Monitoring Execution Traces using Metric Alternating Automata

By

Ghulam Mujtaba

Submitted to the Department of Computer Science in partial fulfillment of the requirement for the degree of Master of Science (M.Sc) at the UNIVERSITÄT DES SAARLANDES

Supervised by

Prof. Bernd Finkbeiner
I hereby declare that this thesis is entirely my own work. In case of any inclusion of the work done by others, the original source has been cited. I have used the resources mentioned in the reference.

Saarbrücken
Ghulam Mujtaba, September 2005
Abstract

We present an automata based verification framework to monitor a running system against a high level specification. Our framework includes specification formalisms to express system properties and verification algorithms to check an execution trace of a system against the intended behavior.

Linear Temporal Logic (LTL) is a widely used specification language to express temporal properties of a system. We present Bounded Temporal Logic (BTL), which extends LTL by parameterizing temporal operators with time bounds. As compared to LTL, BTL is a natural and a more compact formalism to express time-bounded temporal properties.

In automata based verification, alternating automata (AA) are commonly used as intermediate representations of LTL specifications, because of their succinctness and linear translations from LTL formulae. However, the translation from BTL formulae to AA is exponential. We present metric alternating automata (MAA), a variant of AA, and a linear translation mechanism from BTL formulae to MAA.

A collection of algorithms, based on MAA, are presented to monitor an execution trace against a BTL specification. We start with specialized algorithms for different sublogics of BTL, and then present a generic algorithm which handles all the sublogics of BTL (including LTL) and performs as efficiently as the corresponding specialized algorithms for those sublogics.
Acknowledgement

First and foremost I would like to thank Prof. Bernd Finkbeiner for his countless efforts in helping and supporting me during the course of my thesis work. There were times when I had lost hope, but due to his encouragement I am able to complete my work. I would also like to thank Lars Kuhtz for his constructive criticism and useful suggestions. I am overwhelmed with gratitude for Sussane Schneider who had always been there to help us in all sorts of administrative issues.

Last but not the least I would like to thank my family members and friends for their help and support during the long years of my academic career.
# Contents

Abstract i

Acknowledgement iii

1 Preliminaries 1

1.1 Introduction and Motivation .......................... 1

1.2 Background ............................................. 3

1.2.1 Linear Time Temporal Logic ......................... 3

1.2.2 Run-Time Verification ................................. 3

1.2.3 Alternating Automata .................................. 4

1.3 Overview ............................................... 4

2 Metric Alternating Automata 5

2.1 Introduction .............................................. 5

2.2 Syntax and Semantics ..................................... 6

2.3 Run of a Metric Alternating Automaton ................... 7

2.4 Bounded Temporal Logic .................................. 9

2.5 BTL to MAA translation .................................. 11

3 Verification Algorithms 15

3.1 Introduction .............................................. 15

3.2 Forward-Backward Algorithm ............................ 16

3.2.1 Forward Expansion .................................... 17

3.2.2 Backward Evaluation .................................. 18

3.2.3 How it works .......................................... 20

3.2.4 Example ............................................... 21

3.3 Optimized Forward-Backward Algorithm .................. 26

3.3.1 Forward Expansion .................................... 27

3.3.2 Backward Evaluation .................................. 28

3.3.3 How it works .......................................... 29

3.3.4 Example ............................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Optimized Breadth-First Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>3.4.1 Example</td>
<td>36</td>
</tr>
<tr>
<td>4 Generic Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>41</td>
</tr>
<tr>
<td>4.2 The Algorithm</td>
<td>42</td>
</tr>
<tr>
<td>4.2.1 Translation from a Configuration to a DAG</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2 Forward Expansion</td>
<td>45</td>
</tr>
<tr>
<td>4.2.3 Backward Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>4.2.4 How it works</td>
<td>46</td>
</tr>
<tr>
<td>4.2.5 Example</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Slightly-Restricted Bounded Temporal Logic</td>
<td>51</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>55</td>
</tr>
<tr>
<td>A OPrA - A Runtime Monitoring Software</td>
<td>59</td>
</tr>
<tr>
<td>A.1 General Description</td>
<td>59</td>
</tr>
<tr>
<td>A.1.1 Formula Translation</td>
<td>59</td>
</tr>
<tr>
<td>A.1.2 Program Instrumentation</td>
<td>60</td>
</tr>
<tr>
<td>A.1.3 Event Recognizer</td>
<td>61</td>
</tr>
<tr>
<td>A.1.4 Runtime Monitoring</td>
<td>61</td>
</tr>
<tr>
<td>A.2 Specification Script</td>
<td>61</td>
</tr>
<tr>
<td>A.3 Software Architecture</td>
<td>64</td>
</tr>
</tbody>
</table>
Chapter 1

Preliminaries

1.1 Introduction and Motivation

This thesis presents a framework to monitor running systems against high level specifications. Our framework includes specification formalisms to express system properties and verification algorithms to monitor a system execution against the intended behavior. Linear-time Temporal Logic (LTL) [15] is a widely used specification language to model the properties of reactive systems. Several verification tools based on LTL have been developed in academia and in industry. Generally, the temporal properties expressed in LTL are interpreted over an infinite moment in future. LTL can also be used to express temporal properties that are interpreted over a finite moments in future by using a sequence of next operators. However, this approach is adequate only if those finite moments are very small in number. We present Bounded Temporal Logic (BTL) as a compact alternative to LTL. In BTL, temporal operators are parameterized with both finite and infinite time bounds. For example, we want a system to behave in the following way:

Always, when a process $P$ is permitted to enter the critical section $S$, it will eventually leave $S$ within $n$ system steps.

Let $p$ represents the proposition “$P$ enters $S$” and $q$ the proposition “$P$ leaves $S$”, then the above property can be expressed in BTL as follows:

\[ \square_{[0,\infty]}(p \rightarrow \diamond_{[0,n-1]}q). \]

One may argue that the same property can also be expressed in classical LTL, using $n$ next operators, as follows:

\[ \square(p \rightarrow (q \lor \bigcirc(q \lor \bigcirc(...(q \lor \bigcirc q)))...)). \]
However, the size of the specification is exponentially larger compared to the BTL formula.

BTL not only expresses system properties in a compact form, but also leads to better complexity of the verification algorithms, which are discussed in Chapter 4. BTL can also be seen as a special case of metric temporal logic (MTL) with discrete time intervals. MTL, which was introduced by Koymans in [13], is an extension of LTL used to express quantitative temporal properties of reactive systems.

In the automata-theoretic approach to verification, a verification problem is first reduced to an automata-theoretic problem [4], such as taking the intersection of two automata, emptiness checking, membership checking etc. The run-time verification problem can be reduced to the membership checking problem of automata [8], i.e., a trace of the system execution satisfies the specification, if it is accepted by the corresponding specification automaton.

Alternating automata are efficient datastructures for run-time verification due to their succinctness and linear translation from LTL specifications [8]. Alternating automata, in the context of run-time verification, were first studied in [6], where a collection of algorithms based on alternating automata is presented. The algorithms traverse the trace in different ways, i.e., Breadth-First, Depth-First and Reverse. Two of the algorithms, Depth-First and Reverse, need the trace of the system execution to be available offline. The Breadth-First algorithm can work online, but it has an exponential space complexity in the size of input formula.

We introduce metric alternating automata (MAA) and a translation mechanism from BTL formulas to MAA. MAA extend alternating automata by introducing time constraints on transitions. A transition is enabled in MAA, if and only if, the corresponding time constraint is fulfilled. Our algorithms monitor a system execution against an intended behavior by checking whether a given execution trace $\rho$ is accepted by the specification automaton $A$. Three algorithms, the Forward-Backward algorithm, the Optimized Breadth-First algorithm and the Generic algorithm are presented to check $\rho$ against $A$. The Forward-Backward unrolls $A$ into a DAG (directed acyclic graph) $\tilde{G}$ and then traverses $\tilde{G}$ backwards to detect an accepting run $\rho$ in $A$. The space complexity of the algorithm is quadratic in the size of $A$ and linear is the size of $\rho$. The Optimized Breadth-First algorithm, which extends the Breadth-First algorithm presented in [6], has space complexity constant in $\rho$ and exponential in $A$. The algorithms outperform each other for different sublogics of BTL. The Generic algorithm, presented in Chapter 4, combines the Breadth-First and Forward-Backward algorithm to optimize the overall complexity.

For a BTL specification $\varphi$ and an execution trace $\rho$, the Generic algorithm, in the worst case, runs in space $O(C^{1/|\varphi|})$ and time $O(C^{1/|\varphi|} \cdot |\rho|)$, where $C$ is the maximum integer constant appearing in $\varphi$. For the same specification when translated
into classical LTL, the best known algorithms run in space $O(2^{C^*|\varphi|})$ and time $O(2^{C^*|\varphi|} \cdot |\rho|)$.

We also present a restricted sublogic of BTL, called *Slightly-restricted Bounded Temporal Logic* (SBTL). SBTL is strong enough to express most of the system properties commonly used in practice. For SBTL formula, the space complexity of the *Generic* algorithm reduces to $O(|\varphi| \cdot C^2 \cdot 2^{2(|\varphi|)})$.

## 1.2 Background

### 1.2.1 Linear Time Temporal Logic

*Linear time temporal logic* (LTL) is based on a linear model of time. Linear time means that each moment in time has one and only one successor. Time is bounded in the past (i.e., we have a start time) and unbounded in the future (i.e., there are infinitely many moments in the future). Reactive systems are known for their ongoing interaction with their surroundings and their non-terminating behavior [2]. Thus, a system execution is an infinite sequence of states. The correctness of a system execution is proved by checking the temporal ordering of the infinite state sequence. Temporal logic is a simple and natural way to specify the ordering of events without referring to absolute time measures.

Temporal properties are mainly classified into *safety* properties and *liveness* properties [17]. A *safety* property asserts that “nothing bad” will happen in future, while a *liveness* property asserts that “something good” will eventually happen in future. The *Mutual exclusion* problem is a famous example of a *safety* property [2], where we require that no two processes are able to access a shared resource simultaneously. An example of a *liveness* property is the *guaranteed service* [2], where it is required that each request for a certain resource is eventually entertained.

### 1.2.2 Run-Time Verification

*Model checking* (MC) [16] is a widely used technique to prove a system’s design correctness against a formal specification. Despite intensive research, applications of MC are mostly restricted to finite state systems. Software model checking is extremely hard due to the large (possibly infinite) state space to be explored. Thus, MC is mostly used for verifying *Communication Protocols, Hardware Circuits* or some abstract representations of *Software Systems*. Since one can not completely rule out possible differences between the actual implementation and the abstract
model, verifying an abstract model does not guarantee the correctness of the actual system.

One of the alternatives is to check the system while it is running. The research community has proposed run-time verification as an light-weight alternative to MC for Software Systems [6]. In run-time verification, a particular trace of the system execution is verified against a formal specification, instead of exploring the whole state space. Although, not as comprehensive as MC, Run-time verification has a lot of attention in recent years in the area of software verification.

1.2.3 Alternating Automata

In automata based verification, a verification problem is reduced to a known automata-theoretic problem. Unfortunately, most of the operations on nondeterministic automata are very costly and are not feasible in many cases. Alternating automata are efficient data structures for verification purposes. For example, the complementation of a nondeterministic automaton is exponential, while complementation of an alternating automaton is a linear operation.

Alternating automata generalize nondeterministic automata by allowing a choice to be marked as either universal or existential. A universal choice means that a word is accepted if all the paths through the automaton lead to acceptance. An existential choice means that a word is accepted if one of the paths through the automaton leads to acceptance. A run of a nondeterministic automaton is a sequence of states, whereas a run of an alternating automaton, because of universal choice, is a tree.

1.3 Overview

We begin with an introduction of metric alternating automata (MAA) and Bounded Temporal Logic (BTL) in Chapter 2. In Chapter 3, we present verification algorithms based on MAA, and analyze their complexities. In Chapter 4, we present the Generic algorithm and analyze its complexity for different sublogics of BTL. Chapter 5 constants the concluding remarks about our work. A brief tutorial of the online monitoring tool ‘OPrA’, which implements the framework, is presented in the appendix.
Chapter 2

Metric Alternating Automata

2.1 Introduction

In the automata-theoretic approach to verification, a verification problem is first reduced to an automata-theoretic problem, and then solved by the methods known already for automata. The most common practice is to translate the temporal logic specification $\varphi$ into an automaton $A$, and then perform different operations on $A$. For example, runtime monitoring of a system’s execution can be reduced to the membership checking problem. We use a similar approach to the one discussed in [6], where an LTL formula $\varphi$ is translated into an alternating automaton $A$ and then a given execution trace $\rho$ is checked against $A$ for acceptance.

In our framework, we use Bounded Temporal Logic (BTL), presented in Section 2.4, to express time-bounded temporal properties. The translation from a BTL formula to an alternating automaton is exponential. We introduce a variant of alternating automata, called metric alternating automata (MAA). MAA associate time-bounds on transitions, which leads to a linear translation from BTL formulae to MAA.

Metric alternating automata, like alternating automata, allow dual branching modes, that is, a universal branching mode and an existential branching mode. Universal branching, represented by conjunction over subautomata, means that a state sequence is accepted by the automaton if all paths lead to acceptance, whereas existential branching, represented by disjunction over subautomata, means that a state sequence is accepted if any of the path leads to acceptance. Nodes of metric alternating automata are marked as either accepting or rejecting, represented by 1 and 0 respectively. A run of a metric alternating automaton, due to conjunction over subautomata, is a tree instead of a sequence of states in case of nondeterministic automaton. A run tree is accepting if every path in the tree ends at an accepting node.
2.2 Syntax and Semantics

Definition 2.1. (Metric Alternating Automaton) Given a set of clocks $C$, a metric alternating automaton $A$ is defined as follows:

$$A ::= \epsilon_A \quad \text{empty automaton}$$

$$\vdash N \quad \text{automaton node}$$

$$\vdash A \land A \quad \text{conjunction of two automata}$$

$$\vdash A \lor A \quad \text{disjunction of two automata}$$

$$\vdash A^d \quad \text{metric sub-automaton}$$

where $N$ is a node of a metric alternating automaton defined in Definition 2.2, and $d \in \mathbb{N} \cup \{\infty\}$ is a metric.

Definition 2.2. (Node of a Metric Alternating Automaton) Given an input alphabet $Q$, a node $N$ of a metric alternating automaton is defined as follows:

$$N ::= \langle \nu, \text{acc} \rangle \quad \text{leaf node}$$

$$\vdash \langle F, \delta, \text{acc} \rangle \quad \text{timer node}$$

where,

- $\nu \in Q$ is an input letter.
- $\delta$ is a sub-automaton expressing the next-state relation.
- $F$ is a boolean function over $\mathbb{N} \cup \{\infty\}$.
- acc $\in \mathbb{B}$.

In general, a clock of a timed automaton represents the time elapsed since the last reset. In our formalism, a clock $c$ is a decreasing sequence of positive integers, such that $c_{i+1} = c_i - 1$ for all $i > 0$.

Nodes of a metric alternating automaton are classified into two types. A leaf node that represents a state formula $\nu$ and a timer node represents a boolean function $F$ over a set of clocks.

Example 2.1. Figure 2.1 illustrates the construction of a metric alternating automaton $A$ that specifies the timed-language $L$ given below:
Figure 2.1: Metric alternating automaton for the timed language \( L = \{ \sigma | \forall i.((\sigma_i \models a) \rightarrow \exists j > i.(\sigma_j \models b \text{ and } j - i \leq 4)) \} \).

\[ L = \{ \sigma | \forall i.((\sigma_i \models a) \rightarrow \exists j > i.(\sigma_j \models b \text{ and } j - i \leq 4)) \} \]

where \( \sigma_i \) and \( \sigma_j \) represent the \( i^{th} \) and the \( j^{th} \) input letter in \( \sigma \) respectively, and \( i,j \) are positive integers.

The language \( L \) consists of all sequences such that if \( a \) holds at a certain position, then \( b \) must holds within four time units. In simple words, we can say that the maximum distance between the position where \( a \) holds and the position \( b \) holds is at most four time units.

The nodes of the automaton are graphically represented by rounded squares and circles, as shown in Figure 2.1. A rounded square represents an accepting node, while a circle represents a rejecting node. An execution trace is a member of the language \( L \) if it is accepted by \( A \). Acceptance conditions of \( A \) are discussed in the sections to follow.

**Definition 2.3. (Execution Trace)** Given a set \( \nu \) of system variable, an execution trace \( \rho : s_0, s_1, s_2, \ldots \) is an infinite sequence of states, where a state \( s_i \) is a truth assignment to the variables in \( \nu \).

### 2.3 Run of a Metric Alternating Automaton

In automata theory, a run of a nondeterministic automaton is a sequence of states. However, a run of a metric alternating automaton, because of the conjunction over sub-automata, is a tree.
Definition 2.4. (Run Tree) A run tree of a metric alternating automaton is defined as follows:

\[
T ::= \varepsilon_T \quad \text{empty tree} \\
| \ll N, T \rr \quad \text{a node with a subtree} \\
| T \cdot T \quad \text{composition of two subtrees}
\]

where \( N \) represents a node of a metric alternating automaton.

Definition 2.5. (Run) Given an execution trace \( \rho \) and a metric alternating automaton \( A \), a run tree \( T \) with an associated clock \( c \) is called a run of \( \rho \) in \( A \) if one of the following conditions holds:

\( A = \varepsilon \) and \( T = \varepsilon_T \)

\( A = \ll \nu, acc \rr \) and \( T = \ll \ll \nu, acc \rr, \varepsilon_T \rr, \) and \( \rho_0 \models \nu \)

\( A = \ll F, \delta, acc \rr \) and \( \rho \neq \varepsilon, T = \ll \ll F, \delta, acc \rr, T' \rr, F(c) = \text{true} \) and \( T' \), with a clock \( c - 1 \), is a run of \( \rho_1, \rho_2, \rho_3 \ldots \) in \( \delta \).

\( A = A_1 \land A_2 \) and \( T = T_1 \cdot T_2, \) with a clock \( c \), is a run of \( \rho \) in \( A_1 \), and \( T_2, \) with a clock \( c \), is a run of \( \rho \) in \( A_2 \).

\( A = A_1 \lor A_2 \) and \( T \), with a clock \( c \), is a run of \( \rho \) in \( A_1 \) or \( T \), with a clock \( c \), is a run of \( \rho \) in \( A_2 \).

\( A = A_0^d \) and \( T \), with a clock \( d \), is a run of \( \rho \) in \( A_0 \).

Definition 2.6. (Accepting Run) A run \( T \) of a metric alternating automaton \( A \) is accepting if all the branches of \( T \) end at accepting nodes.

Definition 2.7. (Model) An execution trace \( \rho \) is a model of a metric alternating automaton \( A \), denoted as \( \rho \models A \) if there exists an accepting run of \( \rho \) in \( A \).

Definition 2.8. (Language) The language of metric alternating automaton \( A \), denoted as \( L(A) \), is the set of all models of \( A \).

Example 2.2. Figure 2.2 shows two run \( T_1 \) and \( T_2 \) of the metric alternating automaton \( A \) (shown in figure 2.1) against two different input sequences \( \sigma_1 \) and \( \sigma_2 \) given below:

\( \sigma_1 = \ll \neg a, b \rr \rightarrow \ll a, \neg b \rr \rightarrow \ll \neg a, \neg b \rr \rightarrow \ll \neg a, \neg b \rr \rightarrow \ll a, \neg b \rr \rightarrow \ll a, b \rr \)

\( \sigma_2 = \ll a, b \rr \rightarrow \ll a, \neg b \rr \rightarrow \ll \neg a, \neg b \rr \rightarrow \ll \neg a, \neg b \rr \rightarrow \ll a, \neg b \rr \rightarrow \ll \neg a, \neg b \rr \).
Figure 2.2: Runs $T_1$ and $T_2$ of input sequences $\sigma_1$ and $\sigma_2$ in the specification automaton $A$, shown in Figure 2.1.

According to Definition 2.7, $\sigma_1 \models A$ if there is an accepting run of $\sigma_1$ in $A$. The run tree $T_1$ is accepting, by Definition 2.6, as all the branches end at accepting nodes ($n_0, n_1, n_3$). Thus, $\sigma_1$ is a model of $A$. On the other hand, $\sigma_2$ produces $T_2$ which, by our definition, is not accepting, as some of the branches end at the rejecting node ($n_2$), as shown in the figure. Since there does not exist any accepting run for the input sequence $\sigma_2$, the sequence is rejected by the specification automaton $A$ and $\sigma_2$ is not a model of $A$.

### 2.4 Bounded Temporal Logic

We present The Bounded Temporal Logic (BTL) as an alternative specification language to Linear-time Temporal Logic (LTL). LTL, discussed briefly in Chapter 1, does not succinctly express the time-bounded temporal properties. BTL allows one to express time-bounded temporal properties in a nice compact form
by parameterizing temporal operators with both finite and infinite time bounds. The time-bounded properties, when expressed in BTL have size logarithmically smaller compared to LTL.

For example, the property that “Always every p-state is followed by a q-state within 100 time units” can be modelled in BTL as follows:

$$\Box_{[0,\infty)}(p \rightarrow \Diamond_{[0,100)} q).$$

Similarly, the property that “Within 20 system steps a p-state triggers an infinite sequence of q-state” can be expressed in BTL as follows:

$$\Diamond_{[0,19]}(p \rightarrow \Box_{[0,\infty)} q).$$

**Definition 2.9.** Bounded Temporal Logic (BTL) Given a set of propositions $P$, Bounded Temporal Logic BTL can be inductively defined as follows:

$$\varphi := p | \varphi \lor \varphi | \varphi \land \varphi |
\Box_{[x_1,x_2]} \varphi | \Diamond_{[x_1,x_2]} \varphi |
\varphi U_{[x_1,x_2]} \varphi$$

where $p \in P$ is a proposition, $x_1, x_2 \in \mathbb{N} \cup \{\infty\}$ and $x_1 \leq x_2 \leq \infty$.

Given an execution trace $\rho$, a state formula $P$, BTL formulae $\varphi$ and $\psi$, a BTL formula holds at position $0 \leq j < |\rho|$, written as $(\rho, j) \models \varphi$, is formally described as follows:

For a state formula:

$$(\rho, j) \models p \text{ iff the assertion } p \text{ holds at } \rho_j.$$ 

For the boolean connectives:

$$(\rho, j) \models \varphi \land \psi \text{ iff } (\rho, j) \models \varphi \text{ and } (\rho, j) \models \psi,$$

$$(\rho, j) \models \varphi \lor \psi \text{ iff } (\rho, j) \models \varphi \text{ or } (\rho, j) \models \psi.$$ 

For temporal operators

$$(\rho, j) \models \Box_{[x_1,x_2]} \varphi \text{ iff } (\rho, i) \models \varphi \text{ for all } i \in [x_1 + j, x_2 + j]$$

$$(\rho, j) \models \Diamond_{[x_1,x_2]} \varphi \text{ iff } (\rho, i) \models \varphi \text{ for some } i \in [x_1 + j, x_2 + j]$$

$$(\rho, j) \models \varphi U_{[x_1,x_2]} \psi \text{ iff } (\rho, i) \models \psi \text{ for some } i \in [x_1 + j, x_2 + j] \text{ and } (x_1 + j = i \text{ or } (\rho, k) \models \varphi \text{ for all } k \in [x_1 + j, i - 1]).$$
2.5. BTL TO MAA TRANSLATION

where \( x_2 + j < |\rho| \).

\[
\begin{align*}
(\rho, j) &\models \Box_{[x_1, x_2]} \varphi \quad \text{iff} \quad (\rho, i) \models \varphi \text{ for all } i \in [x_1 + j, |\rho| - 1] \\
(\rho, j) &\models \Diamond_{[x_1, x_2]} \varphi \quad \text{iff} \quad (\rho, i) \models \varphi \text{ for some } i \in [x_1 + j, |\rho| - 1] \\
(\rho, j) &\models \varphi \mathcal{U}_{[x_1, x_2]} \psi \quad \text{iff} \quad (\rho, i) \models \psi \text{ for some } i \in [x_1 + j, |\rho| - 1] \text{ and }
\quad (x_1 + j = i \text{ or } (\rho, k) \models \varphi \text{ for all } k \in [x_1 + j, i - 1])
\end{align*}
\]

where, \( x_1 + j < |\rho| \leq x_2 + j \).

\[
\begin{align*}
(\rho, j) &\models \Box_{[x_1, x_2]} \varphi \quad \text{always true} \\
(\rho, j) &\models \Diamond_{[x_1, x_2]} \varphi \quad \text{always false} \\
(\rho, j) &\models \varphi \mathcal{U}_{[x_1, x_2]} \psi \quad \text{always false}
\end{align*}
\]

where, \( x_1 + j \geq |\rho| \).

Given BTL formula \( \varphi \) and \( \Psi \), weak until \( \mathcal{W} \) and dual until \( \mathcal{R} \) operators can be expressed as follows:

\[
\begin{align*}
\bullet \ (\rho, j) &\models \varphi \mathcal{W}_{[x_1, x_2]} \psi \equiv (\rho, j) \models \varphi \mathcal{U}_{[x_1, x_2]} \psi \text{ or } (\rho, j) \models \Diamond_{[x_1, x_2]} \varphi \\
\bullet \ (\rho, j) &\models \varphi \mathcal{R}_{[x_1, x_2]} \psi \equiv (\rho, j) \models \neg(\neg \varphi \mathcal{U}_{[x_1, x_2]} \neg \psi)
\end{align*}
\]

Note 2.1. BTL does not support the the next operator, however \( \Diamond_{[1,1]} \varphi \) can be used to specify that \( \varphi \) holds at the next state.

2.5 BTL to MAA translation

In this section, we present the translation from a BTL formula to a MAA. To make the translation from a specification to an automaton more readable, we define translation functions \( \Psi_0, \Psi_1, \Psi_2, \Psi_3 \) and \( \Psi_4 \) which we later use in the construction of the MAA. Given MAA \( \mathcal{A}, \mathcal{A}_1, \mathcal{A}_2 \), the translation functions \( \Psi_0, \Psi_1, \Psi_2, \Psi_3 \) and \( \Psi_4 \) are be defined as follows:

\[
\begin{align*}
\Psi_0(\mathcal{A}) &= (\langle \lambda x. x > 0, \Psi_0(\mathcal{A}), 0 \rangle) \lor (\langle \lambda x. x = 0, \epsilon, 1 \rangle) \land \mathcal{A} \\
\Psi_1(\mathcal{A}) &= (\langle \lambda x. x > 0, \Psi_1(\mathcal{A}), 1 \rangle) \lor (\langle \lambda x. x = 0, \epsilon, 1 \rangle) \land \mathcal{A} \\
\Psi_2(\mathcal{A}) &= (\langle \lambda x. x > 0, \Psi_2(\mathcal{A}), 1 \rangle \lor \langle \lambda x. x = 0, \epsilon, 1 \rangle) \land \mathcal{A} \\
\Psi_3(\mathcal{A}_1, \mathcal{A}_2) &= (\langle \lambda x. x > 0, \Psi_3(\mathcal{A}_1, \mathcal{A}_2), 0 \rangle \land \mathcal{A}_1) \lor \mathcal{A}_2 \\
\Psi_4(\mathcal{A}) &= (\langle \lambda x. x > 0, \Psi_4(\mathcal{A}), 0 \rangle) \lor \mathcal{A}
\end{align*}
\]
CHAPTER 2. METRIC ALTERNATING AUTOMATA

The translation rules from BTL specification \( \varphi \) to metric alternating automaton \( A(\varphi) \) are given below:

For a state formula \( p \):

\[
A(p) = \langle p, 1 \rangle
\]

For BTL formulae \( \varphi \) and \( \psi \):

\[
\begin{align*}
A(\varphi \land \psi) &= A(\varphi) \land A(\psi) \\
A(\varphi \lor \psi) &= A(\varphi) \lor A(\psi) \\
A(\Box_{[x_1, x_2]} \varphi) &= (\Psi_1((\Psi_2(A(\varphi))))^{x_2-x_1})^{x_1-1} \\
A(\Diamond_{[x_1, x_2]} \varphi) &= (\Psi_0((\Psi_4(A(\varphi))))^{x_2-x_1})^{x_1-1} \\
A(\varphi U_{[x_1, x_2]} \psi) &= (\Psi_0((\Psi_3(A(\varphi), A(\psi))))^{x_2-x_1})^{x_1-1}
\end{align*}
\]

**Note 2.2.** In translation from BTL to automaton, all BTL formulae are assumed to be in negation normal form, that is, all the negations have been pushed to state level such that there is no temporal operator within the scope of negation.

Given BTL formulae \( \varphi \) and \( \psi \), Figure 2.3 shows the construction of metric alternating automata \( A_1, A_2, A_3 \) and \( A_4 \) from the respective BTL formulae \( \Diamond_{[x_1, x_2]} \varphi, \Box_{[x_1, x_2]} \varphi \) and \( \varphi U_{[x_1, x_2]} \psi \).

**Proposition 2.1.** For every BTL formula \( \varphi \), there exists a metric alternating automaton \( A \), and the size of \( A \) is linear in proportion to the size of \( \varphi \).
2.5. BTL TO MAA TRANSLATION

Figure 2.3: Construction of metric alternating automata (MAA) from BTL formulae.
Chapter 3

Verification Algorithms

3.1 Introduction

In this chapter, we present online verification algorithms based on metric alternating automata (MAA). Our algorithms check an execution trace against a formal specification, expressed in bounded temporal logic (BTL). BTL, as discussed in Chapter 2, is a compact version of LTL, where temporal operators are parameterized with discrete time intervals.

To check an execution trace against the intended behavior, we follow a similar approach to the one presented in [6]. The idea is to translate a BTL specification $\phi$ into a metric alternating automaton $A$, and then check whether a given execution trace $\rho$ is accepted by $A$ or not. $\rho$ is accepted by $A$, according to Definition 2.6, if there exists a run $T$ of $\rho$ in $A$, such that every path through $T$ ends at an accepting node.

Three algorithms are presented to check whether a run $T$ of $\rho$ exists in $A$ or not. The first algorithm, called Forward-Backward algorithm, unrolls $A$ into a configuration tree $T$ and then traverses $T$ backwards, starting from leaves, to detect $T$ as a subtree in $T$. The algorithm follows a very simple technique, but it generates a $T$ that grows exponentially in the size of $\rho$. The second algorithm, called Optimized Forward-Backward algorithm, unrolls the automaton into a more compact datastructure, called a configuration DAG $G$, and then traverses $G$ backwards to detect $T$ in $G$. The space complexity of the second algorithm is linear in the size of $|\rho|$ and quadratic in the size of $A$. The third algorithm is an optimized version of the Breadth-First algorithm presented in [6].
3.2 Forward-Backward Algorithm

As discussed in Section 3.1, the Forward-Backward algorithm detects a run \( T \) of an execution trace \( \rho \) in the specification automaton \( A \). The algorithm first translates \( A \) into a configuration tree \( T \), defined in Definition 3.1, and then evaluates \( T \) backwards to detect \( T \) as a subtree in \( T \). At a given system step, \( T \) represents all the system configurations that are consistent with the specification so far. The initial configuration is computed by translating the specification automaton \( A \) into \( T \). The algorithm works in two phases, i.e., the forward expansion and the backward evaluation of \( T \). The forward expansion expands \( T \) from the leaves, while the backward evaluation evaluates \( T \) backwards, starting from the leaves.

**Definition 3.1. (Configuration Tree)** A configuration tree \( T \), generated by unrolling a metric alternating automaton \( A \), is defined recursively as follows:

\[
T := \epsilon_T \quad \text{empty tree} \\
| \langle p, \langle N, T \rangle, res \rangle \quad \text{node with a sub-tree} \\
| \langle p, \langle T \land T \rangle, res \rangle \quad \text{a conjunctive branching point with two sub-trees} \\
| \langle p, \langle T \lor T \rangle, res \rangle \quad \text{a disjunctive branching point with two sub-trees} \\
| \langle p, \langle d, T \rangle, res \rangle \quad \text{a metric sub-tree}
\]

where, \( N \) is a node of \( A \), \( p \) is a parent configuration tree, \( d \in \mathbb{N} \cup \{\infty\} \) and \( res \in \{-1, 0, 1\} \).

**Procedure Forward(\( S_t \))**

<table>
<thead>
<tr>
<th>Input</th>
<th>A set ( S_t ) of node-clock pairs of a configuration tree ( T ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>A set of node-clock pairs + expansion of ( T ) as side effect.</td>
</tr>
<tr>
<td>begin</td>
<td>[ S'_t \leftarrow \emptyset ]</td>
</tr>
<tr>
<td>for each</td>
<td>for each ( \langle X, c \rangle \in S_t ) do</td>
</tr>
<tr>
<td>if ( X = \langle pp, \langle { F }, \delta, acc \rangle, \epsilon_T \rangle, 0 ) then</td>
<td></td>
</tr>
<tr>
<td>( T' \leftarrow \text{translate}(\delta) )</td>
<td></td>
</tr>
<tr>
<td>( X \leftarrow \langle pp, \langle { F }, \delta, acc \rangle, T' \rangle, 0 \rangle )</td>
<td></td>
</tr>
<tr>
<td>( S'_t \leftarrow S'_t \cup \text{Get-Leaves}(X, T', c-1) )</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>( S'_t )</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 3.2. (Translation from an MAA to a Configuration Tree)** For a metric alternating automaton \( A \), the translation function \( \text{translate}(A) \), from \( A \) to a configuration tree, is defined as follows:
\[
\begin{align*}
\text{translate}(\epsilon_A) &= \epsilon_T \\
\text{translate}(\langle \mathcal{N} \rangle) &= \langle \epsilon_T, \langle \mathcal{N}, \epsilon_T \rangle, 0 \rangle \\
\text{translate}(A_1 \lor A_2) &= \langle \epsilon_T, \langle \text{translate}(A_1) \lor \text{translate}(A_2) \rangle, 0 \rangle \\
\text{translate}(A_1 \land A_2) &= \langle \epsilon_T, \langle \text{translate}(A_1) \land \text{translate}(A_2) \rangle, 0 \rangle \\
\text{translate}(A_d) &= \langle \epsilon_T, \langle d, \text{translate}(A_d) \rangle, 0 \rangle
\end{align*}
\]

**Procedure Get-Leaves** \((p, T, c)\)

**Input**: A parent pointer \(p\), a configuration tree \(T\) and a clock \(c\).

**Output**: A set of node-clock pairs of expanded \(T\) assignment of parent pointer in \(T\) as side effect.

\[
\begin{align*}
\text{begin} \\
\text{switch } T \text{ do} \\
\quad \text{case } \langle p', \langle \mathcal{N}, \epsilon_T \rangle, \text{res} \rangle & \quad p' \leftarrow p \\
\quad \quad \quad \text{return } \{ \langle T, c \rangle \} \\
\quad \text{case } \langle p', \langle T_1 \lor T_2 \rangle, \text{res} \rangle & \quad p' \leftarrow p \\
\quad \quad \quad \text{return } \text{Get-Leaves}(T, T_1, c) \cup \text{Get-Leaves}(T, T_2, c) \\
\quad \text{case } \langle p', \langle T_1 \land T_2 \rangle, \text{res} \rangle & \quad p' \leftarrow p \\
\quad \quad \quad \text{return } \text{Get-Leaves}(T, T_1, c) \cup \text{Get-Leaves}(T, T_2, c) \\
\quad \text{case } \langle p', \langle d, T', \rangle, \text{res} \rangle: & \quad p' \leftarrow p \\
\quad \quad \quad \text{return } \text{Get-Leaves}(T, T', d) \\
\text{end} \\
\text{return } \emptyset
\end{align*}
\]

### 3.2.1 Forward Expansion

A **configuration tree** \(T\), maintained by the **Forward-Backward** algorithm, is expanded at every system step to generate all the possible successor configurations. The procedure **Forward** expands \(T\) from the leaves by adding subtrees to \(T\). **Forward** takes as input a set \(S_t\) of pairs \(\langle \mathcal{X}, c \rangle\), where \(\mathcal{X}\) is a leaf of \(T\) and \(c\) is an associated clock. For each node-clock pair \(\langle p, \langle \mathcal{F}, \delta, acc, \rangle T', c \rangle \in S_t\), the subautomaton \(\delta\) is translated into a **configuration subtree** \(T'\), using the function **translate**. Each newly added \(T'\) is then passed to the procedure **Get-Leaves** with clock value \(c - 1\) to compute the set of node-clock pairs for \(T'\). As a side effect, **Get-Leaves** assigns each subtree in \(T'\) a parent pointer. **Forward** returns as output a set \(S'_t\) of node-clock pairs for the expanded \(T\).
The construction of a configuration tree from a metric alternating automaton is shown in the function translate. Each metric subautomaton with a metric $d$ is translated into a metric subtree $T_{\text{metric}}$, such that the clock associated with $T_{\text{metric}}$ is initialized with $d$.

**Procedure** Eval-Back($T$)

**Input**: A configuration tree $T$.

**Output**: A configuration tree.

begin
    $T_x \leftarrow \epsilon_T$

switch $T$ do
    case $\langle pp, \langle N, T' \rangle, res \rangle$
        $res \leftarrow \text{result}(T')$
        if $res \neq 0$ then
            $T_x \leftarrow \text{Eval-Back}(pp)$
    case $\langle pp, \langle T_1 \land T_2 \rangle, res \rangle$
        $res \leftarrow \text{result}(T_1) \text{ and result}(T_2)$
        if $res \neq 0$ then
            $T_x \leftarrow \text{Eval-Back}(pp)$
    case $\langle pp, \langle T_1 \lor T_2 \rangle, res \rangle$
        $res \leftarrow \text{result}(T_1) \text{ or result}(T_2)$
        if $res \neq 0$ then
            $T_x \leftarrow \text{Eval-Back}(pp)$
    case $\langle pp, \langle c, T' \rangle, res \rangle$
        $res \leftarrow \text{result}(T')$
        if $res \neq 0$ then
            $T_x \leftarrow \text{Eval-Back}(pp)$
if $T_x = \epsilon_T$ then
    return $T$
else
    return $T_x$
end

3.2.2 Backward Evaluation

At each system step the expansion of a configuration tree $T$ is followed by the backward evaluation of $T$. In the backward evaluation, $T$ is traversed backwards to detect an accepting run $T$ in $T$. The Backward evaluation of $T$, as the name
suggestions, starts with the leaves and traverses $T$ backwards. A leaf node $X$ with an associated clock $c$ is checked, using the function \textbf{evaluate} [Definition 3.5], whether it is a part of $T$ or not. The result of the evaluation of $X$, either 1 or $-1$, is propagated upwards following the parent pointer. The evaluation result 1 or $-1$ implies that $X$ is a part of $T$ or $X$ is not a part of $T$ respectively. Similarly, the parent subtree $P_X$ of $X$ is then checked whether it belongs to $T$ or not. The recursive process continues until the root of $T$ is reached or a subtree rooted at a disjunctive or a conjunctive branching point is evaluated to 0.

Disjunctive branching points $\bigvee$ and conjunctive branching points $\bigwedge$ of $T$ have a variable $res$ which is used to mark the status of the branching points. For $\bigvee$, $res = 1, res = -1$ or $res = 0$ implies that at least one of the subtrees rooted at $\bigvee$ belongs to $T$, all the subtrees rooted at $\bigvee$ do not belong to $T$ or at least one of the subtrees rooted at $\bigvee$ is not fully evaluated respectively. Similarly, for $\bigwedge$, $res = 1, res = -1$ or $res = 0$ implies that all the subtrees rooted at $\bigwedge$ belong to $T$, at least one of the subtrees rooted at $\bigwedge$ does not belong to $T$ or at least one of the subtrees rooted at $\bigwedge$ is not fully evaluated respectively.

The procedure \textbf{Eval-Back} propagates the result upwards in $T$ by updating the value of the boolean $res$ for each subtree in $T$. \textbf{Eval-Back} returns as output a subtree $T'$ in $T$ that is evaluated to either 1 or $-1$. The existence of $T$ is detected in $T$, if and only if, \textbf{Eval-back} propagates the result 1 till the root of $T$. Similarly, a rejecting sequence is detected, if and only if, the root of $T$ is evaluated to $-1$.

**Definition 3.3.** For a given configuration tree $T = \langle P, X, res \rangle$, the function \textbf{result}(T) returns $res$.

**Definition 3.4.** Given a value $x \in \{-1, 0, 1\}$, the logical operators \textbf{and} and \textbf{or} used in the procedure \textbf{Eval-back} have the following semantics:

\[
\begin{align*}
-1 \quad \text{and} \quad x &= -1 \\
 x \quad \text{and} \quad -1 &= -1 \\
 0 \quad \text{and} \quad 0 &= 0 \\
 1 \quad \text{and} \quad x &= x \\
 x \quad \text{and} \quad 1 &= x \\
-1 \quad \text{or} \quad x &= x \\
 x \quad \text{or} \quad -1 &= x \\
 0 \quad \text{or} \quad 0 &= 0 \\
 1 \quad \text{or} \quad x &= 1 \\
 x \quad \text{or} \quad 1 &= 1
\end{align*}
\]

At the final position of the trace $\rho$, the algorithm applies the accepting condition and evaluates each leaf in $\mathcal{R}$ using function \textbf{eval-final}.
**Procedure** Eval-Tree($S, eval, \rho_i$)

**Input**: A set $S$ of node-clock pairs of a configuration tree, a function $eval$, and the current system state $\rho_i$.

**Output**: A set of configuration trees.

\[
begin
S \leftarrow \emptyset \\
for each \langle p, \langle N, \epsilon_T \rangle, res \rangle \in S do \\
res \leftarrow eval(\langle N, c \rangle, \rho_n) \\
if res \neq 0 then \\
S \leftarrow S \cup Eval-Back(p)
end
\]

**Definition 3.5.** For a given node-clock pair $\langle N, c \rangle$, and a system state $\rho_i$, the function $eval$ is defined as follows:

- $eval(\langle \langle \nu, acc \rangle, c \rangle, \rho_i) = \begin{cases} 1 & \text{if } \rho_i \models \nu \\ -1 & \text{otherwise} \end{cases}$
- $eval(\langle \langle F, \delta, acc \rangle, c \rangle, \rho_i) = \begin{cases} 0 & \text{if } F(c) = \text{true} \\ -1 & \text{otherwise} \end{cases}$
- $eval(\langle \langle F, \epsilon, acc \rangle, c \rangle, \rho_i) = \begin{cases} 1 & \text{if } F(c) = \text{true} \\ -1 & \text{otherwise} \end{cases}$

**Definition 3.6.** For a given node-clock pair $\langle N, c \rangle$, and the final system state $\rho_n$, the function $eval-final$ is defined as follows:

- $eval-final(\langle \langle F, \delta, 0 \rangle, c \rangle, \rho_n) = -1$
- $eval-final(\langle \langle F, \delta, 1 \rangle, c \rangle, \rho_n) = \begin{cases} 1 & \text{if } F(c) = \text{true} \\ -1 & \text{otherwise} \end{cases}$
- $eval-final(\langle N, c \rangle, \rho_n) = eval(\langle N, c \rangle, \rho_n)$

### 3.2.3 How it works

The main module Forward-Backward takes a program trace $\rho$ and a metric alternating automata $A$ and checks whether $\rho$ is a model of $A$. At system step
3.2. FORWARD-BACKWARD ALGORITHM

### Procedure Forward-Backward(\(\mathcal{A}, \rho\))

**Input**: An automaton \(\mathcal{A}\) and a program trace \(\rho\).

**Output**: A boolean.

\[
\begin{array}{l}
\text{begin} \\
\quad \mathcal{R} \leftarrow \text{translate}(\mathcal{A}) \\
\quad S_t \leftarrow \text{Get-Leaves}(\epsilon, \mathcal{R}, \infty) \\
\quad \text{for } n=1 \ldots |\rho| - 1 \text{ do} \\
\quad \quad S \leftarrow \text{Eval-Tree}(S_t, \text{evaluate}, \rho_n) \\
\quad \quad \text{if } \mathcal{R} \in S \text{ and result}(\mathcal{R}) \neq 0 \text{ then} \\
\quad \quad \quad \text{return result}(\mathcal{R}) \\
\quad \quad \quad S_t \leftarrow \text{Forward}(S_t) \\
\quad \text{return } \mathcal{R} \in \text{Eval-Tree}(S_t, \text{eval-final}[Definition\,3.6], \rho_n) \text{ and} \\
\quad \quad \text{result}(\mathcal{R}) = 1 \\
\text{end}
\end{array}
\]

0, a configuration tree \(\mathcal{R}\) is initialized by translating \(\mathcal{A}\) to \(\mathcal{R}\). The procedure Get-Leaves is called to computes a set \(S_t\) of node-clock pairs for \(\mathcal{R}\). For each subsequent system step, Forward-Backward works as follows:

1. Calls the procedure Eval-Tree, with function evaluate as an argument, to evaluate each \(\langle X, c \rangle \in S\). The evaluation result is propagated upwards in \(\mathcal{R}\) using procedure Eval-Back.

2. Terminates with success or failure if \(\mathcal{R}\) is evaluated to 1 or −1 respectively.

3. Calls the procedure Forward to expand \(\mathcal{R}\) and to compute the successor set of node-clock pairs.

4. Repeats step 1, 2 and 3 until \(\rho\) reaches its last state.

#### 3.2.4 Example

In this section, we discuss the working of the Forward-Backward algorithm based on an example automaton \(\mathcal{A}\), shown in Figure 3.1, and an example execution trace \(\rho = [\langle a, \neg p, \neg q \rangle, \langle \neg a, p, \neg q \rangle, \langle \neg a, \neg p, q \rangle]\). The specification automaton \(\mathcal{A}\) represents the BTL formula \(\Diamond_{[0,\infty]} (a \land ( (\Diamond_{[0,2]} p) \cup_{[0,3]} q ))\).

As discussed earlier, the Forward-Backward algorithm unrolls \(\mathcal{A}\) into a configuration tree \(T\) and then detects a run \(T\) of \(\mathcal{A}\) in \(T\) such that \(T\) starts from the root of \(T\) and ends at the accepting nodes. Figure 3.2 shows different configurations...
of $T$ at each system step. Each metric subtree in $T$ is annotated with an associated clock. The dotted lines represent subtrees in $T$ that do not belong to $T$, the thick lines represent the subtrees that belong to $T$, and the normal lines represent subtrees that are not fully evaluated yet.

To check $\rho$ against $\mathcal{A}$ for acceptance, Forward-Backward works as follows:

**Step 1:**

- The nodes of $T$ are evaluated to either $-1, 1$ or $0$, and the procedure $\text{Eval-Back}$ propagates the evaluation result upwards in $T$, as shown in Figure 3.2(1b). The dotted lines show the propagation of the result $-1$, while a thick line shows the propagation of the result $1$. Recall that an evaluation result is propagated only if it is $-1$(failure) or $1$(success). The timer nodes $n_0, n_3$ and $n_5$ are evaluated to $0$ as the associated time constrains are fulfilled.

- The subtrees in $T$ that are evaluated in the previous step are removed and $T$ is reduced to a compact form, as shown in Figure 3.2(1c).

**Step 2:**

- The procedure Forward expands $T$ from the timer nodes $n_0, n_3$ and $n_5$, as shown in Figure 3.2(2a), with clocks decremented by $1$. 

\[ \mathcal{A}(\Diamond_{[0,\infty]}(a \land ( (\Diamond_{[0,1]}p) \cup_{[0,3]} q ))) \]
3.2. **FORWARD-BACKWARD ALGORITHM**

Figure 3.2: A stepwise construction of a *configuration tree*.

- The procedure **Eval−Back** propagates the result of the evaluation of nodes $n_1, n_2, n_3$ and $n_4$ upwards in $T$, as shown in Figure 3.2(2b). Here, the timer node $n_3$ under the clock scope 0 is also evaluated to $-1$, as the associated time constraint is not fulfilled.

- $T$ is again reduced to a compact form, as shown in Figure 3.2(2c).

**Step 3:**

- The procedure Forward expands $T$ from the timer nodes $n_0$ and $n_5$, as shown in Figure 3.2(3a).
• At the final step, the nodes are evaluated by applying the accepting condition. The node \( n_2 \) is state-satisfied and accepting, while all other nodes are either not state-satisfied \((n_1, n_4)\) or rejecting \((n_0, n_3, n_5)\). The propagation of the evaluation result is shown in Figure 3.2(3b).

• \( \rho \) is accepted, as there exists an accepting run of \( \rho \) in \( \mathcal{A} \). Figure 3.2(3b) shows a \( T \) that starts from the root of \( T \) and ends at accepting node \( (n_2 \) under clock 2).

**Claim 3.1.** For a given execution trace \( \rho \) and a metric alternating automaton \( \mathcal{A} \), let \( T \) be the configuration tree generated by the Forward-Backward by unrolling \( \mathcal{A} \). Then, the size \( Z_n \) of \( T \), at position \( n \) in \( \rho \), is always bounded by \( K \sum_{i=0}^{n} x^i \) and the number of leaves \( L_n \) in \( T \) are bounded by \( X^n \), where \( X = \left| \mathcal{A} \right| \) and \( K \in \mathbb{Z}^+ \).

**Proof.**

**Base Case:** At the system step 1, \( Z_1 \leq K \cdot X \) and \( L_1 \leq X \). This is true as the algorithm, at the first step simply translates \( \mathcal{A} \) into \( T \) and the size of \( T \) is linear in \( |\mathcal{A}| \). The number of leaves are at most \( X \), as the number of nodes in \( \mathcal{A} \) are always bounded by \( X \).

**Induction:** At system step \( n \), we assume that \( Z_n \leq K \sum_{i=0}^{n} X^i \) and \( L_n \leq X^n \).

To complete the induction step, we must prove that at system step \( n + 1 \),

\[
Z_{n+1} \leq K \sum_{i=0}^{n+1} X^i \quad \text{and} \quad L_{n+1} \leq X^{n+1}.
\]

As per our assumption, \( L_n \leq X^n \) at system step \( n \). Each leaf adds a subtree of maximum size \( K \cdot X \) by translating a subautomaton in \( \mathcal{A} \) into a configuration subtree. We have at most \( X^n \) new subtrees added to \( T \) at system step \( n + 1 \), each of size at most \( X \). The sum of the sizes of newly generated subtrees is bounded by \( X^n \cdot K \cdot X \).

The upper bound of \( Z_{n+1} \) of \( T \), at system step \( n + 1 \), after expansion can be computed as follows:

\[
Z_{n+1} \leq K \sum_{i=0}^{n} X^i + (X^n)K \cdot X
\]

\[
\Rightarrow Z_{n+1} \leq K \sum_{i=0}^{n} X^i + X^{n+1}
\]

\[
\Rightarrow Z_{n+1} \leq K \sum_{i=0}^{n+1} x^i
\]
The number of leaves \( L_{n+1} \) in the expanded \( T \) are equal to the number of leaves of newly added subtrees at system step \( n + 1 \). Since the size of each newly generated subtree \( T' \) is bounded by \( X \), the number of leaves in each \( T' \) are at most \( X \).

Hence, the upper bound of \( L_{n+1} \) is computed by the following expression:

\[
L_{n+1} \leq (x^n) \times x
\]
\[
\Rightarrow L_{n+1} \leq X^{n+1}
\]

\[\square\]

**Theorem 3.1.** Given a program trace \( \rho \) and a *metric alternating automaton* \( \mathcal{A} \), Forward-Backward runs in time \( \mathcal{O}(|\mathcal{A}|^{|\rho|}) \) and space \( \mathcal{O}(|\mathcal{A}|^{|\rho|}) \).

**Proof.** As shown in Claim 3.1, the size of the configuration tree \( T \), maintained by Forward-Backward, at system step \( |\rho| \) is bounded by \( K \times \sum_{i=0}^{|\rho|} |\mathcal{A}|^i \).

Since the space required by the algorithm is linear in the size of \( T \), the space complexity of Forward-Backward is \( \mathcal{O}(|\mathcal{A}|^{|\rho|}) \).

Similarly, the running time of Forward-Backward is also linear in \( |T| \). The forward expansion unrolls \( \mathcal{A} \) to \( T \), which is a linear process. During backward evaluation none of the edges are visited twice, therefore backward evaluation is also linear. Thus, the running time of Forward-Backward is \( \mathcal{O}(|\mathcal{A}|^{|\rho|}) \).

\[\square\]

**Theorem 3.2.** Given a program trace \( \rho \) and a *metric alternating automaton* \( \mathcal{A} \), Forward-Backward(\( \mathcal{A}, \rho \)) = *true*, if there exists an accepting run of \( \rho \) in \( \mathcal{A} \).

The correctness of the Forward-Backward directly follows from the definition of an accepting run.
3.3 Optimized Forward-Backward Algorithm

The Forward-Backward algorithm presented in Section 3.2, uses a configuration tree $T$ that grows exponentially in the size of an execution trace. Such an exponential growth of $T$ makes the algorithm inefficient for practical purposes.

Recall that in the Forward-Backward algorithm, the procedure Forward computes a set $S_t$ of pairs $\langle X, c \rangle$ for $T$, where $X$ is a node in $T$ under the scope of a clock $c$. For any two node-clock pairs $\langle \langle p, \langle N', T' \rangle, res \rangle, c \rangle$ and $\langle \langle p', \langle N'', T'' \rangle, res \rangle, c' \rangle$ in $S_t$, the Forward-Backward algorithm constructs two similar successor subtrees $T'$ and $T''$, if $N' = N''$ and $c = c'$. We use the term isomorphic to refer to such node-clock pairs. The above observation motivates us to think about replacing all similar subtrees with just one subtree. The idea seems simple, but it may result into a construction which has multiple predecessors of a single successor. We present a modified version of the configuration tree, called configuration DAG $G$ that allows us to have multiple predecessors. We also modify our Forward-Backward algorithm slightly to work on $G$. The new algorithm is called Optimized Forward-backward algorithm.

**Definition 3.7. (Configuration DAG)** A configuration DAG $G$, generated by unrolling $A$, is defined as follows:

\[
G ::= \epsilon_G \quad \text{empty DAG} \\
\mid \langle S_p, \langle N, c \rangle, res \rangle \quad \text{a leaf} \\
\mid \langle S_p, \langle \wedge, S_s \rangle, res \rangle \quad \text{a conjunctive branching point with a set of sub-DAGs} \\
\mid \langle S_p, \langle \vee, S_s \rangle, res \rangle \quad \text{a disjunctive branching point with a set of sub-DAGs}
\]

where, $N$ is a node of $A$, $S_p$ is a set of parent configuration DAGs, $S_s$ is a set of configuration DAGs, $c \in C$ is a clock and $res \in \{-1, 0, 1\}$.

The above definition allows us to join more than two sub-DAGs either conjunctively or disjunctively. A conjunctive branching point $\wedge$ is evaluated to $true$ if all of the sub-DAGs are evaluated to $true$. Whereas, a disjunctive branching point $\vee$ is evaluated to $true$ if any of the sub-DAGs is evaluated to $true$.

**Definition 3.8. (Translation from an MAA to a Configuration DAG )** For a metric alternating automaton $A$ and a clock $c$, the translation function $\text{translate}(A, c)$ is defined as follows:

\[
\begin{align*}
\text{translate}(\epsilon_A, c) &= \epsilon_G \\
\text{translate}(N, c) &= \langle \emptyset, \langle N, c \rangle, 0 \rangle \\
\text{translate}(A_1 \lor A_2, c) &= \langle \emptyset, \langle \vee, \{\text{translate}(A_1, c)\} \cup \{\text{translate}(A_1, c)\} \rangle, 0 \rangle \\
\text{translate}(A_1 \land A_2, c) &= \langle \emptyset, \langle \wedge, \{\text{translate}(A_1, c)\} \cup \{\text{translate}(A_1, c)\} \rangle, 0 \rangle \\
\text{translate}(A_0^d, c) &= \text{translate}(A_0, d)
\end{align*}
\]
### Procedure Get-Leaves \((S_p, G)\)

**Input**: A set \(S_p\) of parent configuration DAGs, a configuration DAG \(G\).

**Output**: A set of nodes of \(G\) + assignment of parent pointers in \(G\) as a side effect.

```markdown
begin
  switch \(G\) do
  case \(\epsilon_G\)
    return \(\emptyset\)
  case \(\langle S_{pp}, \chi, res \rangle\)
    \(S_{pp} \leftarrow S_p\)
    return \(\{\langle S_{pp}, \chi, res \rangle\}\)
  case \(\langle S_{pp}, \langle \lor, S_s \rangle, res \rangle\)
    \(S_{pp} \leftarrow S_p\)
    return \(\bigcup_{E \in S_s} \text{Get-Leaves}(\{G\}, E)\)
  case \(\langle S_{pp}, \langle \land, S_s \rangle, res \rangle\)
    \(S_{pp} \leftarrow S_p\)
    return \(\bigcup_{E \in S_s} \text{Get-Leaves}(\{G\}, E)\)
  end
```

**Definition 3.9.** For a set \(S_b\) of branching points and a child configuration DAG \(G\), the function \(\text{add-child}(S_b, G)\) is defined as follows:

\[
\text{add-child}(S_b, G) = \bigcup_{(S_p, \langle x, S_s \rangle, res) \in S_b} \langle S_p, \langle x, S_s \cup \{G\} \rangle, 0 \rangle
\]

**Definition 3.10.** For a set \(S_b\) of branching points and a child configuration DAG \(G\), the function \(\text{remove-child}(S_b, G)\) is defined as follows:

\[
\text{remove-child}(S_b, G) = \bigcup_{(S_p, \langle x, S_s \rangle, res) \in S_b} \langle S_p, \langle x, S_s - \{G\} \rangle, 0 \rangle
\]

### 3.3.1 Forward Expansion

The forward phase of the *Optimized Forward-Backward* is similar to the forward phase of the *Forward-Backward* algorithm discussed in Section 3.2.1, i.e., to generate all possible successor configurations by expanding the configuration DAG \(G\). However, the *Optimized Forward-Backward* algorithm does extra computations to keep the size of \(G\) smaller. As stated above, the configuration tree \(T\) maintained
Procedure \texttt{Forward}(S_t)

\textbf{Input}: A set $S_t$ of leaves of a configuration DAG $G$.

\textbf{Output}: A set of leaves of expanded $G +$ expansion of $G$ as a side effect.

\begin{verbatim}
begin
$S'_t \leftarrow \emptyset$
for each $X \in S_t$ do
  if $X = \langle S_p, \langle F, \delta, acc \rangle, c \rangle, 0 \rangle$ then
    $G' \leftarrow \text{translate}(\delta, c - 1)$
    $S_p \leftarrow \text{remove-child}(S_p, X)$
    $S_p \leftarrow \text{add-child}(S_p, G')$
  $S'_t \leftarrow S'_t \cup \text{Get-Leaves}(S_p, G')$
return $S'_t$
end
\end{verbatim}

by the \textit{Forward-Backward} algorithm contains multiple copies of the same sub-trees in $T$. The \textit{optimized forward-backward} algorithm avoids such duplication by merging isomorphic node-clock pairs.

We assume that the union operation $\cup$, beside computing union of two sets, merges the isomorphic leaves. For example, two isomorphic leaves $X_1 = \langle S_p, \langle N, c \rangle, \text{res} \rangle$ and $X_2 = \langle S'_p, \langle N, c \rangle, \text{res} \rangle$ can be replaced by a single leaf $X = \langle S_p \cup S'_p, \langle N, c \rangle, \text{res} \rangle$. By using function \texttt{add-child}, $X$ is added to every element of the set $S_p \cup S'_p$. Similarly, using function \texttt{remove-child}, $X_1$ and $X_2$ are removed from $S_p$ and $S'_p$ respectively.

\subsection{3.3.2 Backward Evaluation}

The \textit{backward evaluation} traverses the configuration DAG $(G)$ backwards to detect a \textit{run} $T$ in $G$. In the \textit{Forward-Backward} algorithm, the procedure \texttt{Eval-Back} propagates the result upwards in the configuration tree following a single parent link. However, in the Optimized \textit{Forward-Backward} algorithm, $G$ allows every sub-DAG $G_{\text{sub}}$ in $G$ to have a set $S_p$ of predecessors. The evaluation result of $G_{\text{sub}}$ is propagated upwards in $G$ following every $p \in S_p$, as shown in the procedure \texttt{Eval-Back}. The operations \texttt{and} and \texttt{or} are applied over a set instead of a pair.

The procedure \texttt{Eval-Back} returns a set $S$ of DAGs that are evaluated to either $-1$ or $1$. To keep $G$ in a compact form, \texttt{Eval-Back} removes all the sub-DAGs in $G$ that are evaluated to $-1$ or $1$. The removal of the sub-DAGs results in a situation where we have a disjunctive or a conjunctive branching point $B$ over a set $\{G'\}$ of size $1$. In such a situation, \texttt{Eval-Back} reduces the evaluation of $B$ to the evaluation of $G'$, and replaces $B$ by $G'$. 
3.3. OPTIMIZED FORWARD-BACKWARD ALGORITHM

<table>
<thead>
<tr>
<th>Procedureursive Forward-Backward Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Eval-Back</strong>: A set $S$ of configuration DAGs and a configuration DAG $G$.</td>
</tr>
<tr>
<td><strong>Input</strong>: A set $S$ of configuration DAGs and a configuration DAG $G$.</td>
</tr>
<tr>
<td><strong>Output</strong>: A set of configuration DAGs.</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>$S' \leftarrow {G}$</td>
</tr>
<tr>
<td>for each $(S_p, (x, S_s), res) \in S$ do</td>
</tr>
<tr>
<td>if $x = \land$ then</td>
</tr>
<tr>
<td>$res \leftarrow \text{and}_{E \in S_s} \text{result}(E)$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$res \leftarrow \text{or}_{E \in S_s} \text{result}(E)$</td>
</tr>
<tr>
<td>if $res \neq 0$ then</td>
</tr>
<tr>
<td>$S' \leftarrow S' \cup \text{Eval-Back}(S_p, (x, S_s), res))$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$S_s \leftarrow S_s - {G}$</td>
</tr>
<tr>
<td>if $S_s = {E}$ then</td>
</tr>
<tr>
<td>$S_p \leftarrow \text{remove-child}(S_p, (x, S_s), res))$</td>
</tr>
<tr>
<td>$S_p \leftarrow \text{add-child}(S_p, E)$</td>
</tr>
<tr>
<td>return $S'$</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

3.3.3 How it works

The main module FORWARD-BACKWARD takes a program trace $\rho$ and a metric alternating automaton $\mathcal{A}$ and checks whether $\rho$ is a model of $\mathcal{A}$ or not. At system step 0, a configuration DAG $\mathcal{R}$ is initialized by translating $\mathcal{A}$ to $\mathcal{R}$. The set $S_t$ of leaves of $\mathcal{R}$ is initialized by the procedure Get-Leaves.

For each subsequent system step, Forward-Backward works as follows:

1. Calls the procedure Eval-DAG, with function evaluate as an argument, to evaluate each $(\mathcal{X}, c) \in S_t$.

2. Terminates with success or failure if $\mathcal{R}$ is evaluated to 1 or $-1$ respectively.

3. Calls the procedure Forward to expand $\mathcal{R}$ and to compute, for $S$, the successor set of node-clock pairs.

4. Repeats step 1, 2, 3 and 4 unless $\rho$ reaches its last state.

At the final position of the trace $\rho$, the algorithm applies the accepting condition and evaluates each leaf in $\mathcal{R}$ using the function eval-final.
Procedure `Eval-DAG(S_t, eval, \rho_i)`

Input: A set $S_t$ of leaves of a configuration DAGs, a function `eval` and the current system state $\rho_i$.

Output: A set of configuration DAGs.

begin
  $S = \emptyset$
  for each $(S_p, \mathcal{X}, res) \in S_t$ do
    $res \leftarrow \text{eval}(\mathcal{X}, \rho_i)$
    if $res \neq 0$ then
      $S \leftarrow S \cup \text{Eval-Back}(S_p, (S_p, \mathcal{X}, res))$
  
return $S$.
end

Procedure `Forward-Backward(A, \rho)`

Input: An automaton $A$ and a program trace $\rho$.

Output: A Boolean.

begin
  $R \leftarrow \text{translate}(A, \infty)$
  $S_t \leftarrow \text{Get-Leaves}(\emptyset, R)$
  for $n = 1 \ldots |\rho| - 1$ do
    $S \leftarrow \text{Eval-DAG}(S_t, \text{evaluate}, \rho_n)$
    if $R \in S$ and result($R$) $\neq 0$ then
      return result($R$)
    $S_t \leftarrow \text{Forward}(S_t)$
  
return $R \in \text{Eval-DAG}(S_t, \text{eval-final}, \rho_n)$ and result($R$) $= 1$
end

3.3.4 Example

The working of the `Forward-Backward` algorithm was discussed, in Section 3.2.4, based on an example execution trace and an example specification automaton.

We take the same automaton $A$, shown in Figure 3.1, and the same execution trace $\rho = [(a, \neg p, \neg q), (\neg a, p, \neg q), (\neg a, \neg p, q)]$ to discuss how `Optimized Forward-Backward` works?

The `Optimized Forward-Backward` algorithm unrolls $A$ into a configuration DAG $G$ and then detects a path $T$ in $G$, such that $T$ starts from the root of $G$ and ends at the accepting nodes. Figure 3.3 shows different configurations of $G$ at each system step. The dotted lines represent sub-DAGs in $G$ that do not belong to $T$. 
3.3. OPTIMIZED FORWARD-BACKWARD ALGORITHM

Figure 3.3: A stepwise construction of a configuration DAG

thick lines represent the sub-DAGs that belong to $T$, and normal lines represent sub-DAGs that are not fully evaluated yet. The clocks are pushed down to the level of the leaves. A leaf of $G$ is represented as $N^c_x$, where $N^c_x$ is a node of $A$ and $c$ is an associated clock.

To check $\rho$ against $A$ for acceptance, Forward-Backward works as follow:

**Step 1:**

- The algorithm translates $A$ into $G$, as shown in Figure 3.3(1a).
- The nodes of $G$ are evaluated to $-1, 1$ or $0$, and the procedure Eval-Back propagates the evaluation result upwards in $G$, as shown in Figure 3.3(1b).
The dotted lines show the propagation of the result $-1$, while a thick line shows the propagation of the result $1$. The timer nodes $n_0, n_3^1$ and $n_5^0$ are evaluated to $0$ as the associated time constrains are fulfilled.

- The sub-DAGs in $G$ that are evaluated in the previous step are removed and $G$ is reduced to a compact form, as shown in Figure 3.3(1c).

**Step 2:**

- The procedure Forward expands $G$ from the timer nodes $n_0, n_3^1$ and $n_5^3$ with clocks decremented by $1$. As shown in Figure 3.3(2a), the sub-DAGs with isomorphic nodes $(n_3^3, n_4^1)$ are merged.

- The procedure Eval-Back propagates the result of the evaluation of the nodes $n_1, n_2^3, n_2^0, n_3^0$ and $n_4^1$ upwards in $G$, as shown in Figure 3.3(2b).

- The compact $G$, after reduction, is shown in Figure 3.3(2c).

**Step 3:**

- The procedure Forward expands $T$ from the timer nodes $n_0$ and $n_5^2$, as shown in Figure 3.3(3a).

- At the final step, the nodes are evaluated by applying the accepting condition. The nodes $n_2^2$ and $n_3^2$ are state-satisfied and accepting, while all other nodes are either not state-satisfied or rejecting. The propagation of the evaluation result is shown in Figure 3.3(3b).

- $\rho$ is accepted as there exists a $T$, such that $T$ starts from the root of $G$ and ends at an accepting node $n_2^2$, as shown in Figure 3.3(3b).

**Theorem 3.3.** Given a program trace $\rho$ and a metric alternating automaton $A$ constructed from a BTL formula $\varphi$, FORWARD-BACKWARD runs in time $O(X^2 \cdot (M_\epsilon + 2) \cdot |\rho|)$ and space $O(X^2 \cdot (M_\epsilon + 2) \cdot |\rho|)$, where $M_\epsilon$ is the largest constant appearing in $\varphi$ (excluding $\infty$) and $X = |A|$.

**Proof.** The procedure FORWARD-BACKWARD maintains a configuration DAG $G$ during its execution and the space complexity of FORWARD-BACKWARD is linear in the size of $G$. At each system step, $G$ is expanded by the procedure Forward form the leaves. The size $Z_n$ of $G$ at a system step $n$ is bounded by $K \sum_{i=0}^{n} M$, where $M \in \mathbb{Z}^+$ is the upper bound of the increment in the size of $G$ at each system step.
The procedure $\text{Forward}$ takes a set $S_t$ of leaves $\langle S_t(N, c), \text{res} \rangle$ of $G$, where $N$ is a node of $A$ paired with an associated clock $c$. The size of $S_t$ is always bounded by $X \ast (M_c + 2)$, as the number of nodes in $A$ are bounded by $X$.

$\text{Forward}$ expands $G$ from each leaf $E \in S_t$. Each $E$ generates a sub-DAG $G'$ by unrolling the subautomaton $\delta$, and the size of $G'$ is bounded by $K \ast X$ (as translation from $\delta$ to $G'$ is linear). Since there are at most $X \ast C$ elements in $S_t$ and every element increments the size of $G$ by at most $X$, the upper bound of increment in the size of $G$ is given by $M = K \ast X^2 \ast (M_c + 2)$.

After merging isomorphic leaves, the number of leaves is reduced to at most $X \ast C$, but the number of edges remains the same.

Hence, the space complexity of $\text{FORWARD-BACKWARD}$ is $O(X^2 \ast (M_c + 2) \ast |\rho|)$.

Overall running time of $\text{FORWARD-BACKWARD}$ is also linear is the size of $G$. The forward expansion of $G$ is linear as the construction from a metric alternating automaton to a configuration DAG is linear. During backward evaluation, none of the edges in $G$ is visited twice. Each edge is immediately removed by the procedure $\text{Eval-Back}$, once it is visited during backward evaluation.

The running time of the algorithm, for $\rho$, is therefore $O(X^2 \ast (M_c + 2) \ast |\rho|)$.

**Corollary 3.1.** Given a program trace $\rho$ and a metric alternating automaton $A$, constructed from an LTL formula $\varphi$, $\text{FORWARD-BACKWARD}$ runs in time $O(X^2 \ast |\rho|)$ and space $O(X^2 \ast |\rho|)$, where $X$ is the size of $A$.

**Proof.** BTL is equivalent to the classical LTL if all the temporal operators have lower bounds $0$ and upper bounds $\infty$, except the special case where a bounded diamond $\langle [1, 1] \rangle$ is used in place of the next operator of classical LTL. The clocks associated with $\langle [1, 1] \rangle$ are initialized with $0$ (as shown in the construction of automata in Chapter 2).

Every node $N$ of $A$ can only be paired with a single clock value from the set $\{0, \infty\}$. The number of leaves of the configuration DAG $G$ is bounded by $X$.

Hence, the space complexity of $\text{FORWARD-BACKWARD}$ is $O(X^2 \ast |\rho|)$ and the running time is also $O(X^2 \ast |\rho|)$.

**Definition 3.11.** (Finitely Bounded Temporal Logic) (FBTL) A sublogic of BTL, such that all of the temporal operators are parameterized with finite interval bounds.

**Lemma 3.4.** Given a program trace $\rho$ and a metric alternating automaton $A$, constructed from FBTL formula $\varphi$, $\text{FORWARD-BACKWARD}$ runs in time $O(X^3 \ast (M_c + 1)^2)$ and space $O(X^3 \ast (M_c + 1)^2)$, where $X$ is the size of $A$, and $M_c$ is the maximum constant appearing in $\varphi$. 

\[ \]
Proof. As proved in Theorem 3.3, the size of a configuration DAG (\(G\)) is incremented by at most \(X^2 \ast (M_c + 2)\) at each system step. The factor \(M_c + 2\) in the above expression reduces to \(M_c + 1\), as in FBTL the number of clocks are bounded by \(M_c + 1\). The number of leaves in \(G\) are bounded by \(X \ast (M_c + 1)\) and increment in \(|G|\) at any system step is bounded by \(X^2 \ast (M_c + 1)\).

The maximum number of steps required to fully evaluate \(\phi\) is bounded by \(C_{sum}\), and \(C_{sum}\) is the sum of the lengths of intervals appearing in \(\phi\).

Thus, the total size of \(G\) is bounded by \(X^2 \ast (M_c + 1) \ast C_{sum}\) or \(X^3 \ast (M_c + 1)^2\) as \(C_{sum} \leq X \ast (M_c + 1)\).

The complexity (both space and running time) of FORWARD-BACKWARD, as proved in Theorem 3.3, is linear in \(|G|\). Hence, for \(\phi\), FORWARD-BACKWARD runs is space \(O(X^3 \ast (M_c + 1)^2)\) and time \(O(X^3 \ast (M_c + 1)^2)\).

\[ \square \]

### 3.4 Optimized Breadth-First Algorithm

The Optimized Forward-Backward algorithm, presented in the previous section, has a quadratic and a linear space complexity in the size of input formula \(\phi\) and length of an execution trace \(\rho\) respectively. The algorithm is useful for checking a smaller prefix of \(\rho\) against a relatively larger size of \(\phi\). To monitor a larger (possibly infinite) \(\rho\), the essential requirement is to bring down the space complexity to constant in the size of \(\rho\). One of the solutions is to use the Breadth-First algorithm presented in [6], as the space complexity of the algorithm is independent of the size of \(\rho\).

We present a modified version of the Breadth-First algorithm that works on metric alternating automata. For a metric alternating automaton \(A\) translated from a BTL formula \(\phi\), the algorithm maintains a set \(S\) of system configurations that are consistent with the prefix of an execution trace \(\rho\) seen so far. A configuration \(C \in S\) is a set of pairs \((N, c)\), where \(N\) is a node of \(A\) and \(c \in \{0, 1, ..., M_c, \infty\}\) is an associated clock, and \(M_c\) is the maximum constant appearing in \(\phi\). \(\rho\) is accepted by \(A\) if there exists at least one \(C \in S\) that leads to an accepting system state.

**Definition 3.12. (Configuration)** A Configuration is a set of pairs \((N, c)\), where \(N\) is a node of a metric alternating automata and \(c\) is an associated clock.

**Definition 3.13.** For a metric alternating automaton \(A\), and an associated clock \(c\), the function \(\text{init}(A, c)\) that computes a set \(S\) of configurations is defined as follows:
3.4. OPTIMIZED BREADTH-FIRST ALGORITHM

\[
\begin{align*}
\text{init}(\epsilon_A, c) &= \emptyset \\
\text{init}(\mathcal{N}, c) &= \{\langle \mathcal{N}, c \rangle \} \\
\text{init}(\langle A_0^d \rangle, c) &= \text{init}(\langle A_0, d \rangle) \\
\text{init}(A_1 \lor A_2, c) &= \text{init}(A_1, c) \cup \text{init}(A_2, c) \\
\text{init}(A_1 \land A_2, c) &= \text{init}(A_1, c) \otimes \text{init}(A_2, c)
\end{align*}
\]

where, \( \otimes \) denotes the following:
\[
\{C_1 \ldots C_n\} \otimes \{C'_1 \ldots C'_m\} = \{C_i \uplus C'_j | i = 1 \ldots n, j = 1 \ldots m\}
\]

**Procedure** BREADTH-FIRST \((A, \rho)\)

**Input**: An automaton \(A\) and a program trace \(\rho\)

**Output**: A boolean

```
begin
  S ← init(A)
  for \(n=0 \ldots |\rho| - 2\) do
    \(S' ← \emptyset\)
    for each \(C\) in \(S\) do
      if state-satisfied\((C, \rho_n)\) then
        \(S' ← S' \cup \text{Successor}(C)\)
        \(S ← S'\)
    \(S' ← \emptyset\)
  for each \(C\) in \(S\) do
    if state-satisfied\((C, \rho_{n-1})\) and final\((C)\) then
      \(S' ← S' \cup \{C\}\)
  end
  return \(S' ← \emptyset\)
end
```

**Definition 3.14.** For a configuration \(C\), the function \(\text{Successor}(C)\) is defined as follows:

\[
\text{successor}(C) = \otimes \text{init}(\delta, c - 1)
\]

\[(\langle \mathcal{F}, \delta, \text{acc} \rangle, c) \in C\]

**Definition 3.15.** For a configuration \(C\) and a system state \(s\), the function state-satisfied\((C, s)\) returns true if

for every \(X ∈ C\), \(\text{evaluate}(X, s)\) returns true.

**Definition 3.16.** For a configuration \(C\), the function \(\text{final}(C)\) returns true if

for every \(\langle A, c \rangle ∈ C\), \(A.\text{acc} = 1\).
The algorithm presented above works exactly like the Breadth-First algorithm presented in [6], but the space complexity of the algorithm does change a big times.

For a metric alternating automaton $A$, translated from BTL formula $\varphi$, the space requirement for Breadth-First is bounded by $2^{X \times (M_c + 2)}$, where $X = |A|$ and $M_c$ is the maximum constant appearing in $\varphi$.

The maximum constant $M_c$ in practice is very large as compared to $X$ and therefore the space complexity of the algorithm is much higher for practical purposes. We present an optimization to the above algorithm that reduce the complexity to exponential in $X$ and quadratic in $M_c$. The optimization reduces the size of $C$ by removing redundant entries in $C$, which ultimately reduces the number of configurations, i.e., the size of $S'$. We claim that if the size of $C'$ is $X + y$, then there exists at least $y$ redundant entries in $C$ that can be removed. $C$ is called a compact configuration if there does not exist a redundant entry in $C$.

3.4.1 Example

For a metric alternating automaton $A$, shown in Figure 3.1, and the execution trace $\rho = [\langle a, \neg p, \neg q \rangle, \langle \neg a, p, \neg q \rangle, \langle \neg a, p, q \rangle]$, the Optimized Breadth-First algorithm checks $\rho$ against $A$ by generating all the possible configurations at each system step. The program trace $\rho$ is accepted if at least one of the configurations generated at the initial system step leads to an accepting configuration. We denote a node-clock pair as $\langle N, c \rangle$ in this section.

The set of configurations generated at system steps 1, 2 and 3 are given below:

Step 1: The function init$(A)$, at system step 1, computes the following set of configurations:

$S_1 = \{ \{ n_0^\infty \}, \{ n_2^\infty \}, \{ n_3^\infty \}, \{ n_1^\infty, n_3^1, n_5^3 \}, \{ n_4^\infty, n_1^4, n_3^3 \} \}$

Two configurations $\{ n_1^\infty \}$ and $\{ n_4^\infty, n_3^1, n_5^3 \}$ in $S_1$ are state-satisfied, and are used to generate successor configurations for the next system step.

Step 2: At system step 2, the set $S_2$ of configurations is computed using function Successor as follows:

$S_2 = \text{successor}(\{ n_0^\infty \}) \cup \text{successor}(\{ n_1^\infty, n_3^1, n_5^3 \})$
Thus, the evaluation of non-timer nodes does not depend on the value of the clock.

Case 1: Proof. To prove our claim, we make the following case distinctions:

Claim 3.2. For a system configuration $C$, any of the pair can be picked randomly. Thus, $\langle N, c \rangle \iff \langle N, c' \rangle$ and any of the pair can be picked randomly.

Case 1: $N = \langle \nu, \text{acc} \rangle$

Case 2: $N = \langle \lambda x. x > 0, \delta, 1 \rangle$

Three configurations $\{n_0^\infty, n_1^0, n_5^2\}$ and $\{n_3^1, n_4^0, n_5^2\}$ in $S_2$ are state-satisfied, and are used to generate successor configurations in Step 3.

Step 3: At the system step 3, the set $S_3$ of configurations is computed using function $\text{successor}$ as follows:

$$S_3 = \text{successor}(\{n_0^\infty\}) \cup \text{successor}(\{n_4^0, n_5^2\}) \cup \text{successor}(\{n_3^1, n_4^0, n_5^2\})$$

$$\Rightarrow \{ \{n_0^\infty\}, \{n_1^\infty, n_2^3\}, \{n_3^\infty, n_1^3, n_5^3\}, \{n_1^\infty, n_1^3, n_5^3\}, \{n_1^3, n_5^3\}, \{n_4^0, n_5^2\}, \{n_4^0, n_5^2\}, \{n_5^1, n_5^3\}, \{n_3^1, n_4^0, n_5^2\}, \{n_3^1, n_4^0, n_5^2\} \}$$

The final set $S_3$ of configurations has one configuration $\{n_2^1\}$ that is accepting and state-satisfied, while all other configurations are either not state-satisfied or not accepting. Thus, $\rho$ is accepted by $A$.

Definition 3.17. (Redundant Entry) For a system configuration $C$, an element $e \in C$ is a redundant entry, iff there exists another element $e' \in C$, such that $e'$ subsumes $e$.

Definition 3.18. (Compact Configuration) A configuration $C$ is a compact configuration if $C$ does not contain a redundant entry.

The Optimized Breadth-First algorithm maintains a set $S_c$ of compact configurations. We assume that the union operation $\cup$, besides taking a union of sets, removes redundant entries on fly. The size of each configuration $C$ is bounded by the number of nodes in the specification automaton $A$, as each element $\langle N, c \rangle \in C$ has distinct $N$. The size of $S_c$, as proved in Theorem 3.7, is bounded by $(M_e + 2)^{|A|}$.

Claim 3.2. For a system configuration $C$, let $\langle N, c \rangle$ and $\langle N, c' \rangle$ be two elements in $C$; either $\langle N, c \rangle$ subsumes $\langle N, c' \rangle$ or $\langle N, c' \rangle$ subsumes $\langle N, c \rangle$.

Proof: To prove our claim, we make the following case distinctions:

Case 1: $N = \langle \nu, \text{acc} \rangle$

The evaluation of non-timer nodes does not depend on the value of the clock. Thus, $\langle N, c \rangle \iff \langle N, c' \rangle$ and any of the pair can be picked randomly.

Case 2: $N = \langle \lambda x. x > 0, \delta, 1 \rangle$
• $c > 0$ and $c' > 0$
  The accepting timer node-clock pairs $\langle \mathcal{N}, c \rangle$ and $\langle \mathcal{N}, c' \rangle$ assert that the specification subformula represented by the subautomaton $\delta$ will hold for the next $c$ and $c'$ system steps respectively. The assertion implies that $\langle \mathcal{N}, c \rangle \Rightarrow \langle \mathcal{N}, c' \rangle$ if $c > c'$ and $\langle \mathcal{N}, c' \rangle \Rightarrow \langle \mathcal{N}, c \rangle$ if $c' > c$. Thus, the pair with a greater clock value subsumes the other one.

• $c = 0$ or $c' = 0$
  Since the function $\lambda$ is evaluated to $false$ when applied to 0, $C$ is state satisfied, iff every element in $C$ is evaluated to $true$. Thus, a node-clock pair with a clock value 0 subsumes all other elements in $C$.

**Case 3:** $\mathcal{N} = \langle \lambda x.x > 0, \mathcal{A}, -1 \rangle$

• $c > 0$ and $c' > 0$
  The rejecting timer node-clock pairs $\langle \mathcal{N}, c \rangle$ and $\langle \mathcal{N}, c' \rangle$ assert that the specification subformula represented by the subautomaton $\delta$ will hold within the next $c$ and $c'$ system steps respectively. The assertion implies that $\langle \mathcal{N}, c \rangle \Rightarrow \langle \mathcal{N}, c' \rangle$ if $c > c'$ and $\langle \mathcal{N}, c' \rangle \Rightarrow \langle \mathcal{N}, c \rangle$ if $c' > c$. Thus, the pair with a lesser clock value subsumes the other one.

• $c = 0$ or $c' = 0$
  Same as in the previous case.

**Case 4:** $\mathcal{N} = \langle \lambda x.x = 0, \epsilon_{\mathcal{A}}, 1 \rangle$

• $c > 0$ or $c' > 0$
  Since the function $\lambda$ is evaluated to $false$ when applied to 0, a node-clock pair with a clock value greater than 0, subsumes all the other elements in $C$.

• $c = 0$ and $c' = 0$
  This case makes two pairs equal, and therefore any of the pairs can be picked randomly.

**Lemma 3.5.** For a metric alternating automaton $\mathcal{A}$ and an execution trace $\rho$, let $S$ be the set of configurations generated by BREADTH-FIRST at any position in $\rho$. Then the maximum size of any configuration $C \in S$ is bounded by $X$, where $X = |\mathcal{A}|$. 
3.4. **OPTIMIZED BREADTH-FIRST ALGORITHM**

*Proof.* As proved in Claim 1.2, any two node-clock pairs appearing in configuration $C$ having the same automaton node can be replaced with a single pair. Therefore, every element $(N, c) \in C$ has a distinct $N$, where $N$ is a node in $A$ and $c$ is an associated clock. The number of automaton nodes is bounded by $X$, therefore the maximum size of any $C \in S$ is bounded by $X$.

The correctness of the *Breadth-First* algorithm follows directly from the correctness of *Breadth-First* algorithm presented in [6].

**Theorem 3.6.** Given an execution trace $\rho$ and a metric alternating automaton $A$, $\text{BREADTH-FIRST}(A, \rho) = \text{true}$, if there exists an accepting run of $\rho$ in $A$.

**Theorem 3.7.** Given an execution trace $\rho$ and a metric alternating automaton $A$ constructed from BTL formula $\varphi$, $\text{BREADTH-FIRST}$ runs in space $O(X \ast (M_c + 2)^X)$ and in time $O(X \ast (M_c + 2)^{2X})$, where $X = |A|$ and $M_c$ is the maximum constant appearing in $\varphi$ (excluding $\infty$).

*Proof.* The procedure *BREADTH-FIRST* generates, at each system step, a set $S$ of configurations. A configuration $C \in S$ is a set of node-clock pairs, such that each $(N, c) \in C$ has a distinct $N$. $C$ is mapping from a set of nodes of $A \{N_1...N_X\}$ to the set of clock values $\{0, 1,..., M_c, \infty\}$. Thus, the size of each $C \in S$ is bounded by $X$ and the size of $S$ is bounded by $(M_c + 2)^X$.

Hence, the space complexity of *BREADTH-FIRST* is $O(X \ast (M_c + 2)^X)$.

The running time of *BREADTH-FIRST* is the running time of the procedure *Forward* at each system step times the number of system steps. At each system step, *Forward* takes a configurations set $S_1$ of maximum size $(M_c + 2)^X$ and constructs a successor configurations set $S_2$ of size at most $(M_c + 2)^X$. The size of each configuration $C \in S_1$ is bounded by $X$. Each node-clock pair $P \in C$ generates a successor configuration set $S'$ of size at most $2^X$. Thus, the total number of configurations generated are bounded by $X \ast 2^X \ast (M_c + 2)^X$ or $X \ast (M_c + 2)^{2X}$.

Hence, the running time of *BREADTH-FIRST* is $O(X \ast (M_c + 2)^{2X})$.

**Corollary 3.2.** Given an execution trace $\rho$ and a metric alternating automaton $A$, constructed from an LTL equivalent BTL formula $\varphi$, *BREADTH-FIRST* runs in time $O(2^X \ast |\rho|)$ and space $O(2^X)$, where $X$ is the size of $A$.

*Proof.* For an LTL equivalent BTL formula $\varphi$, the the size of a set of node-clock pairs is bounded by the size of $\varphi$, as each node $N$ in the specification automaton can be paired with either 0 or $\infty$. Thus, for a given execution trace, the complexity
of BREADTH-FIRST for $\varphi$ is reduced to $O(2^{\|\varphi\|})$ in space and $O(2^{2\|\varphi\|} \times |\rho|)$ in time.
Chapter 4

Generic Algorithm

4.1 Introduction

The algorithms presented in Chapter 3 have their strengths and weaknesses based on the type of input formula. The Breadth-First algorithm is better suited for a larger program trace (possibly infinite) with a smaller finitely bounded intervals. For LTL formulae, where the maximum interval bound other than $\infty$ is 1, the Breadth-First algorithm performs to its maximum potential. The Optimized Forward-Backward algorithm has better running time and is useful for a relatively larger specification size and shorter prefix of a program trace. Thus, the Optimized Forward-Backward algorithm is better suited for an input formula that has only the finitely bounded temporal operators.

The above observation leads to a new framework where both algorithms can be combined capitalize on their relative strengths. We introduce a rather inelegant but an effective approach to make the best use of the Forward-Backward and Breadth-First algorithms. The idea is to use the Breadth-First technique on the top level, and use Optimized Forward-Backward technique for subformulae that have only finitely bounded temporal operators.

We present a Generic algorithm that combines the breadth first technique and forward backward technique in a single algorithm. For different sublogics of BTL, the Generic algorithm works as efficiently as the corresponding specialized algorithms for those sublogics. We also introduce a sublogic of BTL called Slightly-Restricted Temporal Logic SBTL. SBTL does not allow any unbounded temporal operators within the scope of bounded temporal operators. We believe that SBTL is strong enough to specify most of the system properties used in practice. The complexity of Generic is reduced considerably when a specification is given in the form of SBTL.
4.2 The Algorithm

This section presents the Generic algorithm, which combines the Breadth-First algorithm and the Optimized Forward-Backward algorithm to optimize the overall complexity. The algorithm translates a metric alternating automaton $A$ into an annotated metric alternating automaton $A_a$. The additional information associated with $A_a$ guides the algorithm to apply either breadth first or forward backward techniques for each metric subautomaton $A_{metric} \in A_a$.

The Generic algorithm works mainly like the Breadth-First algorithm as it computes a set of configurations at each system step. Unlike the Breadth-First algorithm, the evaluation of a configuration cannot be instantly reduced to the evaluation to its successor configurations. Generic maintains a set of sets of configurations in the form of a configuration DAG ($\mathcal{G}$), defined in Definition 4.4. $\mathcal{G}$ is expanded and evaluated backwards in the same way as done in the Optimized Forward-Backward algorithm.

**Definition 4.1. (Annotated Metric Alternating Automaton)** An annotated metric alternating automaton $A$ is defined as follows:

$$A ::= \epsilon_A \text{ empty automaton}$$
$$| N \text{ an automaton node}$$
$$| A \land A \text{ conjunction of two automata}$$
$$| A \lor A \text{ disjunction of two automata}$$
$$| \langle A^d, k \rangle \text{ metric sub-automaton},$$

where $k \in \mathbb{N} \cup \{\infty\}$.

**Definition 4.2.** The function $\text{cons-sum}$ that takes a metric alternating automaton $A$ and returns sum of the constant appearing in $A$, is defined as follows:

$$\text{cons-sum}(\epsilon_A) = 0$$
$$\text{cons-sum}(N) = 0$$
$$\text{cons-sum}(A_0^d) = d + \text{cons-sum}(A_0)$$
$$\text{cons-sum}(A_1 \lor A_2) = \text{cons-sum}(A_1) + \text{cons-sum}(A_2)$$
$$\text{cons-sum}(A_1 \land A_2) = \text{cons-sum}(A_1) + \text{cons-sum}(A_2)$$

**Definition 4.3.** The function $\text{annotate}$ that takes a metric alternating automaton and returns an annotated metric alternating automaton, is defined as follows:

$$\text{annotate}(\epsilon_A) = \emptyset$$
$$\text{annotate}(N) = N$$
$$\text{annotate}(A_0^d) = \langle \text{annotate}(A_0), d + \text{cons-sum}(A_x) \rangle$$
$$\text{annotate}(A_1 \lor A_2) = \text{annotate}(A_1) \lor \text{annotate}(A_2)$$
$$\text{annotate}(A_1 \land A_2) = \text{annotate}(A_1) \land \text{annotate}(A_2)$$
Definition 4.4. (Configuration DAG) A configuration DAG \( G \), generated by unrolling \( \mathcal{A} \), is defined as follows:

\[
G := \epsilon_G \quad \text{empty DAG}
\]

\[
\begin{align*}
&\langle S_p, \langle \mathcal{N}, c \rangle, \text{res} \rangle \quad \text{a leaf represents a node-clock pair} \\
&\langle S_p, \langle C, \text{res} \rangle \rangle \\
&\langle S_p, \langle \wedge, S \rangle, \text{res} \rangle \quad \text{a conjunctive branching point with a set of } G \\
&\langle S_p, \langle \vee, S \rangle, \text{res} \rangle \quad \text{a disjunctive branching point with a set of } G
\end{align*}
\]

where, \( \mathcal{N} \) is a node of \( \mathcal{A} \), \( S_p \) is a set of pointers to parent configuration DAGs, \( c \) is a clock, \( S \) is a set of configuration DAGs, \( C \) is a set of pairs \( \langle \mathcal{N}, c \rangle \), and \( \text{res} \in \{-1, 0, 1\} \).

Note 4.1. There are two types of leaves in the above definition of a configuration DAG. We use the term “type A” for a leaf that represents a node-clock pair, and use to the term “type B” for the leaf that represents a set of node-clock pairs.

Definition 4.5. (Translation from an AMAA to a Configuration DAG) For a metric alternating automaton \( \mathcal{A} \) and an associated clock \( c \), the translation function \( \text{translate}(\mathcal{A}, c) \) is defined as follows:

\[
\begin{align*}
\text{translate}(\epsilon, c) &= \emptyset \\
\text{translate}(\mathcal{N}, c) &= \langle \emptyset, \langle \mathcal{N}, c \rangle, 0 \rangle \\
\text{translate}(\mathcal{A}_1 \lor \mathcal{A}_2, c) &= \langle \emptyset, \langle \lor, \{\text{translate}(\mathcal{A}_1, c)\} \cup \{\text{translate}(\mathcal{A}_2, c)\}, 0 \rangle \\
\text{translate}(\mathcal{A}_1 \land \mathcal{A}_2, c) &= \langle \emptyset, \langle \land, \{\text{translate}(\mathcal{A}_1, c)\} \cup \{\text{translate}(\mathcal{A}_2, c)\}, 0 \rangle \\
\text{translate}(\langle \mathcal{A}_d^0, k \rangle, c) &= \text{translate}(\mathcal{A}_d, d)
\end{align*}
\]

Definition 4.6. Given an annotated metric alternating automaton \( \mathcal{A} \), a clock \( c \), the function \( \text{init}(\mathcal{A}, c) \) is defined as follows:

\[
\begin{align*}
\text{init}(\epsilon, c) &= \emptyset \\
\text{init}(\mathcal{N}, c) &= \{\{\mathcal{N}, c\}, \epsilon_G\} \\
\text{init}(\mathcal{A}_d^0, k, c) &= \{\emptyset, \{\text{translate}(\mathcal{A}_0, d)\}\} \\
\text{init}(\mathcal{A}_d^0, \infty, c) &= \text{init}(\mathcal{A}_0, d) \\
\text{init}(\mathcal{A}_1 \lor \mathcal{A}_2, c) &= \text{init}(\mathcal{A}_1, c) \cup \text{init}(\mathcal{A}_2, c) \\
\text{init}(\mathcal{A}_1 \land \mathcal{A}_2, c) &= \text{init}(\mathcal{A}_1, c) \otimes \text{init}(\mathcal{A}_2, c)
\end{align*}
\]

where, \( \otimes \) denotes the following:

\[
\{\langle C_1, D_1 \rangle \ldots \langle C_n, D_n \rangle\} \otimes \{\langle C'_1, D'_1 \rangle \ldots \langle C'_m, D'_m \rangle\} = \{\langle C_i \cup C'_j, D_i \cup D'_j \rangle \mid i = 1 \ldots n, j = 1 \ldots m\}
\]

Definition 4.7. (Configuration) For a metric alternating automaton \( \mathcal{A} \), a configuration is a pair \( \langle C, D \rangle \), where
• $C$ is a set of pairs $\langle \mathcal{N}, c \rangle$, where $\mathcal{N}$ is a node of $A$ and $c$ is an associated clock.

• $D$ is a set of configuration DAGs.

For an alternating automaton $A$ and an execution trace $\rho$, the algorithm generates a set $S$ of possible system configurations at every system step, using the function $\text{init}$. The initial configuration set $S_1$ is computed by calling $\text{init}$ with argument $A$. $\rho$ is accepted by $A$ if and only if there exists at least one configuration in $S_1$ that leads to an accepting configuration.

A configuration $\langle C, D \rangle \in S$ leads to an accepting configuration if and only if the following holds.

• $C$ is state satisfied.

• All $d \in D$ are evaluated to 1.

• At least one of the successor configurations of $C$ leads to an accepting state.

---

**Procedure** $\text{Generate-DAG}(S)$

**Input**: A set $S$ of configurations.

**Output**: A configuration DAG.

```
begin
  $D' \leftarrow \emptyset$
  for each $\langle C, D \rangle \in S$ do
    $X \leftarrow \langle \emptyset, C, \text{res} \rangle$
    $G_c \leftarrow \langle \emptyset, \langle \land, \{X \cup D\}, 0 \rangle \rangle$
    $D' \leftarrow D' \cup \{G_c\}$
    $G \leftarrow \langle \emptyset, \langle \lor, \{D'\}, 0 \rangle \rangle$
  return $G$
end
```

---

### 4.2.1 Translation from a Configuration to a DAG

In the *Breadth-First* algorithm, the evaluation of a configuration is instantly reduced to the evaluation of its successor configurations. A configuration $\mathcal{C}$ in the configuration set $S$, generated by $\text{Generic}$ using the function $\text{init}$, contains a set $D$ of DAGs. The evaluation of $\mathcal{C}$ can not be reduced to the evaluation of successor configurations unless each $d \in D$ is fully evaluated. Since $\text{Generic}$ generates DAGs for the subautomata representing the subformulas having only
finitely bounded intervals, all \( d \in D \) are evaluated within a finite number of system steps. Every \( CC \in S \) is stored for a finite number of system steps before its evaluation is reduced to the evaluation of successor configurations.

The algorithm stores \( S \) in the form a configuration sub-DAG (\( G \)). The translation from \( S \) to \( G \) is shown in the procedure Generate-DAG. \( G \) is a disjunctive branching point over a set \( D_{sub} \) of configuration sub-DAGs \( \langle S_p, \langle \land, \{ (G, C, 0) \} \cup D \rangle \) constructed by translating every configuration \( \langle C, D \rangle \in S \), where the set \( S_p \) contains the links to the parent configurations.

---

**Procedure** Forward\((S_t)\)

**Input**: A set \( S_t \) of leaves of a configuration DAG \( G \).

**Output**: A set of leaves of the expanded \( G + \) expansion of \( G \) as a side effect.

begin
\[
S'_t \leftarrow \emptyset \\
\text{for each } X \in S_t \text{ do}
\]
switch \( X \) do

case \( \langle S_p, C, res \rangle \)
\[
S_p \leftarrow \text{remove-child} (S_p, X) \\
S' \leftarrow \bigotimes \text{init}(\delta, c - 1) \\
\langle (F, \delta, \text{acc}, c) \rangle_{C} \\
G \leftarrow \text{Generate-DAG}(S') \\
S_p \leftarrow \text{add-child} (S_p, G) \\
S'_t \leftarrow S'_t \cup \text{Get-Leaves}(S_p, G)
\]

case \( \langle S_p, \langle (F, \delta, \text{acc}), c \rangle, 0 \rangle \)
\[
S_p \leftarrow \text{remove-child} (S_p, X) \\
G \leftarrow \text{translate}(\delta, c - 1) \\
S_p \leftarrow \text{add-child} (S_p, G) \\
S'_t \leftarrow S'_t \cup \text{Get-Leaves}(S_p, G)
\]

return \( S'_t \)

end

---

### 4.2.2 Forward Expansion

A configuration DAG \( (G) \) maintained by the Generic algorithm is expanded at each system step by the procedure Forward from the leaves. Forward takes a set \( S_t \) of leaves of \( G \) and expands \( G \) by computing a configuration sub-DAG for each leaf in \( S_t \).
The set $S_t$ contains both type A and type B leaves. The expansion is done differently for the type A and the type B leaves. For type A leaves, $G$ is expanded in the same way as discussed in Section 3.3.1. However, for type B leaves, $G$ is expanded by computing a set $S$ of configurations first, and then translating $S$ to configuration sub-DAG. Similarly for type A leaves, the algorithm merges isomorphic leaves of type B on the fly to avoid producing duplicate sub-DAGs.

### 4.2.3 Backward Evaluation

The backward evaluation of a configuration DAG $G$ starts off from the leaves. The procedure $\text{Eval-DAG}$ takes a set $S_t$ of leaves of $G$ and evaluates every element in $S_t$. The evaluation result is propagated upwards using the procedure $\text{Eval-Back}$ presented in Section 3.3.2. $S_t$ contains both type A and type B leaves. A type A leaf is evaluated in the same way as previously done by the Optimized Forward-Backward algorithm. However, a type B leaf $\langle S_p, C, res \rangle$ is evaluated to $-1$, $1$ and $0$, if one of $\langle N, c \rangle \in C$ is evaluated to $-1$, all $\langle N, c \rangle \in C$ are evaluated to $-1$ or one of $\langle N, c \rangle \in C$ is evaluated to $0$ respectively.

### 4.2.4 How it works

The procedure $\text{GENERIC}$ takes a program trace $\rho$ of length $n$ and a metric alternating automaton $A$ and checks whether $\rho$ is a model of $A$. A set $S$ of initial
4.2. THE ALGORITHM

Procedure GENERIC(\(A, \rho\))

Input: An automaton \(A\) and a program trace \(\rho\).
Output: A Boolean.

begin
\(S \leftarrow \text{init}(A, \infty)\)
\(R \leftarrow \text{Generate-DAG}(S)\)
\(S_t \leftarrow \text{Get-Leaves}(\emptyset, R)\)

for \(n = 1 \ldots |\rho| - 1\) do
\(S_d \leftarrow \text{Eval-DAG}(S_t, \text{evaluate}, \rho_n)\)
if \(R \in S_d\) and result\((R) \neq 0\) then
\(\text{return result}(R)\)

for each \(\langle S_p, X, \text{res} \rangle \in S_d\) do
\(S_p \leftarrow \text{remove-child}(S_p, \langle S_p, X, \text{res} \rangle)\)
\(S_t \leftarrow \text{Forward}(S_t)\)

end

return \(R \in \text{Eval-DAG}(S_t, \text{eval-final})\) and result\((R) = 1\)

end

system configurations is computed and then translated into a configuration DAG \(R\). \(R\) is passed to the procedure Get-Leaves to compute the set \(S_t\) of leaves. The algorithm from step 1 to step \(n - 1\) works as follows:

1. Calls the procedure Eval-DAG to evaluate \(R\) backwards from the leaves.
2. Terminates with success or failure if \(R\), in the previous step, is evaluated to 1 or \(-1\) respectively.
3. Removes all the sub-DAGs in \(R\) that are evaluated to 1 or \(-1\).
4. Calls the procedure Forward to expand \(R\) from the leaves, and computes a set of leaves of expanded \(R\).
5. Repeats step 1, 2, 3, 4 and 5 until \(\rho\) reaches its last state.

At the final position of the trace \(\rho\), the algorithm applies the accepting condition and evaluates each leaf in the configuration DAG using the function eval-final.

4.2.5 Example

Figure 4.2 shows the detection of an accepting run of an execution trace \(\rho = [[\langle a, \neg p, \neg q \rangle], [\langle \neg a, p, \neg q \rangle], [\langle a, \neg p, q \rangle]]\) in the specification automaton \(A\). \(A\), shown in Figure 4.1, is constructed from the BTL formula \(\Diamond_{[0,\infty]}(a \land (\Diamond_{[0,1]}p) U_{[0,\infty]}q))\).
As discussed in previous sections, the Generic algorithm stores system configurations in the form of a configuration DAG $G$. The algorithm tries to detect a path $G$ that starts at the root of $G$ and ends at accepting nodes.

The dotted lines represent sub-DAGs in $G$ that do not belong to $T$, the thick lines represent the sub-DAGs that belong to $T$, and normal lines represent sub-DAGs that are not fully evaluated yet. The type $A$ node of $G$ is represented as $N_x^c$, where $N_x$ is a node of $A$ and $c$ is an associated clock. The type $B$ node is represented by a set of nodes of $A$.

To check $\rho$ against $A$ for acceptence, Generic works follow:

**Step 1:**

- The algorithm generates the initial set $S$ of configurations for $A$ and then translates $S$ into $G$, as shown in Figure 4.2(1a).

- Both the type $A$ and the type $B$ nodes of $G$ are evaluated, and the procedure $\text{Eval-Back}$ propagates the evaluation result upwards in $G$, as shown in Figure 4.2(1b). The dotted lines show the propagation of the result $-1$, while a thick line shows the propagation of the result 1.

- The sub-DAGs in $G$ that are evaluated in the previous step are removed and $G$ is reduced to a compact form, as shown in Figure 3.3(2c).
4.2. THE ALGORITHM

Figure 4.2: A stepwise construction of a configuration DAG

Step 2:

- $\mathcal{G}$ is expanded from the type $A$ nodes $n_1^1$ by constructing a configuration sub-DAGs, translated from subautomaton using the function `translate`. For each type $B$ node $\{n_0\}$ and $\{n_1, n_5\}$, $\mathcal{G}$ is expanded by constructing configuration sub-DAGs that are translated from successor configurations. The expansion of $\mathcal{G}$ is shown in Figure 4.2(2a).

- The procedure `Eval-Back` propagates the evaluation result of the nodes $n_3^0, \{n_2\}, \{n_1, n_2\}$ and $\{n_1, n_5\}$ upwards in $\mathcal{G}$, as shown in Figure 4.2(2b).

- The compact $\mathcal{G}$, after reduction, is shown in Figure 4.2(2c).
CHAPTER 4. GENERIC ALGORITHM

Step 3:

- The procedure Forward expands T from the nodes \{n_0\} and \{n_5\}, as shown in Figure 4.2(3a).

- At the final step, the nodes are evaluated by applying the accepting condition. The nodes \{n_1, n_2\} and \{n_2\} are state-satisfied and accepting, while all other nodes are either not state-satisfied or rejecting. The propagation of the evaluation result is shown in Figure 4.2(3b).

- \(\rho\) is accepted, as there exist two Ts, starting from the root of \(G\) and ending at the accepting nodes \{\{n_1, n_2\}\} and \{\{n_2\}\}, as shown in Figure 4.2(3b).

Theorem 4.1. Given a program trace \(\rho\) and an alternating automaton \(A\), constructed from BTL formula \(\varphi\), GENERIC runs in time \(O(|\rho| \times ((M_c + 2)^{2(X-Y)} + (M_c + 1) \times Y^2))\) and space \(O(X \times (M_c + 1) \times (M_c + 2)^{2(X-Y)} + (M_c + 1) \times Y^2))\), where \(M_c\) is the largest constant appearing in \(\varphi\) (excluding \(\infty\)), \(X = |A|\) and \(Y\) is the sum of the sizes of subautomata representing FBTL subformulae in \(\varphi\).

The correctness of the Generic algorithms follows from the correctness of the Optimized Forward-Backward and the correctness of the Breadth-First algorithm.

Theorem 4.2. Given a program trace \(\rho\) and an automaton \(A\), GENERIC \((A, \rho) = \text{true}\), if there exists an accepting run of \(\rho\) in \(A\).

Theorem 4.3. Given a program trace \(\rho\) and an alternating automaton \(A\), constructed from BTL formula \(\varphi\), GENERIC runs in time \(O(|\rho| \times ((M_c + 2)^{2(X-Y)} + (M_c + 1) \times Y^2))\) and space \(O(X \times (M_c + 1) \times (M_c + 2)^{2(X-Y)} + (M_c + 1) \times Y^2))\), where \(M_c\) is the largest constant appearing in \(\varphi\) (excluding \(\infty\)), \(X = |A|\) and \(Y\) is the sum of the sizes of subautomata representing FBTL subformulae in \(\varphi\).

Proof. Like the Optimized Forward-Backward algorithm, presented in Chapter 3, GENERIC also maintains a configuration DAG \(G\) during its execution. The space complexity of GENERIC is linear in the size of \(G\). At each system step, \(G\) is expanded by the procedure Forward form the leaves. To find out the overall space complexity of the algorithm, we need to find out the increment in \(|G|\) at a given system step.

The procedure Forward takes a set \(S_t\) of leaves of \(G\) and expands \(G\) from each \((S_p, \mathcal{X}, res) \in S_t\). As discussed in Section 4.4.2, \(S_t\) contains both type A and type B leaves. Since all the isomorphic leaves in \(G\) are merged on the fly, the number of the type A nodes is bounded by \(Y \times (M_c + 1)\) (Lemma 3.4), while the number of type B leaves is bounded by \((M_c + 2)^{2(X-Y)}\) (Theorem 3.7).
4.3 Slightly-Restricted Bounded Temporal Logic

This section presents a sublogic of BTL, called Slightly-Restricted Bounded Temporal Logic (SBTL). SBTL does not allow temporal operator to be parameterized infinite intervals to be nested in a subformula guarded by a temporal operators to be parameterized with a bounded interval. The Generic algorithm make use of this restriction to optimize the overall space complexity.

SBTL is strong enough language to specify most of the properties of reactive systems in practice. For example, the property that “always every p-state is followed by a q-states for 5 time units” can be expressed in SBTL as follows:

\[ \square_{[0, \infty)} (p \rightarrow \square_{[0, 5]} q) \]
Similarly, the property that “always every p-state is followed by a q-state within 5 time units” can be expressed in SBTL as follows:

$$\square_{[0,\infty)}(p \rightarrow \Diamond_{[0,4]} q)$$

**Definition 4.4. Slightly-Restricted Bounded Temporal Logic** Let $$\psi$$ be a FBTL formula, a SBTL formula $$\varphi$$ can be defined inductively as follows:

$$\varphi ::= \psi | \varphi \land \varphi | \varphi \land \varphi | \square_{[0,\infty)} \varphi | \Diamond_{[0,\infty)} \varphi | \varphi U_{[0,\infty)} \varphi$$

**Lemma 4.4.** For a BTL formula $$\varphi$$ that does not contain any subformula that corresponds to FBTL, GENERIC runs in time $$\mathcal{O}(X \ast (M_c + 2)X \ast |\rho|)$$ and space $$\mathcal{O}(X \ast (M_c + 2)X)$$, where $$M_c$$ is the largest constant appearing in $$\varphi$$ and $$X = |\varphi|$$. 

*Proof.* For a metric alternating automaton $$A$$, translated from $$\varphi$$, the function $$\text{init}$$ does not generate configuration DAGs. Each configuration $$\langle C', \emptyset \rangle$$ in the configuration set $$S$$, generated by $$\text{init}$$, is translated into a configuration DAG $$G' = \langle S_p, \land, \{G', C, 0\} \rangle, 0 \rangle$$ by the procedure Generate-DAG. In backward evaluation, $$G'$$ is instantly reduced to a single leaf node $$\langle S_p, C, 0 \rangle$$ by the procedure Eval-back. In the next system step, $$\langle S_p, C, 0 \rangle$$ is replaced by the configuration sub-DAG translated from the set of successor configurations of $$C$$.

Thus, $$G$$ is a disjunctive branching point over a set of type B leaves. As we know from Theorem 3.7, the number of type B leaves is always bounded by $$(M_c + 2)^X$$ and the size of the type B leaves is bounded by $$X$$. Hence, the space complexity of GENERIC, for $$\varphi$$, is $$\mathcal{O}(X \ast (M_c + 2)^X)$$.

The running time of GENERIC is linear in the increment in $$|G|$$ at every system step times the number of system steps. Hence, GENERIC runs in the time $$\mathcal{O}(X \ast (M_c + 2)^X) \Box$$

**Lemma 4.5.** For a metric alternating automaton $$A$$, translated from FBTL formula $$\varphi$$, GENERIC runs in time $$\mathcal{O}((M_c + 1)^2 \ast X^3)$$ and space $$\mathcal{O}((M_c + 1)^2 \ast X^3)$$, where $$M_c$$ is the largest constant appearing in $$\varphi$$ and $$X = |A|$$. 

*Proof.* For $$\varphi$$, the function $$\text{init}$$ generates a single configuration $$\langle \emptyset, \{G\} \rangle$$, where $$G$$ is a configuration DAG translated from $$A$$.

The space complexity of GENERIC, as proved in Theorem 4.3, is $$\mathcal{O}(X \ast (M_c + 1)(M_c + 1)^2(X + Y) + (M_c + 1) \ast Y^2))$$, where $$X$$ is the size of $$A$$ and $$Y$$ is the sum
of sizes of subautomata representing FBTL subformulas. We have $X = Y$, as $\varphi$ is a FBTL formula. Substituting $X$ for $Y$ in the above expression, the space complexity of $\text{GENERIC}$, for $\varphi$ reduces to $O((M_c + 1)^2 \times X^3)$.

Similarly, the time complexity of $\text{GENERIC}$ for BTL formula is $O(X \times (M_c + 1)((M_c + 2)^{2(X-Y)} + (M_c + 1) \times Y^2))$. By substituting $X$ for $Y$, the time complexity of $\text{GENERIC}$ gets reduced to $O((M_c + 1)^2 \times X^3)$.

Lemmas 4.4 and 4.5 prove that the Generic algorithm is as efficient as Breadth-First for BTL specification and as efficient as Optimized Forward-Backward for FBTL specification respectively.

**Theorem 4.6.** For a metric alternating automaton $A$, translated from SBTL formula $\varphi$, $\text{GENERIC}$ runs in time $O(|\varphi| \times (M_c + 1) \times (2^{2(X-Y)} + (M_c + 1) \times Y^2))$ and space $O(|\varphi| \times (2^{2(X-Y)} + (M_c + 1) \times Y^2))$, where $M_c$ is the largest constant appearing in $X = |A|$ and $Y$ is the sum of sizes of subautomata that corresponds to FBTL subformulae in $\varphi$.

**Proof.** $\text{GENERIC}$ maintains system configurations in the form of a configuration DAG $(G)$. Theorem 3.3 proves that the space complexity of $\text{GENERIC}$ is linear in the increment in $|G|$ at every system step times $(M_c + 1) \times X$, and the running time is bounded by the increment in $|G|$ at each system step times $|\varphi|$.

In $\varphi$, we have FBTL subformulae of total size $Y$ inside a top level LTL formula of size $X - Y$. For a configuration DAG $G$ generated by unrolling $A$, the number of type $B$ leaves are bounded by $2^{X-Y}$ (corollary 3.2) and the number of type $A$ leaves is bounded by $Y \times (M_c + 1)$ (lemma 3.4).

Each of type $A$ leaf adds a sub-DAG of size at most $Y$, by unrolling the specification subautomaton. Similarly, each of type $B$ leaf adds a sub-DAG of the size at most $(M_c + 2)^{X-Y}$ by translating a set of successor configurations to a configuration sub-DAG. Thus, $Y \times (M_c + 1)$ leaves of type $A$ increment the size of $G$ by at most $Y^2 \times (M_c + 1)$ and $2^{X-Y}$ leaves of type $B$ increment the size of $G$ by at most $2^{2(X-Y)}$. The total increment in the size of $G$ is therefore bounded by $2^{2(X-Y)} + (M_c + 1) \times Y^2$.

Hence, for $\varphi$, $\text{GENERIC}$ runs in space $O(X \times (M_c + 1) \times (2^{2(X-Y)} + (M_c + 1) \times Y^2))$ and in time $O(|\varphi| \times (2^{2(X-Y)} + (M_c + 1) \times Y^2))$. 

**Theorem 4.7.** For an execution trace $\rho$, and a metric alternating automaton $A$, translated from BTL formula $\varphi$, the space complexity of $\text{GENERIC}$ for different sub-logics of $BTL$ is given below:
\(O(2^X)\) if \(\varphi\) is a LTL formula
\(O(X^3 \cdot (M_c + 1)^2)\) if \(\varphi\) is a FBTL formula
\(O(X \cdot (M_c + 2)(2^{(X-Y)} + (M_c + 1) \cdot Y^2))\) if \(\varphi\) is a SRBTL formula
\(O((M_c + 1)(2^{(X-Y)} + (M_c + 1) \cdot Y^2))\) if \(\varphi\) is a BTL formula

where,

- \(X = |A|\).
- \(Y\) is the sum of the sizes of subautomata representing FBTL formulae.
- \(M_c\) is the maximum constant appearing in \(\varphi\) (excluding \(\infty\)).

**Theorem 4.8.** For an execution trace \(\rho\), and a metric alternating automaton \(A\), translated from input formula \(\varphi\), the time complexity of GENERIC for different sub-logics of BTL is given below:

\(O(2^X \cdot |\rho|)\) if \(\varphi\) is an LTL formula
\(O(X^3 \cdot (M_c + 1)^2)\) if \(\varphi\) is a FBTL formula
\(O(|\rho| \cdot (2^{(X-Y)} + (M_c + 1) \cdot Y^2))\) if \(\varphi\) is a SRBTL formula
\(O(|\rho| \cdot ((M_c + 2)^2^{(X-Y)} + (M_c + 1) \cdot Y^2))\) if \(\varphi\) is a BTL formula

where,

- \(X = |A|\).
- \(Y\) is the sum of the sizes of subautomata representing FBTL formula.
- \(M_c\) is the maximum constant appearing in \(\varphi\) (excluding \(\infty\)).
In this thesis, we presented a framework to monitor time-bounded temporal properties of a running system. The specification language BTL not only allows us to express time-bounded temporal properties in a compact form, but also leads to efficient algorithms. *Metric alternating automata* (MAA), with time constrains on transitions, provide a linear translation mechanism from BTL specifications to MAA. The algorithm based on *alternating automata* (AA) can be easily extended to work on MAA.

A collection of specialized algorithms for different sublogics of BTL is presented with their respective complexity analysis. The *Optimized Forward-Backward* algorithm has a better running time and space complexity for FBTL specifications, where all the temporal operators have finite bounds. On the other hand, the *Optimized Breadth-First* algorithm performs much better when all the temporal operators have infinite bounds, i.e., LTL specifications. Normally, BTL specifications contain a mixture of both finitely and infinitely bounded temporal operators. The *Generic* algorithm dynamically applies specialized techniques for different sublogics of BTL. It not only handles all the sublogics of BTL (including LTL), but also performs as efficiently as the specialized algorithms for those sublogics.

Typically, in BTL specifications, the infinitely bounded temporal operators appear on top of the finitely bounded temporal operators. We have formally classified such properties as *Slightly-restricted Bounded Temporal Logic* (SBTL), presented in Chapter 4. The complexity of the *Generic* algorithm is reduced considerably for the SBTL specification.
CHAPTER 5. CONCLUSION
Bibliography


Appendix A

OPrA - A Runtime Monitoring Software

In this chapter we present a software tool, called OPrA (Online Program Analyzer), for online monitoring of a running program. OPrA implements the framework discussed in this thesis. The first version of OPrA only the Forward-Backward algorithm. All the other algorithms will appear in the coming version.

A.1 General Description

The Software tool OPrA (Online Program Analyzer) is an online monitoring tool, that monitors a running program against a high-level specification written in Bounded Temporal Logic (BTL). A given program $P$ is instrumented with additional instructions to emit relevant events. $P$, when runs, emits events which are then checked against the specification by a monitor running in parallel.

The Software is divided into four modules, which are discussed briefly in the following subsections:

A.1.1 Formula Translation

The Formula Translator reads BTL specification script $\Upsilon$ from a text file and translates $\Upsilon$ to a metric alternating automaton $A$. The translation from $\varphi$ to $A$ is done by the Formula Translator as follows:

- Parses $\varphi$ to check against the syntax and semantics of the BTL.
- If $\Upsilon$ is syntactically a and semantically correct, Formula Translator produces a syntax tree $T$ from $\Upsilon$. 

59
• \( T \) is then transformed into the *negation normal form*, such that negation are pushed to the proposition level.

• \( T \), in the negation normal form, is then translated into a *metric alternating automaton* \( A \) according to the translation rules defined in Chapter 2.

• Return the automaton \( A \) as output.

### A.1.2 Program Instrumentation

The *instrumentation module* takes a specification script \( \Upsilon \) and program segment \( \mathcal{P} \), written in \( C^0 \) (a subset of \( C \) language), and produces an instrumented code \( \mathcal{P}' \) that can compiled with any standard \( C^0 \) compiler. \( \mathcal{P}' \) contains additional instructions to emit events, and also the specification script \( \Upsilon' \) inserted, as comments, at the top of the program file. \( \Upsilon' \) is produced after replacing each proposition in \( \Upsilon \) with their respective index numbers. The boolean constants *true* and *false* are replaced with 1 and 0 respectively. Given \( \mathcal{P} \) and \( \Upsilon \), the program instrumentation works as follows:

• Parses \( \varphi \) script to extract the set of predicates \( S_p \) that appears in \( \varphi \). Predicates or boolean functions over program variables. Each Predicate \( \text{pred} \in S_p \) is assigned a unique integer value \( \text{key} \) and then put into a list \( L_p \) of pair \( \langle \text{pred}, \text{key} \rangle \). Keys are assigned in increasing order staring with 2. 0 and 1 are reserved for the boolean constant *false* and *true* respectively.

• Each \( \text{pred} \in S_p \) in the specification script \( \Upsilon \) is replaced by their respective \( \text{key} \), and the resultant specification script \( \Upsilon' \) is written at the top of instrumented program \( \mathcal{P}' \) enclosed in the comments.

• Traverses \( L_p \) to compute a list \( L_{vp} \) of pair \( \langle \text{var}, \text{plist} \rangle \), where \( \text{var} \) belongs to a set of program variables \( V \) that appears in \( \Upsilon \). The list \( L_{vp} \) maps each \( \text{var} \in V \) to a set \( S' \subseteq S_p \), such that, \( \text{var} \) appears in every element of \( S' \).

• Parses \( \mathcal{P} \) to compute a set of program instruction \( \mathcal{P}_i \) that updates the value of any \( E \in V \). This is done my looking at each assignment instruction in the program text. The set \( \mathcal{P}_i \) consists of all the assignment instructions that contains a program variable \( E \in V \), such that \( E \) appears on the left side of the assignment operator.

• For each assignment instruction \( I \in \mathcal{P}_i \) that updates variable a \( \text{var} \in V \), computes the set \( S' \) of predicates, such that each \( P \in S' \) contains \( \text{var} \).

• Insert new instruction after each assignment instruction \( I \in \mathcal{P}_i \) to emit the event.
A.1.3 Event Recognizer

The Event Recognizer recognizes the events emitted by the running program. A change in the value of program variable \( v \) results in changing truth value of the set \( S \) of predicates appearing in the input formula. The module maintains a list \( L \) of pairs \( \langle id, val \rangle \), where \( val \) is a predicate’s truth value and \( id \) uniquely identifies each predicate. The list \( L \) is computed by parsing the specification script \( \Upsilon \) of a BTL formula. Event Dispatcher receives a pair \( \langle id, val \rangle \), from the running program and updates \( L \). After updating \( L \), Event Recognizer notifies monitoring module about the change in system state.

A.1.4 Runtime Monitoring

Runtime Monitor monitors the instrumented program \( \mathcal{P} \) against the formal specification \( \Upsilon \). The stepwise activities of Runtime Monitor are given below:

1. Parses the specification script \( \Upsilon \), and also initializes the list \( L \) of pair \( \langle id, val \rangle \), where \( id \) is an index number of a proposition, and \( val \) represents its truth-value.

2. Invoke Formula translator to translate the specification \( \Upsilon \) to a metric alternating automaton.

3. Initializes the Event Recognizer.

4. Waits for the notification from the Event Recognizer about system’s state change.

5. When receives the notification of state change from the Event Recognizer and verifies the new system state against the specification.

6. Repeat the last two steps unless program terminates without the following results:
   - Program trace satisfies the specification.
   - Failure is detected during execution.

A.2 Specification Script

In this section we define the grammar for the specification script \( \Upsilon \), which we use to write Bounded Temporal Logic BTL formula. The predicates over program variables are enclosed in curly braces. The grammar rules are defined below:
APPENDIX A. OPRA - A RUNTIME MONITORING SOFTWARE

formula ::= (formula )
  | binary-formula
  | unary-formula
  | proposition

unary-formula ::= temporal-operator formula
  | simple-operator formula

binary-formula ::= formula temporal-operator formula
  | formula simple-operator formula

temporal-operator ::= toperator interval
  | formula simple-operator formula

proposition ::= {predicate}
  | constant

interval ::= \epsilon
  | [number,number]
  | [number, -]

toperator ::= U (the temporal operator for strong until)
  | W (the temporal operator for weak until)
  | R (the temporal operator for the dual of until)
  | G (the temporal operator for always)
  | F (the temporal operator for Eventually)
  | X (the temporal operator for next)

simple-operator ::= and (the boolean operator for conjunction)
  | or (the boolean operator for disjunction)
  | xor (the boolean operator for exclusive conjunction)
  | not (the boolean operator for negation)
  | \rightarrow (the boolean operator for implication)
  | \leftrightarrow (the boolean operator for equivalence)

predicate ::= "C language boolean expression"

number ::= "An integer"

constant ::= true|false

The precedence of temporal and non-temporal operators is given below:
Figure A.1: The architecture of the software tool OPrA (Online Program Analyzer).

Precedence Operators
0    !
1    G, F, X
2    and
3    or
4    xor
5    ↔
6    →
7    U, W, R

The operators appear at the same line have equal precedence.
A.3 Software Architecture

The Figure A.1 shows the architecture of the software tool OPrA. The software takes the specification script $\Upsilon$ and a program $P$ as inputs. $\Upsilon$ contains the script for the BTL specification $\varphi$, stating which system's property is to be monitored. $P$ is instrumented in accordance with $\Upsilon$, to produce the instrumented program $P'$. At the same time $\varphi$ is translated to a metric alternating automaton $A$. $P'$ is compiled with $C0$ compiler, and executed. During the execution $P'$ emits events, which are received by the Runtime Monitor. Runtime Monitor checks whether a sequence of events emitted by $P'$ are accepted by $A$ or not.