

Bounded Verification of Past LTL^{*}

Alessandro Cimatti¹, Marco Roveri¹, Daniel Sheridan²

¹ Istituto per la Ricerca Scientifica e Tecnologica (IRST)
Via Sommarive 18, 38050 Povo, Trento, Italy
{cimatti,roveri}@irst.itc.it

² School of Informatics, The University of Edinburgh
Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
d.j.sheridan@sms.ed.ac.uk

Abstract. Temporal logics with past operators are gaining increasing importance in several areas of formal verification for their ability to concisely express useful properties. In this paper we propose a new approach to bounded verification of PLTL, the linear time temporal logic extended with past temporal operators. Our approach is based on the transformation of PLTL into Separated Normal Form, which in turn is amenable for reduction to propositional satisfiability. An experimental evaluation shows that our approach induces encodings which are significantly smaller and more easily solved than previous approaches, in the cases of both model checking and satisfiability problems.

1 Introduction

Temporal logics with past operators are being devoted increasing interest in a number of application areas (e.g. formal verification [12, 5, 10], requirement engineering [13, 17], and automated task planning [2]). In the widely-used setting of Linear Temporal Logics (LTL), past operators do not add expressive power with respect to pure-future: any LTL formula with past operators can be rewritten by only using future-time operators [11]. On the other hand, past operators are very useful in practice, since they help to keep specifications compact, simple, and easy to understand. This practical consideration has a formal counterpart in the fact that LTL with past operators is exponentially more succinct than LTL with pure-future operators [14].

In this paper we tackle the problem of lifting SAT-based verification techniques, which are becoming a prominent technology in many application areas, to deal with past operators. We focus on *bounded* verification for PLTL, where the analysis is limited to behaviors of a fixed number of time steps.

^{*} This work is partially sponsored by the PROSYD EC project, contract number IST-2003-507219, and the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. We thank Paul Jackson, Roberto Sebastiani and Simone Semprini for their useful comments and feedback.

Our interpretation of bounded verification encompasses both Bounded Model Checking and Bounded Satisfiability. *Bounded Model Checking* [4] focuses on design verification: given a model M (typically representing a design) and a formula φ (typically representing a desired property), checking the existence of a counterexample (a behavior of M which violates φ) over k steps is reduced to a purely propositional satisfiability problem, and solved with an efficient SAT solver.

Bounded Satisfiability is more directed to the analysis of requirements, which is gaining a significant practical interest. In fact, we are witnessing the take off of property-based design paradigms (e.g. with the acceptance of the PSL/Sugar language [1] as a IEEE standard language for property specification). This highlights the increased recognition of the importance of properties that are intended to specify the design intent, rather than the design itself. The object of the verification is now a set of requirements, represented as a set of PLTL formulae Γ . Different forms of analysis can be envisaged: for instance, we may be interested in checking whether Γ is *k-satisfiable*, that is, if it admits a model which can be presented within k steps; checking whether a certain formula φ is *k-possible* with respect to Γ , corresponding to $\Gamma \wedge \varphi$ being *k-satisfiable*; and checking whether a certain formula φ is a necessary consequence (an assertion) for Γ , corresponding to $\Gamma \wedge \neg\varphi$ not being *k-satisfiable*. These problems can be easily reduced to checking the (bounded) satisfiability of a generic set of formulae (or, equivalently, of their conjunction). They can also be seen as a bounded model checking problem where the model is completely unspecified; compared to model checking, however, we notice that a model might not even be available at an early stage of the development process. This shift in focus makes the problems significantly different from a pragmatic point of view.

In this paper, we propose a new encoding of PLTL into propositional logic, based on the use of Separated Normal Form (SNF) for PLTL [8]. 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. [9], that shows significant improvements over the original construction presented in [4]. 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 [3]. 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.

This paper is structured as follows. Section 2 covers the syntax and semantics of PLTL. In Section 3 we introduce the Separated Normal Form for PLTL. In Section 4 we discuss how to generate efficient encodings for bounded model

checking of PLTL. Section 5 provides an experimental evaluation of our technique, and we draw some conclusions in Section 6.

2 Linear Temporal Logic with Past Operators

In this paper we consider PLTL, i.e. the Linear Temporal Logic (LTL) augmented with past operators. The starting point is standard LTL, the formulae of which 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 “**Y**esterday”) 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 “**O**nce”) 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 “**H**istorically”) 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 “**S**ince”) 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 “**T**trigger” operator), exactly as in the future case we have $\varphi\mathbf{R}\psi = \neg(\neg\varphi\mathbf{U}\neg\psi)$.

The syntax of PLTL is formally defined as follows:

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

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

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 $(\pi, 0) \models \varphi$.

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

Fig. 1. The semantics of PLTL

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 [14]. 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, **F** and **G**, **U** and **R**, **O** and **H**, and **S** and **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 $\text{NNF}(\varphi)$.

3 Separated Normal Form for PLTL

The Separated Normal Form (SNF) [7] is a clause-like normal form for temporal logic, based on the Separation Theorem [11]. 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 [8]. With respect to [8], 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 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 formulae (e.g. $\mathbf{X}\underline{\mathbf{G}}\varphi$).

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

$$\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 \mathbf{X}(\mathbf{G}f)) \\ \underline{\mathbf{X}}(\mathbf{G}f) \rightarrow \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 \mathbf{X}(f \mathbf{U}g))) \\ \underline{\mathbf{X}}(f \mathbf{U}g) \rightarrow \mathbf{X}(g \vee (f \wedge \underline{\mathbf{X}}(f \mathbf{U}g))) \\ \varphi \rightarrow \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 \mathbf{X}(f \mathbf{R}g))) \\ \underline{\mathbf{X}}(f \mathbf{R}g) \rightarrow \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 \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 \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 \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 \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{Y2X}]}(\{\mathbf{Y}f \rightarrow \varphi\} \cup \Gamma) &\doteq \{f \rightarrow \mathbf{X}\varphi\} \cup \Gamma \\
\text{SNF}_{[\mathbf{Z2X}]}(\{\mathbf{Z}f \rightarrow \varphi\} \cup \Gamma) &\doteq \left\{ \begin{array}{l} \mathbf{start} \rightarrow \varphi \\ f \rightarrow \mathbf{X}\varphi \end{array} \right\} \cup \Gamma
\end{aligned}$$

Fig. 2. Part of the transformation function for SNF.

$\psi(\mathbf{G}f)$ 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 $\{\mathbf{start} \rightarrow f \wedge \underline{\mathbf{X}}\mathbf{G}f, \underline{\mathbf{X}}\mathbf{G}f \rightarrow \mathbf{X}(f \wedge \underline{\mathbf{X}}\mathbf{G}f)\}$. The intuitive interpretation for the SNF variable $\underline{\mathbf{X}}\mathbf{G}f$ is that $\mathbf{G}f$ holds in the next state. Similarly, consider

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

Fig. 3. The transformation functions to deal with combining past and future

the rule $\mathbf{O}(f) \rightarrow g$: the corresponding set of rules is $\{f \vee \mathbf{YO}f \rightarrow g, f \vee \mathbf{YO}f \rightarrow \mathbf{X}\mathbf{YO}f\}$. 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 $\text{SNF}_{[\text{p2p}]}$ and $\text{SNF}_{[\text{f2f}]}$, 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 $\neg\mathbf{start}$.

4 Encoding Bounded Verification of PLTL into SAT

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

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 .

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 [4] 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. ${}_iL_k \doteq \bigwedge_{p \in \mathcal{A}} (p(l) \leftrightarrow p(k))$.

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:

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

where $\llbracket \cdot \rrbracket_k^i$ 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:

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

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, $\mathbf{X}\mathbf{F}g$, 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 [9] 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. [16]).

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 **U** operator, that requires the application of $\text{SNF}_{[\mathbf{U}]}$, 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.

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 $\text{LD}[\cdot]_k^i$. 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, ${}_i \mathbf{F}g]_k^i$ is replaced by ${}_i \mathbf{F}g]_k^i \vee {}_l \mathbf{F}g]_k^l$. The factorization is completed by renaming every occurrence of ${}_l \mathbf{F}g]_k^l$ with a newly introduced variable $\text{ATL}(\mathbf{F}g)$. The same variable is disjuncted to $\text{LD}[\mathbf{F}g]_k^i$ in the case without a loop. The encoding thus becomes, regardless of

the loop-back point,

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

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.

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

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 .

5 Experimental Analysis

In this section, we compare the SNF approach with the method for bounded model checking for PLTL proposed in [3], hereafter referred to as the *direct encoding*, that is a generalisation of the encoding for LTL [4]. The direct encoding is defined by recursively descending the structure of the formula being encoded, and distinguishing between the case without a loop 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 [15] as the SAT solver; the reported bound corresponds to the first satisfiable instance, or to the largest unsatisfiable instance solved within the time limit.

	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 1. The results for Counter(N).

We first ran the test from [3] 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 1, where T.O. indicates a runtime exceeding 1800 secs. The direct encoding suffers from the nesting of the property, which influences the past 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 [10], 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

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 2. The results on the examples from [10].

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 sought³.

The results are reported in Table 2. 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

³ More details on the Formal Tropos problem set can be found at <http://sra.itc.it/tools/t-tool/experiments/cm/>

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.

6 Conclusions and Future Work

In this paper, 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 [3], 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.

References

1. Accellera. *Accelera Property Specification Language: Reference Manual — Version 1.0*.
2. F. Bacchus and F. Kabanza. Control strategies in planning. In *Proceedings of the AAAI Spring Symposium Series on Extending Theories of Action: Formal Theory and Practical Applications*, pages 5–10, Stanford University, CA, USA, March 1995.
3. 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.

4. 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.
5. J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, 2001.
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. M. Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, August 1991.
8. 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.
9. 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.
10. 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.
11. 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.
12. S. Gnesi, D. Latella, and G. Lenzini. Formal verification of cryptographic protocols using history dependent automata. In *Proceedings of the of the 4th Workshop on Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi*, 1999.
13. O. Kupferman, N. Piterman, and M. Vardi. Extended temporal logic revisited. In *Proceedings of the 12th International Conference on Concurrency Theory*, number 2154 in *Lecture Notes in Computer Science*, pages 519–534. Springer Verlag, 2001.
14. 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.
15. 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.
16. 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>.
17. A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 249–263, 2001.