summary: This work provides a definition of vacuity based on the notion of *affect*: a subformula ψ of φ affects φ if there exists a formula ψ' that substituted to ψ in φ changes the validity of φ . Thus a formula φ is vacuous if there is a subformula ψ of φ that does not affect the validity of φ . The authors introduce the notion of minimal set of sub-formulae of a set S of subformulae as the set of subformulae of S that are no subformula of any formula in S . They provide a definition of vacuity with respect to the notion of minimal set of subformulae of a set S of formulae, and shows that vacuity of a formula ϕ can be detected by checking for vacuity taking into consideration only to the minimal subformulae of ϕ . They shows that with logics with polarity it is enough to check the replacement of a subformula by either true or false. Moreover, they provide an enhanced model checking algorithm that keeps into account the possible vacuity conditions. The authors identify a subset of ACTL called w-ACTL and describe how to check efficiently for vacuity formulae of this logic. Vacuity detection for a w-ACTL formula φ is reduced to model checking a *witness* formula obtained from φ by substituting `true` or `false` for a subformula of φ . Introducing the concept of *interesting* operand of a binary temporal operators based on the features of w-ACTL, it is possible to show that vacuity checking in w-ACTL can be restricted to the replacement of just the smallest interesting subformula, which is unique.

5.2.2 “Vacuity Detection in Temporal Model Checking” [15]

Authors: O. Kupferman and M. Vardi

Summary: This work presents several alternative definition of vacuity of a temporal property φ written in CTL*. All of them relies on taking a formula, considering all its subformulae, and checking whether the formulae obtained by substituting the subformula with true and false are satisfiable. This shows how vacuity detection can be reduced to model checking of simplified specifications where the subformulae of interest are replaced by constant truth values. The generation of interesting witnesses is dealt with. In this work the complexity of the problem is also taken into account.

5.2.3 “Enhanced Vacuity Detection in Linear Temporal Logic” [23]

Authors: R. Armony, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer and M. Vardi

Summary: This work is based on pure future LTL. The authors deal with the detection of vacuity with respects to subformulae with multiple occurrences with possibly mixed polarity rather than formula occurrences with pure polarity. To this aim, the authors introduce an universal quantification for propositional variables that relate a variable to any subset of the set of states or of the set of points of computational paths, depending on the semantics adopted: *structure semantics*, defined on states, or *trace semantics*, defined on point on a computation paths. The claim of this work is that it is necessary to investigate vacuity focusing on sensitivity to semantic perturbation rather than syntactic perturbation as generally done [1]: rather than changing subformulae syntactically, the authors suggest changing their semantics which means changing the states or the point on paths in which the subformulae hold; thus, instead of replacing true[false] for pure positive[negative] polarity subformulae like in [1], subformulae are replaced by an universally quantified propositional variable. The authors define a hierarchy of vacuity definitions, the traditional *formula vacuity* [1], and *trace* and *structure vacuity*. Trace vacuity is the strongest one implying the other two, and state vacuity on it's turn is stronger than formula vacuity. The right vacuity definition is that based on trace semantics, the other two depending on design changes or on specification language extensions. The authors give an algorithm to check for trace vacuity that is linear in the size of the product of the size of the formula and the complexity of model-checking the formula. As far as the feedback to users is regarded, the authors claim that if just a yes/no answer is needed, then, only the influence of minimal subformulae must be checked, otherwise, the influence of non-minimal formulae can be more useful to build a meaningful witness of vacuity. Moreover, it is claimed that vacuity should be checked both with respect to subformulae and occurrences of subformulae, because the two cases do not always give the same results.

5.2.4 “Vacuum cleaning CTL formulae” [20]

Authors: M. Purandare and F. Somenzi

Summary: Vacuity is related to techniques for assessing the quality of a set of properties, like *coverage metrics*; quality is defined in terms of “how snugly the properties fit the model”, and checking for vacuity means looking for properties that can be changed restricting the set of states in which they hold, without causing them to fail. In this work efficient algorithms for the detection of vacuity are presented and it is shown that vacuity checking CTL formulae can be done with a small overhead, sometimes even in less time than plain model checking. The idea is that to check a formula φ for vacuity, instead of model checking φ and all the formulae derived by substituting `true` or `false` to each subformula of φ sequentially, it is possible to check φ and all its derivatives in a single bottom-up run through the parse tree of φ .

6 Novel techniques

In the following we describe a novel technique for the bounded satisfiability of PLTL. This technique has been also presented in [7].

6.1 Bounded verification of Past LTL

We propose a new encoding of PLTL into propositional logic, based on the use of Separated Normal Form (SNF) for PLTL [9]. The main idea underlying the SNF reduction is the introduction of additional variables (subsequently referred to as ‘SNF variables’) to take into account the truth value of sub-formulae. 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. [10], that shows significant improvements over the original construction presented in [3]. We carried out an experimental evaluation, where the SNF-based approach proposed in this paper is compared with the direct extension of BMC to past from [2]. The results show that the SNF approach results in a much more efficient implementation, yielding encodings that are smaller (in terms of clauses) and that are solved much more easily by the propositional solver.

The next sections are structured as follows. In Section 6.1.1 we introduce the Separated Normal Form for PLTL. In Section 6.1.2 we discuss how to generate efficient encodings for bounded model checking of PLTL. Section 6.1.3 provides an experimental evaluation of our technique.

6.1.1 Separated Normal Form for PLTL

The Separated Normal Form (SNF) [8] is a clause-like normal form for temporal logic, based on the Separation Theorem [12]. 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:

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

where l_i, l_j are literals (i.e. either atomic propositions or negations of atomic propositions), and **start** is an abbreviation for $\mathbf{Z} \perp$. In the following, the rules are referred to as start, invariant, next, and eventuality rules, respectively.

Every PLTL formula can be mapped onto a formula in SNF which is equisatisfiable [9]. With respect to [9], we generalize the form of the rules to permit general propositional formulae in place of $\bigwedge_i l_i$ and $\bigvee_j l_j$. A further slight difference is that we adopt a 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 PLTL formula γ , we define a transformation that manipulates sets of formulae. We start from the singleton set $\{\mathbf{start} \rightarrow \text{NNF}(\gamma)\}$, which 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 in input a set of formulae, and applies some transformation to a member of the set. The

$$\begin{aligned}
\text{SNF}_{[\mathbf{X}]}(\{\varphi \rightarrow \psi(\mathbf{X} f)\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \varphi \rightarrow \psi(\underline{\mathbf{X}} f) \\ \underline{\mathbf{X}} f \rightarrow \underline{\mathbf{X}} f \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{F}]}(\{\varphi \rightarrow \psi(\mathbf{F} f)\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \varphi \rightarrow \psi(\underline{\mathbf{F}} f) \\ \underline{\mathbf{F}} f \rightarrow \underline{\mathbf{F}} f \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{Y}]}(\{\psi(\mathbf{Y} f) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(\underline{\mathbf{Y}} f) \rightarrow \varphi \\ \underline{\mathbf{Y}} f \rightarrow \underline{\mathbf{Y}} f \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{Z}]}(\{\psi(\mathbf{Z} f) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(\underline{\mathbf{Z}} f) \rightarrow \varphi \\ \underline{\mathbf{Z}} f \rightarrow \underline{\mathbf{Z}} f \end{array} \right\} \cup \Gamma \\
\\
\text{SNF}_{[\mathbf{G}]}(\{\varphi \rightarrow \psi(\mathbf{G} f)\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \varphi \rightarrow \psi(f \wedge \underline{\mathbf{X}}(\mathbf{G} f)) \\ \underline{\mathbf{X}}(\mathbf{G} f) \rightarrow \underline{\mathbf{X}}(f \wedge \underline{\mathbf{X}}(\mathbf{G} f)) \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{U}]}(\{\varphi \rightarrow \psi(f \mathbf{U} g)\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \varphi \rightarrow \psi(g \vee (f \wedge \underline{\mathbf{X}}(f \mathbf{U} g))) \\ \underline{\mathbf{X}}(f \mathbf{U} g) \rightarrow \underline{\mathbf{X}}(g \vee (f \wedge \underline{\mathbf{X}}(f \mathbf{U} g))) \\ \varphi \rightarrow \underline{\mathbf{F}} g \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{R}]}(\{\varphi \rightarrow \psi(f \mathbf{R} g)\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \varphi \rightarrow \psi(g \wedge (f \vee \underline{\mathbf{X}}(f \mathbf{R} g))) \\ \underline{\mathbf{X}}(f \mathbf{R} g) \rightarrow \underline{\mathbf{X}}(g \wedge (f \vee \underline{\mathbf{X}}(f \mathbf{R} g))) \end{array} \right\} \cup \Gamma \\
\\
\text{SNF}_{[\mathbf{O}]}(\{\psi(\mathbf{O} f) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(f \vee \underline{\mathbf{Y}}(\mathbf{O} f)) \rightarrow \varphi \\ \underline{\mathbf{Y}}(f \vee \underline{\mathbf{Y}}(\mathbf{O} f)) \rightarrow \underline{\mathbf{Y}}(\mathbf{O} f) \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{H}]}(\{\psi(\mathbf{H} f) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(f \wedge \underline{\mathbf{Z}}(\mathbf{H} f)) \rightarrow \varphi \\ \underline{\mathbf{Z}}(f \wedge \underline{\mathbf{Z}}(\mathbf{H} f)) \rightarrow \underline{\mathbf{Z}}(\mathbf{H} f) \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{S}]}(\{\psi(f \mathbf{S} g) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(g \vee (f \wedge \underline{\mathbf{Z}}(f \mathbf{S} g))) \rightarrow \varphi \\ \underline{\mathbf{Z}}(g \vee (f \wedge \underline{\mathbf{Z}}(f \mathbf{S} g))) \rightarrow \underline{\mathbf{Z}}(f \mathbf{S} g) \end{array} \right\} \cup \Gamma \\
\text{SNF}_{[\mathbf{T}]}(\{\psi(f \mathbf{T} g) \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \psi(g \wedge (f \vee \underline{\mathbf{Z}}(f \mathbf{T} g))) \rightarrow \varphi \\ \underline{\mathbf{Z}}(g \wedge (f \vee \underline{\mathbf{Z}}(f \mathbf{T} g))) \rightarrow \underline{\mathbf{Z}}(f \mathbf{T} g) \end{array} \right\} \cup \Gamma \\
\\
\text{SNF}_{[\mathbf{Y}2\mathbf{X}]}(\{\mathbf{Y} f \rightarrow \varphi\} \cup \Gamma) &\doteq \{f \rightarrow \underline{\mathbf{X}} \varphi\} \cup \Gamma \\
\text{SNF}_{[\mathbf{Z}2\mathbf{X}]}(\{\mathbf{Z} f \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \mathbf{start} \rightarrow \varphi \\ f \rightarrow \underline{\mathbf{X}} \varphi \end{array} \right\} \cup \Gamma
\end{aligned}$$

Figure 2: Part of the transformation function for SNF.

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. \mathbf{X} and \mathbf{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 formulae (e.g. $\underline{\mathbf{XG}}\varphi$).

$$\begin{aligned} \text{SNF}_{[p2p]}(\{\mathbf{start} \rightarrow \varphi_P\} \cup \Gamma) &\doteq \{\text{NNF}(\neg\varphi_P) \rightarrow \neg\mathbf{start}\} \cup \Gamma \\ \text{SNF}_{[f2f]}(\{\psi_{\neg P} \rightarrow \neg\mathbf{start}\} \cup \Gamma) &\doteq \{\mathbf{start} \rightarrow \text{NNF}(\neg\psi_{\neg P})\} \cup \Gamma \end{aligned}$$

$$\begin{aligned} \text{SNF}_{[\mathbf{startY}]}(\{\mathbf{start} \rightarrow \mathbf{Y} \varphi\} \cup \Gamma) &\doteq \{\mathbf{start} \rightarrow \perp\} \cup \Gamma \\ \text{SNF}_{[\mathbf{startZ}]}(\{\mathbf{start} \rightarrow \mathbf{Z} \varphi\} \cup \Gamma) &\doteq \{\mathbf{start} \rightarrow \top\} \cup \Gamma \end{aligned}$$

Figure 3: The transformation functions to deal with combining past and future

The transformations defining $\text{SNF}(\cdot)$ are reported in Figures 2 and 3. We write Γ for the subset of formulae which are not affected by the transformation, φ and ψ for PLTL formulae in NNF, and f and g for propositional formulae. 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 2, $\text{SNF}_{[\mathbf{X}]}$, $\text{SNF}_{[\mathbf{F}]}$, $\text{SNF}_{[\mathbf{Y}]}$ and $\text{SNF}_{[\mathbf{Z}]}$ are used to rename sub-formulae. 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 $\{\mathbf{start} \rightarrow f \wedge \mathbf{XG}f, \mathbf{XG}f \rightarrow \mathbf{X}(f \wedge \mathbf{XG}f)\}$. The intuitive interpretation for the SNF variable $\mathbf{XG}f$ 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 $\{f \vee \mathbf{YO}f \rightarrow g, f \vee \mathbf{YO}f \rightarrow \mathbf{XYO}f\}$. The intuition here is that the SNF variable $\mathbf{YO}f$ 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 above transformations only introduce SNF variables, and \mathbf{F} , \mathbf{X} , \mathbf{Y} and \mathbf{Z} operators; together, $\text{SNF}_{[\mathbf{Y2X}]}$ and $\text{SNF}_{[\mathbf{Z2X}]}$ replace previous operators with next operators, so that the only remaining operators are \mathbf{F} and \mathbf{X} .

The transformations in Figure 2 rely on past operators appearing on the left side of rules and future operators on the right. The transformations $\text{SNF}_{[p2p]}$ and $\text{SNF}_{[f2f]}$, reported Figure 3, are used to move operators onto the appropriate side (we use φ_P to denote a PLTL formula with at least an occurrence of a past temporal operator applied to a purely propositional formula, and $\varphi_{\neg P}$ to denote a formula with no such occurrences). The other transformations in Figure 3 avoid renaming \mathbf{Y} and \mathbf{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 $\text{SNF}_{[\mathbf{F}]}$ is forbidden in cases where the \mathbf{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 $\text{SNF}_{[\mathbf{X}]}$, $\text{SNF}_{[\mathbf{Y}]}$, and $\text{SNF}_{[\mathbf{Z}]}$. Furthermore, transformations $\text{SNF}_{[\mathbf{Y2X}]}$ and $\text{SNF}_{[\mathbf{Z2X}]}$ must not be used while the right hand side is $\neg\mathbf{start}$.

6.1.2 Encoding Bounded Verification of PLTL 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) which is fixed

a priori. We assume that the number of steps, also called the bound, is denoted k and given. While completeness may be lost, the exploitation of the bound often enables the use of alternate search techniques. The idea of Bounded Model Checking [3] 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 $\llbracket M \models_k \varphi \rrbracket$: 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 $\text{PATH}_k \wedge \llbracket \varphi \rrbracket_k$, 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 \mathcal{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 \mathcal{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. 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. 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 same 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 $p \in \mathcal{A}$, the corresponding propositional variables at k and l are assigned the same truth values, i.e. ${}_l L_k \doteq \bigwedge_{p \in \mathcal{A}} (p(l) \leftrightarrow p(k))$.

Encoding the SNF Rules The problem of k -satisfiability for a PLTL formula φ is obtained by encoding each rule in $\text{SNF}(\varphi)$ over the $k + 1$ time instants, depending on the existence of a loop. The encoding is structured as follows:

$$\bigwedge_{i=0}^k \bigwedge_{\rho \in \text{SNF}(\varphi)} \neg \llbracket \rho \rrbracket_k^i \quad \vee \quad \bigvee_{l=0}^{k-1} \left({}_l L_k \wedge \bigwedge_{i=0}^k \bigwedge_{\rho \in \text{SNF}(\varphi)} \llbracket \rho \rrbracket_k^i \right)$$

where ${}_l \llbracket \cdot \rrbracket_k^i$ stands for the encoding operator over a path of k steps, at step i , with loop-back at l . We use $l \in \mathbb{N}$ to denote the loop-back point, while $l = -$ denotes the absence of a loop.

The rules are encoded as follows:

$$\begin{aligned}
{}_l\llbracket \mathbf{start} \rightarrow f \rrbracket_k^i &\doteq \begin{cases} {}_l\llbracket f \rrbracket_k^i & \text{if } i = 0 \\ \top & \text{otherwise} \end{cases} \\
{}_l\llbracket f \rightarrow g \rrbracket_k^i &\doteq {}_l\llbracket f \rrbracket_k^i \rightarrow {}_l\llbracket g \rrbracket_k^i \\
{}_l\llbracket f \rightarrow \mathbf{X}g \rrbracket_k^i &\doteq \begin{cases} {}_l\llbracket f \rrbracket_k^i \rightarrow {}_l\llbracket g \rrbracket_k^{i+1} & \text{if } i < k \\ {}_l\llbracket f \rrbracket_k^i \rightarrow {}_l\llbracket g \rrbracket_k^{l+1} & \text{if } i = k \text{ and } l \in \mathbb{N} \\ {}_l\llbracket f \rrbracket_k^i \rightarrow \perp & \text{if } i = k \text{ and } l = - \end{cases} \\
{}_l\llbracket f \rightarrow \mathbf{F}g \rrbracket_k^i &\doteq \begin{cases} -\llbracket f \rrbracket_k^i \rightarrow -\llbracket g \vee \underline{\mathbf{X}}\mathbf{F}g \rrbracket_k^i \quad \wedge \\ \quad -\llbracket \underline{\mathbf{X}}\mathbf{F}g \rightarrow \mathbf{X}(g \vee \underline{\mathbf{X}}\mathbf{F}g) \rrbracket_k^i & \text{if } l = - \\ -\llbracket f \rrbracket_k^i \rightarrow -\llbracket g \vee \underline{\mathbf{X}}\mathbf{F}g \rrbracket_k^{\min(i,l)} \quad \wedge \\ \quad {}_l\llbracket \underline{\mathbf{X}}\mathbf{F}g \rightarrow \mathbf{X}(g \vee \underline{\mathbf{X}}\mathbf{F}g) \rrbracket_k^i & \text{if } l \in \mathbb{N} \end{cases}
\end{aligned}$$

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, $\underline{\mathbf{X}}\mathbf{F}g$, representing the fact that the eventuality is to be fulfilled at next state. Then, in the case of no loop-back, the expansion basically 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 [10] 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 formulae 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 lack of space we do not address these issues here (see e.g. [22]).

The number of propositional variables in the encoding is $O((|\mathcal{A}| + n) \cdot k)$, 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 \mathbf{U} operator, that requires the application of $\text{SNF}_{[\mathbf{U}]}$, with the encoding for \mathbf{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 \mathbf{U} . The number of rule instances in the above encoding is $O(n \cdot k^2)$, because of the different loop-back points.

Loop Independence Optimization In order to overcome the quadratic dependence on k , we further develop the encoding, arriving at a formulation with a number of rule instances that is $O(n \cdot k)$. We exploit the fact that the encoding for most of the rules can be written to be the same in both the loop and non-loop cases, and we explicitly factor it out. This is obtained by rewriting the rules in a way that is independent of the actual existence and position of a loop-back, and by factoring them out of the big disjunction over the possible loop-back points. The encoding is structured as follows:

$$\bigwedge_{i=0}^{k-1} \bigwedge_{\rho \in \text{SNF}(\varphi)} \text{LI}[\rho]_k^i \wedge \left(\bigwedge_{\rho \in \text{SNF}(\varphi)} \text{LD}[\rho]_k^k \vee \bigvee_{l=0}^{k-1} \left({}_l L_k \wedge \bigwedge_{\rho \in \text{SNF}(\varphi)} \text{LD}[\rho]_k^l \right) \right)$$

where $\text{LI}[\cdot]_k^i$ and $\text{LD}[\cdot]_k^i$ denote the *loop-independent* encoding and the *loop-dependent* encoding operators. The definition of $\text{LI}[\cdot]_k^i$ for the start, invariant and next rules coincides with $-\llbracket \cdot \rrbracket_k^i$. For the eventuality rule $f \rightarrow \mathbf{F}g$, we first notice that the dependence on l in $\min(i, l)$, in the loop case, can be eliminated with a disjunction of the encodings at i and at l . That is, ${}_l \llbracket \mathbf{F}g \rrbracket_k^i$ is replaced by ${}_i \llbracket \mathbf{F}g \rrbracket_k^i \vee {}_l \llbracket \mathbf{F}g \rrbracket_k^l$. The factorization is completed by renaming every occurrence of ${}_l \llbracket \mathbf{F}g \rrbracket_k^l$ with a newly introduced variable $\text{ATL}(\mathbf{F}g)$. The same variable is disjuncted to $-\llbracket \mathbf{F}g \rrbracket_k^i$ in the case without a loop. The encoding thus becomes, regardless of the loop-back point,

$$\text{LI}[\mathbf{f} \rightarrow \mathbf{F}g]_k^i \doteq \begin{cases} -\llbracket \mathbf{f} \rrbracket_k^i \rightarrow (-\llbracket g \vee \mathbf{X}\mathbf{F}g \rrbracket_k^i \vee \text{ATL}(\mathbf{F}g)) & \wedge \\ -\llbracket \mathbf{X}\mathbf{F}g \rrbracket_k^i \rightarrow \mathbf{X}(g \vee \mathbf{X}\mathbf{F}g) \rrbracket_k^i & \end{cases}$$

The encoding of the loop-dependent part for the start, invariant and next rules coincides with the encoding operator defined in previous section. (For the sake of clarity, we do not make explicit the fact that the invariant rules are independent of the loop, and could therefore be factored out; this fact is however exploited in the implementation.) The case of eventuality is encoded as follows.

$$\text{LD}[\mathbf{f} \rightarrow \mathbf{F}g]_k^k \doteq \begin{cases} -\llbracket \mathbf{f} \rrbracket_k^k \rightarrow \neg \text{ATL}(\mathbf{F}g) & \wedge \\ -\llbracket \mathbf{X}\mathbf{F}g \rrbracket_k^k \rightarrow \mathbf{X}(g \vee \mathbf{X}\mathbf{F}g) \rrbracket_k^k & \text{if } l = - \\ (-\llbracket \mathbf{f} \rrbracket_k^k \wedge \text{ATL}(\mathbf{F}g)) \rightarrow -\llbracket g \vee \mathbf{X}\mathbf{F}g \rrbracket_k^l & \wedge \\ {}_l \llbracket \mathbf{X}\mathbf{F}g \rrbracket_k^k \rightarrow \mathbf{X}(g \vee \mathbf{X}\mathbf{F}g) \rrbracket_k^k & \text{if } l \in \mathbb{N} \end{cases}$$

We remark that “ATL” variables are untimed: unlike the variables in φ and from the SNF variables, they are not replicated $k + 1$ times. We achieve independence from the loop since different characterising clauses are activated, depending on the particular value of l .

6.1.3 Experimental Analysis

In this section, we compare the SNF approach with the method for bounded model checking for PLTL proposed in [2], hereafter referred to as the *direct encoding*, that is a generalisation of the encoding for LTL [3]. The direct encoding is defined by recursively descending the structure of the formula being encoded, and distinguishing between the case without a loop

	Counter(16)		Counter(32)		Counter(64)	
	Direct	SNF	Direct	SNF	Direct	SNF
$P(0)$	0.07 8	0.06 8	0.5 16	0.19 16	8.10 32	0.96 32
$P(1)$	8.43 17	0.27 17	680.94 33	1.50 33	T.O. 37	11.20 65
$P(2)$	256.99 17	1.03 26	T.O. 21	7.33 50		68.60 98
$P(3)$	T.O. 13	3.27 35		27.73 67		282.01 131
$P(4)$		8.89 44		81.59 84		966.92 164

Table 2: The results for Counter(N).

and the case with a loop. In the case without a loop, the truth of a PLTL formula only depends on the finite prefix, and the interpretation of past operators always progresses towards the points closer to the origin (i.e., from i to 0). In the case of the loop, the problem is significantly more complicated: in fact, when interpreting a PLTL formula within the loop, the interpretation of going into the past may correspond either to going into the prefix before the loop-back point, or back to the future. The problem is solved by introducing the notion of past temporal horizon of a formula, that is then used as an upper bound to the number of virtual unrolls needed when generating the encoding for the formula. Similar to the pure-future case, the direct encoding does not introduce additional variables, so that witnesses of the form $\alpha \cdot \beta^k \cdot \beta^\omega$ can be reached with $k = |\alpha| \cdot |\beta|$ steps.

Both methods were implemented in NuSMV [6]. For each problem instance, and for each method, we report the total time required by NuSMV (on a Pentium 4, 1.8GHz processor with 1Gb RAM) to build and solve the encodings up to the reported bound, using zChaff [19] as the SAT solver; the reported bound corresponds to the first satisfiable instance, or to the largest unsatisfiable instance solved within the time limit.

We first ran the test from [2] involving past operators, i.e. the Alternating Bit Protocol (from the NuSMV distribution) with a property of the form

$$\mathbf{G}(\text{sender.state} = \text{waitForAck} \rightarrow \mathbf{YH} \text{sender.state} \neq \text{waitForAck})$$

The direct encoding required 87.2 secs. to generate the encoding and solve the problem, while the SNF-based encoding requires only 56.2 secs. Both methods find a counterexample at depth 17.

In order to stress the ability of the two methods to process past operators and to find short counterexamples, we conceived the Counter(N) problem set: a counter starts at 0, progresses up to N , and then loops back at $N/2$. We evaluate a set of parameterized properties, of the form

$$P(i) \doteq \neg \mathbf{F}(\mathbf{O}((c = N/2) \wedge \mathbf{O}((c = N/2 + 1) \dots \wedge \mathbf{O}(c = N/2 + i) \dots)))$$

The value of i is a measure of the nesting of past operators, while the structure of the property requires that the loop (of length $N/2$) must be traversed backwards several times in order to reach a counterexample.

The results are reported in Table 2, where T.O. indicates a runtime exceeding 1800 secs. The direct encoding suffers from the nesting of the property, which influences the past

PropType	Size 1				Size 1.5				Size 2			
	Direct		SNF		Direct		SNF		Direct		SNF	
EXISTS	0.10	1	0.46	1	1.00	1	2.69	1	32.57	1	45.92	1
POSS_1	0.09	2	0.52	2	1.59	2	3.12	2	42.62	2	53.02	2
POSS_2	0.08	2	0.52	2	1.55	2	3.19	2	43.20	2	52.88	2
POSS_3	0.13	3	0.62	3	2.94	3	3.85	3	64.67	3	63.00	3
POSS_4	0.10	2	0.54	2	1.47	2	3.15	2	42.72	2	53.29	2
POSS_5	0.12	3	0.60	3	2.95	3	3.95	3	66.11	3	63.87	3
POSS_6	18.61	20	7.77	20	1.50	2	3.19	2	41.80	2	52.69	2
POSS_7	18.78	20	8.10	20	0.91	20	2.66	20	32.39	20	45.88	20
POSS_8	19.36	20	7.86	20	1.92	2	3.28	2	43.23	2	53.80	2
POSS_9	0.11	2	0.52	2	1.58	2	3.14	2	41.92	2	53.97	2
POSS_10	21.55	20	10.69	20	2.96	3	3.83	3	64.11	3	63.76	3
POSS_11	0.16	3	0.60	3	2.98	3	3.83	3	66.01	3	63.03	3
POSS_12	22.21	20	8.34	20	T.O.	16	559.50	20	T.O.	9	T.O.	13
ASS_1	21.36	20	9.22	20	T.O.	16	851.10	20	T.O.	9	T.O.	12
ASS_2	21.44	20	9.04	20	T.O.	16	217.08	20	T.O.	9	T.O.	18
ASS_3	22.44	20	9.71	20	T.O.	16	192.77	20	42.31	2	52.98	2
ASS_4	21.72	20	10.70	20	1.54	2	3.12	2	44.38	2	56.29	2
ASS_5	20.59	20	8.80	20	T.O.	16	217.87	20	T.O.	9	T.O.	17
ASS_6	17.91	20	7.89	20	T.O.	16	173.54	20	T.O.	9	1730.54	20
ASS_7	17.52	20	7.81	20	T.O.	16	197.76	20	T.O.	9	T.O.	16
ASS_8	21.70	20	8.62	20	T.O.	16	504.25	20	T.O.	9	T.O.	13
ASS_9	21.12	20	10.69	20	T.O.	16	363.21	20	T.O.	9	T.O.	14
ASS_10	21.51	20	9.50	20	T.O.	16	840.48	20	T.O.	9	T.O.	12
ASS_11	20.77	20	11.42	20	T.O.	16	114.16	20	T.O.	9	T.O.	15
ASS_12	21.81	20	10.75	20	T.O.	16	142.81	20	T.O.	9	1779.20	20

Table 3: The results on the examples from [11].

temporal horizon and therefore requires a larger number of virtual unrolls. Most of the time is in fact spent in the generation of the encodings. On the contrary, the encodings are generated efficiently by the SNF-based method, and the time required by the SAT solver is also very limited. SNF-based encodings seem to yield a significant speed up, even if longer paths need to be explored in order to find a counterexample. Notice however that in this problem set the component related to the model is not very significant. Although the ability to construct counterexamples with virtual unroll of the past might be a win, there is clearly a tradeoff between the time that is saved in searching shortened counterexamples compared to the time that is invested in generating more complex encodings.

As a further step, we compared the SNF and the direct encodings on a test set from the domain of requirement engineering for software systems. The starting point is a description of a real-world scenario written in Formal Tropos [11], a language for the description of

early requirements. The test set is obtained by conversion from the Formal Tropos model, parameterized in the number of instances for each class in the model, to a set of (ground) PLTL formulae. The parameterization sets the number of instances with which each class in the description is populated. Different kinds of checks are performed, ranging from feasibility of built-in or domain-specific properties (EXISTS and POSS), for which witnesses are sought, and assertion violations (ASS), for which counterexamples are sought².

The results are reported in Table 3. We tackle problems for three degrees of instantiation: Size 1 corresponds to one object per class; Size 1.5 corresponds to the instantiation of one object for some classes and two objects for the remaining ones; in Size 2, each class is instantiated twice. The first column identifies the problem; three sets of columns follow, one for each size instantiation. T.O. indicates that the run-time exceeded 1800 secs. The maximum bound was set to 20. The instances which reached the maximal bound or timed out are unsatisfiable. The reported bound represents the length of the witness (for POSS and EXISTS), or of the counterexample (for ASS); or, in case of a timeout, the depth of the largest k for which the analysis was completed.

The results show that, on this class of problems, the direct encoding is somewhat superior on easier instances which are satisfiable with a small bound. However, on the harder instances, often requiring the exploration to higher bounds, the gain obtained by means of the SNF encoding with respect to the direct encoding is uniform. For the hardest problem instances, the speed up becomes very significant, sometimes bigger than an order of magnitude. The use of SNF also allows problems to be tackled that were previously out of reach within the time limit; when both methods time out, SNF is uniformly able to cover problem instances with higher length.

7 Connections with other derivables and future works

Many possible synergies exist between property simulation and other topics of the project. In particular, property assurance is strictly connected with property simulation and property synthesis. The technology core of property assurance is a set of techniques to check sets of formulae for satisfiability, and having an engine able to perform this check efficiently can be of great benefit for property simulation tools that provide their users with the possibility of visually inspecting the behaviours of the system under specification. Some results of the property assurance activity (violating or witness traces) could be provided within the graphical interface of the property simulation tool in terms of wave-forms or automata. Moreover, users' interventions on the traces visualized in the property simulation tool could be fed into the property assurance tool to test the modified traces for validity with respect to the set of properties under examination.

Property synthesis deals with methodological and technological issues regarding the automatic generation of code from specifications, and being the result of property assurance phase a consistent specification that satisfies a set of desired properties, the link between the two phases is clear: during the code-generation phase, no resources would be spent on the

²More details on the Formal Tropos problem set can be found at <http://sra.itc.it/tools/t-tool/experiments/cm/>

analysis of the specification if its correctness is given for granted. As far as the refinement-based methodology we propose is regarded, property synthesis would be a natural extension to the last step of methodology to close the gap to circuits that can be put into silicon.

We have proposed the use of Separated Normal Form for the generation of encodings for bounded verification of Linear Temporal Logic with Past. We have shown the effectiveness of the approach by an experimental comparison with the previously available direct method [2], where our SNF-based approach is able to gain up to one order of magnitude.

The SNF transformation appears to bring the benefits of a pure future encoding without the usual exponential blowup associated with past to future transformations; this is believed to be a result of the bounded nature of the encoding, and future work will examine fully the theoretical implications of this. For the experimental work, the similarity between SNF and alternating automata calls for a comparison with this, and other, automata techniques.

Broadening the scope of the work, we expect that the techniques presented will be amenable to SAT-based induction in order to achieve completeness. Similarly, the SNF encoding is particularly suitable for use with incremental SAT solvers. These systems have proved useful for bounded model checking to reduce the amount of work involved in iterating up to a bound; for requirements verification the amount of repeated work currently necessary when testing multiple formulae with respect to a set of requirements will be reduced. Finally, we plan to extend the work to make use of non-Boolean SAT solvers to avoid the Booleanization of the data paths.

The work done so far on novel techniques deals with LTL, including both future and past operators, and does not take into account regular expressions that constitute a relevant part of PSL/Sugar. Consequently, a natural future direction of research is that of investigating techniques to cope with SEREs, both by trying and extending the SNF-based approach, and by exploring the possibility of integrating other techniques via, for example, an SNF encoding of Alternating Automata, which seem to be an interesting choice when it is needed to handle regular expressions-based specifications.

Another future research direction is the one related to vacuity detection, vacuity conditions identification, and meaningful witnesses generation. It seems reasonable to exploit LTL SAT techniques to solve these problems, but further work still needs to be done to understand at which extent these techniques can be used as they are, and, above all, how to integrate vacuity related proof obligations in our framework.

References

- [1] 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.
- [2] M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS'03*, Lecture Notes in Computer Science, Warsaw, Poland, April 2003. Springer-Verlag.

- [3] 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 *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
- [4] R. Bloem, R. Cavada, A. Cimatti, I. Pill, and M. Roveri. Notes of the 23 jan 2004 phone meeting.
- [5] R. Bloem and B. Jobstmann. Property synthesis - draft, 1st march 2004.
- [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 *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.
- [7] A. Cimatti, M. Roveri, and 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.
- [8] M. Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, August 1991.
- [9] M. Fisher and P. Noël. 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.
- [10] A. Frisch, D. Sheridan, and 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.
- [11] A. Fuxman, L. Liu, , M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements in Tropos: Some experimental results. In *Proceedings of the 11th IEEE International Requirements Engineering Conference*, Monterey Bay, California USA, September 2003. ACM-Press.
- [12] 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. Springer-Verlag, 1989.
- [13] P. Gastin and D. Oddoux. Ltl with past and two-way very-weak alternating automata. In Branislav Rován and Peter Vojtás, editors, *Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings*, volume 2747 of *Lecture Notes in Computer Science*, pages 439–448. Springer-Verlag, 2003.

- [14] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 1993.
- [15] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1999.
- [16] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th IEEE Symp. Logic in Computer Science (LICS'2002)*, pages 383–392, Copenhagen, Denmark, July 2002. IEEE Comp. Soc. Press.
- [17] S. Merz and A. Sezgin. Emptiness of linear weak alternating automata. Technical report, LORIA, December 2003.
- [18] A. Morzenti, M. Pradella, P. San Pietro, and P. Spoletini. Model-checking trio specifications in spin. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 542–561. Springer-Verlag, 2003.
- [19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, Las Vegas, June 2001.
- [20] M. Purandare and F. Somenzi. Vacuum cleaning ctl formulae. In E. Brinksma and K. Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.
- [21] P. H. Schmitt and J. Goubault-Larrecq. A tableau system for linear-time temporal logic. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 130–144. Springer-Verlag, 1997.
- [22] D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. Technical Report APES-82-2002, APES Research Group, March 2004. Available from <http://www.dcs.st-and.ac.uk/~apes/apesreports.html>.
- [23] M. Vardi, R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, and A. Tiemeyer. Enhanced vacuity detection in linear temporal logic, 2003.