Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 76
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 76
Espoo 2002 HUT-TCS-A76

ON MODEL CHECKING SAFETY PROPERTIES

Timo Latvala

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 76
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 76
Espoo 2002 HUT-TCS-A76

ON MODEL CHECKING SAFETY PROPERTIES

Timo Latvala

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Timo Latvala

ISBN 951-22-6265-7
ISSN 1457-7615

Otamedia Oy
Espoo 2002

ABSTRACT: Safety properties are an interesting subset of general temporal properties for systems. In the linear time paradigm, model checking of safety properties is simpler than the general case, because safety properties can be captured by finite automata. This work discusses the theoretical and some of the practical issues related to model checking LTL properties.

Our first contribution is a theorem relating abstraction for Coloured Petri nets as defined by Lakos [36] and preservation of safety properties. We show that a subset of the safety properties are preserved for this abstraction framework. Our other contribution is an efficient algorithm for translating LTL safety properties to finite automata. Minor contributions include new proofs for some old complexity results regarding LTL and safety properties.

The implementation of the translation algorithm is also experimentally evaluated. Experiments support the feasibility of the approach. In many tests the implementation is quite competitive when compared to algorithms translating full LTL to Büchi automata. The implementation can also check if an LTL formula is pathologic. The check performs well according to experiments.

KEYWORDS: Computer aided verification, model checking, LTL, safety properties, abstraction, Coloured Petri nets

CONTENTS

1 Introduction 1
- 1.1 Contributions and Results . . . 2
- 1.2 Related Work . . . 2
- 1.3 Outline . . . 3

2 Preliminaries 4

3 Automata Theoretic Foundations 6

4 Linear Temporal Logic 9
- 4.1 Syntax and Semantics of LTL . . . 9
- 4.2 Expressiveness and Complexity . . . 11
- 4.3 Safety and Liveness Properties . . . 12
- 4.4 Deciding safety . . . 15

5 Abstraction and Safety Properties 18
- 5.1 Coloured Petri Nets . . . 18
- 5.2 Abstraction and Petri Nets . . . 20
- 5.3 Temporal Logic and Refinement . . . 23
- 5.4 An Example . . . 25

6 Model Checking Safety Properties 28
- 6.1 Detecting Bad Prefixes . . . 29
- 6.2 Informativeness . . . 31
- 6.3 Translation Algorithm . . . 34
- 6.4 Finite Trace Semantics for LTL . . . 40

7 Implementation 42
- 7.1 Translation . . . 42
- 7.2 Checking Pathologic Safety . . . 44

8 Translation Experiments 46
- 8.1 Random Formulae . . . 47
  - Syntactically Safe Formulae . . . 47
  - General Formulae . . . 49
- 8.2 Model Checking Case Studies . . . 50

9 Discussion 53

References 55

1 INTRODUCTION

Developing reliable systems is not an easy task. If the system has *concurrency* it is even harder. When concurrency is introduced in a system, phenomena which are not present in sequential systems manifest themselves. The inherent *non-determinism* of concurrent systems can give rise to subtle errors which are very hard to understand and can be difficult to reproduce.

Concurrency is a devious source of complexity. Even a simple system can exhibit complex behaviour when concurrency is allowed. This is also obvious from many examples in concurrency theory. Determining if a finite automaton accepts any string or a given string can be decided with simple linear-time algorithms. In the concurrent case, i.e. deciding if the intersection of k finite automata accept any string or a given string, the best known algorithms decide the problem in exponential- and linear-time respectively.

One of the ways introduced to aid designers in designing correct concurrent systems is *model checking* [8, 51]. Introduced roughly 20 years ago, model checking has already revolutionised the way hardware systems are designed, and can be considered industry practice today [23].

The basic idea of model checking is simple. Both the system and the properties the system should have, are expressed as mathematical models. Special algorithms allow comparison of the system against the properties and if the system model violates a property, a violating execution can be displayed. If the model and the properties have been specified correctly, no error will go unnoticed. In the ideal case, all of these stages are automatic and very little human intervention is required.

Unsurprisingly, model checking has its limitations. The perhaps most acute problem is how to enable model checking to cope with the ever increasing size and complexity of systems. For some classes of systems, the methods scale quite well. This has made possible the success of model checking in hardware systems. Finding methods which scale for systems which are asynchronous and data intensive seems to be more challenging. Consequently, concurrent software systems are still debugged mostly using traditional methods. The problems related to scaling in model checking are referred to as the *state explosion* problem [60].

In this work we focus on efficient model checking of *safety properties*, using the automata theoretical approach [62, 34, 63]. Safety properties describe properties of the system which have finite counterexamples or, more informally, properties requiring that “nothing bad happens”. A typical safety property requires e.g. that the value of x always is greater than three. Many common properties such as invariants are safety properties which makes safety properties very interesting.

In the automata theoretic approach to model checking both the system and the property to be verified are described as automata. The property holds if all of the executions of the system automaton also are executions of the property automaton. Usually the property is not given as an automaton but in some temporal logic such as linear temporal logic (LTL). There are also other logics such as CTL which can be used for specification. In this work we will mostly restrict ourselves to properties expressed using LTL.

This work discusses the theoretical and some of the practical issues of

model checking LTL safety properties. The most relevant complexity result are presented and analysed. Coloured Petri Nets are one of the formalisms used to describe concurrent systems. We investigate which properties, especially safety properties, are preserved when abstractions defined in [36] are used. Most of this work is dedicated to investigating how to efficiently compile an LTL formula into an automaton, when the given formula describes a safety property. Efficient compilation of the formulas facilitates the verification of larger and more complex systems.

1.1 Contributions and Results

We develop an efficient translation of safety LTL formulae to finite automata, based on the algorithm presented by Kupferman and Vardi [32]. The algorithm has been implemented and extensive experiments have been performed. Our results show that the algorithm scales better than algorithms for translating general LTL formulae to automata. Currently, the implementation is not the fastest of the available translators. However, the experiments indicate that using finite automata for safety model checking results in a real difference in performance for practical models, especially when the property does not hold. The implementation also includes the first implementation to our knowledge of an algorithm for deciding if a formula is a pathologic safety formula.

The work also has some strictly theoretical contributions. Minor contributions include new proofs for some of the complexity results related to safety model checking. A more significant contribution is that we show that the abstraction/refinement framework introduced by Lakos can be used to aid the abstraction when model checking safety properties. We prove that the abstractions in the framework preserve a subset of the safety properties in LTL. We also extend the result to some branching time properties. The feasibility of the approach is argued with a small example.

1.2 Related Work

Model checking of safety properties has been investigated by number of authors. Alpern and Schneider [2] were the first to give a formal definitions of safety and liveness. The work of Sistla [53] on characterising safety of LTL formulas syntactically continues this work and adds to it significantly. Most of the automata theoretic insight into safety and liveness comes from Kupferman's and Vardi's [32] paper. Many important notions are defined there for the first time, among them the notions of informativeness for prefixes and classification of LTL formulae into intentionally, accidentally and pathologically safe. The paper also introduces a translation from LTL formulae to finite automata, which is the basis for the algorithm in this work. Many complexity results are also due to them. Geilen [21] also considers translating LTL into finite automata. His approach reformulates some of the results of Kupferman and Vardi using their notion of informativeness. The focus of the paper is on presenting a tableau algorithm for run-time monitors of LTL properties. Havelund and Rosu [25] also focus on monitoring executions of systems. They present a dynamic programming algorithm which checks

sequences against properties specified in a linear temporal logic with past operators. An algorithm for model checking past temporal logic specifications is also presented in [3].

Lakos [36] has defined and introduced most of the concepts related to abstraction and refinement used in this work. He also proved that the refinements used are in some sense behaviour respecting. Lewis [41] continued Lakos work and investigated refinement especially in the context of incremental development. Lewis also proves that given certain conditions, the refined net is weakly bisimilar to the original net. The approach of Padberg et al. [49] is close to the results presented in this paper. They show how a rule-based approach for morphisms can be used to stepwise refine nets while preserving invariants.

1.3 Outline

We begin in Section 2 by introducing Kripke structures, the system model used in this work, and by defining some fundamental concepts. Section 3 gives the automata theoretic foundations, while setting the stage for the automata theoretic approach to model checking employed in this work. Section 4 defines LTL and presents the relevant complexity theoretical results and clarifies the connection between LTL and automata on infinite words. The important concepts of safety and liveness are also defined and discussed in this section. In Section 5 abstraction for Coloured Petri Nets w.r.t. model checking safety properties is investigated. The translation algorithm from LTL to finite automata is given in Section 5. Section 6 discusses implementation issues while Section 7 focuses on experimentally evaluating the performance of the algorithm. Section 8 discusses the results and speculates on possible future work.

Formal languages. A very important concept in this work is the concept of languages. Let Σ be a finite set called the alphabet. A finite *word* of length n over Σ is a mapping $w : \{1, 2, \ldots, n\} \rightarrow \Sigma$. Words are also in many cases presented as *strings* $w = \sigma_0\sigma_1 \ldots \sigma_n$, where $\sigma_i \in \Sigma$. A *language* of finite words over Σ is a set L of finite words. We can also talk about *infinite words*. They are mappings $w : \mathbb{N} \rightarrow \Sigma$. Languages are defined as in the finite word case.

Regular expressions. One way we will define languages in this work is using regular expressions. We define the syntax of regular expressions w.r.t. an alphabet.

- Every letter from the alphabet is a regular expression.
- If α and β are regular expressions, then so are ϵ, $(\alpha \cup \beta)$, $(\alpha\beta)$ and α^*

Every regular expression defines a language. The letter $\sigma \in \Sigma$ defines the one-word language $\{\sigma\}$. By ϵ we denote the empty string and $(\alpha \cup \beta)$ is the union of the languages of α and β. With $(\alpha\beta)$ we denote the concatenation of the languages of α and β. In some cases the shorthand $\alpha^i = \alpha\alpha \cdots \alpha$, i.e. α i times, is used. The Kleene star, α^* is defined through the union:

$$\alpha^* = \epsilon \cup \bigcup_{i \in \mathbb{N}^+} \alpha^i$$

In many cases our alphabet will be 2^Σ. In this case we will use boolean terms over Σ to define sets of letters. If $\Sigma = \{a, b\}$, then $a \vee b$ denotes $\{\{a\}, \{b\}, \{a, b\}\}$ while $\neg a$ denotes $\{\emptyset, \{b\}\}$. The expression $\top$ can be seen as a shorthand for 2^Σ.

Formal models. All formal reasoning requires a formal model of the system under inspection. In this work we will consider the common model where time is discrete and no concept of duration exists. This means that the ordering between events is relevant, but time between events is not. At each point in time, the system can be described by its *state*. The behaviour of the system is the possible sequences of states of the system. All behaviours of the system are considered infinite. For the class of systems we are especially focusing on, reactive systems, this assumption is easy to justify. Reactive systems continuously react to inputs from the environment and they have no natural terminating state. Abstractly, their behaviour can be seen as infinite. It is of course possible that the system, e.g. due to a programming error, enters a state from which it cannot proceed. This can, however, be simulated by having the system loop in the same state.

The notions above can be formalised using the Kripke structure model. The model is very simple and abstract, but it will be sufficient for our purposes most of the time. Later we will also introduce higher-level formalisms which are closer to programming languages. These mainly function as *generators* of Kripke structures.

Definition 1 *A Kripke structure is a tuple $M = \langle S, \delta, s_0, \pi \rangle$, where*

- *S is a set of states,*

- *$\delta \subseteq S \times S$ is the transition relation obeying the condition that $\forall s \in S : \exists s' \in S : (s, s') \in \delta$,*
- *s_0 is the initial state of the system, and*
- *$\pi : S \rightarrow 2^{AP}$ is a labelling function which assigns a set of atomic propositions to each state.*

An execution of a Kripke structure M is an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$, where s_0 is the initial state of M and $(s_i, s_{i+1}) \in \delta$.

The set of states S can be either finite or infinite. Most definitions in this work are oblivious to this, however, a few of the algorithms require finiteness for termination.

We can also define the *language* of a Kripke structure. An execution σ can be projected onto the alphabet 2^{AP} by using the labelling function π. This projected sequence can be considered a word in $(2^{AP})^\omega$. The set of executions of the Kripke structure generates a set of infinite words, the language of the Kripke structure, denoted $\mathcal{L}(M)$.

The relation between executions and infinite words will allow us to use automata theory to specify behaviours of systems. This is one of the fundamental ideas which underlies the automata theoretic approach to model checking.

3 AUTOMATA THEORETIC FOUNDATIONS

Finite automata on finite an infinite words are essential constructs for the automata theoretic approach to verification. This section introduces *alternating automata* and *non-deterministic* automata.

Just as finite automata on finite words are equivalent to regular languages finite automata on infinite words are equivalent to *omega-regular languages*(c.f. [59]). Omega-regular languages are like the normal regular languages but an additional operator, ω, is allowed for omega-regular expressions. The expression $(a \cup b)(ba)^\omega$ characterises all strings which start with a or b and are followed by infinitely many ba:s. In the following we consider words defined over an alphabet Σ.

Let X be a finite set and $\mathcal{B}^+(X)$ the set of all positive Boolean formulas over X including the abbreviations **true** and **false**. A set $Y \subseteq X$ satisfies a formula $\theta \in \mathcal{B}^+(X)$ iff θ is satisfied by setting all the elements in Y to true and all elements in $X \setminus Y$ to false.

For the familiar non-deterministic automaton, if Q is a set of states, a transition relation δ can be defined as $\delta \subseteq Q \times \Sigma \times Q$. A transition $\delta(q, \sigma) = \{q_1, q_2, q_3\}$ maps a state and a letter $\sigma \in \Sigma$ to a set of states. The non-deterministic nature of the automaton allows it to move to several states in one transition. Alternating automata generalise this by allowing the automaton use a bounded number of copies of itself which work non-deterministically. Formally, transitions are mapped to arbitrary positive formulas in $\mathcal{B}^+(Q)$. As an example, if we have the transition $\delta(q, \sigma) = q_1 \wedge (q_2 \vee q_3)$, the automaton moves to the states q_1 and non-deterministically to q_2 or q_3. Let $w = \sigma_0\sigma_1 \ldots$ be a word and let $w^i = \sigma_i\sigma_{i+1} \ldots$ denote the suffix of w starting from the i:th position. The automaton above accepts a suffix w^l from q if it accepts w^{l+1} both from q_1 and from either q_2 or q_3. In this framework the non-deterministic transition above is expressed as $\delta(q, \sigma) = q_1 \vee q_2 \vee q_3$ Non-deterministic automata are thus automata where only the or-connective is allowed in the transition relation. Non-determinism captures existential choice with the perfect guessing capability of the automata. Non-determinism can easily capture existential style questions such as "accept any word which has the property p". Alternating automata can succinctly express both universal and existential choice.

When a word $w = \sigma_0\sigma_1 \ldots$ is read by an automaton it induces *runs* of the automaton. For a non-deterministic automaton a run can be seen as a function $r : \mathbb{N} \to Q$, where $r(0)$ is an initial state and for every $i \geq 0$, $r(i+1)$ is in $\delta(r(i), \sigma_i)$. Each position is mapped to a state of the automaton and the run must respect the transition relation of the automaton. Due to non-determinism, one word induces several runs.

Runs for an alternating automaton are not so simple. An alternating automaton can be seen as making copies of it self, when "and" appears in a transition. A run of an alternating automaton is thus better viewed as a labelled tree rather than as a path as for non-deterministic automata. A *tree* is a non-empty set $T \subseteq \mathbb{N}^*$, where for every $x \cdot c \in T$ with $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$ we have $x \in T$. The elements of T are called *nodes* and the empty word ϵ is the *root* of T. For $x \cdot c \in T$, $x \in T$ is the unique *parent* of $x \cdot c$, and respectively all $x \cdot c \in T$ are the *children* x. A node without children is called a *leaf*. The

level of a node is its distance from the root ϵ. A *path* $\pi = x_0 x_1 \ldots$ of a tree is a maximal sequence of nodes such that x_0 is the root ϵ and x_i is the parent of x_{i+1} for all $i \geq 0$. A Σ-*labelled tree* is a pair $\langle T, V \rangle$, where T is a tree and $V : T \rightarrow \Sigma$ maps each node of T to a letter in Σ.

Definition 2 *An alternating automaton is tuple* $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$ *where*

- Σ *is the input alphabet,*
- Q *is a finite set of states,*
- $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$ *is a transition function,*
- $Q_0 \subseteq Q$ *is a set of initial states and,*
- $F \subseteq Q$ *is a set of final states.*

A run of $\mathcal{A}$ *over an infinite word* $w = \sigma_0 \sigma_1 \ldots$ *is a* Q*-labelled tree* $\langle T_r, r \rangle$*, where* $T \subseteq \mathbb{N}^*$ *and* $r(\epsilon) \in Q_0$*. For every node* $x \in T_r$ *with* $\delta(r(x), \sigma_{|x|+1}) = \theta$ *there is a possibly empty set* $\{r(x \cdot c) \mid x \cdot c \in T_r\}$ *which satisfies* θ*. With the Büchi accepting condition,* $\mathcal{A}$ *accepts a run* $\langle T_r, r \rangle$ *if all infinite paths* $\pi \subseteq T_r$ *visit at least one state in* F *infinitely often. A word is accepted if there exists an accepting run for it.*

If $\delta(r(x), \sigma_i) = \mathbf{true}$, then x does not need to have any children. Thus all branches of the tree need not be infinite in the run. On the other hand **false** must not appear in a run, since **false** is not satisfiable.

The special cases of *non-deterministic* and *deterministic* are easy to define. An automaton $\mathcal{A}$ is non-deterministic iff $\delta(q, \sigma)$ uses only disjunctions. $\mathcal{A}$ is deterministic iff $\delta(q, \sigma) \in (Q \cup \mathbf{false})$ and $|Q_0| = 1$. The Büchi accepting condition is the obvious: a run $r : \mathbb{N} \rightarrow Q$ is accepted if at least one state in F is visited infinitely often in the run and a word is accepted if has an accepting run.

Alternating automata can also accept finite words. A run on a finite word $w = \sigma_0 \sigma_1 \ldots \sigma_n$ is a finite Q-labelled tree $\langle T_r, r \rangle$ with $T \subseteq \mathbb{N}^{\leq n}$, where $\mathbb{N}^{\leq n}$ is set of $\mathbb{N}$-words not longer than n. Otherwise, a run is defined in the same way as in the infinite word case. A run is accepted iff for all nodes x of level n we have that $r(x) \in F$.

The set of words an automaton $\mathcal{A}$ accepts is denoted $\mathcal{L}(\mathcal{A})$ and is called the language of $\mathcal{A}$. If $\mathcal{L}(\mathcal{A}) = \emptyset$ the automaton is called *empty*.

Alternation does not increase the expressive power of finite automata. Alternating automata on finite words define a regular language and alternating automata on infinite words an omega-regular language (c.f. [63]). However, both in the finite and the infinite word case, alternating automata can be exponentially more succinct than non-deterministic automata. The translation of an alternating automaton to a finite automaton constructs an non-deterministic automaton which is exponentially larger. In the general case, the blow-up is unavoidable. The intuitive idea behind the translation is that the finite automaton guesses a run tree of the alternating automaton. At a given point of a run, the finite automaton keeps a whole level in memory. When it reads the next symbol it guesses the next level.

All finite automata are closed under union, intersection and complementation. While complementing non-deterministic automata involves an exponential penalty, alternating automata on infinite words can be complemented with only a quadratic blow up [31] and alternating automata on finite words complemented in linear time (c.f. [63]).

In many applications it is important to determine if the automaton is *empty*. For a non-deterministic automaton on finite words, determining if the automaton is empty can be done in linear time simply by checking if any final state is reachable from an initial state using the normal graph traversal algorithms. The problem can be shown to be NLOGSPACE-complete using the reachability method (c.f. [63]).

An automaton on infinite words is non-empty if there exists a path from an initial state to a final state, and the final state can be reached from itself. Despite the algorithmically more challenging task, the linear time bound can be maintained in the following way. The strongly connected components (SCC) of the automaton can be computed in linear time [56]. If a non-trivial SCC contains a final state the automaton is non-empty. Using the reachability method this problem can also be shown to be NLOGSPACE-complete.

Unsurprisingly emptiness checking for alternating automata is much more challenging. For both the finite word and infinite word case, it is in fact PSPACE-complete.

Proposition 3 *([6]) The non-emptiness problem for alternating automata is PSPACE-complete*

Proof:
An alternating automaton can be translated into a non-deterministic automaton with an exponential blow-up [6]. Non-deterministic automata can be tested for emptiness in logarithmic space and thus if we do the translation and the emptiness checking on-the-fly, we get a polynomial space algorithm.

To prove PSPACE-hardness of the emptiness problem we can reduce, as we later shall see, the validity problem for LTL to the emptiness problem(c.f. [63]). □

4 LINEAR TEMPORAL LOGIC

Temporal logic [50] is a popular way of specifying properties of reactive systems. There are two basic variants of temporal logic, linear and branching [37]. In linear temporal logic (LTL), introduced to the verification setting by Pnueli [50], any given point in time has only one future, while branching time logics [37] allows several possible futures. The perhaps most known branching time logic is computation tree logic (CTL), introduced in [15].

There has been a two decade long debate, albeit currently not so intense, among researchers in the concurrency community which paradigm, the branching or the linear, is superior in reasoning about concurrency. To the author's knowledge, the most recent contribution to this debate is [64].

In this work, we almost exclusively focus on the linear paradigm. The primary reason is that current research indicates [32] that the concept of safety does not seem to be as fruitful in the branching time paradigm.

LTL allows properties of systems be specified easily, especially compared to e.g. first order logic. The great innovation of Pnueli [50] was that this modal logic was suitable for this task. Common properties like invariants, fairness and causal relationships can be concisely expressed without the horde of quantifiers that first order logic would require.

LTL also enjoys a complexity advantage compared to full first order logic. It is expressive enough in most cases. In contrast, solving the first order logic model checking problem is non-elementary (c.f. [9]).

4.1 Syntax and Semantics of LTL

The syntax of LTL consists of atomic propositions, the normal boolean connectives, and *temporal operators*. Let AP be a set of atomic propositions. Well-formed formulae of LTL are constructed in the following way:

- **true**, **false** and every $p \in AP$ are well-formed formulae
- If ψ and φ are well-formed formulae, then so are $\psi \wedge \varphi$, $\psi \vee \varphi$, $\psi \, U \, \varphi$, $\psi \, V \, \varphi$, $\neg\varphi$ and $X\varphi$.

LTL is interpreted over infinite sequences of atomic propositions, i.e. infinite words in $(2^{AP})^\omega$. A model (or word) $\pi = \sigma_0\sigma_1\sigma_2\ldots$, where $\sigma_i \subseteq AP$, is a mapping $\pi : \mathbb{N} \rightarrow 2^{AP}$. By π^i we denote the suffix $\pi^i = \sigma_i\sigma_{i+1}\sigma_{i+2}\ldots$ and π_i denotes the prefix $\pi_i = \sigma_0\sigma_1\ldots\sigma_i$. For an LTL formula ψ and a model π, we write $\pi^i \models \psi$, "the suffix π^i is a model of ψ". The semantics of the models relation $\models$ is defined inductively in the following way.

- For all π^i we have that $\pi^i \models \mathbf{true}$ and $\pi^i \not\models \mathbf{false}$.
- For atomic propositions $p \in AP$, $\pi^i \models p$ iff $p \in \sigma_i$
- $\pi^i \models \psi_1 \vee \psi_2$ iff $\pi^i \models \psi_1$ or $\pi^i \models \psi_2$.
- $\pi^i \models \psi_1 \wedge \psi_2$ iff $\pi^i \models \psi_1$ and $\pi^i \models \psi_2$.
- $\pi^i \models X\psi$ iff $\pi^{i+1} \models \psi$.
- $\pi^i \models \neg\psi$ iff $\pi^i \not\models \psi$.

- $\pi^i \models \psi_1 \, U \, \psi_2$ iff there exists $k \geq i$ such that $\pi^k \models \psi_2$ and for all $i \leq j < k$ $\pi^j \models \psi_1$.
- $\pi^i \models \psi_1 \, V \, \psi_2$ iff for all $k \geq i$, if $\pi^k \not\models \psi_2$, then there is $i \leq j < k$ such that $\pi^j \models \psi_1$.

Usually we do not write $\pi^0 \models \psi$ but simply $\pi \models \psi$. Other commonly used abbreviations are $\mathbf{F}\psi = \mathbf{true} \, U \, \psi$, $\mathbf{G}\psi = \mathbf{false} \, V \, \psi$, and the normal abbreviations for the boolean connectives $\Rightarrow, \Leftrightarrow$. Of interest is also the *unless*-operator W which is defined by the equivalence $\psi_1 \, W \, \psi_2 \equiv \psi_1 \, U \, \psi_2 \vee \mathbf{G}\psi_1$. A sufficient set of operators which can express all LTL-properties is $\vee, U, X, \neg$. Note also the duality between until and release, $\neg(\psi_1 \, U \, \psi_1) \equiv \neg\psi_1 \, V \, \neg\psi_2$.

The operator X is the so called next-operator which requires that a formula is true in the next position of the execution. The binary operator U is called the until-operator. $\psi_1 \, U \, \psi_2$ means that eventually ψ_2 will be true, and until then ψ_1 is true. This version of the until-operator is called *reflexive* because the operator is satisfied if ψ_2 is true immediately. The dual of until, V, is called the release-operator. The formula $\psi_1 \, V \, \psi_2$ requires that ψ_2 is true if ψ_1 has not been true at an earlier point of time. In this case ψ_1 and ψ_2 must be simultaneously true at some point. Note that ψ_1 is not required to eventually become true. The operator $\mathbf{G}$ has the meaning "globally" or "henceforth". It requires that a formula is true in all positions from the current onward. The dual of $\mathbf{G}$ is $\mathbf{F}$, called "finally" or "eventually". The meaning of $\mathbf{F}\psi$ is that ψ must be true at the current point or at some point in the future. The unless-operator, W, also known as the weak until operator, says that the first argument holds at least up until the second argument. The second argument is never required to hold though.

An LTL formula ψ specifies a language $\mathcal{L}(\psi) = \{\pi \in (2^{AP})^\omega | \; \pi \models \psi\}$. The connection between the executions of a Kripke structure and the models of an LTL formula is now clear. The executions generate words over 2^{AP}, which can also be interpreted as models of an LTL formula. Thus, given a Kripke structure M and an LTL formula ψ, we write $M \models \psi$ iff the projection to the atomic propositions of the LTL formula of each execution of the Kripke structure M is a model of ψ. Sometimes this is referred to as the universal model checking problem. The dual of the universal model checking problem is the existential model checking problem where we ask if any execution of the Kripke structure satisfies the given formula.

Example 4 *Writing simple properties in LTL is fairly straightforward. Specifying an invariant is easy. Let p be the atomic proposition having the meaning that the variable x is greater than zero. Claiming that this is an invariant is easy:*

$$\mathbf{G}p$$

Requiring that x will always return to state where it is greater than zero is not much more difficult:

$$\mathbf{GF}p$$

Causal relationships are also easily expressed. If p is an atomic proposition meaning that "A goes up" and q means "A comes down" formalising "if A

goes up it must eventually come down" gives:

$$\mathbf{G}(p \Rightarrow \mathbf{F}q)$$

Formalising using temporal logic is not always easy. Especially more complex properties require care. "If p before q then also eventually r" becomes:

$$p\,V\,\neg q \Rightarrow \mathbf{F}r$$

4.2 Expressiveness and Complexity

LTL is fairly expressive as can be seen from the examples above. It characterises a well-defined subset of the *omega-regular languages*. The precise subset LTL formulae characterise is the star-free omega-regular languages [58], the omega-regular languages which are formed without using the Kleene star. This means that, despite LTL being fairly expressive, it cannot express fairly simple properties such as "in every second state p is true". LTL has been extended in several ways in order to achieve full omega-regularity. See e.g. [64] for a survey.

Many of the complexity measures in this section will use the size of the formula as a parameter. The size of a formula is defined through the set of subformulas of a formula ψ. It is also called the closure of ψ and is denoted $cl(\psi)$. The size of a formula is defined as the cardinality of the closure.

As LTL characterises a subset of the omega-regular languages, LTL can be translated to an automaton on infinite words. The translation to alternating automata is easy to present. We assume that the formula ψ is in positive normal form (PNF), i.e. negations only appear before propositions. Any LTL formula can without a significant blow-up be rewritten in PNF using the duality between U and V.

The translation straight-forwardly follows the semantics of LTL. Using the expressiveness of alternation it is easy to translate the boolean operators $\vee$ and $\wedge$. Also the next state operator X is easily translated. The translation of the binary temporal operators is based on two equalities:

- $\psi_1\,U\,\psi_2 \equiv \psi_2 \vee (\psi_1 \wedge X(\psi_1\,U\,\psi_2))$
- $\psi_1\,V\,\psi_2 \equiv \psi_2 \wedge (\psi_1 \vee X(\psi_1\,V\,\psi_2))$

With these 'recursive' definitions of until and release which relate the truth of an until or a release formula to its truth in the next state, it is easy use the transition relation to define their semantics. The automaton has $|cl(\psi)|$ states.

Proposition 5 *[4, 63] Given an LTL formula ψ, we can construct an alternating Büchi automaton $\mathcal{A}_\psi = \langle 2^{AP}, cl(\psi), \delta, \{\psi\}, F\rangle$, such that $\mathcal{L}(A_\psi) = \mathcal{L}(\psi)$.*

Proof:
The alphabet is 2^{AP}, the set of states is the set of sub-formulas of ψ and the set of accepting states contains all the states (formulas) of the form $\varphi_1\,V\,\varphi_2$. The transition function δ is defined in the following way for each $\sigma \in 2^{AP}$:

- $\delta(\mathbf{true}, \sigma) = \mathbf{true}$
- $\delta(\mathbf{false}, \sigma) = \mathbf{false}$
- $\delta(p, \sigma) = \mathbf{true}$ if $p \in \sigma$
- $\delta(p, \sigma) = \mathbf{false}$ if $p \notin \sigma$
- $\delta(\neg p, \sigma) = \mathbf{true}$ if $p \notin \sigma$
- $\delta(\neg p, \sigma) = \mathbf{false}$ if $p \in \sigma$
- $\delta(\psi_1 \vee \psi_2, \sigma) = \delta(\psi_1, \sigma) \vee \delta(\psi_2, \sigma)$
- $\delta(\psi_1 \wedge \psi_2, \sigma) = \delta(\psi_1, \sigma) \wedge \delta(\psi_2, \sigma)$
- $\delta(X\varphi, \sigma) = \delta(\varphi, \sigma)$
- $\delta(\psi_1 \, U \, \psi_2, \sigma) = \delta(\psi_2, \sigma) \vee (\delta(\psi_1, \sigma) \wedge \psi_1 \, U \, \psi_2)$
- $\delta(\psi_1 \, V \, \psi_2, \sigma) = \delta(\psi_2, \sigma) \wedge (\delta(\psi_1, \sigma) \vee \psi_1 \, V \, \psi_2)$

□

Many translations have been presented which translate an LTL formula to a Büchi automaton. See e.g. [22, 11, 12, 18, 55, 20]. However, quoting Vardi [63] "this presentation has the advantage that it separates the logics from the combinatorics". The logics is handled by the translation from LTL to alternating automata, while the combinatorics is dealt with in the translation from an alternating Büchi automaton to a non-deterministic Büchi automaton.

Expressiveness usually implies a heavy baggage of complexity. This is also the case for LTL. The classical problems of *satisfiability* and *validity*, i.e. determining for a given LTL formula φ if there exists a model π such that $\pi \models \varphi$ and determining if all models satisfy φ respectively, are PSPACE-complete in the size of the formula [54]. Polynomial space algorithms for both problems are available through the translation to an alternating automaton and a reduction to the emptiness problem. If the automaton is empty the corresponding formula is not satisfiable. The validity problem can of course be reduced to the satisfiability problem by negating the formula under inspection.

Deciding the *model checking problem*, does $M \models \varphi$, given a Kripke structure M and a LTL formula φ, can also be answered using automata theoretic techniques. First, the negation of the formula is translated into an automaton. Then, the intersection of the system and the formula automaton is taken. If the intersection automaton is empty, the formula holds. This procedure takes time $|M| \times 2^{O(|\varphi|)}$. We will cover this more in depth later.

4.3 Safety and Liveness Properties

Temporal properties can be classified into different categories. One possible classification is splitting the properties into safety and liveness properties. Lamport [37] informally described safety properties as properties which require that "something bad never happens" and liveness properties as properties which require that "something good eventually happens". A typical safety property is an invariant, which e.g. requires that the value of a variable is positive. An example of a liveness property is that the system will always eventually return to its initial state.

Divding properties into safety and liveness properties has proven its use in many cases. For many logics and proof systems concerned with properties of concurrent systems, a method for proving safety properties has been developed first which has been followed by a more general method which also can handle liveness properties.

The formal definition of safety and liveness properties uses the language analogy. We first define what a safety language is and then simply define that a safety formula is a formula which defines a safety language. The terms language and property can be considered synonyms in this context.

We again consider languages $L \subseteq \Sigma^\omega$ of infinite words over some alphabet Σ. A finite word $x \in \Sigma^*$ is called a *bad prefix* for L if for any $y \in \Sigma^\omega$ $x \cdot y \notin L$. The finite word x cannot be extended in any way to a word of L.

Definition 6 *A language L is called a safety language iff every $w \in \Sigma^\omega \setminus L$ has a finite bad prefix.*

In other words, if we can identify the bad prefixes for a safety language it is enough that we observe a finite part of a word to determine that it is not part of the language.

Let $\bar{L}$ denote the complement of the language, i.e. $\bar{L} = \Sigma^\omega \setminus L$. If $\bar{L}$ is a safety language, we call L a *co-safety language*. Every $w \in L$ has a *good prefix* $x \in \Sigma^*$ such that $x \cdot y \in L$ for $y \in \Sigma^\omega$ iff L is a co-safety language.

There are also other ways of defining safety languages. The definition of [2] defines safety languages in the following way. A language $L \subseteq \Sigma^\omega$ is a safety language iff for all $w \in \Sigma^\omega$: if $\forall i \geq 0, \exists u \in \Sigma^\omega$ such that $w_i u \in L$ then $w \in L$. In other words, given a safety language L, any word w for which all prefixes w_i can be extended to a word in L must also be in L. Our definition focuses on the fact that each word not in L must have a finite bad prefix. The equivalent definitions defines the safety languages in complementary ways.

Safety languages can be subdivided into subclasses. A language L is closed under *stuttering* if $\sigma_0\sigma_1 \ldots \sigma_i\sigma_i\sigma_{i+1} \ldots$ is in L then $\sigma_0\sigma_1 \ldots \sigma_i \ldots \sigma_i\sigma_{i+1} \ldots$ and also $\sigma_0\sigma_1 \ldots \sigma_i\sigma_{i+1} \ldots$ are in L. Repeating any σ_i finitely many times or removing all repetitions will not affect the membership of a word in a language closed under stuttering. If a language is both a safety language and closed under stuttering it is called a *safety language with stuttering*. Lamport [38] calls this class of languages safety languages. The definition used in this work is according to [32]. It is equivalent to the definitions in [14], [2] and [47]. Stuttering is interesting because all *partial order reductions* require that the property is insensitive to stuttering [60].

Sistla [53] further introduces the term *strong safety languages*. A language L is a *strong safety language* iff

- L is a safety language with stuttering, and
- $\forall w = \sigma_0\sigma_1 \ldots \sigma_{i-1}\sigma_i\sigma_{i+1} \ldots \in L$ also $w' = \sigma_0\sigma_1 \ldots \sigma_{i-1}\sigma_{i+1} \ldots \in L$ for all $i > 0$.

Even if any σ_i is deleted, except σ_0, the resulting sequence should still be in L. According to Sistla, the motivation for the second condition is that even if the system is not observed at times, the observed behaviour should still satisfy the property. This class of properties is also of special interest to real time monitoring, as it is the only class of properties which can be monitored successfully without always starting the system from its initial state.

The verification community generally agrees that many properties which are verified are safety properties. This is not surprising since it is usually simple to think of many safety properties a system should have, while formulating

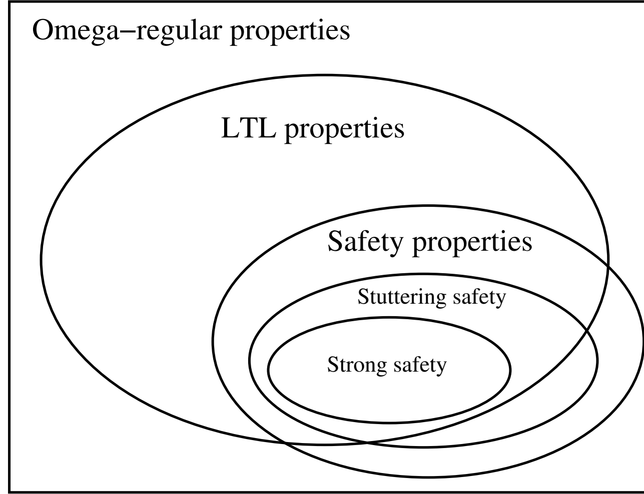

Figure 1: Hierarchy of omega-regular safety languages

many liveness properties is more challenging. Although some simple properties such as termination cannot be expressed within the framework of safety properties, some fairly complex properties are expressible.

Example 7 *We already mentioned that all invariants are safety properties. Thus, if we have an atom $no_deadlock$ defined, deadlock freedom can be expressed as an invariant.*

$$\mathbf{G}\, no_deadlock$$

All invariants are also strong safety properties. No matter how many states are removed from the model, the invariant will of course still hold.

Some properties about resource allocation are also safety properties. The property that all answers are preceded by a request can be formulated in the following way. Let the atomic propositions ans and req have the expected meanings. The formula defining the property is:

$$\neg ans\, W\, req$$

In some applications it is important that once we enter a certain state there is no way to leave. This property can be expressed by:

$$\mathbf{G}(p \Rightarrow \mathbf{G}p)$$

Liveness properties cannot be characterised by observing finite prefixes of executions. This is already suggested by the informal characterisation "eventually something good happens".

Definition 8 *A language L over an alphabet Σ is a liveness language iff the set $\{\pi_i \mid \pi \in L\}$ is Σ^*.*

The definition captures the intuition that any finite sequence can be extended to satisfy a liveness property.

Example 9 *Fairness is a typical liveness property and it required for proving many liveness properties. Strong fairness, i.e. if a request is made infinitely often it is also granted infinitely often can expressed in the following way:*

$$\mathbf{GF}\, req \Rightarrow \mathbf{GF}\, grant$$

A classical property for normal sequential programs is that all runs terminate.

$$\mathbf{F}\,terminate$$

Many properties related to resource allocation are also liveness properties. Let the atomic proposition p have the meaning that we have the resource and the atomic proposition q that we are ready to give the resource away. Then the property "once we have the resource, we will have it at least until we are ready to give it away" is expressed by the formula

$$\mathbf{G}(p \Rightarrow p\,U\,q).$$

In the above definitions of liveness, only languages where no word has a bad prefix, were accepted. This means that languages where some words have bad prefixes are neither safety nor liveness languages. Manna and Pnueli [47] call a language a progress language if some of its words do not have bad prefixes. The division of properties into safety and progress properties is semantically meaningful. The only properties which are both safety and progress properties are the ones which are equivalent to **true** (c.f. [47]).

4.4 Deciding safety

Knowing which formulas are safety formulas is not always easy. Clearly some formulas are safety formulas by the virtue of their structure, e.g. positive formulas using only the temporal operator $\mathbf{G}$.

An easy syntactic characterisation can be given using *past temporal logic* formulae. The past operators consider a finite past unlike their future counterparts, which refer to an infinite future. Past temporal logic uses operators like since, previously and once. A formula $\mathbf{G}\varphi$, where φ only contains past modalities, is safety formula and any safety property expressible with LTL is expressible in this way [47].

For the future fragment of temporal logic, Sistla [53] has presented a syntactic characterisation of safety formulae. We call these syntactically safe formulae.

Proposition 10 *([53]) Every propositional formula is a safety formula, and if ψ and φ are safety formulae then so are $\psi \vee \varphi$, $\psi \wedge \varphi$, $X\psi$, $\mathbf{G}\psi$, and $\psi\,V\,\varphi$.*

Proof:
The proof proceeds by induction on the formula structure and is somewhat different (hopefully also simpler) than the original proof by Sistla. We need only consider the cases $\vee, \wedge, X$ and V because $\mathbf{G}\psi \equiv \mathbf{false}\,V\,\psi$. For each case, we show that applying the operator preserves the property that each counterexample has a bad prefix.

We start with the base case. Let ψ be a propositional formula and π be a model such that $\pi \not\models \psi$. Specifically it is the first state in the model which fails and can thus be used as a bad prefix.

Let ϕ and φ be safety formulae and π a model such that $\pi \not\models \psi$, where ψ is one of the following.

- $\psi = \phi \vee \varphi$. By the semantics of the or-operator, both $\pi \not\models \phi$ and $\pi \not\models \psi$. As both are safety properties they have finite bad prefixes π_ψ and π_φ. One of prefixes will be valid for both as they are from the same model one of them will be a subsequence of the other.

- $\psi = \phi \wedge \varphi$. By the semantics of the and-operator a bad prefix for either ϕ or φ will do. This of course exists by the safety of ϕ and φ.

- $\psi = X\phi$. The semantics of X implies that $\pi^1 \not\models \psi$. Then by the safety of ϕ there exists an i such that π_i^1 is a bad prefix for ϕ. Any $\sigma_0\pi_i^1$, where $\sigma_0 \in 2^{AP}$, is then a bad prefix for ψ.

- $\psi = \phi V \varphi$. If $\pi \not\models \phi V \varphi$ then there exists an $i \geq 0$ such that $\pi^i \not\models \varphi$ and $\pi^j \not\models$ for all $0 \leq j < i$. Without loss of generality we fix our i to be the smallest such number. By the safety of ϕ and φ there exists bad prefixes π_k^i for ϕ and π_l for φ. There are now two possible cases:

  - $k \leq l$: Now π_l is a bad prefix for ψ because, π_l^k is a bad prefix for ϕ and since π_l includes π_k it is also a bad prefix for ψ.
  - $k > l$: Now π_k is a bad prefix for ψ. The argument is symmetric to the argument above.

□

Sistla [53] mentions that the syntactic fragment is also expressively complete for LTL safety properties, but no actual proof is provided. By leaving out the X-operator from the fragment, a fragment which expresses safety with stuttering is obtained.

Proposition 11 *([53]) Every positive formula using only the Release operator V expresses a safety property with stuttering.*

The result is not surprising because it is well-known that if the next-operator X is omitted, LTL can only express stuttering insensitive properties (c.f. [9]).

For strong safety properties Sistla [53] has proved a stronger result. The result does not restrict itself to strong safety properties of LTL. Using only **G**, LTL is expressively complete w.r.t. omega-regular languages and strong safety properties.

Proposition 12 *([53]) The omega-regular strong safety properties are exactly those expressed by positive formulae using only the temporal operator* **G**.

The above characterisations of safety properties, although useful, do not always help us in answering the question, given an LTL formula ψ, is it a safety formula. Often formulas are not given in the normal form. Thus it is an interesting problem decide if a given formula represents a safety property.

Sistla [53] solved the problem and presented an algorithm for LTL formulas, which Kupferman and Vardi [32] later generalised to general omega-regular properties expressed as alternating automata. The result is somewhat discouraging as it shows that deciding safety is PSPACE-complete. However using the same argument which has been used to defend the feasibility of LTL model checking, i.e. formulas tend to be so short that the procedure is useful in practice, checking for safety should be feasible in most cases.

As there is a linear translation in the number of states from LTL to alternating Büchi automata, we present the more general result concerning alternating automata.

Proposition 13 *([32, 53]) Deciding if the language of an alternating Büchi automaton is a safety language is PSPACE-complete.*

Proof:
Let the given automaton be $\mathcal{A}$. The automaton can be translated into an exponentially larger Büchi automaton $\mathcal{A}'$. Denote by $\mathcal{A}'_l$ the same Büchi automaton where all states have been marked as accepting. If $\mathcal{L}(\mathcal{A}'_l) \subseteq \mathcal{L}(\mathcal{A}')$ then the language of $\mathcal{A}$ is a safety language [2, 53]. This condition is equivalent to that $\mathcal{L}(\mathcal{A}'_l) \cap \mathcal{L}(\bar{\mathcal{A}}') = \emptyset$. Complementing an alternating automaton can be done with a quadratic blow-up and as we have previously shown, intersecting two automata and doing an emptiness check can be done in polynomial space.

The PSPACE-hardness result is due to Sistla [53]. It is possible to reduce the LTL validity problem to safety checking. Any LTL formula φ is valid iff $\varphi \vee \mathbf{F}p$ is a safety formula, where p is an atomic proposition not appearing in φ. This holds because the expression is equivalent to $\mathbf{true}$, a safety formula, only when φ is valid. A formula equivalent to a safety formula is also a safety formula. □

5 ABSTRACTION AND SAFETY PROPERTIES

One of the most important ways of combating state explosion is the use of abstraction. The idea of abstraction that models should only include relevant details pervades almost all disciplines of software engineering and computer science. An interesting question is how abstraction can aid model checking of safety properties. We consider this in the context of the abstraction/refinement framework for Coloured Petri nets presented by Lakos [36].

In the previous sections, we considered model checking when a Kripke structure M constructed in advance. Usually the Kripke structure is described implicitly using some formalism for describing systems. One such formalism is Petri nets.

Petri nets are a class of widely used modelling formalisms for concurrent and distributed systems. In the basic model, a system is represented by a set of possible control states, transitions with preconditions between the control states and an initial state. More high-level versions of Petri nets allow type information for the control states and complex functions on the transitions. Petri nets model both states and actions explicitly and are generally considered a versatile formalism for modelling different kinds of systems. Here we use Coloured Petri nets introduced by Jensen [30].

Abstraction is usually applied on a high-level formalism which generates a Kripke structure. The idea is that if irrelevant details are omitted from the high-level model, the resulting Kripke structure will be smaller and thus easier to model check. Lakos [36] has presented an abstraction/refinement framework for modelling with Coloured Petri Nets (CPNs). The framework is designed to support *incremental development*. Lakos suggests that most software development proceeds in an incremental manner, and therefore it is natural that the models of software we use should be built likewise. The process would begin with the design of an abstract model, which is later refined to encompass more and more details, finally resulting in the complete model. Lewis and Lakos [42] describe several projects where incremental modelling has been used.

The framework presented by Lakos restricts refinement to three forms, namely type refinement, node refinement, and subnet refinement. Lakos argues that with these three forms of refinement are equally powerful as his earlier more general proposal [35]. Lewis and Lakos have implemented a version of Maria which supports this refinement framework [42]. One of the most attractive features of this framework is that given two nets it is possible to statically check if one is a refinement of the other.

An interesting question is, which temporal logic properties are preserved, especially safety properties, by the refinements which the framework allows. This could make it possible to prove properties on the abstract net, which presumably has a smaller reachability graph than the refined net, and infer that the same properties hold for the refined net.

5.1 Coloured Petri Nets

We quite closely follow the notations of [36] in order to make it easy for readers to check details from this paper.

We denote the functions over Σ with $\Phi\Sigma = \{X \rightarrow Y | \, X, Y \in \Sigma\}$, the multisets over a colour set X by $\mu X = \{X \rightarrow \mathbb{N}\}$, and the finite sequences over a colour set X by $\sigma X = \{x_1 x_2 \ldots x_n \,|\, x_i \in X\}$. We define CPNs within the context of a given universe of colour sets Σ.

Definition 14 *A CPN is a tuple* $\langle P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0\rangle$ *where*

- P *is a finite set of places,*
- T *is a finite set of transitions,* $P \cap T = \emptyset$,
- A *is a set of arcs,* $A \subseteq P \times T \cup T \times P$,
- $C : P \cup T \rightarrow \sigma$ *assigns colours to places and transitions,*
- $E : A \rightarrow \Phi\Sigma$ *determines the arc inscriptions,*
- $\mathbb{M} = \mu\{(p, c) | \, p \in P, c \in C(p)\}$ *is the set of markings,*
- $\mathbb{Y} = \mu\{(y, c) | \, t \in T, c \in C(t)\}$ *is the set of steps, and*
- M_0 *is the initial marking,* $M_0 \in \mathbb{M}$.

Markings are defined as multisets over (place, colour) pairs and steps are multisets over (transition, colour) pairs.

The following notation is convenient in many definitions. For a CPN N, a node $x \in P \cup T$, and a set of nodes $X \subseteq P \cup T$ we define:

- the *preset* of x, ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in A\}$
- the *postset* of x, $x^\bullet = \{y \in P \cup T \mid (x, y) \in A\}$
- the *border* of X, $bd(X) = \{x \in X \mid \exists y \in (P \cup T) \setminus X : \ y \in {}^\bullet x \cup x^\bullet\}$
- the *environment* of X, $env(X) = \{y \in (P \cup T) \setminus X \mid \exists x \in X : {}^\bullet x \cup x^\bullet\}$

The preset of x is thus the nodes which have an arc *to* x while the postset of x are the nodes which have an arc *from* x. Similarly, the border of X is the nodes in X which have an arc to or from a node not in X while the environment X is the nodes not in X which have an arc to or from a node in X.

A change of state, i.e. the effect of firing of a set of transitions, is defined in two parts using the positive and negative incremental effects.

Definition 15 *The incremental effects* $E^+, E^- : \mathbb{Y} \rightarrow \mathbb{M}$ *of the occurrence of a step* $Y \in \mathbb{Y}$ *are given by:*

- $E^+(Y) = \sum_{(t,m) \in Y} \sum_{(t,p) \in A} \{p\} \times E(t, p)(m)$
- $E^-(Y) = \sum_{(t,m) \in Y} \sum_{(p,t) \in A} \{p\} \times E(p, t)(m)$

Definition 16 *A step* $Y \in \mathbb{Y}$ *is enabled in a marking* $M \in \mathbb{M}$ *if* $M \geq E^-(Y)$. *We denote this* $M[Y\rangle$. *An enabled step* Y *in a marking* M *can occur and lead to a marking* $M' \in \mathbb{M}$, *denoted* $M[Y\rangle M'$, *such that* $M' = M - E^-(Y) + E^+(Y)$.

Next we consider sequences of steps $Y^* = Y_1 Y_2 \ldots Y_n \in \sigma Y$. A step sequence Y^* is enabled in a marking M_0 and may occur leading to a marking M_n, denoted $M_0[Y^*\rangle M_n$, if there exists markings $M_1, M_2, \ldots M_{n-1}$ and steps $Y_1, Y_2, \ldots, Y_n$ such that $M_0[Y_1\rangle M_1$, $M_1[Y_2\rangle M_2, \ldots$, $M_{n-1}[Y_n\rangle M_n$. A step Y is *realisable* by a sequence Y^* in marking M, leading to a marking M', if $M[Y^*\rangle M'$ and $\sum_{y \in Y^*} y = Y$. When Y is realisable by Y^* in M we can see Y^* as a decomposition of Y into smaller steps.

In order to use temporal logic to specify properties on nets we must define the concept of an execution.

Definition 17 *An execution ξ of a CPN N is an infinite sequence of markings $\xi = M_0 M_1 M_2 \ldots$ such that M_0 is the initial marking and $M_i[Y_i\rangle M_{i+1}$ for some enabled step Y_i in M_i. If no step is enabled in a marking M_i, an execution is made infinite by setting $M_{i+1} = M_i$.*

Repeating the last state is standard trick for being compatible with the infinite sequence semantics of temporal logic. Deadlocking sequences are considered infinite by repeating the last state.

We can now define what it means for a temporal logic formula to hold for a Petri net. Given an LTL formula ψ we evaluate each atomic proposition occurring in ψ in all markings of ξ. Let $eval(\xi)$ be the infinite sequence in $(2^{AP})^\omega$ when each atomic proposition has been evaluated in the markings of the sequence. For a CPN N we write that $N \models \psi$ if for all executions ξ of N we have that $eval(\xi) \models \psi$.

5.2 Abstraction and Petri Nets

In many programming languages the main mechanisms for abstraction and refinement are subtyping or subclassing. Usually, subclassing only requires that a subclass has syntactically the same methods as a superclass. Essentially, this does not restrict the behaviour of the subclass in any way. Another approach is to require that the refinement must preserve bisimilarity or some other appropriate equivalence relation. In most cases this is too strong a requirement because it constraints the possibilities for refining too much. Lewis [41] discusses this issue in depth in his PhD thesis. The abstraction/refinement framework introduced by Lakos introduces behavioural compability as the key notion. Informally, any refinement must ensure that "every refined behaviour has a corresponding abstract behaviour". Furthermore, three key forms of refinement are introduced and it is argued that these three forms and their compositions are all you need. The three forms of refinement are type refinement, subnet refinement and node refinement.

Consider the simple order processing system of Figure 2. Orders are received and stored in the placed pending orders. Next, pending orders are registered and processed. After this step, it is still possible to cancel the order. If the order is not cancelled, the order goes into production, and is ready for delivery when the requested item has been produced. Finally the produced item is delivered.

Type refinement allows the refined net to replace types with compatible subtypes. A subtype can, e.g., introduce new data components into a token. The subtype must be compatible with the supertype in the sense that the

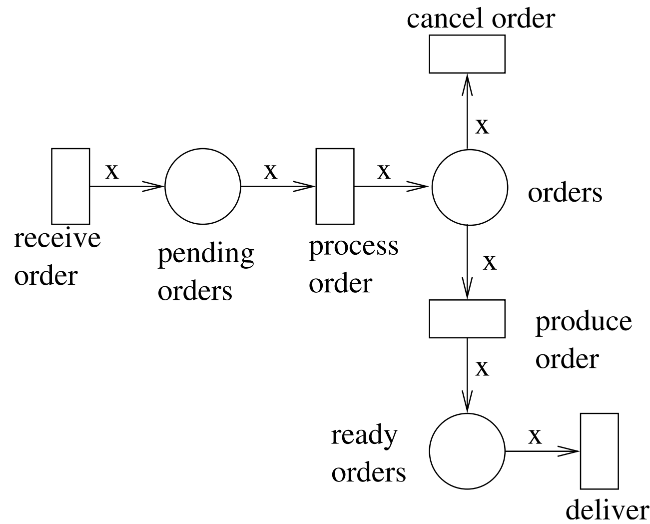

Figure 2: A simple order processing system

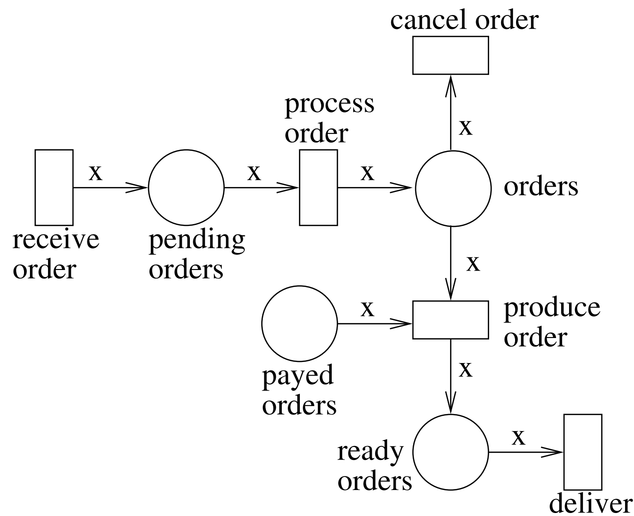

Figure 3: Subnet refinement

subtype can be used everywhere the parent type can. Usually this means that the values of the subtype can be projected onto the supertype. The arc inscriptions must be changed so that they fit the new tokens, however consistency with the old arc inscriptions must be maintained. Given a marking of the refined net, the corresponding marking of the abstract net is obtained by projecting the subtype onto the supertype. Thus, every refined behaviour has a corresponding abstract behaviour. The refinement can however restrict behaviour w.r.t. the abstract behaviour, as even though a token satisfies the abstract requirements of a transition the refined conditions may not be satisfied resulting in a deadlock not present in the abstract net. In our example we could e.g. refine the type for the orders to include more information. Instead of just including the name of the item, the type could also include information on the urgency of the order.

Subnet refinement allows places, transitions, and arcs to be added to the net. The additions may not add new behaviour to the abstract part of the net, but they can restrict behaviour. The corresponding abstract marking of a re-

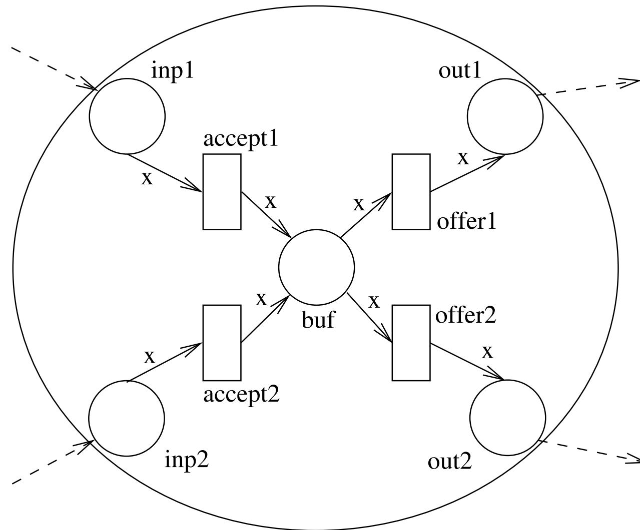

Figure 4: Canonical place refinement.

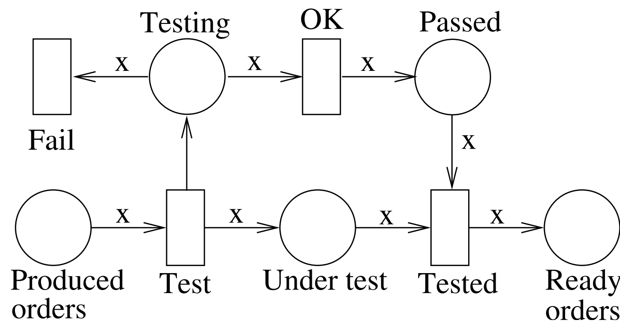

Figure 5: A node refinement for the order processing system

fined net is obtained by ignoring the added components. The order processing system could be refined by adding a check that the customer has payed the order before anything is produced. This restricts the behaviour of the net, because it introduces a possible deadlock if the customer has not payed his order. Figure 3 shows the order processing system with the refinement.

The third form of refinement is node refinement. In node refinement a place or a transition is replaced by a place or a transition bordered subnet. Lakos advocates the use of a canonical base for both refinements. By further refining the canonical bases an arbitrary node refinement can be accomplished. The basis for the canonical refinement can be seen in Figure 4. It is important that the refinement maintains the abstract marking, which is accomplished by the use of the canonical basis. As an example, we could refine the place "ready orders" to a new subnet which would test the ordered product for errors and only after the product has passed the tests, it would be ready for delivery. Figure 5 shows the place refinement for the "ready orders" place.

The mathematical framework of the theory is built around morphisms between nets. A refinement (or abstraction) is given by a morphism $\phi : N \rightarrow N'$ between the refined and the abstract net. In other words a morphism is in this context a mapping from nodes and arc inscriptions of the refined net to the abstract net, obeying certain restrictions. The first restriction is that a

morphism must be surjective w.r.t. P', T' and A', because refinement may only add components and not delete them. Thus ϕ^{-1} is always well defined.

We are mostly interested in *complete* steps, as firing many transitions in the refined net can correspond to firing one transition in the abstract net.

Definition 18 *([35]) Given a morphism $\psi : N \rightarrow N'$, a step Y of N is complete if $\forall t' \in T' : \forall t \in bd(\psi^{-1}(t')) = \{t'\} \times \psi(Y)(t')$.*

In other words a step Y is complete if the border transitions occur with matching modes.

Now we are ready to define behaviour respecting morphisms. Lakos calls a behaviour respecting morphism *system morphisms* [35].

Definition 19 *A **system morphism** $\phi : N \rightarrow N'$ is a mapping from N to N' such that:*

- *ψ is surjective w.r.t. P', T' and A'.*
- *ψ is linear and total over both $\mathbb{M}$ and $\mathbb{Y}$.*
- *$\forall M \in \mathbb{M} : \forall Y \in \mathbb{Y} : Y$ is complete and realisable as $Y_1 Y_2 \dots Y_n$ at marking $M \implies \psi(Y)$ is realisable as $\psi(Y_1)\psi(Y_2)\dots\psi(Y_n)$ at marking $\psi(M)$.*
- *For any reachable marking M, for all $Y \in \mathbb{Y}$, if Y is complete then $\phi(M + E^+(Y) - E^-(Y)) = \phi(M) + \phi(E^+)(\phi(Y)) - \phi(E^-)(\phi(Y))$.*

System morphism can be composed and the result will be a system morphism [35]. The three previously defined refinements can be expressed with system morphisms and are therefore behaviour respecting in the sense above. Essentially, any system morphism can be seen as a refinement.

5.3 Temporal Logic and Refinement

The general idea of abstraction is that we can prove properties of the refined net by proving the properties on the abstract net.

Let N be a refined net of N' with a corresponding system morphism $\phi : N \rightarrow N'$. What we actually wish to prove is that if $N' \models \varphi$ then all executions ξ of the refined net N are models φ when viewed through the morphism, i.e. $\phi(\xi) \models \varphi$. The abstract behaviour of the refined net should be the same as the behaviour of the abstract net. The refined nets can however introduce new deadlocks and can therefore have executions which have no corresponding abstract execution.

Because the refined the can introduce new deadlocks, intuitively the abstract behaviour should preserve all safety properties which do not require that something will happen in the future, not even in a bounded number of steps. In other words, the refinements should preserve the stuttering safety properties, and as it turns out this is indeed the case.

Theorem 20 *$N' \models \varphi$ implies that for all executions ξ of N we have that $\phi(\xi) \models \varphi$, when φ is a stuttering safety formula.*

Proof:
The proof proceeds by induction on the structure of the formula. Let $N' \models \varphi$.

Let φ be a propositional formula and $N' \models \psi$. Because ϕ is surjective w.r.t. P', clearly $\phi(M_0) = M'_0$ and the claim follows.

Let $\varphi = f \vee g$. By the assumption $N' \models f$ or $N' \models g$. The induction hypothesis then gives $\phi(\xi) \models f$ or $\phi(\xi) \models g$ and thus $\phi(\xi) \models f \vee g$. Proving $\varphi = f \wedge g$ proceeds in a similar manner.

Now let $\varphi = f\, V\, g$ and $N' \models \varphi$. System morphisms guarantee that for a finite sequence of markings $M_i, M_{i+1}, \ldots, M_k$ in the refined net there is a corresponding sequence $\phi(M_i), \phi(M_{i+1}), \ldots, \phi(M_k)$ in the abstract net. Equivalently, the absence of a sequence in the abstract net guarantees its absence in the refined net. However, when we consider infinite executions of the refined net, it is possible that they have deadlocked and repeat the last state even though the projected execution would not deadlock in the abstract net. Consider the projection of an execution $\phi(\xi)$. If ξ is not a deadlocking execution we can immediately conclude that $\phi(\xi) \models \varphi$ because $\phi(\xi)$ is an execution of the abstract net. Now consider the case where from some k onward $\xi(i) = \xi(i+1)$ for all $i \geq k$ because ξ is a deadlocking execution. By the semantics of V, $f\, V\, g$ can hold if $\xi^i \models f$ for all $i \geq 0$ or there exists j such that $\xi^j \models g$ and $\xi^l \models f$ for all $0 \leq l \leq j$. In the case no such j exists, $\phi(\xi)^i \models f$ for all $i \geq 0$ by the induction hypothesis and thus $\phi(\xi) \models \varphi$. If the bound j exists there are two possibilities: $j > k$ or $j \leq k$. If $j > k$ the situation reduces to the case where $\phi(\xi)^i \models f$ for all $i \geq 0$. If $j \leq k$, by the induction hypothesis $\phi(\xi)^i \models f$ for $0 \leq i \leq j$ and we can also apply the induction hypothesis to show that $\phi(\xi)^j \models g$. From this we can conclude that $\phi(\xi) \models f\, V\, g$.

$\mathbf{G}f$ is covered by the last case as $\mathbf{G}f \equiv \mathbf{false}\, V\, f$. □

The question which naturally follows is can we prove that even more properties are preserved. The depressing but probable answer is no. For all temporal operators which demand that something will eventually occur, like X, U and $\mathbf{F}$, it is easy to construct counterexamples where the properties are not preserved in the refined net. It should however be noted that refinement notions which preserve too much information are usually very restrictive. Lewis [41] dwells into the issue why refinements which preserve e.g. bisimilarity are often inappropriate.

It is however possible to strengthen the notion refinement in order to preserve more properties. Lewis [41] has investigated how refinement must be strengthened in order to achieve weak bisimilarity. Intuitively, this should hold if the refinement does not introduce any new deadlocks. In [41] the notion of "at least as live" is defined and weak bisimilarity is proven to hold.

Proposition 21 *([41]) If the net N is a refinement of N' by the system morphism $\phi : N \rightarrow N'$ and N is at least as live as N', then N is bisimilar to N'.*

The construction works by labelling all refined actions which do have counterparts in the abstract net as τ-actions. Thus the projected executions can include some repetitions, i.e. stuttering. The notion of "at least as live" is

however computationally very heavy. Currently no better technique than constructing the full reachability graph exists to determine if the refined net is at least as live as the abstract net. For combating state explosion this approach is useless.

The proposition proven by Lewis has an interesting corollary. When the refined net is at least as live as the abstract net, the next-operator free full branching time logic CTL^*_{-X} is preserved.

Corollary 22 *If the net N is a refinement of N' by the system morphism $\phi : N \to N'$ and N is at least as live as N', for all $\varphi \in CTL^*_{-X}$ holds $N \models \varphi \Longleftrightarrow N' \models \varphi$.*

The result follows directly from known properties of bisimulation. See e.g. [7] for a discussion on the subject.

As corollary 22 suggests, it is possible still generalise Theorem 20 to encompass some branching time properties. We consider allowing the use of the path quantifiers $\mathbf{A}$ (for all paths) and $\mathbf{E}$ (for some path) with stuttering free safety formulas of Theorem 20. These form a restricted subset of the full branching time logic CTL^*.

We first consider formulae containing the existential path quantifier $\mathbf{E}$. A simple analysis quickly concludes that formulae containing $\mathbf{E}$ are not preserved by refinement. A formula $\mathbf{E}\psi$ holds in the abstract net if one execution is a model for ψ. As the refined net has less behaviour that the abstract net, it is clear that formulae containing $\mathbf{E}$ are not preserved.

For formulae containing the universal path quantifier $\mathbf{A}$ the situation is the same as for LTL. With a very similar proof as above we obtain the following slight generalisation of Theorem 20.

Theorem 23 *For any positive $ACTL^*$ formula φ containing only the temporal operator V, if for the abstract net $N' \models \psi$ then for all execution ξ of the refined net $\phi(\xi) \models \varphi$.*

The results presented above are related to the results presented by Padberg et al. [49]. They present an approach for stepwise refinement using a rule-based approach based on invariant preserving morphisms. Our results are more general in the sense that we preserve a larger fragment of the safety properties.

5.4 An Example

Consider the order processing system presented previously. It can be seen as a simple example on how incremental design could work. First an abstract model, again see Figure 2, is created which only has the most basic features of the system. Orders are received and processed. Cancelling an order is already possible in the basic model. If an order is not cancelled it is eventually delivered to the customer. We could be interested in making sure that an order is not delivered unless it has been processed. The property can be formalised in the following way:

$$\neg p_i \, W \, q_i,$$

where p_i is a proposition meaning that "the order i has been delivered" and q_i is a proposition meaning that "the order i has been processed". In the abstract model this is easy to verify e.g. using model checking.

Next we start adding features to the model. As previously, we want to make sure only payed orders are produced. This can be done by adding one transition to the model. The result can be seen in Figure 3. Since this is a legal subnet refinement we can by Theorem 20 conclude that the property still holds.

There is no need to stop adding features. As previously, the place "ready orders" can be refined to also model the final testing of the products. No product is shipped without testing. The place refinement can be seen in Figure 5. Again, we can by Theorem 20 conclude that the property still holds.

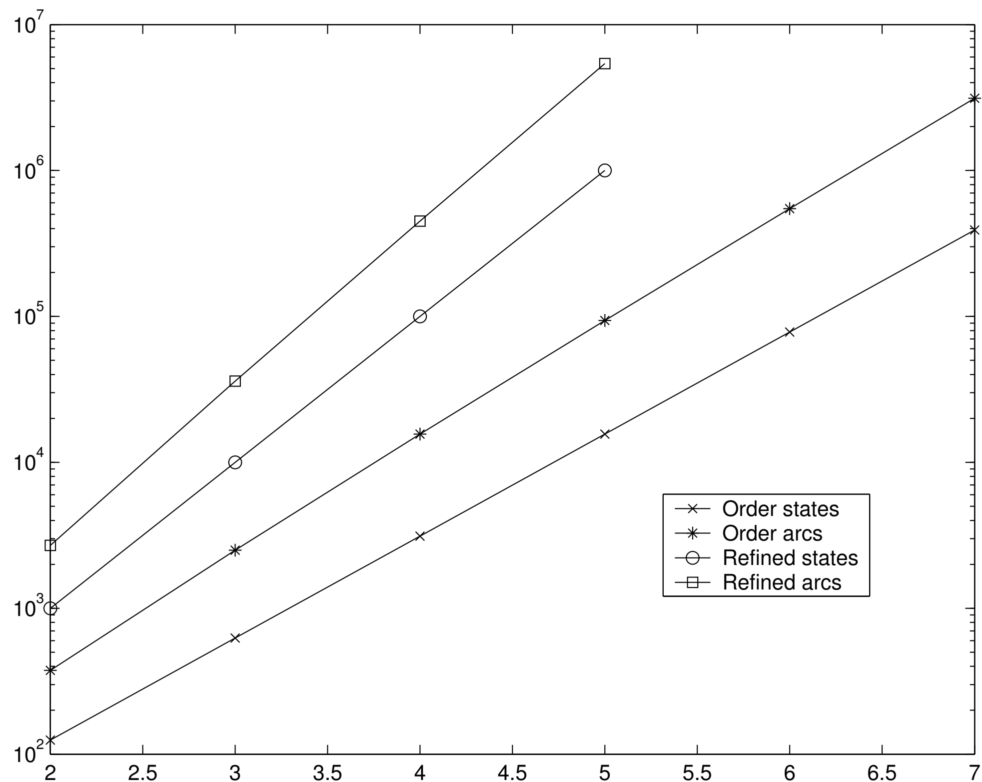

Figure 6: Statistics for the reachability graph of the order processing system.

The question which naturally arises is how much harder is it to apply model checking directly on the refined system. One way of comparing this is to compare the sizes of the reachability graphs, i.e. Kripke structures of the models. In order for this to be possible, we must close the models and add an environment which makes orders. The environment we modelled is perhaps the simplest possible. It chooses non-deterministically from a pool of N possible orders and makes the order. Each order can only be made once. In the refined model it also pays the order when it is made. The models were analysed by the reachability analyser Maria [46]. For the abstract model we computed the reachability graph for $N = 2, 3, \ldots, 7$. As could be expected, the reachability graph of the refined model grew so quickly, we only computed it for $N = 2, 3, 4, 5$.

Figure 6 shows the sizes of the reachability graphs. The x-axis gives the value of the parameter N and the y-axis the number of states or arcs. Note that the scale on the y-axis is logarithmic. As can be expected, in all cases the growth is exponential. The models are in this case so simple that the curves can be well described by a functional dependency of the form $y = 10^{aN+b}$. Table 1 contains the values for the parameters of the curves, which have been computed using the least squares method. By examining the curves and parameters, it is evident that the rate of growth for reachability graph of the refined model is exponentially faster compared to the rate of growth for the reachability graph of the abstract model. This translates to an exponential saving when model checking the abstract system instead of the refined system.

Table 1: Parameter values for fitting the curve $y = 10^{aN+b}$ to sizes of the reachability graphs.

	Abstract		Refined	
	a	b	a	b
states	0.6990	0.6990	1	1
arcs	0.7829	1.0386	1.100	1.2433

We conjecture that a similar approach could be taken when a more complex system is designed. For a realistic system the abstract model would not be as trivial, as at least the fundamental properties of the system must be visible from its behaviour. The philosophy of this approach is quite similar to the B-method [1] and other "correctness by design" methods.

6 MODEL CHECKING SAFETY PROPERTIES

Model checking has gained wide recognition as a useful technique for ensuring the correctness of designs. Despite the statespace explosion problem, many impressively large designs have been proven correct or serious errors in them have been found. The two perhaps most attractive features of model checking are its very high level of automation and its ability to produce a concrete error trace. The error trace is valuable when measures to correct the design are taken.

The safety properties are of special interest in model checking. In the previous section we showed that many interesting properties belong to the safety fragment. Indeed, the name safety suggests that many critical properties belong to this fragment. A liveness property that demands that a request will be eventually served is important but a safety property that demands that railway control system never lets two trains collide is critical. There are also other reasons why the safety fragment is interesting. Here we mention three of them:

1. industrial specifications are mostly concerned with safety properties,
2. treating safety properties as a special case allows more efficient verification, and
3. safety properties are closely related to specifications for testing and runtime verification.

The second and third points merit some further explanation.

As previously defined, safety properties can be described by the bad prefixes for a certain property. This means that when model checking a safety property, it can be reduced to a search for bad prefixes where as full LTL model checking requires searching for bad cycles. Searching for bad prefixes can be considerably easier than searching for bad cycles in the system. Verifying liveness properties also usually requires that the issues of fairness are taken into account. Almost all proofs of liveness properties require fairness assumptions.

Safety properties are also related to testing and runtime verification. These are techniques for gaining confidence that the system works as it should, as opposed to proving that it works correctly. Testing and runtime verification are falsification techniques, i.e. they are aimed at finding errors. Although the bug-hunting aspect of model checking is often emphasised model checking can also prove properties. There has been interest in integrating testing and model checking in recent years and several papers combine model checking techniques and testing [19, 28, 25, 21]. One possibility is let the property guide the testing [28], or simply monitor a real system and check that the system does not execute a bad prefix [25, 21]. Because testing and runtime verification only observe finite executions, it is only possible to validate safety properties. This is why safety properties are very interesting from the testing-perspective.

Before proceeding to the details of model checking safety properties, let us review how the traditional automata theoretic approach [34, 63] to model checking works. The system under inspection is modelled as a Kripke model

M and the specification is given as an LTL formula φ. The Kripke model M can be see as an automaton accepting the language $\mathcal{L}(M)$. It is also possible to create an automaton on infinite words $\mathcal{A}$ which exactly accepts $\mathcal{L}(\varphi)$ [48]. Clearly, a system M has the property φ if the languages have no common words. This is equivalent to that $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) = \emptyset$. The following procedure can thus be applied to model check M against φ [34, 10].

1. Construct a Büchi automaton $\mathcal{A}_{\neg\varphi}$ with the language $\mathcal{L}(\neg\varphi)$.
2. Construct the Kripke model M of the system, interpret it as a Büchi automaton.
3. Compute the product Büchi automaton $\mathcal{B} = M \times \mathcal{A}_{\neg\varphi}$ which is an automaton with the language $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi)$.
4. Check if $\mathcal{L}(\mathcal{B}) = \emptyset$

If the the language of $\mathcal{B}$ is empty, the property holds. If the property does not hold, it is possible to extract a violating execution. This is one of the most attractive features of model checking, compared to other methods which only inform you that there is an error in the model. Combining several of the steps mentioned above and performing them in an interleaving manner is called on-the-fly model checking.

There are several way of performing the emptiness check. In an explicit state context, the most straight forward way is to compute the maximal strongly connected components (SCC) of the product automaton, which can be done in linear time [56]. We remind the reader that a SCC is non-trivial if it contains more than one state or the single state has a self loop. The product is empty if no non-trivial SCC contains a state belonging to the set of final states. Another way is to use the nested depth-first algorithm of [10]. The SCC based algorithm has been extended in many ways to take into account different fairness constraints [43, 16, 39, 40].

Model checking safety properties does not differ much from the procedure above. The steps are the same, but some of the procedures differ. In the first step, instead of constructing a Büchi automaton we construct an automaton on finite words which captures the bad prefixes of the formula. The two following steps, two and three, are essentially the same. The final step, performing the emptiness check is different. An automaton on finite words is empty, if no state in the set of final states is reachable. Thus, there is no need to compute the SCCs of the product automaton or something similar - reachability of a final state is sufficient. Using on-the-fly methods, we can simultaneously build the product automaton and check if the current state is a final state. There is no need to build the whole product automaton if we reach a final state. In some cases this can speed up model checking significantly. Usually the product automaton is built in a depth-first or a breadth-first order. This is not required. Heuristics can be applied to the construction, which can result in shorter running times to find an error.

6.1 Detecting Bad Prefixes

To model check safety properties we would like to construct automata on finite words which recognise all bad prefixes for a given LTL formula. Our

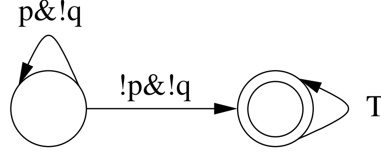

Figure 7: Automaton for the bad prefixes of $\psi = p\,W\,q$

main motivation is that reasoning about automata on finite words is easier than reasoning about infinite words. An LTL formula can be translated into Büchi automaton which is exponential in the size of the formula. How well can we do with safety formulae and automata on finite words?

In the following we introduce some concepts from [32]. Let $L \subseteq \Sigma^\omega$ be safety language. We denote by $pref(L)$ the set of *all* bad prefixes of L. Just as we defined a co-safety language as a language whose complement is safety language, we define $co\text{-}pref(L) = pref(\bar{L})$. A set $X \subseteq pref(L)$ is a *trap* for L iff every word $w \notin L$ has at least one bad prefix in X. The difference between $pref(L)$ and a trap X for L is that X need not contain all the bad prefixes of L, only enough to detect all words not in L. All traps of L are denoted $trap(L)$. For a formula ψ, instead of writing the cumbersome $trap(\mathcal{L}(\psi))$ we simply write $trap(\psi)$.

Example 24 *Consider the safety formula $\psi = p\,W\,q$. A sequence does not satisfy the formula if it has a state where p does not hold and there is no previous state for which q holds. In this case we then have that $pref(L(\psi)) = \{\neg q, p\}^*\{\neg p, \neg q\}\top^*$. Figure 7 shows a corresponding deterministic automaton.*

Constructing $pref(L)$ is easy by hand for simple languages but in the general case it is hard. The construction is similar to complementation because we should detect prefixes for words not in the language. This is also evident from the example above. Complementing a non-deterministic automaton is of exponential complexity. Unfortunately, Kupferman and Vardi have proved that the same holds for constructing an automaton for $pref(L(\mathcal{A}))$, when the language is specified by a non-deterministic Büchi automaton $\mathcal{A}$.

Proposition 25 *([32]) Given a (co-)safety non-deterministic Büchi automaton $\mathcal{A}$, the size of the finite automaton that recognises $(co-)pref(\mathcal{L}(\mathcal{A}))$ is $2^{\theta(|\mathcal{A}|)}$.*

As can be seen from the proposition, it does not help even if the automaton specifies the disallowed words. The requirement of detecting *all* bad prefixes is simply too harsh. When the property is specified as an LTL formula the situation is even bleaker. The complexity is doubly exponential.

Proposition 26 *([32]) Given a safety LTL formula ψ of size n, the size of an automaton for $pref(\psi)$ is $2^{2^{O(n)}}$ and $2^{2^{\Omega(\sqrt{n})}}$.*

In the light of the extreme lower bounds proven above, constructing an automaton for $pref(\psi)$ is not feasible. However, detecting all bad prefixes may not be necessary. Potentially, all benefits gained from having $pref(\psi)$

could also be had from having an automaton for $X \in trap(\psi)$. For model checking purposes it is enough to detect one bad prefix of each computation and not all of them. We define the following concepts.

Definition 27 *([32]) Given a safety formula ψ, a finite automaton $\mathcal{A}$ is tight for ψ iff $\mathcal{L}(\mathcal{A}) = pref(\psi)$. A finite automaton is fine for ψ iff $\mathcal{L}(\mathcal{A}) = X$ for some $X \in trap(\psi)$.*

An automaton which is fine for a safety formula ψ does not recognise all bad prefixes, but it will still identify all bad computations. Almost all benefits of a tight automaton can also be enjoyed with a fine automaton. Tight automata have one distinct advantage: as they recognise all bad prefixes they also recognise the minimal bad prefixes. Model checking using tight automata might result in shorter counterexamples.

It is currently an open problem if there are feasible constructions for fine automata for safety LTL formulae. The proofs that Kupferman and Vardi present for the exponential complexity of recognising *co-pref*(L), when L is a safety language given by a co-safety non-deterministic Büchi automaton, are not valid for the case where we only require that the finite automaton is a trap for L. The same is true for their proof of the doubly exponential complexity of recognising $pref(\psi)$, where ψ is a safety formula.

6.2 Informativeness

While LTL safety formulae in general are difficult to deal with, clearly at least some well behaved formulae can easily be translated. This is readily seen by just observing the structure of the Büchi automata which efficient translations tools produce. When they translate safety formulae, the resulting Büchi automaton could be used as is in many cases as the automaton on finite words. For example, given the negation of the formula $p\,W\,q$, the LTL to Büchi automata tool [20] produces an automaton equivalent to the one in Figure 7. The question is, for which kind of formulas is this possible.

Kupferman and Vardi [32] try to capture the notion of when formulas are well-behaved and can be translated easily. For safety formulas, the key notion is that do the bad prefixes for the formula "tell the whole story" of why the formula is violated by a computation. A prefix which tells the whole story of why a formula is violated is called *informative*.

We consider LTL formulae in positive normal form. Let ψ be an LTL formula and π a finite computation $\pi = \sigma_0\sigma_1 \ldots \sigma_n$. The computation π is *informative* for ψ iff there exists a mapping $L : \{0, \ldots, n+1\} \to 2^{cl(\neg\psi)}$ such that the following conditions hold:

- $\neg\psi \in L(0)$,
- $L(n+1)$ is empty, and
- for all $0 \leq i \leq n$ and $\varphi \in L(i)$, the following hold.
  - If φ is a propositional assertion, it is satisfied by σ_i.
  - If $\varphi = \varphi_1 \vee \varphi_2$ then $\varphi_1 \in L(i)$ or $\varphi_2 \in L(i)$.

- If $\varphi = \varphi_1 \wedge \varphi_2$ then $\varphi_1 \in L(i)$ and $\varphi_2 \in L(i)$.
- If $\varphi = X\varphi_1$, then $\varphi_1 \in L(i+1)$.
- If $\varphi = \varphi_1 U \varphi_2$ then $\varphi_2 \in L(i)$ or $[\varphi_1 \in L(i)$ and $\varphi_1 U \varphi_2 \in L(i+1)]$.
- If $\varphi = \varphi_1 V \varphi_2$ then $\varphi_2 \in L(i)$ and $[\varphi_1 \in L(i)$ or $\varphi_1 V \varphi_2 \in L(i+1)]$.

If π is informative for ψ, the mapping L is called the *witness* for $\neg\psi$ in π. Using the notion of informativeness, safety formulae can be classified into three different categories [32].

- A safety formula ψ is *intentionally* safe iff all the bad prefixes for ψ are informative.
- A safety formula ψ is *accidentally* safe iff every computation that violates ψ has an informative prefix. The formula ψ can in other words have bad prefixes which are not informative. Every computation is, however, guaranteed to have at least one informative prefix.
- A safety formula ψ is *pathologically* safe if there is a computation that violates ψ and has no informative bad prefix.

We can illustrate the difference between the categories with a few examples.

Example 28 *The formulas Xp and $p V q$ are examples of intentionally safe formulas. For Xp we have that $pref(Xp) = \top\neg p\top^*$. We can construct a mapping L which satisfies all possible bad prefixes. Let $L(0) = X\neg p, L(1) = \neg p$ and $L(2 \dots n) = \emptyset$. In a similar manner we can construct a mapping for $p V q$.*

Accidentally safe formulas usually contain some forms of redundancy. Consider the accidentally safe formula $\psi = \mathbf{G}(p \vee (Xq \wedge X\neg q))$. The set of bad prefixes is $pref(\psi) = (\top)^*\neg p\top^*$. In this case it is impossible to construct a mapping L for some prefixes. Consider the bad prefix $\{p\}\emptyset$. We can set $L(0) = \{\neg\psi\}$ and $L(1) = \{\neg\psi, \neg p, Xq \vee X\neg q\}$. The construction of L cannot be finished because the informativeness requirements are in conflict. On one hand the length of the witness is two and therefore $L(2)$ should be empty, while on the other hand the next state operator requires $L(2)$ to be non empty. The prefix $\{p\}\emptyset$ is not informative for ψ. Any violating computation will however contain an informative bad prefix.

Pathologic formulae have even more redundancy than accidentally safe formulae. Consider the formula $\psi = [\mathbf{G}(q \vee \mathbf{FG}p) \wedge \mathbf{G}(r \vee \mathbf{FG}\neg p)] \vee \mathbf{G}q \vee \mathbf{G}r$. The formula is actually equivalent to $\mathbf{G}q \vee \mathbf{G}r$. Thus the set of bad prefixes is $pref(\psi) = \top^*(\neg r\top^*\neg q \cup \neg q\top^*\neg r \cup \neg q \wedge \neg r)\top^*$. No violating computation will have an informative prefix because it is impossible to fulfil the requirement of informativeness. For instance, the formula $\mathbf{FG}p$ becomes $\mathbf{GF}\neg p$ when you negate it. Thus, informativeness requires that $\mathbf{GF}\neg p$ is in each $L(i), 1 \leq i \leq n$. This violates the requirement that $L(n)$ should be empty.

Consider the translation of LTL formulas to alternating Büchi automata given in Section 4.2. It maps subformulas of the given formula to states, and the transitions follow the semantics of LTL. Any finite run of the automaton which which ends in **true** is accepted, and can actually be seen as inducing a mapping L. Let ψ be a safety LTL formula and $\mathcal{A}_{\psi}^{true}$ denote the corresponding alternating automaton where the set of accepting states is empty. $\mathcal{A}_{\neg\psi}^{true}$ will accept exactly the computations which have an informative bad prefix for ψ. Let $fin(\mathcal{A}_{\neg\psi}^{true})$ the automaton when regarded as an automaton on finite words.

Proposition 29 *([32]) For a formula ψ, the automaton $fin(\mathcal{A}_{\neg\psi}^{true})$ accepts exactly the bad prefixes for ψ which are informative.*

Proof:
Let $\pi = \sigma_0\sigma_1\ldots\sigma_n$ be a word accepted by $fin(\mathcal{A}_{\neg\psi}^{true})$. The word induces a run $\langle T, r\rangle$ which maps positions of the prefix to sets subformulas (sets of states of the automaton) of $\neg\psi$. At each level i of the tree, the nodes comprise the set $L(i)$. Because both the automaton and the witness L obey the same semantics (easily verified by comparing the definition of L and the transition relation of $\mathcal{A}_{\neg\psi}$), this will induce a valid witness mapping L.

Now, let $\pi = \sigma_0\sigma_1\ldots\sigma_n$ be an informative bad prefix for ψ. With arguments symmetric to the ones above it easy to see that the π induces a an accepting run. □

The result above immediately implies that $fin(\mathcal{A}_{\neg\psi}^{true})$ can be very useful, and that we can model check most safety formulae without constructing something which is doubly exponential in the length of the formula. We get the following corollary.

Corollary 30 *([32]) Let ψ be a safety formula.*

- *If ψ is intentionally safe, then $fin(\mathcal{A}_{\neg\psi}^{true})$ is tight for ψ.*
- *If ψ is accidentally safe, then $fin(\mathcal{A}_{\neg\psi}^{true})$ is fine for ψ.*

In practice, this means that as long as a safety formula is not pathologically safe we can use a singly exponential construction for the automata. If the formula is pathologic, we might possibly miss a counterexample using this construction. Any counterexample detected by $fin(\mathcal{A}_{\neg\psi}^{true})$ is, however, a valid counterexample. This raises the question how can we recognise pathologically safe formulas in order to avoid them. Pathologic formulas are not needed as such, as they are always equivalent to non pathological formula.

The syntactically safe formulas introduced by Sistla are, perhaps as expected, well-behaved and they always have informative bad prefixes. For a syntactically safe formula ψ, the negation of the formula can only contain the temporal operators U and X. If $\pi \models \neg\psi$, then by the semantics of U and X it is easy to see that for some i, π_i must be an informative prefix for ψ. Both U and X must be satisfied after a finite number of states. Thus we get the following proposition.

Proposition 31 *([32]) If an LTL formula ψ is syntactically safe, then ψ is intentionally or accidentally safe.*

For the remaining safety formulas a general procedure must be applied. Unfortunately, deciding if a formula is pathologic is hard in the general case. We must decide if the formula has any violating computations which lack informative bad prefixes. The problem can be reduced to the satisfiability problem of LTL, which is a PSPACE-complete problem. The original result is due to Kupferman and Vardi where they suggest (but do not present) a reduction to the safety of an LTL formula, also a PSPACE-complete problem.

Proposition 32 *([32]) Deciding whether a given formula ψ is pathologically safe is PSPACE-complete*

Proof:
The automaton $\mathcal{A}_{\psi}^{true}$ accepts exactly all computations which have informative prefixes. Pathologic formulas have violating computations which are not informative. Thus, a formula is not pathologic if every computation that satisfies $\neg\psi$ is accepted by $\mathcal{A}_{\neg\psi}^{true}$. This can be verified by checking the containment of $\mathcal{L}(\mathcal{A}_{\neg\psi})$ in $\mathcal{L}(\mathcal{A}_{\neg\psi}^{true})$. As we have noted before, containment of alternating automata can be done in polynomial space [31]. Later in this work, we will dwell in to the specific details of implementing this check.

Let ψ be an LTL formula and p an atomic proposition not appearing in ψ. Consider the formula $\varphi = \psi \wedge \mathbf{F}p$. Now, ψ is satisfiable iff φ is pathologic. If ψ is satisfiable, we must consider if there are computations that satisfy $\neg\varphi = \neg\psi \vee \mathbf{G}\neg p$ which do not have finite prefixes with a witness L. Let π be computation such that $\pi \models \mathbf{G}\neg p$ and $\pi \not\models \neg\psi$. Because ψ cannot be equivalent to $\mathbf{F}p$ such a model must exist. The model π has no informative prefix because $\mathbf{G}$ cannot have a finite witness. Consequently, φ has computations which have no informative bad prefix and is therefore pathologic. To prove the other direction, we consider the negative case when ψ is unsatisfiable. In this case $\neg\varphi = \neg\psi \vee \mathbf{G}\neg p$ is valid. Thus any prefix for any model will serve as an informative witness. $\square$

6.3 Translation Algorithm

The construction given in the previous section gives us a way of constructing an automaton for informative bad prefixes of a safety property ψ. In most cases, however, we prefer to deal with normal finite automata rather than alternating automata. Some research has focused on using alternating automata directly [19].

One way to get a finite automaton is to translate the alternating automaton resulting from the construction. Another possibility is to define the translation directly to finite automata. Kupferman and Vardi [32] have presented one direct translation from an LTL formula. The translation is based on the reverse deterministic automaton defined in [48].

The automaton associates states to subsets S in $cl(\neg\psi)$. The set of formulas associated with a state represent the unfulfilled conditions of the formula. Thus, a successful run will start in a initial state which contains $\neg\psi$ and will finish in the single accepting state which maps to $\emptyset$, signifying that there are no obligations left to fulfil.

An accepting run of the automaton for a finite prefix $\pi = \sigma_0\sigma_1 \ldots \sigma_n$ will induce a witness L, where states of the run r correspond to the witness,

showing that π is informative for ψ

The original construction of Kupferman and Vardi [32] is the following. Let $S \subseteq cl(\neg\psi)$ be a state and $\sigma \in 2^{AP}$ a letter. The single predecessor S' in $\delta^{-1}(S, \sigma)$ contains exactly all the propositional assertions in $cl(\neg\psi)$ that are satisfied by σ, and all formulas φ in $cl(\neg\psi)$ for which the following hold.

- If $\varphi = \varphi_1 \vee \varphi_2$, then $\varphi \in S'$ iff $\varphi_1 \in S'$ or $\varphi_2 \in S'$.
- If $\varphi = \varphi_1 \wedge \varphi_2$, then $\varphi \in S'$ iff $\varphi_1 \in S'$ and $\varphi_2 \in S'$.
- If $\varphi = X\varphi_1$, then $\varphi \in S'$ iff $\varphi_1 \in S$.
- If $\varphi = \varphi_1 \, U \, \varphi_2$, then $\varphi \in S'$ iff $\varphi_2 \in S'$ or $[\varphi_1 \in S'$ and $\varphi_1 \, U \, \varphi_2 \in S]$.
- If $\varphi = \varphi_1 \, V \, \varphi_2$, then $\varphi \in S'$ iff $\varphi_2 \in S'$ and $[\varphi_1 \in S'$ or $\varphi_1 \, V \, \varphi_2 \in S]$.

For efficiency reasons it usually also justified to implement the translations for the derived operators **G** and **F**.

- If $\varphi = \mathbf{G}\varphi_1$, then $\varphi \in S'$ iff $\bot$
- If $\varphi = \mathbf{F}\varphi_1$, then $\varphi \in S'$ iff $\varphi_1 \in S'$ or $\mathbf{F}\varphi_1 \in S$.

As can been seen above, translating **G** is impossible in the sense that no finite witness can be given for **G**. Those familiar with bounded model checking [5] will notice that the translation is very similar to what is called the bounded semantics without a loop in [5].

The above implicit representation is suitable for use in tool which, e.g. uses BDDs to manipulate boolean formulas. Using the above definition in an explicit state tool requires that we compute the automaton first. An algorithm which does this, starts from the empty set as the initial state, and then using the rules above computes the predecessors for each state until no new states are found. The algorithm given below computes an automaton. Note that the algorithm will not work correctly if the iteration over the subformulas in $cl(\psi)$ is not done in some increasing subformula order.

Input: A safety formula ψ in positive normal form.
Output: A finite automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$.
proc $translate(\psi)$
$\Sigma = 2^{AP}; F := \{\emptyset\};$
$Q := X := F;$
while$(X \neq \emptyset)$ **do**
    S :="some element in X"; $X := X \setminus \{S\}$;
    for each $\sigma \in 2^{AP}$ **do**
        for each $\varphi \in cl(\psi)$ **do**
            switch(φ) **begin**
                case $p = q$ or $p = \neg q$ for $q \in AP$:
                if (p is satisfied by σ) **then** $S' := S' \cup \{p\}$
                case $\varphi = \psi_1 \vee \psi_2$:
                if ($\psi_1 \in S'$ or $\psi_2 \in S'$) **then** $S' := S' \cup \{\varphi\}$;
                case $\varphi = \psi_1 \wedge \psi_2$:
                if ($\psi_1 \in S'$ and $\psi_2 \in S'$) **then** $S' := S' \cup \{\varphi\}$;

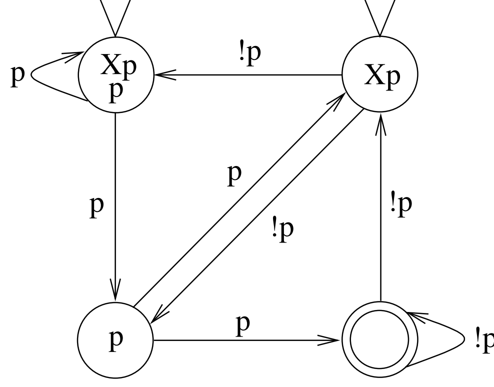

Figure 8: A finite automaton for the bad prefixes of $X\neg p$.

case $\varphi = X\psi_1$:
if $(\psi_1 \in S)$ **then** $S' := S' \cup \{\varphi\}$;
case $\varphi = \psi_1 \, U \, \psi_2$:
if $(\psi_2 \in S'$ or $(\psi_1 \in S'$ and $\varphi \in S))$ **then** $S' := S' \cup \{\varphi\}$;
case $\varphi = \psi_1 \, V \, \psi_2$:
if $(\psi_2 \in S'$ and $(\psi_1 \in S'$ or $\varphi \in S))$ **then** $S' := S' \cup \{\varphi\}$;
end
od
if$(\psi \in S')$ **then** $Q_0 := Q_0 \cup \{S'\}$;
$\delta = \delta \cup \{(S', \sigma, S)\}$;
$X := X \cup \{S'\}$; $Q := Q \cup \{S'\}$
od
od

Example 33 *Consider the safety formula $X\neg p$. Lets construct the automaton which accepts all bad prefixes of the formula ($X\neg p$ is intentionally safe). The negation of the formula is Xp. In this case set of atomic propositions is $AP = \{p\}$and the resulting power set, and thus the alphabet the automaton will be $\Sigma = 2^{AP} = \{\emptyset, \{p\}\}$. The states are subsets of $cl(Xp) = \{p, Xp\}$.*

We start going backwards using the definitions above from the state $S = \emptyset$, and compute the unique predecessor $S' = \delta^{-1}(S, \sigma)$ for each $\sigma \in \Sigma$.

- *If $\sigma = \emptyset$, i.e. $\sigma = \neg p$, then $S' = \emptyset$ because no subformula is propositionally consistent with σ and Xp cannot be added because $p \notin S$. Thus this will be a self loop.*
- *If $\sigma = p$, then $S' = \{p\}$ because p is propositionally consistent with σ and, as above, Xp cannot be added because $p \notin S$.*

The only new state is $S = \{p\}$. The predecessor S' are the following.

- *If $\sigma = \neg p$, then $S' = \{Xp\}$ because no atomic proposition is propositionally consistent with σ but since $p \in S$, the next rule applies.*
- *If $\sigma = p$, then $S' = \{p, Xp\}$ since σ is propositionally consistent with p and, as above, the next rule applies since $p \in S$.*

This resulted in two new states. First set $S = \{Xp\}$. The predecessors S' are the following.

- *If $\sigma = \neg p$, then $S' = \emptyset$ because p is not propositionally consistent with σ and the next rule cannot be used as $p \notin S$.*

- *If $\sigma = p$, then $S' = \{p\}$ as p is propositionally consistent with σ and, as above, the next rule does not apply.*

We did not get any new states so we continue with $S = \{p, Xp\}$. The predecessors S' are the following.

- *If $\sigma = \neg p$, then $S' = \{Xp\}$ because $p \in S$ but p is not propositionally consistent with σ.*

- *If $\sigma = p$, then $S' = \{p, Xp\}$ as $p \in S$ and p is propositionally consistent with σ.*

No new states were added, so the generation of the automaton is complete. The resulting automaton can be seen in Figure 8.

As can bee seen from the example, the algorithm is not optimal in any sense. The automaton produced has nice properties such as being reverse deterministic, which makes it an efficient choice for backwards symbolic search, but clearly the automata produced are unnecessarily big. The size of the automata is bounded by $2^{|cl(\psi)|}$. For this naive algorithm, the worst case behaviour will be realised often.

By examining a few automata we quickly notice that states which only differ in the labelling of the atomic propositions can be easily joined, as long as the transitions are relabelled. This is easily accomplished by allowing conjunctions of propositions and their negations to appear on the labels. This technique is also used by most translators from LTL to automata.

The optimisation above can lead to significant savings but it is still possible to be even more aggressive. If we allow arbitrary boolean expressions on the arcs we can simplify even further. All non temporal subformulas, i.e. the subformulas which have a 'and' or an 'or' at the root of their parse tree, can be handled by correct labelling of the arcs. Only temporal subformulas require that we add states to the automaton. The reason for this can be seen from original construction of Kupferman and Vardi. Only temporal formulas refer to other states than the current state.

Another way to look at the optimisations above is that they prune the possible state space of the automaton. Instead of every subset of $cl(\neg\psi)$ being a possible state of the automaton, only some subformulas need to be considered. We define the restricted closure $rcl(\psi)$ of a formula ψ in the following way:

- All temporal subformulas $\varphi \in cl(\psi)$, i.e. formulas with a temporal operator at the root of their parse tree, belong to $rcl(\psi)$.

- If a formula $X\varphi$ belongs to $rcl(\psi)$ then $\varphi \in rcl(\psi)$.

- If no other rule applies, then ψ belongs to $rcl(\psi)$.

Temporal subformulas must belong to the restricted closure because they refer to other than the current state. There are two special cases when other formulas are also included. The first case is the immediate subformula of a next-operator. In this case the subformula must be kept to ensure that it will true in the next state. The second case is when ψ is a propositional expression, when the reason is that $rcl(\psi)$ cannot be empty, because this will result in an automaton with no states.

The formulation of the optimised version looks quite similar to the normal algorithm, but there a two important changes. The normal closure is replace with the restricted closure. This of course results in much smaller automata because, the potential statespace is smaller. However, because only some formulas are allowed to label states having a simple inclusion test $\psi \in S$ will not work, because some of the subformulas of ψ may not be allowed to label S. Instead we must evaluate if the formulas in S satisfy ψ. We define $sat(\psi, S)$ in the following way:

- $sat(\mathbf{true}, S) = \mathbf{true}$
- $sat(\mathbf{false}, S) = \mathbf{false}$
- $sat(\psi, S) = \mathbf{true}$ if $\psi \in S$.
- $\psi = \psi_1 \vee \psi_2$: $sat(\psi, S) = \mathbf{true}$ if $sat(\psi_1, S)$ or $sat(\psi_2, S)$.
- $\psi = \psi_1 \wedge \psi_2$: $sat(\psi, S) = \mathbf{true}$ if $sat(\psi_1, S)$ and $sat(\psi_2, S)$.
- Otherwise $sat(\psi, S) = \mathbf{false}$

This will work correctly since temporal formulas are not eliminated from the restricted closure and the Boolean binary operators can always be evaluated by the rules described above. The inclusion test does not need to be replaced everywhere as in some cases we know that it is sufficient and there is no need for the more complex test. To fully reap the benefits from using the restricted closure, the way the atomic propositions label states is also changed. Only atomic propositions required by the restricted closure are allowed to label state. The algorithm as it is presented here will produce an automaton where there are many transitions from one state to another state. In an implementation these arcs would of course be joined to conserve memory. Note also the change in how the initial state is treated.

Input: A safety formula ψ in positive normal form.
Output: A finite automaton $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, F \rangle$.
proc opt-translate(ψ)
$F := \{\emptyset\}$; $\Sigma := 2^{AP}$;
$Q := X := F$;
while($X \neq \emptyset$) **do**
    S :="some set in X; $X := X \setminus \{S\}$
    for each $\sigma \in 2^{AP}$ **do**
        $S' := \sigma$;
        for each $\varphi \in rcl(\psi)$ **do**
            switch(φ) **begin**
                case $\varphi = \psi_1 \vee \psi_2$:

```
                if (sat(ψ1, S') or sat(ψ2, S')) then S' := S' ∪ {φ};
                case φ = ψ1 ∧ ψ2:
                if (sat(ψ1, S') and sat(ψ2, S')) then S' := S' ∪ {φ};
                case φ = Xψ1:
                if (ψ1 ∈ S) then S' := S' ∪ {φ};
                case φ = ψ1 U ψ2:
                if (sat(ψ2, S') or (sat(ψ1, S') and φ ∈ S)) then S' := S' ∪ {φ};
                case φ = ψ1 V ψ2:
                if (sat(ψ2, S') and (sat(ψ1, S') or φ ∈ S)) then S' := S' ∪ {φ};
            end
            if σ ∉ rcl(ψ) then S' := S' \ {σ};
        od
        if(sat(ψ, S')) then Q0 := Q0 ∪ {S'};
        δ = δ ∪ {(S', σ, S)};
        X := X ∪ {S'}; Q := Q ∪ {S'}
    od
od
```

The correctness of the algorithm relies on that the automaton will accept all informative prefixes and only them.

Theorem 34 *Given a formula $\neg\psi$, the opt-translate algorithm produces an automaton $\mathcal{A}_{\neg\psi}$ which accepts a prefix π iff π is informative for ψ.*

Proof:
Let $\pi = \sigma_0\sigma_1\ldots\sigma_n$ be an informative prefix for ψ and have a witness $L : \{0, 1, \ldots, n+1\} \to 2^{cl(\neg\psi)}$. The witness L induces a run $r : \{0, 1, \ldots, n\} \to Q$ of $\mathcal{A}_{\neg\psi}$ such that $sat(L(i), r(i)) = \mathbf{true}$. This will also hold for $L(n + 1)$, which is empty, and thus $\mathcal{A}_{\neg\psi}$ accepts the run.

For the other direction, consider an accepting run $r : \{0, 1, \ldots, n\} \to Q$ of $\mathcal{A}_{\neg\psi}$. The run induces a witness L such that $L(i) = \{\varphi \mid sat(\varphi, r(i)) \wedge \varphi \in cl(\neg\psi)\}$. □

The theoretical worst case bound for the size complexity of the automata produced by the optimised algorithm is somewhat better than for the basic algorithm. Let $tf(\psi)$ denote the temporal subformulas of ψ. It can be proven that, when the next operator is excluded, the construction is exponential only temporal formulas can cause exponential growth.

Theorem 35 *The number of states of $\mathcal{A}_\psi$ is bounded by $2^{|tf(\psi)|}$ when the next-free subset of LTL is considered.*

Proof:
For a set with cardinality n, the possible number of subsets is 2^n. All temporal subformulas belong to $rcl(\psi)$. When the next-operator is excluded, no other formula can belong to $rcl(\psi)$. Hence, the number of states is bounded by $2^{|tf(\psi)|}$. □

In the unrestricted case, when the next-operator is included, the theoretical bound is the same as for the unoptimised algorithm. In practice, the optimised algorithm will perform better in most cases. The performance of the algorithm in practice is studied experimentally in Section 8.

The algorithm above can produce fairly small automata in most cases. In some cases it will, however, produce automata which are quite big and have many initial states. The formulas which produce these usually have a binary boolean operator at root of the parse tree and a few next state operators suitably placed to minimise the pruning effect of the optimised algorithm. Many initial states often indicate that the automata has many different runs for the same words, presented in a redundant fashion. The most straightforward way to deal with this is to produce a determinise the automaton. This works surprisingly well in practice. Determinising the automaton usually makes it a lot smaller. We speculate that this is because the non-determinism is usually caused by boolean operator at the root of the parse tree, and as the arc expressions can express these constraints succinctly, there is no need to create new states to handle the non-deterministic runs. Determinising results in fewer states with more complicated arcs. This is usually a good trade off, because more states in the automaton can cause a multiplicative increase in time and space complexity while more complex arc expressions only slightly increase time complexity.

6.4 Finite Trace Semantics for LTL

Instead of dealing with LTL safety properties within the normal automata theoretic framework it is possible to use different semantics altogether for LTL. One possibility is to give semantics to the usual LTL operators over finite traces, as has been done in [26, 24, 13]. Another possibility is to use a past temporal logic [47] as has been done in [25].

Giving finite trace semantics to LTL can be seen as approximating the infinite trace semantics with finite traces. This view is especially apparent in the bounded semantics without a loop for bounded model checking [5]. The drawback of giving finite trace semantics to LTL is that some nice properties are lost, notably duality between some of the operators. E.g. in the finite trace case the duality $\neg(\psi_1\, U\, \psi_2) \equiv \neg\psi_1\, V\, \neg\psi_2$ no longer holds. Both Havelund and Rosu [26, 24] and Drusinsky [13] focus mostly on monitoring execution traces in real time while actual model checking is left out. It is an open problem what kind of complexity model checking for Kripke structures has using finite trace semantics.

Another possibility for defining LTL over finite traces is to use some form of past temporal logic. This approach is essentially taken by [25]. They present a dynamic programming algorithm which computes the satisfaction of a finite trace, given a past LTL formula. The procedure can both perform off-line monitoring and be implemented inline by instrumenting the source code of the program to be monitored. Also this approach is geared for monitoring single executions an not model checking Kripke structures. It is, however, trivial to extend to model checking.

All of the approaches presented above are more geared towards monitoring executions in real time rather than solving the traditional model checking problem, while this is the focus of this work. All approaches are related, but when focusing on a single execution, as is done in the monitoring case, some of the complexity baggage can be dropped. LTL can be model checked in linear time in the size of the formula and model, if the model is given as a

finite prefix and a loop. However, it is conceivable that one could construct a translation from finite trace semantics to finite automata. Havelund and Rosu [26] report that they have extended a normal LTL to Büchi automata procedure to include finite trace semantics. No details are given however.

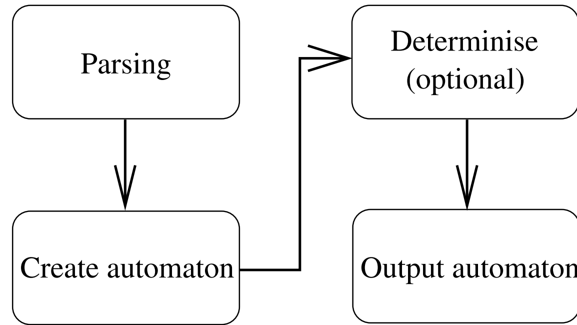

Figure 9: The different stages of the translation.

7 IMPLEMENTATION

We have implemented the optimised translation algorithm for safety LTL formulae and also the check for determining if a formula is pathologic.

The implementation is BDD-based and uses a BDD-library [44] developed by Jørn Lind-Nielsen. BDDs are used to represent sets of formulas efficiently. Especially the translation algorithm heavily employs manipulation of sets, which can easily be implemented with BDDs. However, BDDs can also incur a certain overhead making the algorithm slower in some cases compared to algorithms using simpler set representations.

The tool, *scheck*, has been implemented using ANSI C++ and it should compile on most platforms where a C++-compiler supporting templates is available. The implementation is available online under the terms of the GNU GPL from `http://www.tcs.hut.fi/~timo/scheck`.

7.1 Translation

The implementation of the translation algorithm is split into four separate stages. The first stage simply parses the input formula and transforms it into positive normal form. Optionally it can also perform some simple checks such as check for syntactic safety on the formula. The next stage builds a symbolic transition relation characterising the given formula. After the optimisations presented in the last section have been performed on the transition relation, the finite automaton is constructed. The third stage optionally performs some automata theoretic transformations, such as determinisation of the automaton. The fourth and the last stage outputs the automaton to the desired file or stream. Figure 9 shows a flow chart of the different stages.

The basic idea of the second stage is to construct a symbolic transition relation which adheres to the translation rules given in the previous section. Next, symbolic reachability analysis is used to construct the automaton. We now explain in detail the construction of the transition relation. The subformulas of the formula are enumerated in a preorder fashion (obtained from the parse tree). To represent the states, $2 * N$ BDD variables are reserved, where N is the number of subformulas, i.e. $|cl(\psi)| = N$. One variable describes if the subformula belongs to the current state and one variable is for the next state. Let $var(\varphi)$ denote the variable of φ and $var'(\psi)$ the next state variable for φ. The transition relation is the conjunction of the following

rules over each subformula φ of ψ.

Input: A formula ψ in positive normal form.
Output: A symbolic transition relation R.
proc $\mathit{transition}(\psi)$
$R := \mathbf{true}$;
for each $\varphi \in cl(\psi)$ **do**
  switch(φ) **begin**
    case $\varphi = \psi_1 \vee \psi_2$:
      $R = R \wedge (var(\varphi) \leftrightarrow (var(\psi_1) \wedge var(\psi_1)))$;
    case $\varphi = \psi_1 \wedge \psi_2$:
      $R = R \wedge (var(\varphi) \leftrightarrow (var(\psi_1) \vee var(\psi_1)))$;
    case $\varphi = p, p \in AP$:
      skip;
    case $\varphi = X\psi_1$:
      $R = R \wedge (var(\varphi) \leftrightarrow var'(\psi_1))$;
    case $\varphi = \psi_1 \, U \, \psi_2$:
      $R = R \wedge (var(\varphi) \leftrightarrow (var(\psi_2) \vee (var(\psi_1) \wedge var'(\varphi)))$;
    case $\varphi = \psi_1 \, V \, \psi_2$:
      $R = R \wedge var(\varphi) \leftrightarrow (var(\psi_2) \wedge (var(\psi_1) \vee var'(\varphi)))$;
  end
od

The atomic propositions are handeled by quantification. Using this transition relation we will get an automaton which corresponds to the original construction of Kupferman and Vardi. The optimisations mentioned in the previous section are easy to implement in the BDD framework as they correspond to variable quantification. Any non-temporal variable can be quantified away from the transition relation if it is not the top most formula or it does not have a next-operator as its parent formula. Using quantification will result in significantly smaller automata. It is somewhat difficult to treat the initial state as a special case in the BDD framework, which is the reason why the top most formula is not treated as efficiently as presented in the previous section.

The third stage of the translation is an optional determinisation of the automaton. Experiments show that in almost all cases determinisation makes the automaton smaller. A deterministic automaton also has shorter model checking times, because it causes less branching in the product automaton. See the section on experiments for more details. If the automata are to be used for monitoring executions of software, determinisation is mandatory. Alternatively the determinisation can be performed on-the-fly while monitoring. For monitoring to work, the current state of the automaton must be known at all times. Before the third stage, the automaton is converted to an explicit representation. Determinisation is easier when the automaton is in an explicit form. The arcs are still represented as BDDs since this allows easy manipulation of the arcs. Because we allow boolean expression on the arcs of the automaton, determinisation is somewhat more complicated than the usual algorithms for determinising an automaton.

The last stage of the translation outputs the automaton to a file or a stream. Here, the only challenge is to output the arc labelling, represented as BDDs, succinctly using only $\wedge$ and $\vee$ and negation in front of the propositions. Currently the implementation uses a fairly simple algorithm which outputs a BDD in disjunctive normal form.

7.2 Checking Pathologic Safety

Implementing, a check for if a formula is pathologic involves implementing an emptiness check for the intersection of two automata. Recall that an LTL formula ψ is pathologic iff $\mathcal{L}(\mathcal{A}_{\neg\psi}) \not\subseteq \mathcal{L}(\mathcal{A}^{true}_{\neg\psi})$. This is equivalent to that $\mathcal{L}(\mathcal{A}_{\neg\psi} \times \bar{\mathcal{A}}^{true}_{\neg\psi}) \neq \emptyset$.

In our implementation this will involve the following steps when we are given an LTL formula ψ.

1. Construct a Büchi $\mathcal{A}_{\neg\psi}$ automaton corresponding to the negation of ψ.
2. Construct a *deterministic* finite automaton $\mathcal{B}_{\neg\psi}$, which accepts all informative bad prefixes of ψ.
3. Interpret $\mathcal{B}_{\neg\psi}$ as a Büchi automaton and construct the complement $\bar{\mathcal{B}}_{\neg\psi}$.
4. Construct the product automaton $\mathcal{C} = \mathcal{A}_{\neg\psi} \times \bar{\mathcal{B}}_{\neg\psi}$.
5. Check if $\mathcal{L}(\mathcal{C}) = \emptyset$.

The reason we require that $\mathcal{B}_{\neg\psi}$ is deterministic is that complementing a nondeterministic Büchi automaton is complicated and has an exponential time lower bound [52], while complementing a deterministic Büchi automaton can be done in linear time [33]. The procedure outlined above is not optimal in complexity theoretical sense but it works quite well while the size of $\mathcal{B}_{\neg\psi}$ does not explode. An optimal approach would use alternating automata.

We have presented how all steps can be performed except the complementation of the deterministic Büchi automaton. The original formulation of Kurshan [33] is slightly complicated so we prefer to follow the presentation Vardi given in his lecture notes [61]. Let $\mathcal{A} = \langle \Sigma, Q, \delta, s_0, F \rangle$ be a deterministic Büchi automaton. The complement $\bar{A} = \langle \Sigma, \bar{Q}, \bar{\delta}, \bar{s_0}, \bar{F} \rangle$ can be computed with the following operations.

- $\bar{Q} = Q \times \{0\} \cup (Q - F) \times \{1\}$,
- $\bar{s_0} = s_0 \times \{0\}$,
- $\bar{F} = (S - F) \times \{1\}$, and
- for all states $q \in Q$ and symbols $a \in \Sigma$:

$$\begin{aligned}
\bar{\delta}((q,0),a) &= \begin{cases} \{(\delta(q,a),0)\}, & \text{if } \delta(q,a) \in F \\ \{(\delta(q,a),0),(\delta(q,a),1)\}, & \text{if } \delta(q,a) \notin F \end{cases} \\
\bar{\delta}((q,1),a) &= \{(\delta(q,a),1)\}, \quad \delta(q,a) \notin F
\end{aligned}$$

Clearly the the size of the complement is at most twice the size of the original automaton. It it an open question if there are more efficient ways to complement a deterministic Büchi automaton.

In the implementation we first compute explicit state representations of $\mathcal{A}_{\neg\psi}$ and $\mathcal{B}_{\neg\psi}$. Next, the deterministic automaton $\mathcal{B}_{\neg\psi}$ is complemented using the procedure above. Finally the product is computed and an emptiness check is performed using Tarjan's algorithm for finding SCCs. Everything except the complementation is standard model checking technology. Because the implementation for constructing $\mathcal{A}_{\neg\psi}$ is simple, the tool has an interface for using an external translator to construct the Büchi automaton $\mathcal{A}_{\neg\psi}$.

8 TRANSLATION EXPERIMENTS

In order to evaluate the ideas presented in this work we conducted some experiments. There were three questions we were especially interested in.

1. How well does the size of the automata scale in the average case compared to other translators?

2. How well does the implementation work in practical situations?

3. Is checking formulas for pathologic safety feasible?

Three different tests were used to investigate the tool. The two first tests are based on random formulae and random state spaces and the third test measures model checking performance on different system models.

Three translation tools were used as reference: a state of the art tool by Paul Gastin and Dennis Oddoux [20], the translator packaged with the Spin tool [29], version 3.4.16, and an efficient implementation of the algorithm described in [22] by Mäkelä, Tauriainen and Rönkkö [45]. In the following we refer to the tool of Gastin and Oddoux as *ltl2ba*, to the tool of Mäkelä et al. as *lbt* and to the translator of Spin simply as *spin*. As earlier mentioned, our tool is referred to as *scheck*. All of the reference tools are translators which can translate any LTL formula to Büchi automaton.

For the two first tests which involve random formulae and random statespaces we have used the LTL to Büchi translator test bench by Tauriainen and Heljanko [57]. The tool includes facilities for randomly generating LTL formulae and measuring different statistics such as the size of the generated automaton and generation time. For the third, test we interfaced the tools with the reachability analyser Maria [46].

The first test generates random syntactically safe formulae. Most safety formula encountered in practice will probably be of this form. For instance, all examples in Section 3 of this work are syntactically safe. The idea is to measure how well the tools can cope with typical safety formulae. Statistics measured are the number of states and transitions in the automata produced, the time to generate the automata and the size of the product of a random state space of twenty states and the automaton. The number states and transitions in the generated automaton and generation give an indication of the general performance of the translator while the size of the product statespace is depends on both the size of the generated automaton and the structure of the automaton. Automata which have small product statespaces can at an early stage 'decide' if the current sequence under inspection cannot satisfy the given formula.

The second test is in a sense a generalisation of the first. Now we randomly generate any formula and use the implemented check for pathologic formulae to see if its a safety formula which can be used in the tests. This means that many generated formulae will be rejected but we will also test formulae which are not covered by the syntactic fragment. The measures are the same as in the first test. The main purpose of this test is to prove the feasibility of identifying pathologic formulae. This test is only performed for *scheck*, as none of the other tools can check if a formula is pathologic.

The third test takes a more practical approach. We use the model checker and reachability analyser Maria [46] to benchmark the translators. We have modelled several well-known distributed algorithms with the Maria tool and measure how fast some safety properties can be model checked using the different translators. Maria uses simple reachability analysis if the specification is given as an finite automaton and it uses a Tarjan's SCC algorithm based model checking algorithm [40] for Büchi automata. We also measure the size of the product state space. The idea of the test is to give us some insight into how well the tools perform with 'real' models and if we can gain anything by treating safety properties as a special case in practice.

All tests were conducted on a with a machine with a 266 MHz Pentium II processor with 128 MB of memory. The machine runs Debian GNU/Linux 3.0. All of the programs have been compiled using gcc version 2.95.4. In all tests the *scheck* tool was set to generate deterministic automata, because initial tests had shown that this produced the smallest automata. We have collected the models used and other relevant files such as configuration files to a webpage: `http://www.tcs.hut.fi/~timo/scheck/licthesis`.

8.1 Random Formulae

For both tests based on random formulae we used the tool of Tauriainen and Heljanko, presented in [57]. With the tool it is possible to generate random formulae which contain only certain LTL operators, which is of course very useful when we want to restrict ourselves to the syntactically safe fragment of LTL. There are four parameters which control the generation of the random formulae.

- The number of nodes in the parse tree of the formula.
- The number of *different* atomic propositions which can occur in a formula.
- Priorities for boolean constants and atomic propositions.
- Priorities for temporal operators and logical connectives.

The generation algorithm is such that the priorities for the boolean constants and atomic proposition does not affect the number of temporal and logical connectives in the formula and vice versa. In other words, Boolean constants compete with the atomic propositions for occurrences independently of temporal and logical connectives.

The objective of the tests was to see how the tools scale when we increase the length of the formulae. Three sets of formulas were generated for each length, starting from five up to 22. The results for each length were averaged over the three sets. In all tests the maximum number atomic propositions was six. Atomic propositions were also preferred over boolean constants. The priorities for true and false were set at three and at 15 for atomic propositions.

Syntactically Safe Formulae
To generate syntactically safe formulae the priorities were set to 25 for until, 15 for next, finally, 'and', 'or', and zero for all other temporal and logical connectives. We generated three sets of 1000 formulas for each length.

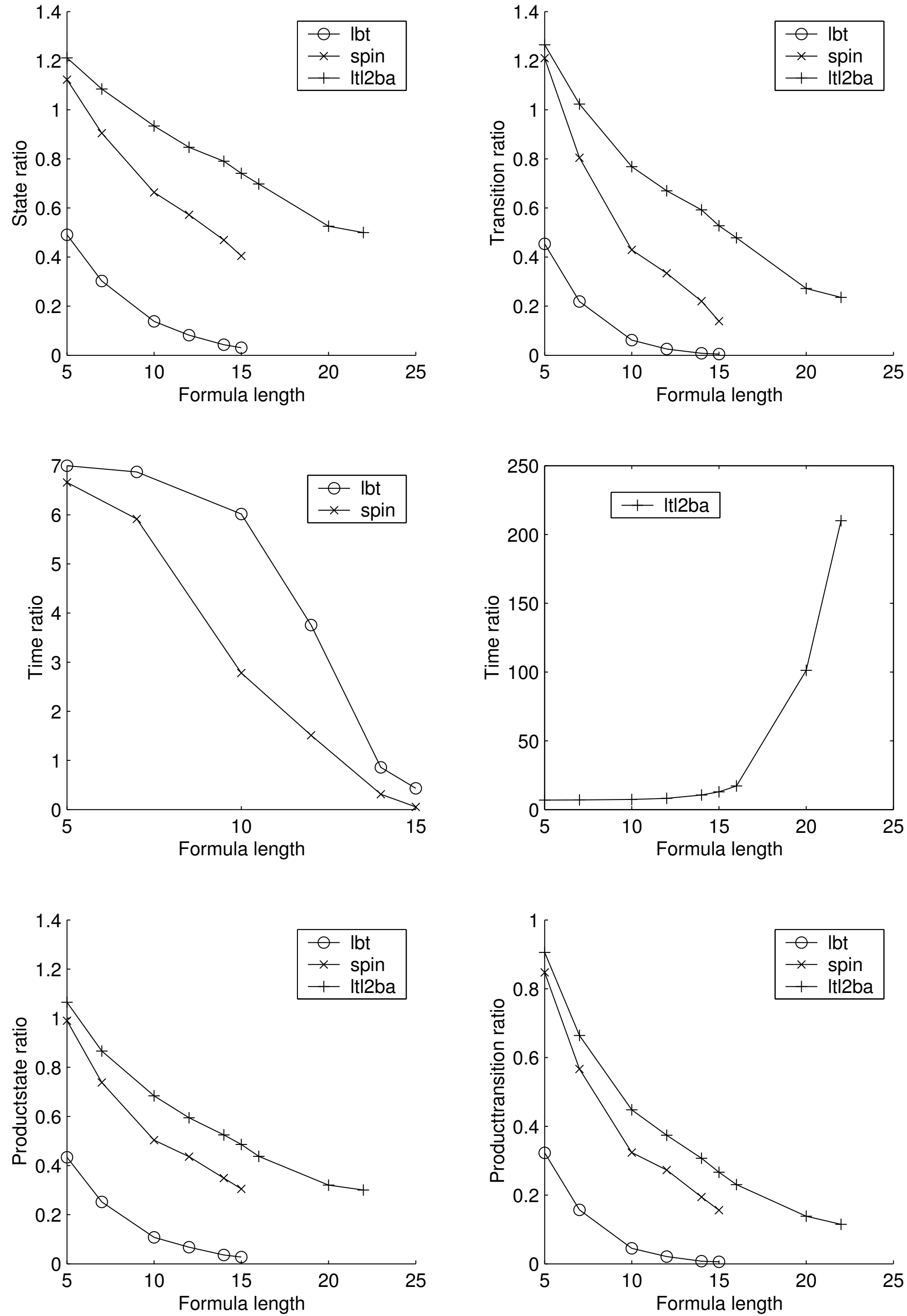

Figure 10: Comparison of the tools with syntactically safe formulae.

We measured the size of the generated automata, the time to generate the automata and the size of the product statespace. To compare to the other tools we computed the mean $E(M(proc, L))$ over the three runs for each procedure $proc$, formula length L and measure M. The ratio of the means $\frac{E(M(scheck,L))}{E(M(proc,L))}$ have been computed in Figure 10.

When we compare the size of the automatons generated, i.e. number of states and transitions, *scheck* seems to be very competitive. Especially the procedures based on [22] cannot compete well. When the formulas are short *spin* and *ltl2ba* are able to compete, but when the length of the formulas grows, *scheck* clearly scales better than the other tools. At the time when the measurements were made *scheck* did not check for a "sink state" in its deterministic automata. If the tests were rerun, *scheck* would probably narrow down the small lead *spin* and *ltl2ba* has in short formulae. Long formulae are not affected as much by the removal of one sink state. Note that in the number of transitions *scheck* scales even better compared to the other tools. One reason is probably that *scheck* generates deterministic automata.

Generation time gives a different picture of how well the tools perform. The tools based on [22] have an advantage with short formulae but do not scale as well. *ltl2ba* is however much faster than *scheck* in all cases. It scales better and it is faster for short formulae. It is possible that the BDD implementation of *scheck* here gives the other tools an competitive advantage. The statistical correlation between the number generated transitions in the automata and the time used is almost one for the three other tools while it is about 0.7 for *scheck*. In other words, for the other tools the number of states produced directly affects the running time while for *scheck* the relation is not as straight-forward. Possibly, this is because BDDs can sometimes be a suboptimal choice for set representation and give an overhead which the compactness of BDDs do not compensate.

The tool in [57] generates a random statespace of twenty states for each formula. We compute the product of this and the generated automaton. Comparison of the size of the product statespaces gives an indication on how well the automata "guide" the model checker and how well the automaton can determine when enough has been generated of the system to decide the property. In essence it gives an evaluation of the structure of the automata. Automata which generate small product statespaces have a good structure and can quickly determine that a property is (un)satisfiable. Here we expected *scheck* to do well, because the automata it generates are deterministic. The results also confirm this. *scheck* generates smaller product statespaces than all three other tools.

General Formulae

To generate general formulae we set the priorities for all temporal connectives to a non-zero value. The check for pathologic safety formulas in *scheck* was enabled so that *scheck* would not try to translate liveness formulae or pathologic safety formulae. The procedure is sound for liveness formulas, but incomplete, so there is no need to check for safety first. We used *ltl2ba* to generate the Büchi automata needed when checking if a formula is pathologic.

One hundred formulas and their negation were generated for each length

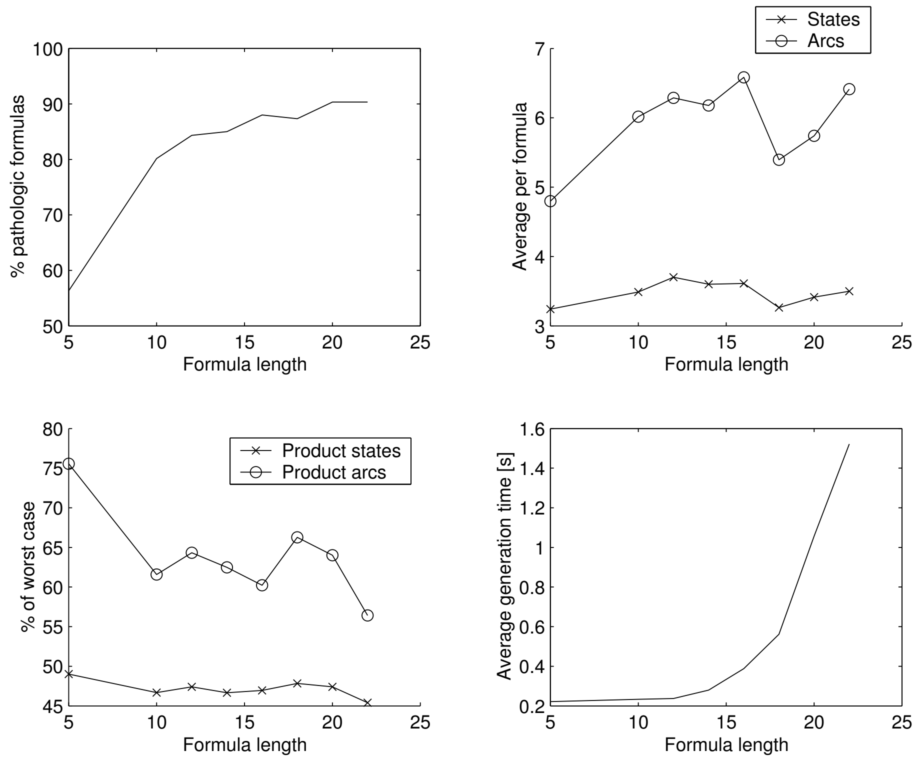

Figure 11: *scheck* performance for general formulae.

ranging from five to 22. The results for *scheck* are summarised in Figure 11. As can been seen from the first plot, most of the generated pathologic or liveness formulas and the percentage grows when formula length increases. This is of course not surprising as the temporal operators often describing liveness properties are more likely to occur. As can be seen in the second plot, the size of the automata grows very slowly with increasing formula length. Only the non-pathological formulas are taken into account. As so many formulas has been rejected we cannot conclude that general formulas are easier than syntactically safe formula. There is not enough data for this. The figures for the product statespace show how much of the potential product statespace was generated when the automaton was synchronised with a random statespace. Again *scheck* is able to keep the product statespace small but here again we cannot conclude anything definite because so many of the formulas were rejected. The generation time shows the familiar exponential increase which usually manifests itself sooner or later when solving PSPACE-complete problems. Our suspicion is that why *scheck* can require exponential time but not generate exponential automata is due to inefficiencies when using BDDs in *scheck*. One conclusion which is not affected by high rejection ratio of the formulas is that *scheck* can clearly scale well when identifying pathologic formulas.

8.2 Model Checking Case Studies

To compare the performance of the tools in a practical setting, the four tools were connected to the Maria tool. We modelled three distributed algorithms which all had a parameter, such as the number of processes, which could

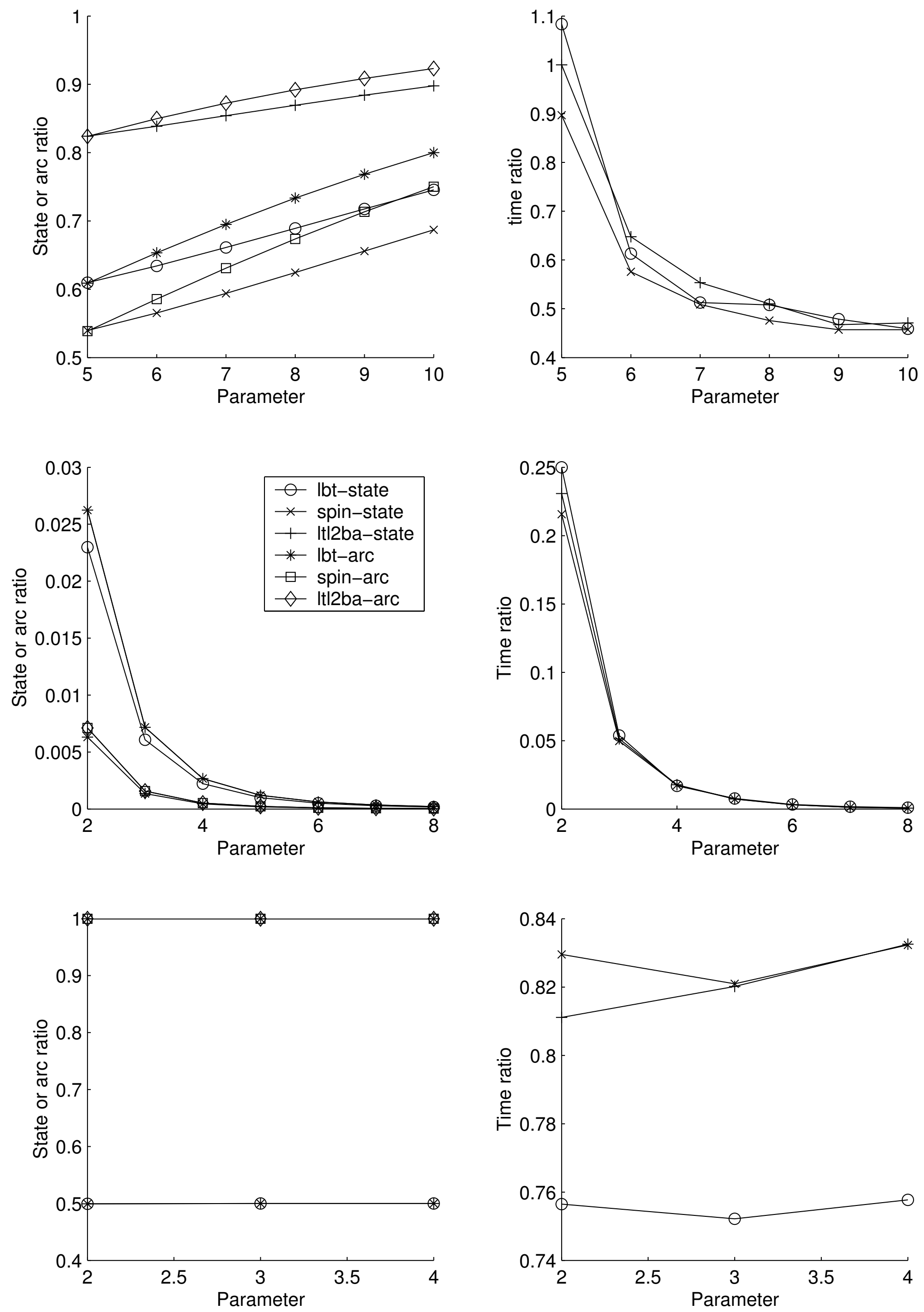

Figure 12: Comparison of the tools with practical models. Top: the echo algorithm, middle: the alternating bit protocol and bottom: readers writers problem.

be increased. On each model we checked a safety property three times for each parameter value. We report the average of the three runs. The statistics compared are size of the product statespace, which gives an indication of the memory use of the model checker, and the time used for model checking.

Because the model checking algorithm is different for *scheck* than the other tools, this comparison more investigates the benefit of being able to use simpler algorithms on safety properties rather than directly comparing the tools themselves. If the model checking algorithm for the other tools would e.g. be the nested depth-first search algorithm of [10], the results would be slightly different.

The three models used were:

1. A model of the echo algorithm with a parameterised number of participants.

2. A model of the alternating bit protocol with varying parameterised channel sizes.

3. A model of the concurrent readers, exclusive writing problem with a parameterised number of processes.

The parameter we modify in the echo algorithm is the number of nodes participating in the network. For the alternating bit protocol the channel buffer size is changed and for the readers writers problem we set the number of readers, which has been fixed to be equal to the number of writers. For two of the models, the echo algorithm and readers writers problem, a property which is satisfied was model checked, while a failing property was model checked for the alternating bit protocol.

In Figure 12 we have plotted the ratio of the averages for the different measures, i.e. the average of the result for *scheck* divided by the average of the result for the other tool. The first row shows the results for the echo algorithm, the second row for the alternating bit protocol, and the third row for the readers writers problem.

In all cases *scheck* produces smaller or equally big product statespaces. Model checking using it is always faster than the other tools. For the echo algorithm it is interesting to note that although the ratio of the product statespace shows that the other tools are catching up, the time ratio still continues to decrease indicating the opposite. Holtzmann and Etessami observed [18] that when the property fails, an algorithm which is not heavily optimised can perform better than optimised algorithms. This can also been seen in these experiments for the alternating bit protocol. Especially in these cases, *scheck* is far superior to the other tools. This can probably be attributed to that model checking can stop immediately after the final state has been entered in the automaton, as there is no need to find a loop. For the readers writers problem the performance of the other algorithms is almost on par with *scheck*. This is the reason the first figure is a bit unclear. Both *spin* and *ltl2ba* produce the same result as *scheck*, while *lbt* produces twice as many states and arcs. This is probably due to that all algorithms are able to produce nearly optimal automata for the simple property verified.

9 DISCUSSION

One of the questions this work set out to answer was is it worth the effort to treat safety as a special case when model checking systems. In light of the results presented in this work, the question can be answered affirmatively. The implementation of the translation procedure presented in this work, *scheck*, produces smaller automata than the state of the art of the LTL to Büchi automata translators. In some cases the difference is exponential other times negligible. Clearly also the resulting product statespaces are smaller for *scheck*. This is probably because *scheck* produces deterministic automata. The fact that determinising would result in much smaller automata came as a pleasant surprise. It is a well-known that determinising a non-deterministic automaton can result in an exponentially larger automaton. Safety properties can be expressed using reverse deterministic automata. Apparently, the expressiveness of non-determinism is not needed in the forward case either. Automata generation time was the one area where the results for *scheck* were disappointing. Although *scheck* generates the automaton for almost any formula in a few seconds, this is quite slow compared ltl2ba which in most cases would only use a few hundreths of a second. One of the reasons could be that *scheck* uses BDDs to manage sets, which sometimes can cause overhead. A non-BDD implementation would probably perform better.

One of the most important tests for any tool is how well it performs in practical situations. In this category *scheck* was competitive. The resulting product statespaces for *scheck* were always smaller or at least as small as for the other tools. Especially if a counterexample existed, model checking using *scheck* was much faster. The comparisons were made against a model checking algorithm based on Tarjan's SCC algorithm, which means the results could be different if the comparison was done against a tool using the nested depth-first algorithm [10]. As the number of different models used in the experiment was small, more experiments are needed to confirm our results. There are also model checking algorithms where checking Büchi acceptance is more complex than simple reachability. One example is the LTL_{-X} model checker for net unfoldings presented in [17]. Using a special algorithm for the safety subset has been proposed as a much simpler alternative [27].

The results presented for CPN abstractions in this work makes incremental development more feasible. Because many safety properties can be verified on the abstract designs, correctness of the refined designs is easier to ensure. The results can also be seen to suggests how a design should be abstracted in order to preserve safety properties. Unfortunately, fully automatic abstraction is probably not feasible, so human intervention will be required when abstraction is done. Correctness of each refinement/abstraction step can however be ensured as demonstrated by the implementation described in [42]. It could be interesting to see how well this kind of design and verification methodology would perform in practice. From our perspective the most interesting question is, could safety properties for very large designs be successfully verified in this way.

In order to be able to benefit from treating safety properties as a special case we must be able to recognise safety formulae. There are two ways in

which this can be done. Either we only use the syntactically safe subset of LTL, which is easy to recognise or we check if a formula is pathologic. The feasibility of checking if a formula is pathologic mostly hinges on the feasibility of producing deterministic finite automata from the properties which then are interpreted as deterministic Büchi automata. If the automata are not deterministic we must complement non-deterministic Büchi automata, a non-trivial task. However, as this work has shown, producing deterministic finite automata from LTL safety properties is feasible. The experiments also confirm that checking if a formula is pathologic is feasible. This means that both options for recognising safety properties are available and can be used. Especially a tool which only supports the fundamental temporal operators will benefit from being able to check if a formula is pathological because there are some fairly obvious safety properties which are not syntactically safe.

The worst case bound for the size of the automata in this work is still $2^{|cl(\psi)|}$, although it is simpler for the case where the next-operator is omitted. For some algorithms, the bound $O(2^{|tf(\psi)|})$ has been proven [11, 25]. We conjecture that this is also possible for the automata construction for safety properties. An optimisation of the treatment of the next-operator subformulas could perhaps facilitate this change.

To produce even smaller automata faster than *scheck*, another approach is probably required. It would be interesting to see if starting from alternating automata as in [20] could facilitate an efficient translation.

ACKNOWLEDGEMENTS

The author thanks Keijo Heljanko for fruitful discussions and comments on the paper. The financial support of Helsinki Graduate School in Computer Science and Engineering, the Academy of Finland (project 47754), the Wihuri Foundation and Tekniikan Edistämissäätiö (Foundation for Technology) is gratefully acknowledged.

References

[1] J.R. Abrail. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verication. Model-Checking Techniques and Tools.* Springer, 2001.

[4] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching time model checking. In *Computer Aided Verification (CAV'97)*, volume 818 of *LNCS*, pages 142–155. Springer, 1994.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Constructions and Analysis of Systems (TACAS'98)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[6] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[8] E.M. Clarke and E.A. Emerson. Design and synthesis of syncronization of skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[9] E.M Clarke and B-H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1637–1790. Elsevier, 2001.

[10] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[11] J-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceeding of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *LNCS*, pages 253–271, Berlin, 1999. Springer.

[12] M. Daniele, F. Giunchiglia, and M.Y Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 249–260, Berlin, 1999. Springer.

[13] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[14] A.E. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983.

[15] A.E. Emerson and Clarke E.M. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[16] E.A. Emerson and C-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[17] Javier Esparza and Keijo Heljanko. A new unfolding approach to LTL model checking. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, volume 1853 of *LNCS*, pages 475–486. Springer, July 2000.

[18] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Concurrency Theory (CONCUR'2000)*, volume 1877 of *LNCS*, pages 153–167. Springer, 2000.

[19] B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. In *Proceedings of the First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[20] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Computer Aided Verification (CAV'2001)*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.

[21] M.C.W. Geilen. On the construction of monitors for temporal logic properties. In *RV'01 - First Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[22] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[23] Richard Goering. Model checking expands verification's scope. *Electric Engineering Today*, February 1997.

[24] K. Havelund and G. Rosu. Java pathexplorer — A runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS'01)*, 2001.

[25] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.

[26] Klaus Havelund and Grigore Rosu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, Research Institute for Advanced Computer Science, May 2001.

[27] Keijo Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets.* PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering, February 2002.

[28] J. Helovuo and S. Leppänen. Exploration testing. In *Application of Concurrency in System Design (ACSD'2001)*, pages 201–210. IEEE, 2001.

[29] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[30] K. Jensen. *Coloured Petri Nets*, volume 1. Springer, Berlin, 1997.

[31] O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. In *Israeli Symposium on Theory of Computing and Systems*, pages 147–158. IEEE Computer Society Press, 1997.

[32] O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[33] R.P. Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Science*, 35:59–71, 1987.

[34] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeon University Press, Princeton, New Jersey, 1994.

[35] Charles Lakos. On the abstraction of coloured Petri nets. In *Application and Theory of Petri Nets (ICATPN'97)*, volume 1248 of *LNCS*, pages 42–61. Springer, 1997.

[36] Charles Lakos. Composing abstractions of coloured Petri nets. In *Application and Theory of Petri Nets 2000*, volume 1825 of *LNCS*, pages 323–345. Springer, 2000.

[37] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[38] L Lamport. Logical foundations. In *Distributed Systems - Methods and Tools for Specification*, volume 192 of *LNCS*. Springer, 1982.

[39] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.

[40] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *Applications and Theory of Petri Nets (ICAPTN'2001)*, volume 2075 of *LNCS*, pages 242–262. Springer, 2001.

[41] G.A. Lewis. *Incremental Specification and Analysis in the Context of Coloured Petri Nets*. PhD thesis, Department of Computing, University of Tasmania, 2002.

[42] G.A. Lewis and C.A. Lakos. Incremental state space construction for coloured petri nets. In *Application and Theory of Petri Nets 2001*, volume 2075 of *LNCS*, pages 263–282. Springer, 2001.

[43] O. Lichtenstein and A. Pnueli. Checking that finite state programs satisfy their linear specifination. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[44] Jørn Lind-Nielsen. Buddy - A binary decision diagram package, 2000. `http://www.itu.dk/research/buddy/`.

[45] M. Mäkelä, H. Tauriainen, and M. Rönkkö. lbt: LTL to Büchi conversion, 2001. `http://www.tcs.hut.fi/Software/maria/tools/lbt/`.

[46] Marko Mäkelä. Maria: modular reachability analyser for algebraic system nets. In *Application and Theory of Petri Nets 2002*, volume 2360 of *LNCS*, pages 434–444. Springer, 2002.

[47] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[48] Vardi M.Y. and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[49] J. Padberg, K. Hoffmann, and M. Gajewsky. Stepwise introduction and preservation of safety properties in algebraic high-level net systems. In *Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *LNCS*, pages 249–265. Springer, 2000.

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

[51] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1981.

[52] S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institute of Science, 1989.

[53] A.P. Sistla. Safety, liveness, and fairness in temporal logic. *Formal Aspects in Computing*, 6:495–511, 1994.

[54] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

[55] F. Somenzio and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proceedings of the International Conference on Computer Aided Verification (CAV2000)*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000.

[56] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[57] Heikki Tauriainen and Keijo Heljanko. Testing LTL formula translation into Büchi automata. *STTT - International Journal on Software Tools for Technology Transfer*, 4(1):57–70, 2002.

[58] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.

[59] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier, 1990.

[60] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.

[61] M. Vardi. Automata-theoretic approach to design verification. webpage, 1999. `http://www.wisdom.weizmann.ac.il/~vardi/av/notes/lec2.ps`.

[62] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[63] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, pages 238–266. Springer, 1996. LNCS 1043.

[64] M.Y. Vardi. Branching vs. linear time: Final showdown. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2001)*, pages 1–22. Springer, 2001.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A63	Nisse Husberg, Tomi Janhunen, Ilkka Niemelä (Eds.) Leksa Notes in Computer Science. October 2000.
HUT-TCS-A64	Tuomas Aura Authorization and Availability - Aspects of Open Network Security. November 2000.
HUT-TCS-A65	Harri Haanpää Computational Methods for Ramsey Numbers. November 2000.
HUT-TCS-A66	Heikki Tauriainen Automated Testing of Büchi Automata Translators for Linear Temporal Logic. December 2000.
HUT-TCS-A67	Timo Latvala Model Checking Linear Temporal Logic Properties of Petri Nets with Fairness Constraints. January 2001.
HUT-TCS-A68	Javier Esparza, Keijo Heljanko Implementing LTL Model Checking with Net Unfoldings. March 2001.
HUT-TCS-A69	Marko Mäkelä A Reachability Analyser for Algebraic System Nets. June 2001.
HUT-TCS-A70	Petteri Kaski Isomorph-Free Exhaustive Generation of Combinatorial Designs. December 2001.
HUT-TCS-A71	Keijo Heljanko Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets. February 2002.
HUT-TCS-A72	Tommi Junttila Symmetry Reduction Algorithms for Data Symmetries. May 2002.
HUT-TCS-A73	Toni Jussila Bounded Model Checking for Verifying Concurrent Programs. August 2002.
HUT-TCS-A74	Sam Sandqvist Aspects of Modelling and Simulation of Genetic Algorithms: A Formal Approach. September 2002.
HUT-TCS-A75	Tommi Junttila New Canonical Representative Marking Algorithms for Place/Transition-Nets. October 2002.
HUT-TCS-A76	Timo Latvala On Model Checking Safety Properties. December 2002.

ISBN 951-22-6265-7
ISSN 1457-7615