CONTENTS

CHAPTER 1
INTRODUCTION

Program verification is a well-studied area in computer science. For the class of programs usually described as "sequential" programs, many formalisms have been developed to verify their correctness. These programs usually terminate and follow the following general pattern: they initially accept some input, perform some computation, and then yield a final output. (Eg. a program which sorts a given list; a program to implement some graph algorithm). They can thus be viewed as a transformation from initial states to final states. For such systems, formalisms such as Hoare's logic can be used to check their correctness.

With the advent of concurrent programming, there has been increasing interest in a class of programs that operate continuously, or ideally, operate without terminating. Operating systems, network communication protocols and air-traffic control systems are examples in this class. The normal behaviour of these programs is a non-terminating computation, which maintains an ongoing interaction with the environment. Such programs are described as *reactive systems*. During their interaction with the environment, intermediate outputs of the program could influence subsequent inputs to the program. Typical examples in this class are concurrent and distributed programs. Concurrent programs can be thought of as reactive systems as one can view each process as interacting with the other processes. The conceptual complexity of concurrency increases the likelihood of a program containing errors and hence automatic verification methods to check if a program meets its specification is of considerable importance. Temporal logic is a tool to specify and verify correctness of such programs.

The class of Modal logics were originally developed to study the different "modes" of truth. Temporal logic is a special type of modal logic; it provides a formal system for qualitatively describing and reasoning about how truth of assertions changes over time. In [12] Pnueli gave reasons as to why temporal logic might be a useful tool to specify and verify correctness of continuously running, nonterminating programs. Some operators in temporal logic such as "sometime" and "always" are useful in describing the program's behaviour over a period of time.

Temporal logic is now an active area of research. It has been used as a tool in specification, verification and synthesis of concurrent programs. The variant of temporal logic initially presented by Pnueli is the linear version [12, 13, 10, 2, 5]. In this version, time is viewed as linear, i.e. each moment has a unique successor. Thus this logic is interpreted over linear sequences. In this presentation, we'll deal with the propositional linear-time temporal logic (LTL).

A linear specification can be written as an LTL formula. The formula describes the computations of the program – a program *satisfies* the specification if all of its computations satisfy the formula.

There are many ways by which one can check if a concurrent program satisfies a specification written as an LTL formula. One such effective way is known as *model checking*. Model checking works on finite-state programs and by viewing the program as a structure over which the formula is interpreted, checks if the program satisfies the formula [2, 9].

There is an intimate connection between linear-time temporal logic and the theory of automata on infinite words. This connection was first explicitly pointed out in [19] (see [18]). It turns out that model-checking can be done with this automata-theoretic approach.

The theory of automata on infinite sequences was established in the sixties by Büchi [1]. Büchi was interested in decision problems in mathematical logic - he showed that finite automata provide a normal form for the monadic second-order theory of one successor.(See [15]).

Vardi and Wolper showed the basic result that temporal logic formulas can be viewed as finite-state acceptors of sets of infinite computations. In fact, they gave a construction (which henceforth we'll refer to as the

Vardi-Wolper construction), which, given an LTL formula α , will construct a Büchi automaton that accepts precisely the computations satisfied by the formula [18, 17]. This automaton is of size $2^{O(|\alpha|)}$, where $|\alpha|$ is the size of α.

An LTL formula is said to be *satisfiable* if there exists a model which satisfies it. The Vardi-Wolper construction shows how to reduce the problem of satisfiability of an LTL formula to the problem of non-emptiness of Büchi automata (with, of course, an exponential blow-up).

Now, the computation of any reactive system is essentially an infinite sequence of states. Also, every state is described by a finite set of atomic propositions. So, the program itself can be viewed as a Büchi automaton and the language accepted by this automaton is precisely the runs of the program.

The automata theoretic approach to model-checking then is as follows: Given a program P and an LTL formula α, one can model-check whether P satisfies α by the following method :

1. Build the Büchi automaton $\mathcal{A}_{\sim\alpha}$ for the negation of the formula, i.e. for $\sim\alpha$. This is called the *property automaton.*
2. Model the global behaviour of P as a Büchi automaton, $\mathcal{A}_P$. This is called the *program automaton.*
3. Construct the product automaton of these automata which will accept an infinite word iff it is accepted by both automata
4. Check the product automaton for emptiness. The program satisfies the specification if and only if this language is empty.

The non-emptiness of a Büchi automaton is basically a reachability problem and is known to be decidable in linear time [3] and is also known to be $NLOGSPACE$-complete under LOG-space reductions [18].

Theoretically, the model-checking as well as the satisfiability problems for LTL is known to be $PSPACE$-complete [3]. In practice, applications to model checking face two complexity related limits :

- The size of both the program and property automata, since the execution time is proportional to the product of the sizes of these automata.
- The size of that part of the product automaton that has to be kept in main memory in order to check for emptiness.

The second problem was addressed in [4], where the authors propose a simple cycle-detection method (rather than checking for strongly connected components). This uses a simple depth-first search (DFS) strategy and only needs a small part of the automaton to be in main memory : the part corresponding to the computation of the depth-first search it is currently exploring. Such a strategy could thus work on a product-automaton constructed "on-the-fly", i.e. parts of the automaton can be constructed as and when required. Hence, if the property that is being checked does not hold, the algorithm could detect this very early after visiting only a small part of the state-space. To construct the product-automaton on-the-fly, it would be desirable to construct both the program and property automata in an on-the-fly manner.

When modeling a concurrent system as a Büchi automaton, the standard approach is to describe the concurrency as nondeterminism. Thus the program automaton describes all interleaved executions of the concurrent program. This results in an exponential growth of the automaton with the number of processes in the program. Thus, the concept of generating the state-space on-the-fly is a very pragmatic approach to this problem - in such a verification, only the state-space required is constructed and as soon as the algorithm detects a computation which does not meet the specification, it terminates. Of course, in the worst case, (i.e. when the program is correct!), the algorithm will construct the whole state-space.

The program automaton can be constructed on-the-fly as it is usually a simple product of the state-spaces of the processes. This report deals with the on-the-fly construction of the property automaton. Such an algorithm was presented in [6]. In this report, we'll extend this work to action-based linear-time temporal logic. We want such an extension so that we would be able to give efficient on-the-fly algorithms for other logics (mainly Product LTL introduced in [14]). We give these details in the conclusions.

The algorithm we present is a pragmatic approach to construct an automaton from a temporal logic formula. We give its proof of correctness in a slightly different way and in a more detailed fashion than in [6] - we show that the construction is correct by associating it with the Vardi-Wolper construction.

The algorithm is presented in a way such that it can be used for generating the program automaton on-the-fly during a *depth-first search*. The result is that we obtain a verification procedure where the product-automaton is constructed on-the-fly (by constructing both the program and property automata on-the-fly), during a depth-first search which checks for emptiness.

The rest of this report is organised as follow. In the next chapter, we give the definitions and describe the action-based LTL we'll be working with. Then, we give an adaptation of the Vardi-Wolper construction to action-based LTL. Then, we present our adaptation of the algorithm presented in [6] to action-based LTL and give details of how it can be used to solve the satisfiability and model-checking problems. We then conclude with a discussion on how the work presented here can be extended to partial-order based logics in future work.

CHAPTER 2

PRELIMINARIES

2.1 Büchi Automata on infinite words

Büchi automata are nondeterministic finite automata equipped with an acceptance condition that is appropriate for infinite words. Let ω denote the set of natural numbers. Let us also fix a finite alphabet Σ. An infinite word (or ω-word) over Σ is a function $\tau : \omega \rightarrow \Sigma$. Thus an infinite word is just an infinite sequence of letters from Σ.

An ω-word is accepted by the automaton if it can read the word from left to right and meet some final state infinitely often.

Definition 2.1 A Büchi automaton over an alphabet Σ is of the form $\mathcal{A} = (Q,\ Q_{in},\ \rightarrow,\ F)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is the set of initial states, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states.

A *run* of A on an ω-word $\sigma = a_0 a_1 \ldots$ from Σ^ω is an infinite sequence $\rho = q_0 q_1 \ldots$ such that $q_0 \in Q_{in}$ and $\forall i \geq 0,\ q_i \xrightarrow{a_i} q_{i+1}$.

A run $\rho = q_0 q_1 \ldots$ is said to be *accepting* if $\mathrm{Inf}(\rho) \cap F \neq \emptyset$, where $\mathrm{Inf}(\rho) = \{q \in Q \mid \exists^\omega n : q_n = q\}$ is the set of states met infinitely often by the run.

$\mathcal{A}$ accepts $\sigma \in \Sigma^\omega$ if there is an accepting run of $\mathcal{A}$ on σ. Now, $\mathcal{L}(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \mathcal{A}\ \textit{accepts}\ \sigma\}$ is the ω-language recognized by $\mathcal{A}$.

Example : Let $\Sigma = \{a,\ b\}$. Let the state-space of the Büchi automaton be as below :

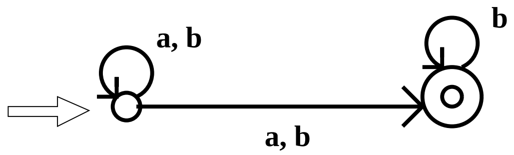

FIG. 2.1 : Example of Büchi automaton

[The final state is the state with concentric circle and the initial state is the one pointed to by the double-arrow]

Then, $\sigma \in \mathcal{L}(\mathcal{A})$ iff σ has only a finite number of a's.

We'll introduce a different kind of Büchi automata called Labeled Generalized Büchi Automata (LGBA), which will be convenient for our purposes.

Definition 2.2 A *Generalized Büchi automaton* (GBA) over Σ is a structure $\mathcal{A} = (Q, Q_{in}, \rightarrow, \mathcal{F})$ where

Q is a finite set of states

$Q_{in} \subseteq Q$ is a set of initial states

$\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation

$\mathcal{F} \subseteq 2^Q$ is a set of sets of final states $\{F_1, \ldots F_r\}$

As before, given an ω-word $\sigma = a_0a_1 \ldots \in \Sigma^\omega$, a *run* of $\mathcal{A}$ for σ is an infinite sequence $q_0q_1 \ldots \in Q^\omega$ where $q_0 \in Q_{in}$ and $\forall i : q_i \xrightarrow{a_i} q_{i+1}$.

An *accepting run* ρ is a run such that for each final set $F_i \in \mathcal{F}$, there exists at least one state $q \in F_i$ that appears infinitely often in ρ.

A *Labeled Generalized Büchi automaton* ($LGBA$) over Σ is a triple $\langle \mathcal{A}, D, L \rangle$ where $\mathcal{A}$ is a GBA over Σ, D is some finite domain, and $L : Q \longrightarrow 2^D$ is a labeling function from the states of $\mathcal{A}$ to the subsets of the domain D (a state has a set of labels from D).

A run $\rho = q_0 q_1 \ldots$ over $\mathcal{A}$ is said to accept $(d_0 d_1 \ldots, a_0 a_1 \ldots) \in D^\omega \times \Sigma^\omega$ if ρ is a run for $a_0 a_1 \ldots$ and $\forall i \geq 0 : d_i \in L(q_i)$.

An LGBA $\mathcal{A}$ accepts $(\alpha, \beta) \in D^\omega \times \Sigma^\omega$ if there exists an accepting run of $\mathcal{A}$ which accepts it.

The language recognized by the LGBA is $\mathcal{L}(\mathcal{A}) = \{(\alpha, \beta) \in D^\omega \times \Sigma^\omega \mid \mathcal{A} \text{ accepts } (\alpha, \beta)\}$.

2.2 Linear-time Temporal Logic

The underlying structure of time in linear-time temporal logic is a totally ordered set $(S, <)$ where $(S, <)$ is isomorphic to the natural numbers with their usual ordering $(\mathcal{N}, <)$. The time we talk about is discrete, has an initial moment (corresponding to 0) which has no predecessors and is infinite in the future.

This concept of time is quite appropriate when we want to reason about behaviours of concurrent programs. Computers are generally discrete digital devices and programs which we'll be interested in will start at some initial configuration and run forever.

Syntax

Let us first fix a finite underlying set of propositions $\mathcal{P}$ and an action alphabet Σ. The set of formulae of $LTL(\Sigma)$ is the least set of formulae generated by the following rules :

- T, F and each $p \in \mathcal{P}$ are formulas in the logic.
- If α and β are formulas in the logic, and $a \in \Sigma$, then so are $\sim\alpha$, $\langle a \rangle \alpha$, $\alpha \vee \beta$ and $\alpha U \beta$.

Semantics

The models of $LTL(\Sigma)$ are ordered pairs $M = (\sigma, \rho)$ where σ is an infinite sequence of subsets of propositions and ρ is an infinite sequence of actions, i.e. $(\sigma, \rho) \in (2^{\mathcal{P}})^\omega \times \Sigma^\omega$. Let $M = (\sigma, \rho) = (P_0 P_1 \ldots, a_0 a_1 \ldots)$. For any ω-word $\zeta = x_0 x_1 \ldots$, we'll write ζ^i to denote the suffix of ζ starting at x_i. We'll write $M \models \alpha$ to mean that the formula α is satisfied by the model M. We define the semantics of $LTL(\Sigma)$ by structural induction of the formulae:

- $M \models T$, $M \not\models F$ for any M
- $M \models p$ iff $p \in P_0$ where $p \in \mathcal{P}$
- $M \models {\sim}\alpha$ iff $M \not\models \alpha$
- $M \models \alpha \vee \beta$ iff $M \models \alpha$ or $M \models \beta$
- $M \models \langle a \rangle \alpha$ iff $a_0 = a$ and $(\sigma^1, \rho^1) \models \alpha$
- $M \models \alpha U \beta$ iff $\exists j \geq 0$ such that $(\sigma^j, \rho^j) \models \beta$ and $\forall 0 \leq i < j$, $(\sigma^i, \rho^i) \models \alpha$.

We introduce other derived operators as abbreviations :
$\alpha \wedge \beta = {\sim}({\sim}\alpha \vee {\sim}\beta)$, $\quad O\alpha = \bigvee_{a \in \Sigma} \langle a \rangle \alpha$
$\Diamond\alpha = T\ U\ \alpha$, $\quad \Box\alpha = {\sim}\Diamond{\sim}\alpha$
We also introduce dual operators for $\alpha U \beta$ and $\langle a \rangle \alpha$ formulas :
$\alpha V \beta = {\sim}({\sim}\alpha\ U\ {\sim}\beta)$, $\quad [a]\alpha = {\sim}\langle a \rangle {\sim}\alpha$

LTL$^+$(Σ)
For convenience, we'll work with a positive-form of LTL :
$LTL^+(\Sigma) = T \mid F \mid p \mid {\sim}p \mid \alpha \vee \beta \mid \alpha \wedge \beta \mid O\alpha \mid \langle a \rangle \alpha \mid [a]\alpha \mid \alpha U \beta \mid \alpha V \beta$
where $p \in \mathcal{P}$ and $a \in \Sigma$.

Given a formula $\alpha \in LTL(\Sigma)$, we can convert it into an equivalent $LTL^+(\Sigma)$ formula by pushing the negation inside using the duals of U and $\langle a \rangle$. Note that the size of the resulting formula will only be linear in the size of α.

CHAPTER 3
A GLOBAL CONSTRUCTION

We give here a global automaton construction of an $LGBA$ which will accept all the models of the given formula φ – we've adapted the Vardi-Wolper construction to action-based LTL.

Given $\varphi \in LTL^{+}$, we'll first define the closure of φ. Note that the closure might include formulas which are not in $LTL^{+}(\Sigma)$ (like $\sim\langle a\rangle T$).

Definition 3.1 For $\varphi \in LTL^{+}(\Sigma)$, define $CL(\varphi)$ to be the smallest set of formulas such that :

- $\varphi \in CL(\varphi)$
- $T,\ F \in CL(\varphi)$
- $\sim p \in CL(\varphi) \Rightarrow p \in CL(\varphi)$
- $p \in CL(\varphi) \Rightarrow \sim p \in CL(\varphi)$
- $\alpha \vee \beta \in CL(\varphi) \Rightarrow \alpha, \beta \in CL(\varphi)$
- $\alpha \wedge \beta \in CL(\varphi) \Rightarrow \alpha, \beta \in CL(\varphi)$
- $O\alpha \in CL(\varphi) \Rightarrow \alpha \in CL(\varphi)$
- $\langle a\rangle\alpha \in CL(\varphi) \Rightarrow \alpha \in CL(\varphi)$
- $[a]\alpha \in CL(\varphi) \Rightarrow \langle a\rangle\alpha,\ \sim\langle a\rangle T \in CL(\varphi)$
- $\alpha U \beta \in CL(\varphi) \Rightarrow \alpha, \beta, O(\alpha U \beta) \in CL(\varphi)$
- $\alpha V \beta \in CL(\varphi) \Rightarrow \alpha, \beta, O(\alpha V \beta) \in CL(\varphi)$

Let the number of until formulas in $CL(\varphi)$ be m and let them be $EV = \{\alpha_1 U \beta_1, \ldots, \alpha_m U \beta_m\}$. Note that if φ does not have any until formulas, EV will be empty.

We now define atoms, which will serve as the states of the automaton we'll define. Basically atoms which we define here are propositionally consistent subsets of formulas. Unlike the original Vardi-Wolper construction which allow only maximal consistent subsets, we allow any downward consistent subset of the closure to be an atom. Note that temporally the atoms we define may contain contradictions, but these atoms will never be a part of any infinite run.

The atoms of φ are defined by :

Definition 3.2 $A \subseteq CL(\varphi)$ is an atom if :

1. $T \in A,\ F \notin A$
2. $\forall p \in CL(\varphi),\ p \in A \Leftrightarrow \sim p \notin A$
3. $\alpha \vee \beta \in A \Rightarrow \alpha \in A\ or\ \beta \in A$
4. $\alpha \wedge \beta \in A \Rightarrow \alpha \in A\ and\ \beta \in A$
5. $\alpha U \beta \in A \Rightarrow \beta \in A\ or\ (\alpha \in A\ and\ O(\alpha U \beta) \in A)$
6. $\alpha V \beta \in A \Rightarrow \alpha, \beta \in A\ or\ (\beta \in A\ and\ O(\alpha V \beta) \in A)$
7. $[a]\alpha \in A \Rightarrow \sim\langle a\rangle T \in A\ or \langle a\rangle \alpha \in A$

Let $\mathcal{AT}$ be the set of all atoms of φ. The $LGBA$ we construct will be $\langle \mathcal{B}, 2^{\mathcal{P}}, L_{\mathcal{B}} \rangle$ where $\mathcal{B} = (\mathcal{AT}, \rightarrow, A_{in}, \mathcal{F})$,

$\rightarrow$ is defined by : $\forall A, B \in \mathcal{AT}, \forall a \in \Sigma, A \xrightarrow{a} B$ iff

- $\forall \langle b\rangle \alpha \in A,\ b = a\ and\ \alpha \in B$
- $\forall O\alpha \in A,\ \alpha \in B$

- $\forall \sim\langle b\rangle T \in A,\ b \neq a$

$$A_{in} = \{A \in \mathcal{AT} \mid \varphi \in A\}$$

$\mathcal{F} = \{F_1, \ldots F_m\}$ where $F_i = \{A \in \mathcal{AT} \mid \alpha_i U \beta_i \notin A \text{ or } \beta_i \in A\}$
Note that if EV is empty then so is $\mathcal{F}$ and every run of the automaton will be an accepting run.

An atom is labeled by a single set in $2^{\mathcal{P}}$: $\forall A \in\ calAT$, $L_{\mathcal{B}}(A) = \{A \cap \mathcal{P}\}$

Now, we show that $\mathcal{B}$ accepts the models of φ.

Theorem 3.3 $\mathcal{L}(\mathcal{B}) =$ *Models of* φ.

Proof
($\subseteq$) **Soundness**
Let $x = (P_0P_1\ldots, a_0a_1\ldots) \in \mathcal{L}(\mathcal{B})$ be accepted by the run $A_0A_1\ldots$ where $P_i \in L_{\mathcal{A}}(A_i)$, i.e. $P_i = A \cap \mathcal{P}$.

We'll prove by induction on the structure of the formula ϕ that $\forall i, \phi \in A_i \Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \phi$

Base case :
If $\phi = p$: $p \in A_i \Rightarrow p \in P_i \Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models p$
If $\phi = \sim p$: $\sim p \in A_i \Rightarrow p \notin P_i \Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \sim p$
If $\phi = \sim\langle a\rangle T$: $\sim\langle a\rangle T \in A_i \Rightarrow a_i \neq a$. Hence $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \sim\langle a\rangle T$.

Induction step

- $\phi = \alpha \vee \beta$
  $\alpha \vee \beta \in A_i \Rightarrow \alpha \in A_i$ or $\beta \in A_i$
  $\Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha$ or $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \beta$ (by ind.

hypo.)

$\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \alpha \vee \beta$

- $\phi = \alpha \wedge \beta$
  $\alpha \wedge \beta \in A_i \Rightarrow \alpha, \beta \in A_i$
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \alpha$ and $(P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \beta$ (by ind. hypo.)
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \alpha \wedge \beta$

- $\phi = O\alpha$
  $O\alpha \in A_i \Rightarrow \alpha \in A_{i+1} \Rightarrow (P_{i+1} P_{i+2} \ldots, a_{i+1} a_{i+2} \ldots) \models \alpha$ (by ind. hypo.)
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models O\alpha$

- $\phi = \langle a \rangle \alpha$
  $\langle a \rangle \alpha \in A_i \Rightarrow \alpha \in A_{i+1}$ and $a_i = a$
  $\Rightarrow (P_{i+1} P_{i+2} \ldots, a_{i+1} a_{i+2} \ldots) \models \alpha$ and $a_i = a$
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \langle a \rangle \alpha$

- $\phi = [a]\alpha$
  $[a]\alpha \in A_i \Rightarrow \langle a \rangle \alpha \in A_i$ or $\sim\langle a \rangle T \in A_i$
  If $\langle a \rangle \alpha \in A_i$, then $\alpha \in A_{i+1}$ and $a_i = a$
  $\Rightarrow (P_{i+1} P_{i+2} \ldots, a_{i+1} a_{i+2} \ldots) \models \alpha, a_i = a$
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \langle a \rangle \alpha$
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models [a]\alpha$
  If $\sim\langle a \rangle T \in A_i$, then $(P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \sim\langle a \rangle T$
  $\Rightarrow (P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models [a]\alpha$

- $\phi = \alpha_s U \beta_s$
  Since $A_i A_{i+1} \ldots$ is an accepting run, $\exists k \geq i$ such that $A_k \in F_s$. Let the least such k be j.

  - If $j = i$, then since $\alpha_s U \beta_s \in A_i$ and $A_i \in F_s$, we must have $\beta_s \in A_i$. Hence $(P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \beta_s$ and $(P_i P_{i+1} \ldots, a_i a_{i+1} \ldots) \models \alpha_s U \beta_s$
  - If $j > i$, then clearly $\forall i \leq l < j, \alpha_s U \beta_s \in A_l$ and $\beta_s \notin A_l$
    Since A_l is an atom, $\alpha_s \in A_l$. Hence by the induction hypothesis, $\forall i \leq l < j, (P_l P_{l+1} \ldots, a_l a_{l+1} \ldots) \models \alpha_s$. Now, $\alpha_s U \beta_s \in A_{j-1}$ and $\beta_s \notin A_{j-1} \Rightarrow O(\alpha_s U \beta_s) \in A_{j-1}$. Hence $\alpha_s U \beta_s \in$

A_j. Since $A_j \in F_s$, we must have $\beta_s \in A_j$. By the induction hypothesis, $(P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \beta_s$. Hence $\forall i \leq l < j, (P_lP_{l+1}\ldots, a_la_{l+1}\ldots) \models \alpha_s$ and $(P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \beta_s$. Thus $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha_s U \beta_s$.

- $\phi = \alpha V \beta$
  $\alpha V \beta \in A_i$. First recall that $\alpha V \beta \equiv \sim(\sim\alpha\ U\ \sim\beta)$. From the properties of atoms, it follows that one of the following happens :
  - $\forall j \geq i, \beta \in A_j$ and $\alpha V \beta \in A_j$. In this case, $\forall j \geq i, (P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \beta \Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha V \beta$.
  - $\exists j \geq i : \forall i \leq l < j, \beta \in A_l, \alpha V \beta \in A_l$ and $\alpha, \beta \in A_j$. Then, $\forall i \leq l < j, (P_lP_{l+1}\ldots, a_la_{l+1}\ldots) \models \beta$ and $(P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \alpha$ and $(P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \beta$. Hence $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha V \beta$.

Now, since $\varphi \in A_0$, $(P_0P_1\ldots, a_0a_1\ldots) \models \varphi$. Hence x is a model of φ.

($\supseteq$) **Completeness**

Let $(P_0P_1\ldots, a_0a_1\ldots)$ be a model of φ.
$\forall i$, let $A_i = \{\alpha \in CL(\varphi) \mid (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha\}$.

- First, conditions 1 to 6 for being an atom are satisfied by A_i. This follows from the semantics of $LTL(\Sigma)$: For e.g.,

  (1) Let $p \in CL(\varphi)$. Now $p \in A_i \iff (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models p \iff p \in P_i \iff (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \not\models \sim p \iff \sim p \notin A_i$.

  (2) $\alpha \vee \beta \in A_i \Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha \vee \beta$
  $\Rightarrow (P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \alpha$ or $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \beta$. Also, $\alpha, \beta \in CL(\varphi)$. Hence $\alpha \in A_i$ or $\beta \in A_i$.

  Similarly, the other conditions can be checked.

- $\forall i, A_i \xrightarrow{a_i} A_{i+1}$ in $\mathcal{B}$:
  - Let $\langle b \rangle \alpha \in A_i$. Then $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \langle b \rangle \alpha \Rightarrow a_i = b$ and $(P_{i+1}P_{i+2}\ldots, a_{i+1}a_{i+2}\ldots) \models \alpha \Rightarrow a_i = b$ and $\alpha \in A_{i+1}$.

– Let $O\alpha \in A_i$. Then $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models O\alpha$
$\Rightarrow (P_{i+1}P_{i+2}\ldots, a_{i+1}a_{i+2}\ldots) \models \alpha \Rightarrow \alpha \in A_{i+1}$.

– Let $\sim\langle b\rangle T \in A_i$. Then $(P_iP_{i+1}\ldots, a_ia_{i+1}\ldots) \models \sim\langle b\rangle T \Rightarrow b \neq a_i$ or $(P_{i+1}P_{i+2}\ldots, a_{i+1}a_{i+2}\ldots) \not\models T \Rightarrow b \neq a_i$

Hence $\forall i, A_i \xrightarrow{a_i} A_{i+1}$ in $\mathcal{B}$.

- $A_0 \in A_{in}$: Since $(P_0P_1\ldots, a_0a_1\ldots) \models \varphi$, $\varphi \in A_0$. Hence $A_0 \in A_{in}$.
- $\forall i, L_{\mathcal{B}}(A_i) = P_i$:
$\forall p \in P_i, p \in A_i$ and $\forall p \notin P_i, p \notin A_i$. Hence $A_i \cap \mathcal{P} = P_i$. Therefore $\forall i, L_{\mathcal{B}}(A_i) = P_i$.

Hence $\rho = A_0A_1\ldots$ is a run for $(P_0P_1\ldots, a_0a_1\ldots)$ in $\mathcal{B}$.
Now, ρ must be an accepting run. For assume it is not. Then $\exists r \exists j : \forall k \geq j, A_k \notin F_r$.
Thus $\forall k \geq j, \alpha_r U \beta_r \in A_k$ and $\beta_r \notin A_r$
$\Rightarrow \forall k \geq j, (P_kP_{k+1}\ldots, a_ka_{k+1}\ldots) \models \alpha_r U \beta_r$ and $(P_kP_{k+1}\ldots, a_ka_{k+1}\ldots) \not\models \beta_r$ which contradicts $(P_jP_{j+1}\ldots, a_ja_{j+1}\ldots) \models \alpha_r U \beta_r$ since β_r never gets satisfied.

Hence ρ is an accepting run for $(P_0P_1\ldots, a_0a_1\ldots)$ in $\mathcal{B}$.
Hence the theorem. ■

CHAPTER 4
THE ALGORITHM

4.1 A Description of the algorithm

4.1.1 The Goal

By the previous chapter, we know that we can construct an $LGBA$ which will accept the models of φ and that this can be used for the satisfiability and model-checking problems. In this chapter, we give a construction of an $LGBA$ for a formula which works in an "on-the-fly" manner. Using this algorithm, parts of the automaton can be computed as and when required. The algorithm we give is an adaptation of the algorithm for LTL given in [6]. The algorithm, given a formula $\varphi \in LTL(\Sigma)$, will generate an LGBA $\mathcal{A}$ which will accept all the models of φ.

The algorithm works by building a graph which will be the nodes of the LGBA defined by the algorithm. The nodes of the graph are labeled by sets of formulas which are obtained by decomposing formulas according to their boolean structure. Temporal formulas are handled by just deciding what should be true at the node and what must be true at any next node. For this the identities:

$$\alpha U \beta \equiv \beta \vee (\alpha \wedge O(\alpha U \beta))$$

$$\alpha V \beta \equiv \beta \wedge (\alpha \vee O(\alpha V \beta))$$

$$[a]\alpha \equiv \sim\langle a\rangle T \vee \langle a\rangle \alpha$$

will be useful.

4.1.2 The Data Structure

The data structure used for representing the graph nodes is described below.

A graph node has six fields :

Name A string that is the name of the node

Incoming A set of elements of the form (q, X) where q is the name of a node and $X \subseteq \Sigma$. This set represents the incoming edges to this node in the graph. If $(q, X) \in Incoming$ such that $a \in X$, then there is an edge from q labeled a to this node. There is a special name *Init* used to indicate that the node is an initial node.

New This is a set of temporal formulas that the algorithm will try to satisfy. These formulas are those which have not been processed.

Old This is the list of formulas that the algorithm has decided to satisfy, and hence all the formulas here have already been processed.

$Next^+$ These are the temporal next-state formulas which this node satisfies. Thus these properties assert all the formulas that must be true in any successor node.

$Next^-$ These are the set of actions which are not allowed to occur from the node. Thus no transition from this node can be an action in $Next^-$.

The algorithm will construct nodes and store them as a list of nodes called *Nodes_Set*. In all such nodes the *New* field will be empty. At the end *Nodes_Set* will define, with the help of the *Incoming* fields, the graph constructed by the algorithm.

4.1.3 A description of the algorithm

The algorithm starts off with a node, say p_0, which has its *New* field set to $\{\varphi\}$, $Incoming = \{Init\}$ and with *Old*, $Next^+$, and $Next^-$ empty.

The formulas in New must be viewed as obligations which have to be satisfied. Formulas in Old represent the set of obligations that are satisfied, provided the obligations in New are satisfied. Thus when New becomes empty, Old contains the set of formulas which the node satisfies.

In the beginning the algorithm has only one obligation φ to satisfy.

When the current node is q the algorithm processes the formulas in $New(q)$. It removes a formula η from New and tries all possible ways to satisfy it. When looking at η it looks at only the outermost operator and processes η according to this. If $\eta = p$ or $\sim p$ where $p \in \mathcal{P}$, clearly the only way to satisfy it is to make it true at that node. If Old already has the negation of η (where $Neg(\sim p) = p$) then η cannot be satisfied at this node, and hence the node is abandoned. Otherwise η is simply added to Old. If $\eta = T$ then it gets trivially satisfied; T is just added to Old. If $\eta = F$, then again the node is abandoned.

The other cases are:

$\eta = \alpha \wedge \beta$
: Clearly the only way to satisfy this is to satisfy α as well as β. Hence α and β are added to New as new obligations and $\alpha \wedge \beta$ is added to Old. Of course, if either α or β is already in Old, then it is not put into New as it is something which has already been processed.

$\eta = \alpha \vee \beta$
: Here there are two ways of satisfying η. Hence the node splits into two identical nodes (only names are different) and α is added to New of one node while β is added to the other. η is added to Old of both nodes.

$\eta = \langle a \rangle \alpha$
: Since this is a next-state requirement, it is simply added to $Next^{+}$.

$\eta = [a]\alpha$
: Recall $[a]\alpha \equiv \sim\langle a \rangle T \vee \langle a \rangle \alpha$. Hence there are two ways of satisfying η. The node is hence split. To make $\sim\langle a \rangle T$ hold in one node, a is added

to $Next^-$ field of it. In the other node $\langle a \rangle \alpha$ is added to $Next^+$. η is added to Old of both nodes.

$\eta = \alpha U \beta$
We use the equivalence:

$$\alpha U \beta \equiv \beta \vee (\alpha \wedge O(\alpha U \beta))$$

The node is split. In one node, β is added to New, while in the other, α and $O(\alpha U \beta)$ are added to New. Again, η is added to Old of both nodes.

$\eta = \alpha V \beta$
Recall $\alpha V \beta \equiv (\beta \wedge \alpha) \vee (\beta \wedge O(\alpha V \beta))$. The node is split - in one node, α and β are added to New while in the other β and $O(\alpha V \beta)$ are added to New. Again, η is added to Old of both nodes.

When New becomes empty, the node q is fully processed and can be added to $Nodes\_Set$. If there is already a node, say w, in $Nodes\_Set$ with the same Old, $Next^+$, and $Next^-$ fields, then since the future of q future is the same as that of w, the node q gets merged with w; this is done by updating the node in $Nodes\_Set$ by adding all the incoming edges of q to $Incoming$ of w. If no such copy is in $Nodes\_Set$ then q is added to $Nodes\_Set$.

Now if there are two formulas $\langle a \rangle \alpha$ and $\langle b \rangle \beta$ in $Next^+$, $(a \neq b)$ then there can be no successor node as one demands action a and the other action b. Hence no new node is spawned. Also if $\langle a \rangle \alpha \in Next^+$ and $a \in Next^-$ then action a is demanded by $\langle a \rangle \alpha$ but is disallowed by $Next^-$; here too no new nodes are spawned. Otherwise, a new node is spawned, say q', such that $New(q')$ contains the obligations which $Next^+(q)$ demands of q'. The Old , $Next^+$, $Next^-$ fields of q' are empty. If $Next^+(q)$ mentions any $\langle a \rangle \alpha$ then only action a can be performed and thus $Incoming$ of q' is set to $\{(q, a)\}$. Otherwise it is set to $\{(q, \Sigma - Next^-(q))\}$ because any action not disallowed by q can happen.

The nodes spawned are processed in DFS order. Note that new nodes are spawned only by nodes which get added to $Nodes\_Set$. Since each new node gets processed in a finite amount of time and since the number of nodes in $Nodes\_Set$ can only be finite, it follows that the algorithm will terminate and output a finite automaton.

4.2 The Algorithm

We give the algorithm in pseudo-code. For conciseness, we use functions New1, New2 and Next1 which are defined by the following table.

η	$New1(\eta)$	$Next1(\eta)$	$New2(\eta)$
$\alpha \vee \beta$	$\{\alpha\}$	$\emptyset$	$\{\beta\}$
$\alpha U \beta$	$\{\alpha\}$	$\{O(\alpha U \beta)\}$	$\{\beta\}$
$\alpha V \beta$	$\{\beta\}$	$\{O(\alpha V \beta)\}$	$\{\alpha, \beta\}$

TABLE 4.1 : Definition of NEW1, NEXT1 and NEW2

new_name is a function which generates new strings on successive calls. Neg is a function defined as: $Neg(p) = \sim p$; $Neg(\sim p) = p$.

The Algorithm :

```
EXPAND(Node, Nodes_Set)
if New(Node) = ∅
  then if ∃ N ∈ Nodes_Set with Old(N) = Old(Node) and
          Next⁺(N) = Next⁺(Node) and Next⁻(N) = Next⁻(Node)
          then Incoming(N) ← Incoming(N) ∪ Incoming(Node)
               return (Nodes_Set)
          else if ∃⟨a⟩α, ⟨b⟩β ∈ Next⁺(Node) with a ≠ b
                  or ∃ a ∈ Next⁻(Node), ⟨a⟩α ∈ Next⁺(Node)
                  then return (Nodes_Set ∪ {Node})
                  else if ∄⟨a⟩α ∈ Next⁺(Node)
                          then return (EXPAND([Name ← new_name(),
                                    Incoming ← {(Name(Node),
                                      Σ \ Next⁻(Node))},
                                    New ← STRIPNEXTS(Next⁺(Node)),
                                    Old ← ∅,
                                    Next⁺ ← ∅, Next⁻ ← ∅],
```

$\{Node\} \cup Nodes\_Set)$
else **return** $(\textsc{expand}([Name \leftarrow new\_name(),$
$Incoming \leftarrow \{(Name(Node),\ \{a\})\ \},$
$New \leftarrow \textsc{stripnexts}(Next^{+}(Node)),\ Old \leftarrow \emptyset,$
$Next^{+} \leftarrow \emptyset,\ Next^{-} \leftarrow \emptyset],\ \{Node\} \cup Nodes\_Set))$
else *let* $\eta \in New;$
$New(Node) \leftarrow New(Node) \setminus \{\eta\};$
switch
case $\eta = P$ *or* $\sim\!P$ *or* T *or* F :
if $\eta = F$ *or* $\textsc{Neg}(\eta) \in Old(Node)$
then return $(Nodes\_Set)$
else $Old(Node) \leftarrow Old(Node) \cup \{\eta\};$
return $(\textsc{expand}(Node,\ Nodes\_Set));$

case $\eta = \alpha U \beta$, *or* $\alpha V \beta$, *or* $\alpha \vee \beta$:
$Node1 \leftarrow [Name \leftarrow \textsc{new\_name}(), Incoming \leftarrow Incoming(Node),$
$New \leftarrow New(Node) \cup (\textsc{New1}(\eta) \setminus Old(Node)),$
$Old \leftarrow Old(Node) \cup \{\eta\},\ Next^{+} \leftarrow Next^{+}(Node) \cup \textsc{Next1}(\eta),$
$Next^{-} \leftarrow Next^{-}(Node)]$
$Node2 \leftarrow [Name \leftarrow \textsc{new\_name}(), Incoming \leftarrow Incoming(Node),$
$New \leftarrow New(Node) \cup (\textsc{New2}(\eta) \setminus Old(Node)),$
$Old \leftarrow Old(Node) \cup \{\eta\},\ Next^{+} \leftarrow Next^{+}(Node),$
$Next^{-} \leftarrow Next^{-}(Node)]$
return $(\textsc{expand}(Node2,\ \textsc{expand}(Node1, Nodes\_Set)))$

case $\eta = \alpha \wedge \beta$:
$Old(Node) \leftarrow Old(Node) \cup \{\eta\}$
$New(Node) \leftarrow New(Node) \cup (\{\alpha,\ \beta\} \setminus Old(Node))$
return $(\textsc{expand}(Node,\ Nodes\_Set))$

case $\eta = O\alpha,\ \langle a \rangle \alpha$:
$Old(Node) \leftarrow Old(Node) \cup \{\eta\}$
$Next^{+}(Node) \leftarrow Next^{+}(Node) \cup \{\eta\}$
return $(\textsc{expand}(Node, Nodes\_Set))$

case $\eta = [a]\alpha$:
$Node1 \leftarrow [Name \leftarrow \textsc{new\_name}(), Incoming \leftarrow Incoming(Node),$
$New \leftarrow New(Node),\ Old \leftarrow Old(Node) \cup \{\eta\},$
$Next^{+} \leftarrow Next^{+}(Node),\ Next^{-} \leftarrow Next^{-}(Node) \cup \{a\}]$
$Node2 \leftarrow [Name \leftarrow \textsc{new\_name}(), Incoming \leftarrow Incoming(Node),$
$New \leftarrow New(Node),\ Old \leftarrow Old(Node) \cup \{\eta\},$
$Next^{+} \leftarrow Next^{+}(Node) \cup \{\langle a \rangle \alpha\},\ Next^{-} \leftarrow Next^{-}(Node)]$
return $(\textsc{expand}(Node2,\ \textsc{expand}(Node1, Nodes\_Set)))$

```
CREATE-GRAPH(φ)
  return (EXPAND([Name ← NEW_NAME(), Incoming ← {Init},
      New ← {φ}, Old ← ∅, Next⁺ ← ∅, Next⁻ ← ∅], ∅))
```

STRIPNEXTS$(S) = \{\alpha \mid O\alpha \in S \text{ or } \langle a\rangle\alpha \in S \text{ for some } a \in \Sigma\}$

4.3 The LGBA defined by the algorithm

Let t be a function from the set of nodes (finished and unfinished) to $2^{CL(\varphi)}$ which stands for the temporal formulas at a node q. It is defined by

$$t(q) = Old(q) \cup New(q) \cup Next^{+}(q) \cup \{\sim\langle a\rangle T \mid a \in Next^{-}(q)\}$$

(We require t to be defined only on finished nodes (i.e. on $Nodes\_Set$) for describing the $LGBA$, but we require the extended t when doing the proof of correctness.)

The LGBA defined by the algorithm is $\langle \mathcal{A},\ D,\ L_{\mathcal{A}}\rangle$ where
$\mathcal{A} = (\mathcal{NS}, Q_{in}, \rightarrow, \mathcal{F}')$
$\mathcal{NS}$ = Nodes of the graph returned by the algorithm = $Nodes\_Set$
Q_{in} = Nodes q such that $Init \in Incoming(q)$
$p \xrightarrow{a} q$ iff $\exists(p,\ X) \in Incoming(q)$ such that $a \in X$
$\mathcal{F}' = \{F'_1, \ldots, F'_m\}$ where
$\forall 1 \leq i \leq m : F'_i = \{q \in Nodes\_Set \mid \alpha_i U \beta_i \notin t(q) \text{ or } \beta_i \in t(q)\}$

$D = 2^{\mathcal{P}}$ is the domain
and the labeling function is given by
$L_{\mathcal{A}}(q) = \{X \subseteq \mathcal{P} \mid Pos(q) \subseteq X \text{ and } X \cap Neg(q) = \emptyset\}$ where
$Pos(q) = \mathcal{P} \cap Old(q)$ and $Neg(q) = \{p \in \mathcal{P} \mid\ \sim p \in Old(q)\}$.

i.e. $L_{\mathcal{A}}(q)$ is the set of all assignments consistent with the proportional assignments in $Old(q)$.

The LGBA can be used for both the satisfiability and model-checking problems. It can be used for satisfiability using an algorithm which checks for its emptiness. Note that since $L_{\mathcal{A}}(q) \neq \emptyset$ for any q, the algorithm need not look at the labels; it has to just check the graph and see if the underlying GBA has an accepting run; this can be done by using either the strongly-connected component approach, or [4].

Checking for emptiness can be done on-the-fly; the algorithm can stop if it detects the existence of an accepting run.

As described in the introduction, model-checking can be done by intersecting the program automaton with the automaton constructed by the algorithm, in an on-the-fly manner. Note that a node in the program automaton can be associated with a state in the program automaton, only if the set of propositions true in that state in the program is a label of the node. Since the program automaton has no final states, the product automaton is a simple synchronous product. Hence the product automaton can be generated as and when needed during a depth-first search which checks for non-emptiness.

4.4 Proof of correctness

To show that the *LGBA* $\mathcal{A}$ defined by the algorithm captures all the models of φ, we'll show that the global Vardi-Wolper automaton ($\mathcal{B}$) and $\mathcal{A}$ accept precisely the same language. Since the automaton $\mathcal{B}$ captures the models of φ, this would prove the correctness of the algorithm.

4.4.1 Completeness proof of the algorithm

We show here that any model accepted by $\mathcal{B}$ is also accepted by $\mathcal{A}$. We do this by mapping accepting runs of the automaton $\mathcal{B}$ to accepting runs over the algorithm automaton $\mathcal{A}$.

To pull out this run, we define the following :

f is a function from the set of nodes (finished and unfinished) to $2^{\mathcal{AT}}$ which intuitively stands for those atoms which can extend a node q. It is defined by

$$f(q) = \{A \in \mathcal{AT} \mid t(q) \subseteq A\}$$

Now, we give a series of lemmas which will lead to the completeness proof.

Lemma 4.4 *When a node q is split in the algorithm (lines 30-39 and 51-58), into two nodes $q1$ and $q2$, the following holds*

$$f(q) = f(q1) \cup f(q2)$$

Similarly, when a node q is updated to become a new node q' (lines 24-28 and 41-49), the following holds:

$$f(q) = f(q')$$

Proof

(a)

- $\eta = \alpha \vee \beta$ or $\alpha U \beta$ or $\alpha V \beta$
  $t(q_1) = t(q) \cup$ NEW1(η) $\cup$ NEXT1(η) and $t(q_2) = t(q) \cup$ NEW2(η)
  Since $t(q) \subseteq t(q_1)$ and $t(q) \subseteq t(q_2)$, $f(q_1) \cup f(q_2) \subseteq f(q)$.
  Now let $A \in f(q)$. Since $\eta \in t(q) \subseteq A$, NEW1$(\eta)$ $\cup$ NEXT1$(\eta) \subseteq A$ or NEW2$(\eta) \subseteq A$. Thus $t(q_1) \subseteq A$ or $t(q_2) \subseteq A$. Hence $A \in f(q_1) \cup f(q_2)$.
  Therefore $f(q) = f(q_1) \cup f(q_2)$.

- $\eta = [a]\alpha$
  $t(q_1) = t(q) \cup \{\sim\langle a\rangle T\}$ and $t(q_2) = t(q) \cup \{\langle a\rangle\alpha\}$.
  Since $t(q) \subseteq t(q_1)$ and $t(q) \subseteq t(q_2)$, $f(q_1) \cup f(q_2) \subseteq f(q)$.
  Now let $A \in f(q)$. Since $[a]\alpha \in t(q) \subseteq A$, $\sim\langle a\rangle T \in A$ or $\langle a\rangle\alpha \in A$. Thus $t(q_1) \subseteq A$ or $t(q_2) \subseteq A$. Hence $A \in f(q_1) \cup f(q_2)$.
  Therefore $f(q) = f(q_1) \cup f(q_2)$.

(b)

- $\eta = p$, $\sim p$, or T
  $t(q') = t(q)$. Hence $f(q) = f(q')$.

- $\eta = \alpha \wedge \beta$
  $t(q') = t(q) \cup \{\alpha, \beta\}$
  First, since $t(q) \subseteq t(q')$, $f(q') \subseteq f(q)$.
  Let $A \in f(q)$. Then, since $\alpha \wedge \beta \in A$, $\alpha, \beta \in A$. Hence $A \in f(q')$.
  Thus $f(q) = f(q')$.

- $\eta = O\alpha$ or $\langle a \rangle \alpha$
  $t(q') = t(q)$. Hence $f(q) = f(q')$.

Hence the lemma. ■

When a node p is spawned with its Old, $Next^+$ and $Next^-$ fields empty, the algorithm starts processing the nodes in New. From this point onwards, one can view the algorithm as creating a tree rooted at p. The tree gets modified as formulas in New are processed. When a node (which must be a leaf) splits, we can view this as a creation of two children, since the algorithm will start expanding each child (eventually). When a node is processed and it's fields get modified, we view this as the creation of a single child. When a node is abandoned, we mark the node *bad* - no new edges come out of this node. Finally a tree is produced which is rooted at p. We'll call a leaf node *good* if it is not bad. Note that the good leaves at the end of the construction have their New fields empty and are exactly the nodes added to $Nodes\_Set$. The proof of the following lemma will work by induction on the number of steps which have been performed so far by the algorithm, which have modified the tree.

Lemma 4.5 *Let p be a rooted node and $q_1, q_2, \ldots, q_n$ be the leaves of the tree which is formed. Let $A \in f(p)$. Then there is a good leaf q_j such that $A \in f(q_j)$ and $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow q_j \in F_i'$*

Proof

Let $A \in f(p)$. We'll prove the following claim :

Claim : At any point in the construction of the tree rooted at p, there is a good leaf q such that $A \in f(q)$ and $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q)$ or $\beta_i \in t(q))$

Proof of claim : by induction on the number of steps in the algorithm which has changed the tree so far.

Base case : Initially, p is the only leaf and it is good (since it's not marked bad, i.e. abandoned). Hence for $q_j = p,\ A \in f(q_j)$ and since $Old(p) = \emptyset$, we have $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q)$ or $\beta_i \in t(q))$.

Induction step : Assume some leaf q gets processed and η is the formula being processed. By the induction hypothesis, we have some good leaf q_j which satisfies the claim before η is processed. If $q \neq q_j$ then trivially we have the claim to be true after η is processed because q will still satisfy the claim. So, let's assume that q is a leaf which satisfies the induction hypothesis. Hence $A \in f(q)$ and $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q)$ or $\beta_i \in t(q))$.

- $\eta = p, \sim p, T, F$

  If the node gets updated, q gets a child q' which is good. (lines 27-28). Now by Lemma 4.4, $f(q) = f(q')$. Hence $A \in f(q')$. Since no until formula has been added to Old and $t(q) = t(q')$, q' satisfies the claim for the new tree.

  Now, if the node is abandoned and marked bad (lines 25-26), then either $\eta = F$ or $Neg(\eta) \in Old(q)$. But since $A \in f(q)$, it cannot be the case that $F \in A$ nor the case that $\eta,\ Neg(\eta) \in A$. This contradiction proves that such a node q will not be abandoned and marked bad.

- $\eta = \alpha \vee \beta, aVb, [a]\alpha$

  q gets two children q_1 and q_2, both good, due to the split. By Lemma

4.4, $f(q) = f(q_1) \cup f(q_2)$. Hence $A \in f(q_c)$ for some $c \in \{0, 1\}$. Now since no until formulas have been added to Old(q_c) and since $t(q) \subseteq t(q_c)$, it's clear that $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q_c)$ or $\beta_i \in t(q_c))$. Hence q_c satisfies the claim.

- $\eta = \alpha \wedge \beta, O\alpha, \langle a \rangle \alpha$
  q gets a good child q'. By Lemma 4.4, $f(q) = f(q')$. Hence $A \in f(q')$. Again, since no until formulas have been added to q' and since $t(q) = t(q')$, we have $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q')$ or $\beta_i \in t(q'))$. Hence q' satisfies the claim.

- $\eta = \alpha_j U \beta_j$
  q gets two children q_1 and q_2, both of which are good. Now, $\forall 1 \leq i \leq m,\ i \neq j,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q')$ or $\beta_i \in t(q'))$ where $q' = q_1$ or q_2 since $t(q) \subseteq t(q')$ and no $\alpha_i U \beta_i$ formula $(i \neq j)$ has been added to Old. If $A \notin F_j$, then by Lemma 4.4, $A \in f(q_c)$ for some $c \in \{0, 1\}$. Hence q_c would satisfy the claim. So, assume $A \in F_j$. First, since $\alpha_j U \beta_j \in t(q) \subseteq A$ and $A \in F_j$, we must have $\beta_j \in A$. Now, $t(q_2) = t(q) \cup \{\beta_j\}$. Since $t(q) \subseteq A$ and $\beta_j \in A$, $t(q_2) \subseteq A$. Hence $A \in f(q_2)$. Also, since $\beta_j \in t(q_2)$, we have $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q_2)$ or $\beta_i \in t(q_2))$. Hence q_2 satisfies the claim.

Hence the claim is true.

At the end of the construction, from the claim, it follows that there exists a good leaf q such that $A \in f(q)$ and $\forall 1 \leq i \leq m,\ A \in F_i \Rightarrow (\alpha_i U \beta_i \notin Old(q)$ or $\beta_i \in t(q))$

$\Rightarrow (\alpha_i U \beta_i \notin t(q)$ or $\beta_i \in t(q))$ (since $New(q) = \emptyset$, since q is good)

$\Rightarrow q \in F'_i$

Hence the lemma. ■

Lemma 4.6 *Let $A \in f(q)$ and let $A \xrightarrow{a} A'$. Then $\exists q' \in \mathcal{NS}$ such that $q \xrightarrow{a} q'$, $A' \in f(q')$ and $\forall 1 \leq i \leq m$: if $A' \in F_i$, then $q' \in F'_i$.*

Proof

First, it is clear that there is no $\sim\langle a\rangle T$ or $\langle b\rangle\alpha$ $(b \neq a)$ in $t(q)$, for otherwise it would belong to A as well and $A \xrightarrow{a} A'$ will not be possible in $\mathcal{B}$. Hence after q was processed by the algorithm, a node p' must have been spawned with its New field set as the formulas in $Next^{+}(q)$ stripped of their O's and $\langle a\rangle$'s.

Now, if $\alpha \in t(p)$, then $O\alpha$ or $\langle a\rangle\alpha \in t(q) \;\Rightarrow O\alpha$ or $\langle a\rangle\alpha \in A \;\Rightarrow \alpha \in A'$. Hence $t(p) \subseteq A'$ and $A' \in f(p')$.

By Lemma 4.5, there exists a good leaf, and hence a node in $\mathcal{NS}$, say q', in the tree rooted at p', such that $A' \in f(q')$ and $\forall 1 \leq i \leq m,\; A \in F_i \Rightarrow q' \in F_i'$. Also, $\exists (q, X) \in Incoming(q')$ such that $a \in X$. Hence $q \xrightarrow{a} q'$. This proves the lemma. ■

Lemma 4.7 *Let* $A_0 \in A_{in}$. *Then* $\exists q_{in} \in Q_{in}$ *such that* $A_0 \in f(q_{in})$.

Proof

The algorithm starts with a node p_0 with $New(p_0) = \{\varphi\}$. Since $A_0 \in A_{in}$, $\varphi \in A_0 \;\Rightarrow t(p_0) \subseteq A_0 \;\Rightarrow A_0 \in f(p_0)$. Lemma 4.5 guarantees the existence of some good leaf q_0, which hence gets into $\mathcal{NS}$, in the tree rooted at p_0 such that $A_0 \in f(q_0)$. Also, any leaf of the tree rooted at p_0 has $Init \in Incoming$ and hence $q_0 \in Q_{in}$. This proves the lemma. ■

Lemma 4.8 $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$

Proof

Let $(P_0P_1\ldots, a_0a_1\ldots) \in \mathcal{L}(\mathcal{B})$ and let $A_0A_1\ldots$ be an accepting run for it in $\mathcal{B}$. We'll exhibit an accepting run of $\mathcal{A}$ that accepts $(P_0P_1\ldots, a_0a_1\ldots)$.

First, by Lemma 4.7, $\exists q_0 \in Q_{in}$ such that $A_0 \in f(q_0)$. We'll construct an accepting run $q_0q_1\ldots$ for $(P_0P_1\ldots,\; a_0a_1\ldots)$ by using Lemma 4.6 repeatedly. Assume we've constructed a partial run $q_0\ldots q_t$ so far such that $A_t \in f(q_t)$ (this is true in the beginning for the partial run q_0). Now since $A_t \xrightarrow{a_t} A_{t+1}$

and $A_t \in f(q_t)$, using Lemma 4.6, $\exists q_{t+1}$ such that $q_t \xrightarrow{a_t} q_{t+1}$, $A_{t+1} \in f(q_{t+1})$ and $\forall 1 \leq i \leq m$, $A_{t+1} \in F_i \Rightarrow q_{t+1} \in F_i'$. Thus we have extended the partial run to $q_0 \ldots q_t q_{t+1}$ with $A_{t+1} \in f(q_{t+1})$. Continuing in this fashion, we can build an infinite run $\rho = q_0 q_1 \ldots$ such that $\forall j : A_j \in f(q_j)$ and $\forall 1 \leq i \leq m$, $A_j \in F_i \Rightarrow q_j \in F_i'$. Since $A_0 A_1 \ldots$ meets every final set infinitely often, the run ρ also meets all final sets infinitely often. Hence ρ is an accepting run in $\mathcal{A}$.
Now, for any i, since $A_i \in f(q_i)$, $t(q_i) \subseteq A_i$. Also $P_i = A_i \cap \mathcal{P}$.
If $p \in Pos(q_i)$, then $p \in t(q_i) \subseteq A_i \Rightarrow p \in P_i$.
If $p \in Neg(q_i)$, then $\sim p \in t(q_i) \subseteq A_i \Rightarrow p \notin A_i \Rightarrow p \notin P_i$.
Hence $P_i \subseteq \mathcal{P}$, $Pos(q_i) \subseteq P_i$ and $Neg(q_i) \cap P_i = \emptyset$. Hence $P_i \in L_{\mathcal{A}}(q_i)$. Thus ρ is an accepting run for $(P_0 P_1 \ldots, a_0 a_1 \ldots)$. Hence the lemma. ■

Lemma 4.8 proves the completeness of the algorithm.

4.4.2 Soundness proof of the algorithm

Lemma 4.9 *For any node $q \in \mathcal{NS}$ and $d \in L_{\mathcal{A}}(q)$, let $A = t(q) \cup Pos(d) \cup Neg(d) \cup \{T\}$ where $Pos(d) = d$ and $Neg(d) = \{\sim p \mid p \in \mathcal{P} \setminus d\}$. Then A is an atom.*

Proof
First, clearly A is a subset of $CL(\varphi)$. Now, refer to conditions (1) to (7) which must be satisfied for a set of formulas to be an atom. Condition (1) is satisfied as $T \in A$ and since F is never part of any node in $Nodes\_Set$, $F \notin A$. Also, since $Pos(q) \subseteq d$ and $Neg(q) \cap d = \emptyset$, it's clear that condition (2) is also satisfied by A. For (3) to (7), observe that whenever a formula on the left-hand side of these conditions are thrown into Old, the required formulas get thrown into New, and these formulas will eventually get into Old and hence into A. For example, for condition (3), if $\alpha \vee \beta \in A$, then it must belong to $t(q)$. So, when it was processed, either α or β would have been put into New. But any node thrown into New will eventually get into all the finished nodes under this node. Hence A will have either α or β. Similarly, other conditions can be verified.

■

Lemma 4.10 $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$

Proof

Let $(P_0P_1\ldots, a_0a_1\ldots)$ be accepted by a run $\sigma = q_0q_1\ldots$ in $\mathcal{A}$, where $\forall i$: $P_i \in L_{\mathcal{A}}(q_i)$. Consider $A_i = t(q_i) \cup Pos \cup Neg \cup \{T\}$ where $Pos = P_i$ and $Neg = \{\sim p \mid p \notin P_i\}$. First, by Lemma 4.9, A_i is an atom, for all i. (Note that $A_i \in f(q_i)$). Also, it's clear that $L_{\mathcal{B}}(A_i) = P_i$.

We'll show that $A_0A_1\ldots$ is an accepting run in $\mathcal{B}$ for $(P_0P_1\ldots, a_0a_1\ldots)$. First, $A_0 \supseteq t(q_0)$ and $\varphi \in t(q_0)$. Hence $\varphi \in A_0 \Rightarrow A_0 \in A_{in}$.

Now, let $q_i \xrightarrow{a_i} q_{i+1}$. Clearly, since q_{i+1} was spawned from q_i, it's clear that $\langle b\rangle\alpha \notin A_i$ (where $b \neq a_i$) and $a_i \notin Next^{-}(q_i)$. Now, $\forall O\alpha \in t(q_i)$, $\alpha \in t(q_{i+1})$ (by construction) and $\forall\langle a_i\rangle\alpha \in t(q_i)$, $\alpha \in t(q_{i+1})$.

Hence we have :

- $\forall\langle b\rangle\alpha \in A_i,\ b = a_i$
- $\forall O\alpha \in A_i,\ \alpha \in A_{i+1}$
- $\forall\langle a_i\rangle\alpha \in A_i,\ \alpha \in A_{i+1}$
- $\sim\langle a_i\rangle T \notin A_i$

Hence $A_i \xrightarrow{a_i} A_{i+1}$ in $\mathcal{B}$.

Also, if $q_i \in F_r$ for some r, then either $\alpha_r U\beta_r \notin t(q_i)$ or $\beta_r \in q_i$.

If $\alpha_r U\beta_r \notin t(q_i)$, then $\alpha_r U\beta_r \notin A_i$.

If $\beta_r \in t(q_i)$, then $\beta_r \in A_i$.

Hence $A_i \in F'_r$. Hence we have a run $\rho = A_0A_1\ldots$ in $\mathcal{B}$ for $(P_0P_1\ldots, a_0a_1\ldots)$. Since σ meets each F_i infinitely often, ρ meets each F'_i infinitely often. Hence ρ is an accepting run for $(P_0P_1\ldots, a_0a_1\ldots)$. Hence $(P_0P_1\ldots, a_0a_1\ldots) \in \mathcal{L}(\mathcal{B})$.

Hence the lemma. ∎

Lemma 4.10 proves the soundness of the algorithm.

Theorem 4.11 *$\mathcal{A}$ accepts the models of φ*

Proof

By Lemma 4.10 and Lemma 4.8, we know that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. From Theorem 3.3, it follows that $\mathcal{A}$ accepts the models of φ. ■

CHAPTER 5
CONCLUSION

We've presented a pragmatic approach to the model-checking problem for action-based LTL. We've adapted this from the algorithm described in [6]. The on-the-fly approach seems to yield a practical algorithm for model-checking as well as satisfiability. Also, the algorithm is very simple-minded and straightforward. We hope to extend this method to other logics.

As we described in the introduction, when concurrent or distributed systems are modeled as non-deterministic transition systems, there is an exponential blow-up in the state-space with the number of processes in the program. This is usually called the *state-explosion problem* and there have been many suggestions on how to tackle it. It has been noticed that in many specifications, many interleaved runs of a system can be equated as all of them represent the same partial-order stretch of behaviour [16, 8]. Hence to check certain properties, it is sufficient to check one of the runs in such a set of equivalent runs. This has led to many efficient partial-order verification methods [16, 8, 11, 7].

We are, in particular, interested in a subset of LTL called *Product-PTL*, introduced by Thiagarajan in [14]. When programs are modeled as a network of communicating automata that synchronize on common actions, there is a natural independence relation amongst the actions - two actions are independent if the the set of processes which synchronize on these actions are disjoint. This independence relation naturally defines an equivalence relation over the runs of the network. Now, a formula in LTL is said to be trace-consistent if for any two runs in the same equivalence class, either both or neither of them satisfy the formula. Thus, for any trace-consistent formula, one needs to check only one representative from each class. Though most useful specifications are trace consistent, a syntactic characterization of such formulas is believed to be hard.

Product-PTL defines an interesting subset of trace-consistent formulas of LTL. This logic has been shown to have strong relationships with networks of communicating automata. [14]. We hope to extend the on-the-fly automata theoretic approach for $LTL(\Sigma)$ presented here to an on-the-fly algorithm for *Product-PTL* which will construct networks of communicating automata. In fact, this was our motivation when we extended the on-the-fly algorithm to action-based LTL. Preliminary work has shown that using the algorithm we've presented, we can, without much difficulty, give an on-the-fly verification procedure for *Product-PTL*.

REFERENCES

1. Büchi, J.R.,(1962) On a decision method in restricted second order arithmetic, in *Proc. of the International Congress of Logic, Methodology and Philosophy of Science*, 1-12

2. Clarke, E.M., E.A. Emerson, A.P. Sistla, (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications

3. Clarke, E.M., A.P.Sistla, (1985) The complexity of propositional linear temporal logic in, *Journal of the Association of Computing Machinery, 32* : 733-749

4. Courcoubetis C., M.Vardi, P.Wolper, M.Yannakakis, (1992) Memory efficient algorithms for the verification of temporal logic properties, in *Formal methods in System Design 1*, 275-288

5. Emerson, E.A., (1990) Temporal and Modal Logic, in J.V.Leeuwen (ed.), *Handbook of theoretical computer science*, Vol. B, Elsevier Science Publ., Amsterdam, 997-1072

6. Gerth, R., D.Peled, M.Y.Vardi, P.Wolper,(1995) Simple On-the-fly Verification of Linear Temporal Logic

7. Godefroid, P., P.Wolper, (1991) Using Partial Orders for Efficient Verification of Deadlock Freedom and Safety Properties, in *LNCS 575, Springer-Verlag Berlin-Amsterdam-New York*, 332-343

8. Katz, S., D.Peled, (1992) Interleaving Set Temporal Logic, in *Theoretical Comp. Sci. 73(3)*, 21-43

9. Lichtenstein, O., A.Pnueli, (1985) Checking that finite state concurrent programs satisfy their linear specifications, in *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, 97-107

10. Manna, Z., A.Pnueli, Verification of concurrent programs : the temporal framework, in *The correctness problem in computer science*, R.S.Boyer and J.S.Moore (eds.), 215-73, London Academic Press

11. Peled, D., All from One, One for All : On model-checking using representatives

12. Pnueli, A., (1977) The temporal logic of programs, in *Proc. 18th ANN. IEEE Symp. on Foundations of Computer Science* 46-57

13. Pnueli, A., (1991) The temporal logic of concurrent programs, *Theoretical Computer Science, 13*, 45-60

14. Thiagarajan P.S., (1995) A Trace Consistent Subset of PTL, *LNCS 962, CONCUR '95 : Concurrency theory. Insup Lee, Scott A Smolka (eds.), 6th International Conference, Philadelphia, USA*

15. Thomas, W., (1990) Automata on Infinite Objects, in J.V.Leeuwen (ed.), *Handbook of theoretical computer science*, Vol. B, Elsevier Science Publ., Amsterdam, 133-191

16. Valamari, A., (1990) Stubborn Sets for Reduced State Space Generation, LNCS 483, 491-515

17. Vardi, M.Y., An Automata-Theoretic Approach to Linear Temporal Logic

18. Vardi, M.Y., P.Wolper, (1994) Reasoning about infinite computations, *Information and Computation 115(1)*, 1-37

19. Wolper, P., M.Y.Vardi, A.P.Sistla, (1983) Reasoning about infinite computation paths, in *Proc. 24th IEEE Symposium on Foundations of Computer Science*, 185-194, Tucson