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 herby 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

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:

$$\Box_{[0,\infty]}(p \rightarrow \Diamond_{[0,n-1]} q).$$

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

$$\Box(p \rightarrow (q \vee \bigcirc(q \vee \bigcirc(...(q \vee \bigcirc q)...)))).$$

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 ρ is accepted by the specification automaton $\mathcal{A}$. Three algorithms, the *Forward-Backward* algorithm, the *Optimized Breadth-First* algorithm and the *Generic* algorithm are presented to check ρ against $\mathcal{A}$. The *Forward-Backward* unrolls $\mathcal{A}$ into a DAG (directed acyclic graph) $\mathcal{G}$ and then traverses $\mathcal{G}$ backwards to detect an accepting run ρ in $\mathcal{A}$. The space complexity of the algorithm is quadratic in the size of $\mathcal{A}$ and linear is the size of ρ. The *Optimized Breadth-First* algorithm, which extends the *Breadth-First* algorithm presented in [6], has space complexity constant in ρ and exponential in $\mathcal{A}$. 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 φ and an execution trace ρ, the *Generic* algorithm, in the worst case, runs in space $\mathcal{O}(C^{|\varphi|})$ and time $\mathcal{O}(C^{|\varphi|} * |\rho|)$, where C is the maximum integer constant appearing in φ. For the same specification when translated into classical LTL, the best known algorithms run in space $\mathcal{O}(2^{C*|\varphi|})$ and time $\mathcal{O}(2^{C*|\varphi|} * |\rho|)$.

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 $\mathcal{O}(|\varphi| * C^2 * 2^{2(|\varphi|)})$.

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 φ into an *automaton* $\mathcal{A}$, and then perform different operations on $\mathcal{A}$. 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 φ is translated into an *alternating automaton* $\mathcal{A}$ and then a given execution trace ρ is checked against $\mathcal{A}$ 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* $\mathcal{A}$ is defined as follows:

$$\begin{array}{rll} \mathcal{A} ::= & \epsilon_{\mathcal{A}} & \textit{empty automaton} \\ | & \mathcal{N} & \textit{automaton node} \\ | & \mathcal{A} \wedge \mathcal{A} & \textit{conjunction of two automata} \\ | & \mathcal{A} \vee \mathcal{A} & \textit{disjunction of two automata} \\ | & \mathcal{A}^d & \textit{metric sub-automaton} \end{array}$$

where $\mathcal{N}$ 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 $\mathcal{N}$ of a *metric alternating automaton* is defined as follows:

$$\begin{array}{rll} \mathcal{N} ::= & \langle \nu, acc \rangle & \textit{leaf node} \\ | & \langle \mathcal{F}, \delta, acc \rangle & \textit{timer node} \end{array}$$

where,

- $\nu \in Q$ is an input letter.
- δ is a sub-automaton expressing the next-state relation.
- $\mathcal{F}$ 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 ν and a timer node represents a boolean function $\mathcal{F}$ over a set of clocks.

Example 2.1. Figure 2.1 illustrates the construction of a *metric alternating automaton* $\mathcal{A}$ that specifies the *timed-language* $\mathcal{L}$ given below:

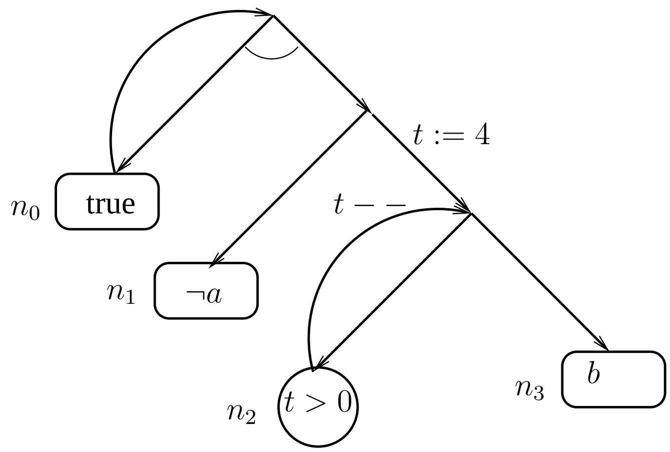

Figure 2.1: Metric alternating automaton for the timed language $\mathcal{L} = \{\sigma | \forall i.((\sigma_i \models a) \rightarrow \exists j > i.(\sigma_j \models b)$ and $j - i \leq 4)\}$

.

$\mathcal{L} = \{\sigma | \forall i.((\sigma_i \models a) \rightarrow \exists j > i.(\sigma_j \models b)$ and $j - i \leq 4)\}$,
where σ_i and σ_j represent the i^{th} and the j^{th} input letter in σ respectively, and i, j are positive integers.

The language $\mathcal{L}$ 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 $\mathcal{L}$ if it is accepted by $\mathcal{A}$. Acceptance conditions of $\mathcal{A}$ are discussed in the sections to follow.

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

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:

$$
\begin{array}{lll}
T ::= & \epsilon_T & \textit{empty tree} \\
\mid & \langle \mathcal{N}, T \rangle & \textit{a node with a subtree} \\
\mid & T\,.T & \textit{composition of two subtrees}
\end{array}
$$

where $\mathcal{N}$ represents a node of a a *metric alternating automaton*.

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

$\mathcal{A} = \epsilon_{\mathcal{A}}$ *and* $T = \epsilon_T$

$\mathcal{A} = \langle \nu, acc \rangle$ *and* $T = \langle \langle \nu, acc \rangle, \epsilon_T \rangle$, *and* $\rho_0 \models \nu$

$\mathcal{A} = \langle \mathcal{F}, \delta, acc \rangle$ *and* $\rho \neq \epsilon$, $T = \langle \langle \mathcal{F}, \delta, acc \rangle, T' \rangle$, $\mathcal{F}(c) =$ **true** and T'*, with a clock* $c-1$*, is a run of* $\rho_1, \rho_2, \rho_3 ...$ *in* δ.

$\mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2$ *and* $T = T_1.T_2$, T_1*, with a clock* c*, is a run of* ρ *in* $\mathcal{A}_1$*, and* T_2*, with a clock* c*, is a run of* ρ *in* $\mathcal{A}_2$.

$\mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2$ *and* T *,with a clock* c*, is a run of* ρ *in* $\mathcal{A}_1$ *or* T *, with a clock* c*, is a run of* ρ *in* $\mathcal{A}_2$.

$\mathcal{A} = \mathcal{A}_0^d$ *and* T*, with a clock* d*, is a run of* ρ *in* $\mathcal{A}_0$.

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

Definition 2.7. (Model) An *execution trace* ρ is a model of a *metric alternating automaton* $\mathcal{A}$, denoted as $\rho \models \mathcal{A}$ if there exits an accepting run of ρ in $\mathcal{A}$.

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

Example 2.2. Figure 2.2 shows two run $\mathcal{T}_1$ and $\mathcal{T}_2$ of the *metric alternating automaton* $\mathcal{A}$ (shown in figure 2.1) against two different input sequences σ_1 and σ_2 given below:

$\sigma_1 = \langle \neg a, b \rangle \rightarrow \langle a, \neg b \rangle \rightarrow \langle \neg a, \neg b \rangle \rightarrow \langle \neg a, \neg b \rangle \rightarrow \langle a, \neg b \rangle \rightarrow \langle a, b \rangle$
$\sigma_2 = \langle a, b \rangle \rightarrow \langle a, \neg b \rangle \rightarrow \langle \neg a, \neg b \rangle \rightarrow \langle \neg a, \neg b \rangle \rightarrow \langle a, \neg b \rangle \rightarrow \langle \neg a, \neg b \rangle$.

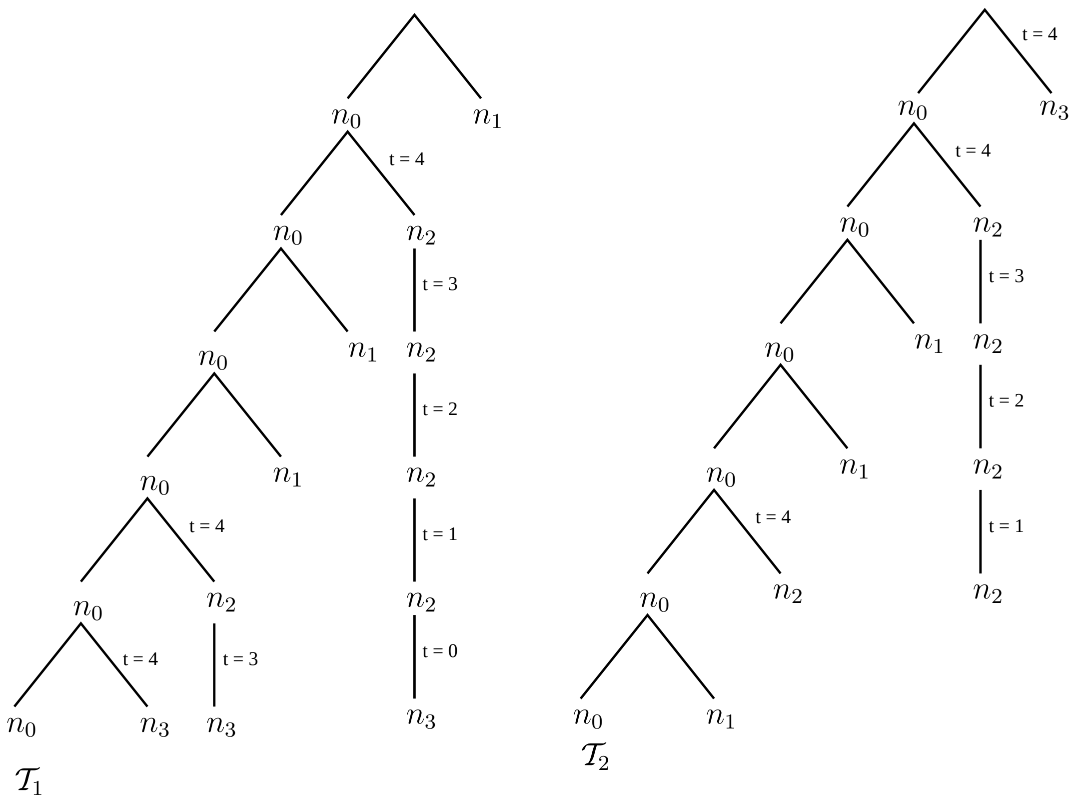

Figure 2.2: Runs $\mathcal{T}_1$ and $\mathcal{T}_2$ of input sequences σ_1 and σ_2 in the specification automaton $\mathcal{A}$, shown in Figure 2.1.

According to Definition 2.7, $\sigma_1 \models \mathcal{A}$ if there is an accepting run of σ_1 in $\mathcal{A}$. The run tree $\mathcal{T}_1$ is accepting, by Definition 2.6, as all the branches end at accepting nodes (n_0, n_1, n_3). Thus, σ_1 is a model of $\mathcal{A}$. On the other hand, σ_2 produces $\mathcal{T}_2$ 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 exists any accepting run for the input sequence σ_2, the sequence is rejected by the specification automaton $\mathcal{A}$ and σ_2 is not a model of $\mathcal{A}$.

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:

$$\begin{aligned} \varphi := \; & p \\ | \; & \varphi \vee \varphi \\ | \; & \varphi \wedge \varphi \\ | \; & \Box_{[x_1,x_2]}\varphi \\ | \; & \Diamond_{[x_1,x_2]}\varphi \\ | \; & \varphi \mathcal{U}_{[x_1,x_2]}\varphi \end{aligned}$$

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* ρ, a state formula P, BTL formulae φ and ψ, a BTL formula holds at position $0 \leq$ j $< |\rho|$, written as (ρ, j) $\models \varphi$, is formally described as follows:

For a state formula:

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

For the boolean connectives:

$$\begin{array}{lll} (\rho, j) \models \varphi \wedge \psi & \text{iff} & (\rho, j) \models \varphi \text{ and } (\rho, j) \models \psi \\ (\rho, j) \models \varphi \vee \psi & \text{iff} & (\rho, j) \models \varphi \text{ or } (\rho, j) \models \psi. \\ (\rho, j) \models \varphi & \text{iff} & (\rho, j) \not\models \varphi \end{array}$$

For temporal operators

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

where $x_2 + j < |\rho|$.

$$
\begin{array}{lll}
(\rho, j) \models \square_{[x_1,x_2]}\varphi & \text{iff} & (\rho, i) \models \varphi \text{ for all } i \in [x_1 + j, |\rho| - 1] \\
(\rho, j) \models \diamondsuit_{[x_1,x_2]}\varphi & \text{iff} & (\rho, i) \models \varphi \text{ for some } i \in [x_1 + j, |\rho| - 1] \\
(\rho, j) \models \varphi\mathcal{U}_{[x_1,x_2]}\psi & \text{iff} & (\rho, i) \models \psi \text{ for some } i \in [x_1 + j, |\rho| - 1] \text{ and} \\
 & & (\ x_1 + j = i \text{ or } (\rho, k) \models \varphi \text{ for all } k \in [x_1 + j, i - 1]\ )\ ,
\end{array}
$$

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

$$
\begin{array}{ll}
(\rho, j) \models \square_{[x_1,x_2]}\varphi & \text{always true} \\
(\rho, j) \models \diamondsuit_{[x_1,x_2]}\varphi & \text{always false} \\
(\rho, j) \models \varphi\mathcal{U}_{[x_1,x_2]}\psi & \text{always false,}
\end{array}
$$

where, $x_1 + j \geq |\rho|$.

Given BTL formula φ and Ψ, weak until $\mathcal{W}$ and dual until $\mathcal{R}$ operators can be expressed as follows:

- $(\rho, j) \models \varphi\mathcal{W}_{[x_1,x_2]}\psi \equiv (\rho, j) \models \varphi\mathcal{U}_{[x_1,x_2]}\psi$ or $(\rho, j) \models \diamondsuit_{[x_1,x_2]}\varphi$
- $(\rho, j) \models \varphi\mathcal{R}_{[x_1,x_2]}\psi \equiv (\rho, j) \models \neg(\neg\varphi\mathcal{U}_{[x_1,x_2]}\neg\psi)$

Note 2.1. BTL does not support the the next operator, however $\diamondsuit_{[1,1]}\varphi$ can be used to specify that φ 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 Ψ_0, Ψ_1, Ψ_2, Ψ_3 and Ψ_4 which we later use in the construction of the MAA. Given MAA $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2$, the translation functions Ψ_0, Ψ_1, Ψ_2, Ψ_3 and Ψ_4 are be defined as follows:

$$
\begin{array}{lll}
\Psi_0(\mathcal{A}) & = & (\langle \lambda x.\ x > 0, \Psi_0(\mathcal{A}), 0\rangle) \vee ((\langle \lambda x.\ x = 0, \epsilon, 1\rangle) \wedge (\mathcal{A})) \\
\Psi_1(\mathcal{A}) & = & (\langle \lambda x.\ x > 0, \Psi_1(\mathcal{A}), 1\rangle) \vee ((\langle \lambda x.\ x = 0, \epsilon, 1\rangle) \wedge (\mathcal{A})) \\
\Psi_2(\mathcal{A}) & = & (\langle \lambda x.\ x > 0, \Psi_1(\mathcal{A}), 1\rangle \vee \langle \lambda x.\ x = 0, \epsilon, 1\rangle) \wedge \mathcal{A} \\
\Psi_3(\mathcal{A}_1, \mathcal{A}_2) & = & (\langle \lambda x.\ x > 0, \Psi_2(\mathcal{A}_1, \mathcal{A}_2), 0\rangle \wedge \mathcal{A}_1) \vee \mathcal{A}_2 \\
\Psi_4(\mathcal{A}) & = & (\langle \lambda x.\ x > 0, \Psi_3(\mathcal{A}), 0\rangle) \vee \mathcal{A}
\end{array}
$$

The translation rules from BTL specification φ to *metric alternating automaton* $\mathcal{A}(\varphi)$ are given below:

For a state formula p:

$$\mathcal{A}(p) \quad = \quad \langle p, 1 \rangle$$

For BTL formulae φ and ψ:

$$\begin{array}{lcl}
\mathcal{A}(\varphi \wedge \psi) & = & \mathcal{A}(\varphi) \wedge \mathcal{A}(\psi) \\
\mathcal{A}(\varphi \vee \psi) & = & \mathcal{A}(\varphi) \vee \mathcal{A}(\psi) \\
\mathcal{A}(\Box_{[x_1,x_2]}\varphi) & = & (\Psi_1((\Psi_2(\mathcal{A}(\varphi)))^{x_2-x_1}))^{x_1-1} \\
\mathcal{A}(\Diamond_{[x_1,x_2]}\varphi) & = & (\Psi_0((\Psi_4(\mathcal{A}(\varphi)))^{x_2-x_1}))^{x_1-1} \\
\mathcal{A}(\varphi\mathcal{U}_{[x_1,x_2]}\psi) & = & (\Psi_0((\Psi_3(\mathcal{A}(\varphi),\mathcal{A}(\psi)))^{x_2-x_1}))^{x_1-1}
\end{array}$$

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 φ and ψ, Figure 2.3 shows the construction of metric alternating automata $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ and $\mathcal{A}_4$ from the respective BTL formulae $\Diamond_{[x_1,x_2]}\varphi$, $\Box_{[x_1,x_2]}\varphi$ and $\varphi\mathcal{U}_{[x_1,x_2]}\psi$.

Proposition 2.1. For every BTL formula φ, there exists a *metric alternating automaton* $\mathcal{A}$, and the size of $\mathcal{A}$ is linear in proportion to the size of φ.

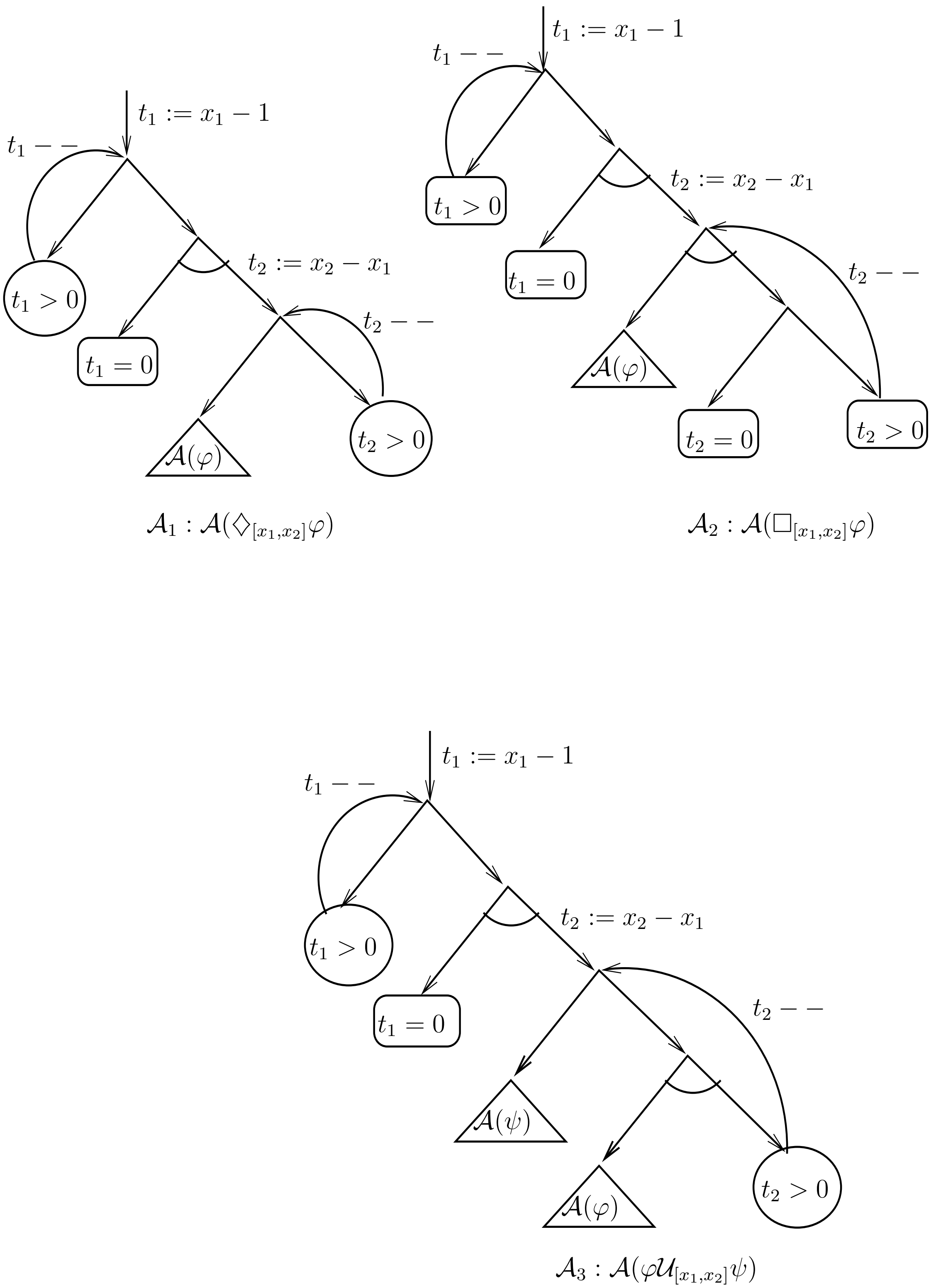

Figure 2.3: Construction of *metric alternating automata (MAA)* from BTL formulae.

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 φ into a *metric alternating automaton* $\mathcal{A}$, and then check whether a given execution trace ρ is accepted by $\mathcal{A}$ or not. ρ is accepted by $\mathcal{A}$, according to Definition 2.6, if there exists a *run* $\mathcal{T}$ of ρ in $\mathcal{A}$, such that every path through $\mathcal{T}$ ends at an accepting node.

Three algorithms are presented to check whether a run $\mathcal{T}$ of ρ exists in $\mathcal{A}$ or not. The first algorithm, called *Forward-Backward* algorithm, unrolls $\mathcal{A}$ into a *configuration tree* T and then traverses T backwards, starting from leaves, to detect $\mathcal{T}$ as a subtree in T. The algorithm follows a very simple technique, but it generates a T that grows exponentially in the size of ρ. The second algorithm, called *Optimized Forward-Backward* algorithm, unrolls the automaton into a more compact datastructure, called a *configuration DAG* $\mathcal{G}$, and then traverses $\mathcal{G}$ backwards to detect $\mathcal{T}$ in $\mathcal{G}$. The space complexity of the second algorithm is linear in the size of $|\rho|$ and quadratic in the size of $\mathcal{A}$. 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* $\mathcal{T}$ of an *execution trace* ρ in the specification automaton $\mathcal{A}$. The algorithm first translates $\mathcal{A}$ into a *configuration tree* T, defined in Definition 3.1, and then evaluates T backwards to detect $\mathcal{T}$ 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 $\mathcal{A}$ 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* $\mathcal{A}$, is defined recursively as follows:

$T ::=$	ϵ_T	*empty tree*
$\mid$	$\langle p, \langle \mathcal{N}, T\rangle, res\rangle$	*node with a sub-tree*
$\mid$	$\langle p, \langle T \wedge T\rangle, res\rangle$	*a conjunctive branching point with two sub-trees*
$\mid$	$\langle p, \langle T \vee T\rangle, res\rangle$	*a disjunctive branching point with two sub-trees*
$\mid$	$\langle p, \langle d, T\rangle, res\rangle$	*a metric sub-tree*

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

Procedure `Forward`(S_t)

Input : A set S_t of node-clock pairs of a configuration tree T.
Output: A set of node-clock pairs + expansion of T as side effect.
begin
  $S'_t \leftarrow \emptyset$
  for *each* $\langle \mathcal{X}, c\rangle \in S_t$ **do**
    if $\mathcal{X} = \langle pp, \langle\langle \mathcal{F}, \delta, acc\rangle, \epsilon_T\rangle, 0\rangle$ **then**
      $T' \leftarrow$ **translate**(δ)
      $\mathcal{X} \leftarrow \langle pp, \langle\langle \mathcal{F}, \delta, acc\rangle, T' \rangle, 0\rangle$
      $S'_t \leftarrow S'_t \cup$ `Get-Leaves`$(\mathcal{X}, T', c-1)$
  return S'_t
end

Definition 3.2. (**Translation from an MAA to a Configuration Tree**) For a metric alternating automaton $\mathcal{A}$, the translation function **translate**($\mathcal{A}$), from $\mathcal{A}$ to a *configuration tree*, is defined as follows:

$$
\begin{array}{lcl}
\textbf{translate}(\epsilon_{\mathcal{A}}) & = & \epsilon_T \\
\textbf{translate}(\mathcal{N}) & = & \langle \epsilon_T, \langle \mathcal{N}, \epsilon_T \rangle, 0 \rangle \\
\textbf{translate}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \langle \epsilon_T, \langle \textbf{translate}(\mathcal{A}_1) \vee \textbf{translate}(\mathcal{A}_1) \rangle, 0 \rangle \\
\textbf{translate}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \langle \epsilon_T, \langle \textbf{translate}(\mathcal{A}_1) \wedge \textbf{translate}(\mathcal{A}_1) \rangle, 0 \rangle \\
\textbf{translate}(\mathcal{A}_x^d) & = & \langle \epsilon_T, \langle d, \textbf{translate}(\mathcal{A}_x) \rangle, 0 \rangle
\end{array}
$$

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
  switch T **do**
    case $\langle p', \langle \mathcal{N}, \epsilon_T \rangle, res \rangle$
      $p' \leftarrow p$
      return $\{\langle T, c \rangle\}$
    case $\langle p', \langle T_1 \vee T_2 \rangle, res \rangle$
      $p' \leftarrow p$
      return `Get-Leaves(` T, T_1, c) $\cup$ `Get-Leaves(`T, T_2, c)
    case $\langle p', \langle T_1 \wedge T_2 \rangle, res \rangle$
      $p' \leftarrow p$
      return `Get-Leaves(` T, T_1, c) $\cup$ `Get-Leaves(`T, T_2, c)
    case $\langle p', \langle d, T' \rangle, res \rangle$*:*
      $p' \leftarrow p$
      return `Get-Leaves(`T, T', d)
  return $\emptyset$
end

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 $\langle \langle p, \langle \mathcal{F}, \delta, acc, \rangle T' \rangle, c \rangle \in S_t$, the subautomaton δ 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_{metric}, such that the clock associated with T_{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 $\langle pp, \langle \mathcal{N}, T' \rangle, res \rangle$
      $res \leftarrow$ **result**(T')
      if $res \neq 0$ **then**
        $T_x \leftarrow$ `Eval-Back`(pp)
    case $\langle pp, \langle T_1 \wedge T_2 \rangle, res \rangle$
      $res \leftarrow$ **result**(T_1) **and** **result**(T_1)
      if $res \neq 0$ **then**
        $T_x \leftarrow$ `Eval-Back`(pp)
    case $\langle pp, \langle T_1 \vee T_2 \rangle, res \rangle$
      $res \leftarrow$ **result**(T_1) **or** **result**(T_1)
      if $res \neq 0$ **then**
        $T_x \leftarrow$ `Eval-Back`(pp)
    case $\langle pp, \langle c, T' \rangle, res \rangle$
      $res \leftarrow$ **result**(T')
      if $res \neq 0$ **then**
        $T_x \leftarrow$ `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* $\mathcal{T}$ in T. The *Backward evaluation* of T, as the name suggests, starts with the leaves and traverses T backwards. A leaf node $\mathcal{X}$ with an associated clock c is checked, using the function **evaluate** [Definition 3.5], whether it is a part of $\mathcal{T}$ or not. The result of the evaluation of $\mathcal{X}$, either 1 or -1, is propagated upwards following the parent pointer. The evaluation result 1 or -1 implies that $\mathcal{X}$ is a part of $\mathcal{T}$ or $\mathcal{X}$ is not a part of $\mathcal{T}$ respectively. Similarly, the parent subtree $P_{\mathcal{X}}$ of $\mathcal{X}$ is then checked whether it belongs to $\mathcal{T}$ 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 $\mathcal{T}$, all the subtrees rooted at $\bigvee$ do not belong to $\mathcal{T}$ 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 $\mathcal{T}$, at least one of the subtrees rooted at $\bigwedge$ does not belong to $\mathcal{T}$ or at least one of the subtrees rooted at $\bigwedge$ is not fully evaluated respectively.

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

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

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

$$\begin{array}{rrrcr}
-1 & \textbf{and} & x & = & -1 \\
x & \textbf{and} & -1 & = & -1 \\
0 & \textbf{and} & 0 & = & 0 \\
1 & \textbf{and} & x & = & x \\
x & \textbf{and} & 1 & = & x \\
-1 & \textbf{or} & x & = & x \\
x & \textbf{or} & -1 & = & x \\
0 & \textbf{or} & 0 & = & 0 \\
1 & \textbf{or} & x & = & 1 \\
x & \textbf{or} & 1 & = & 1
\end{array}$$

At the final position of the trace ρ, the algorithm applies the accepting condition and evaluates each leaf in $\mathcal{R}$ using function **eval-final**.

Procedure `Eval-Tree(`S, ***eval***, ρ_i`)`

Input : A set S of node-clock pairs of a *configuration tree*, a function **eval**, and the current system state ρ_i.
Output: A set of *configuration trees*
.

begin
  $S \leftarrow \emptyset$
  for *each* $\langle p, \langle \mathcal{N}, \epsilon_T \rangle, res \rangle \in S$ **do**
    $res \leftarrow$ **eval**$(\langle \mathcal{N}, c \rangle, \rho_n)$
    if $res \neq 0$ **then**
      $S \leftarrow S \cup$ `Eval-Back`(p)
end

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

$$\textbf{evaluate}(\langle\langle \nu, acc \rangle, c \rangle, \rho_i) = \begin{pmatrix} 1 & if \rho_i \models \nu \\ -1 & otherwise \end{pmatrix}$$

$$\textbf{evaluate}(\langle\langle \mathcal{F}, \delta, acc \rangle, c \rangle, \rho_i) = \begin{pmatrix} 0 & \text{if } \mathcal{F}(c) = true \\ -1 & otherwise \end{pmatrix}$$

$$\textbf{evaluate}(\langle\langle \mathcal{F}, \epsilon, acc \rangle, c \rangle, \rho_i) = \begin{pmatrix} 1 & \text{if } \mathcal{F}(c) = true \\ -1 & otherwise \end{pmatrix}$$

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

$$\textbf{eval-final}(\langle\langle \mathcal{F}, \delta, 0 \rangle, c \rangle, \rho_n) = -1$$

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

$$\textbf{eval-final}(\langle \mathcal{N}, c \rangle, \rho_n) = \textbf{evaluate}(\langle \mathcal{N}, c \rangle, \rho_n)$$

3.2.3 How it works

The main module `Forward-Backward` takes a program trace ρ and a metric alternating automata $\mathcal{A}$ and checks whether ρ is a model of $\mathcal{A}$. At system step

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

Input : An automaton $\mathcal{A}$ and a *program trace* ρ.
Output: A boolean.
begin
  $\mathcal{R} \leftarrow$ **translate**$(\mathcal{A})$
  $S_t \leftarrow$ `Get-Leaves`$(\epsilon, \mathcal{R}, \infty)$
  for *n=1 ...* $|\rho| - 1$ **do**
    $S \leftarrow$ `Eval-Tree`$(S_t,$ **evaluate**$, \rho_n)$
    if $\mathcal{R} \in S$ *and **result***$(\mathcal{R}) \neq 0$ **then**
      return result$(\mathcal{R})$
    $S_t \leftarrow$ `Forward`(S_t)
  return $\mathcal{R} \in$ `Eval-Tree`$(S_t,$ **eval-final**$[Definition 3.6], \rho_n)$ and **result**$(\mathcal{R}) = 1$
end

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 $\langle \mathcal{X}, c \rangle \in S$. 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 `Foward` to expand $\mathcal{R}$ and to compute the successor set of node-clock pairs.

4. Repeats step 1, 2 and 3 until ρ 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 $\Diamond_{[0,\infty]}(a \wedge ((\Diamond_{[0,2]} p) \, \mathcal{U}_{[0,3]} \; q \;))$.

As discussed earlier, the *Forward-Backward* algorithm unrolls $\mathcal{A}$ into a *configuration tree T* and then detects a run $\mathcal{T}$ of $\mathcal{A}$ in *T* such that $\mathcal{T}$ starts from the root of *T* and ends at the accepting nodes. Figure 3.2 shows different configurations

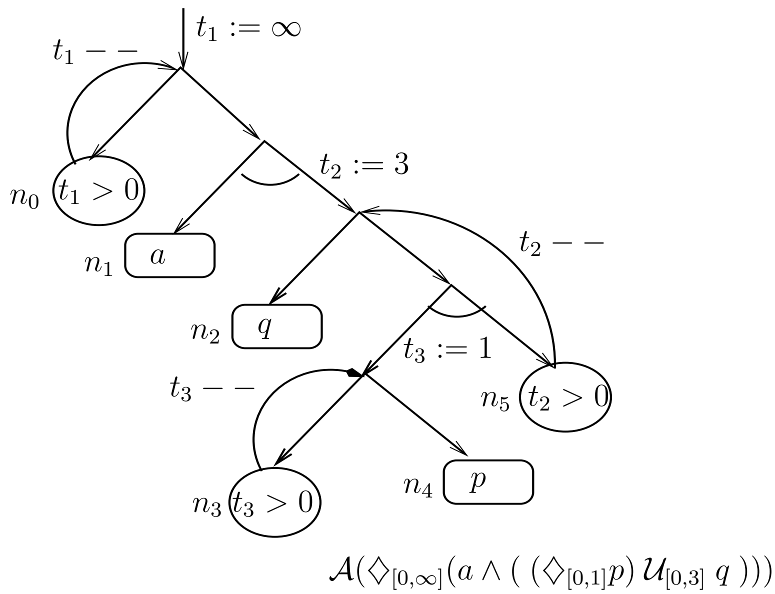

Figure 3.1: A Construction of a *metric alternating automaton* from the BTL formula $\Diamond_{[0,\infty]}(a \wedge ((\Diamond_{[0,1]}p) \,\mathcal{U}_{[0,3]}\, q))$.

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 $\mathcal{T}$, the thick lines represent the subtrees that belong to $\mathcal{T}$, and the normal lines represent subtrees that are not fully evaluated yet.

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

Step 1:

- The algorithm translates $\mathcal{A}$ into *T*, as shown in Figure 3.2(1a).
- The nodes of *T* are evaluated to either $-1, 1$ or 0, and the procedure `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.

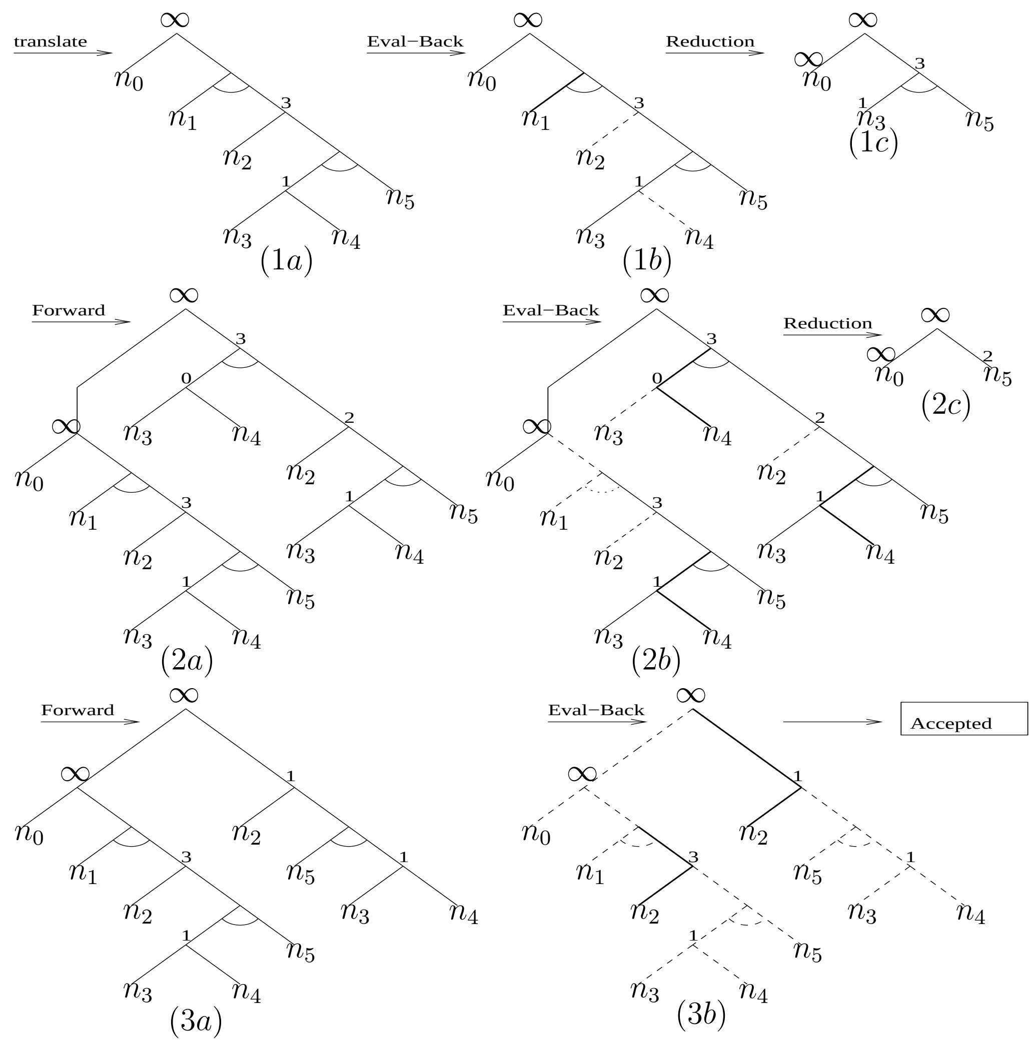

Figure 3.2: A stepwise construction of a *configuration tree* .

- The procedure `Eval-Back` propagates the result of the evaluation of nodes n_1, n_21, 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).

- ρ is accepted, as there exists an accepting run of ρ in $\mathcal{A}$. Figure 3.2(3b) shows a $\mathcal{T}$ 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* ρ 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 ρ, 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 $X = |\mathcal{A}|$ and $K \in \mathbb{Z}^+$.

Proof. **Base Case:** At the system step 1, $Z_1 \leq K * X$ 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 $n+1$,

$$Z_{n+1} \leq K * \sum_{i=0}^{n+1} X^i \text{ and } L_{n+1} \leq X^{n+1}.$$

As per our assumption, $L_n \leq X^n$ at system step n. Each leaf adds a subtree of maximum size $K * X$ 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 $X^n * K * X$.

The upper bound of Z_{n+1} of T, at system step $n+1$, after expansion can be computed a follows:

$$Z_{n+1} \leq K * \sum_{i=0}^{n} X^i + (X^n)K * X$$

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

$$\Rightarrow Z_{n+1} \leq K * \sum_{i=0}^{n+1} x^i$$

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:

$$L_{n+1} \leq (x^n) * x$$
$$\Rightarrow L_{n+1} \leq X^{n+1}$$

□

Theorem 3.1. Given a program trace ρ 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 $K * \sum_{i=0}^{|\rho|} |\mathcal{A}|^i$.

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 $\mathcal{O}(|\mathcal{A}|^{|\rho|})$.

□

Theorem 3.2. Given a program trace ρ and a *metric alternating automaton* $\mathcal{A}$, *Forward-Backward*$(\mathcal{A}, \rho)$ = **true**, if there exists an accepting run of ρ 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 $\langle \mathcal{X}, c\rangle$ for *T*, where $\mathcal{X}$ is a node in T under the scope of a clock c. For any two node-clock pairs $\langle\langle p, \langle \mathcal{N}, T'\rangle, res\rangle, c\rangle$ and $\langle\langle p', \langle \mathcal{N}', T'\rangle, res\rangle, c'\rangle$ in S_t, the *Forward-Backward* algorithm constructs two similar successor subtrees T' and T'', if $\mathcal{N} = \mathcal{N}'$ 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* $\mathcal{G}$ that allows us to have multiple predecessors. We also modify our *Forward-Backward* algorithm slightly to work on $\mathcal{G}$. The new algorithm is called *Optimized Forward-backward* algorithm.

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

$$
\begin{array}{lll}
G ::= & \epsilon_G & \textit{empty DAG} \\
| & \langle S_p, \langle \mathcal{N}, c\rangle, res\rangle & \textit{a leaf} \\
| & \langle S_p, \langle \wedge, S_s\rangle, res\rangle & \textit{a conjunctive branching point with a set of sub-DAGs} \\
| & \langle S_p, \langle \vee, S_s\rangle, res\rangle & \textit{a disjunctive branching point with a set of sub-DAGs}
\end{array}
$$

where, $\mathcal{N}$ is a node of $\mathcal{A}$, 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* $\bigwedge$ is evaluated to $true$ if all of the sub-DAGs are evaluated to $true$. Whereas, a *disjunctive branching point* 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 $\mathcal{A}$ and a clock c, the translation function **translate**($\mathcal{A}$, c) is defined as follows:

$$
\begin{array}{lll}
\textbf{translate}(\epsilon_{\mathcal{A}}, c) & = & \epsilon_G \\
\textbf{translate}(\mathcal{N}, c) & = & \langle \emptyset, \langle \mathcal{N}, c\rangle, 0\rangle \\
\textbf{translate}(\mathcal{A}_1 \vee \mathcal{A}_2, c) & = & \langle \emptyset, \langle \vee, \{\textbf{translate}(\mathcal{A}_1, c)\} \cup \{\textbf{translate}(\mathcal{A}_1, c)\}\rangle, 0\rangle \\
\textbf{translate}(\mathcal{A}_1 \wedge \mathcal{A}_2, c) & = & \langle \emptyset, \langle \wedge, \{\textbf{translate}(\mathcal{A}_1, c)\} \cup \{\textbf{translate}(\mathcal{A}_1, c)\}\rangle, 0\rangle \\
\textbf{translate}(\mathcal{A}_0^d, c) & = & \textbf{translate}(\mathcal{A}_0, d)
\end{array}
$$

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.

begin
  switch G **do**
    case ϵ_G
      return $\emptyset$
    case $\langle S_{pp}, \mathcal{X}, res\rangle$
      $S_{pp} \leftarrow S_p$
      return $\{\langle S_{pp}, \mathcal{X}, res\rangle\}$
    case $\langle S_{pp}, \langle\vee, S_s\rangle, res\rangle$
      $S_{pp} \leftarrow S_p$
      return $\biguplus_{E \in S_s}$ `Get-Leaves(`$\{G\}, E$`)`
    case $\langle S_{pp}, \langle\wedge, S_s\rangle, res\rangle$
      $S_{pp} \leftarrow S_p$
      return $\biguplus_{E \in S_s}$ `Get-Leaves(`$\{G\}, E$`)`
end

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

$$\textbf{add-child}(S_b, \mathcal{G}) \;=\; \bigcup_{\langle S_p, \langle x, S_s\rangle, res\rangle \in S_b} \langle S_p, \langle x, S_s \cup \{\mathcal{G}\}\rangle, 0\rangle$$

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

$$\textbf{remove-child}(S_b, \mathcal{G}) \;=\; \bigcup_{\langle S_p, \langle x, S_s\rangle, res\rangle \in S_b} \langle S_p, \langle x, S_s - \{\mathcal{G}\}\rangle, 0\rangle$$

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* $\mathcal{G}$. However, the *Optimized Forward-Backward* algorithm does extra computations to keep the size of $\mathcal{G}$ smaller. As stated above, the *configuration tree* T maintained

Procedure `Forward(`S_t`)`

Input : A set S_t of leaves of a *configuration DAG G*.
Output: A set of leaves of expanded *G* + expansion of *G* as a side effect.
begin
  $S'_t \leftarrow \emptyset$
  for ***each*** $\mathcal{X} \in S_t$ **do**
    if $\mathcal{X} = \langle S_p, \langle\langle \mathcal{F}, \delta, acc\rangle, c\rangle, 0\rangle$ **then**
      $G' \leftarrow$ **translate**$(\delta, c-1)$
      $S_p \leftarrow$ **remove-child**$(S_p, \mathcal{X})$
      $S_p \leftarrow$ **add-child**(S_p, G')
      $S'_t \leftarrow S'_t \uplus$ `Get-Leaves`(S_p, G')
  return S'_t
end

by the *Forward-Backward* algorithm contains multiple copies of the same subtrees in *T*. The *optimized forward-backward* algorithm avoids such duplication by merging isomorphic node-clock pairs.

We assume that the union operation $\uplus$, beside computing union of two sets, merges the isomorphic leaves. For example, two *isomorphic leaves* $\mathcal{X}_1 = \langle S_p, \langle \mathcal{N}, c\rangle, res\rangle$ and $\mathcal{X}_2 = \langle S'_p, \langle \mathcal{N}, c\rangle, res\rangle$ can be replaced by a single leaf $\mathcal{X} = \langle S_p \cup S'_p, \langle \mathcal{N}, c\rangle, res\rangle$. By using function **add-child**, $\mathcal{X}$ is added to every element of the set $S_p \cup S'_p$. Similarly, using function **remove-child**, $\mathcal{X}_1$ and $\mathcal{X}_2$ are removed from S_p and S'_p respectively.

3.3.2 Backward Evaluation

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

The procedure `Eval-Back` returns a set S of DAGs that are evaluated to either -1 or 1. To keep *G* in a compact form, `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 $\mathcal{B}$ over a set $\{G'\}$ of size 1. In such a situation, `Eval-Back` reduces the evaluation of $\mathcal{B}$ to the evaluation of G', and replaces $\mathcal{B}$ by G'.

Procedure `Eval-Back(`S, G`)`

Input : A set S of *configuration DAGs* and a *configuration DAG G*.
Output: A set of *configuration DAGs*.

```
begin
    S′ ← {G}
    for each ⟨S_p, ⟨x, S_s⟩, res⟩ ∈ S do
        if x = ∧ then
            res ← and_{E ∈ S_s} result(E)
        else
            res ← or_{E ∈ S_s} result(E)
        if res ≠ 0 then
            S′ ← S′ ∪ Eval-Back(S_p, ⟨S_p, ⟨x, S_s⟩, res⟩)
        else
            S_s ← S_s − {G}
            if S_s = {E} then
                S_p ← remove-child(S_p, ⟨S_p, ⟨x, S_s⟩, res⟩)
                S_p ← add-child(S_p, E)
    return S′
end
```

3.3.3 How it works

The main module `FORWARD-BACKWARD` takes a program trace ρ and a metric alternating automaton $\mathcal{A}$ and checks whether ρ 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 $\langle \mathcal{X}, c \rangle \in S_t$.

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

3. Calls the procedure `Foward` 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 ρ reaches its last state.

At the final position of the trace ρ, the algorithm applies the accepting condition and evaluates each leaf in $\mathcal{R}$ using the function **eval-final**.

Procedure `Eval-DAG(`S_t, ***eval***, ρ_i`)`

Input : A set S_t of leaves of a *configuration DAGs*, a function **eval** and the current system state ρ_i.
Output: A set of *configuration DAGs*.
begin
  $S = \emptyset$
  for ***each*** $\langle S_p, \mathcal{X}, res \rangle \in S_t$ **do**
    $res \leftarrow$ **eval**$(\mathcal{X}, \rho_i)$
    if $res \neq 0$ **then**
      $S \leftarrow S \cup$ `Eval-Back`$(S_p, \langle S_p, \mathcal{X}, res \rangle)$
  return S.
end

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

Input : An automaton $\mathcal{A}$ and a *program trace* ρ.
Output: A Boolean.
begin
  $\mathcal{R} \leftarrow$ **translate**$(\mathcal{A}, \infty)$
  $S_t \leftarrow$ `Get-Leaves`$(\emptyset, \mathcal{R})$
  for *n=1 ...* $|\rho| - 1$ **do**
    $S \leftarrow$ `Eval-DAG`$(S_t,$ **evaluate**$, \rho_n)$
    if $\mathcal{R} \in S$ *and* ***result***$(\mathcal{R}) \neq 0$ **then**
      return result$(\mathcal{R})$
    $S_t \leftarrow$ `Forward`(S_t)
  return $\mathcal{R} \in$ `Eval-DAG`$(S_t,$ **eval-final**$, \rho_n)$ and **result**$(\mathcal{R}) = 1$
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 $\mathcal{A}$, shown in Figure 3.1, and the same execution trace $\rho = [\langle a, \neg p, \neg q \rangle, \langle \neg a, p, \neg q \rangle, \langle \neg a, \neg p, q \rangle]$ to discuss how *Optimized Forward-Backward* works?

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

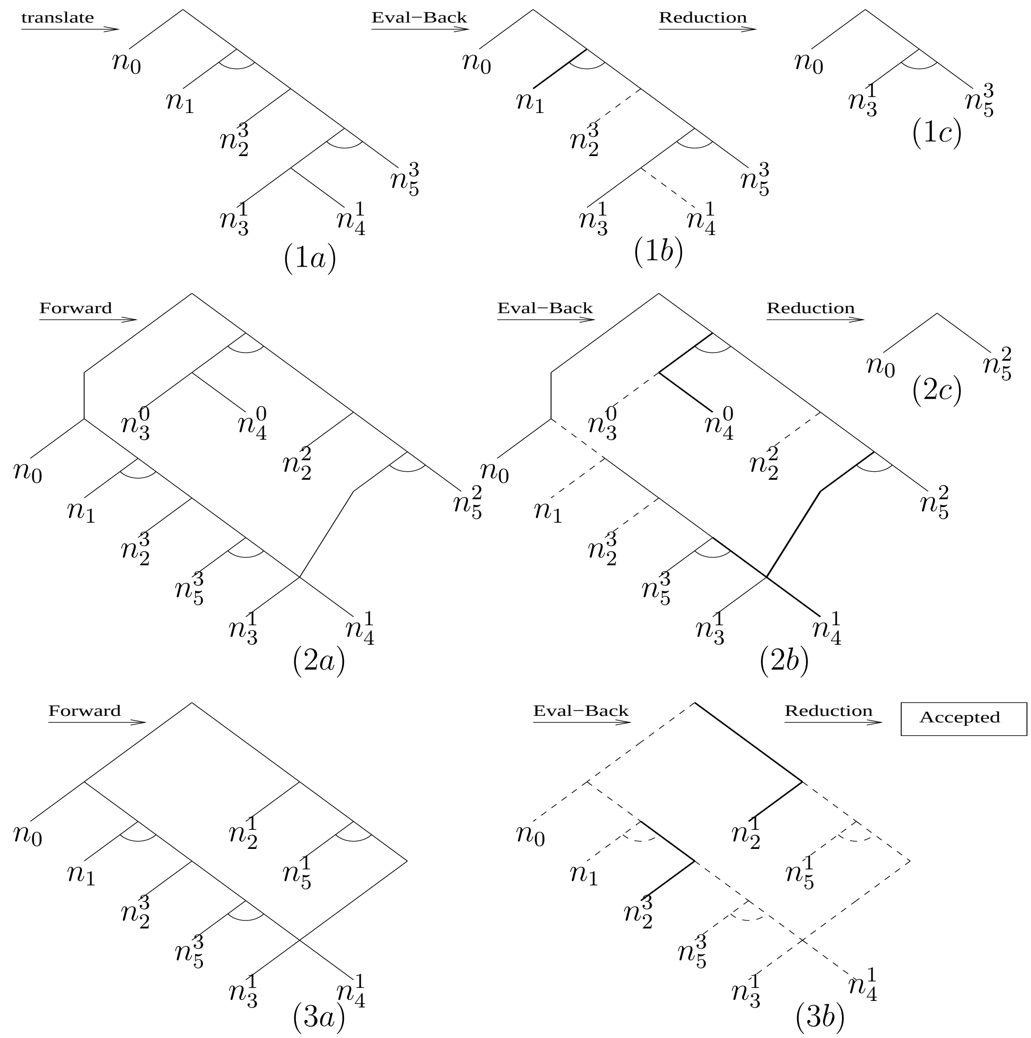

Figure 3.3: A stepwise construction of a *configuration DAG*

thick lines represent the sub-DAGs that belong to $\mathcal{T}$, 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 $\mathcal{G}$ is represented as $\mathcal{N}_x^c$, where $\mathcal{N}_x$ is a node of $\mathcal{A}$ and c is an associated clock.

To check ρ against $\mathcal{A}$ for acceptence, *Forward-Backward* works as follow:

Step 1:

- The algorithm translates $\mathcal{A}$ into $\mathcal{G}$, as shown in Figure 3.3(1a).
- The nodes of $\mathcal{G}$ are evaluated to $-1, 1$ or 0, and the procedure `Eval-Back` propagates the evaluation result upwards in $\mathcal{G}$, 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_5^3 are evaluated to 0 as the associated time constrains are fulfilled.

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

Step 2:

- The procedure Forward expands $\mathcal{G}$ 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_3^1, n_4^1) are merged.
- The procedure `Eval-Back` propagates the result of the evaluation of the nodes $n_1, n_2^3, n_2^2, n_3^0, n_4^0$ and n_4^1 upwards in $\mathcal{G}$, as shown in Figure 3.3(2b).
- The compact $\mathcal{G}$, 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_2^3 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).
- ρ is accepted as there exists a $\mathcal{T}$, such that $\mathcal{T}$ starts from the root of $\mathcal{G}$ and ends at an accepting node n_2^2, as shown in Figure 3.3(3b).

Theorem 3.3. Given a program trace ρ and a *metric alternating automaton* $\mathcal{A}$ constructed from a BTL formula φ, `FORWARD-BACKWARD` runs in time $\mathcal{O}(X^2 * (M_c + 2) * |\rho|)$ and space $\mathcal{O}(X^2 * (M_c + 2) * |\rho|)$, where M_c is the largest constant appearing in φ (excluding ∞) and $X = |\mathcal{A}|$.

Proof. The procedure `FORWARD-BACKWARD` maintains a *configuration DAG* $\mathcal{G}$ during its execution and the space complexity of `FORWARD-BACKWARD` is linear in the size of $\mathcal{G}$. At each system step, $\mathcal{G}$ is expanded by the procedure `Forward` form the leaves. The size Z_n of $\mathcal{G}$ 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 $\mathcal{G}$ at each system step.

The procedure `Forward` takes a set S_t of leaves $\langle S_p\langle\mathcal{N}, c\rangle, res\rangle$ of $\mathcal{G}$, where $\mathcal{N}$ is a node of $\mathcal{A}$ paired with an associated clock c. The size of S_t is always bounded by $X * (M_c + 2)$, as the number of nodes in $\mathcal{A}$ are bounded by X.

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

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

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

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

The running time of the algorithm, for ρ, is therefore $\mathcal{O}(X^2 * (M_c + 2) * |\rho|)$. □

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

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

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

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

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 ρ and a *metric alternating automaton* $\mathcal{A}$, constructed from FBTL formula φ, `FORWARD-BACKWARD` runs in time $\mathcal{O}(X^3 * (M_c + 1)^2)$ and space $\mathcal{O}(X^3 * (M_c + 1)^2)$, where X is the size of $\mathcal{A}$, and M_c is the maximum constant appearing in φ.

Proof. As proved in Theorem 3.3, the size of a *configuration DAG* ($\mathcal{G}$) is incremented by at most $X^2 * (M_c + 2)$ 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 $\mathcal{G}$ are bounded by $X*(M_c+1)$ and increment in $|\mathcal{G}|$ at any system step is bounded by $X^2 * (M_c + 1)$.

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

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

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

□

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 φ and length of an execution trace ρ respectively. The algorithm is useful for checking a smaller prefix of ρ against a relatively larger size of φ. To monitor a larger (possibly infinite) ρ, the essential requirement is to bring down the space complexity to constant in the size of ρ. 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 ρ.

We present a modified version of the *Breadth-First* algorithm that works on *metric alternating automata*. For a *metric alternating automaton* $\mathcal{A}$ translated from a BTL formula φ, the algorithm maintains a set S of *system configurations* that are consistent with the prefix of an *execution trace* ρ seen so far. A configuration $C \in S$ is a set of pairs $\langle \mathcal{N}, t \rangle$, where $\mathcal{N}$ is a node of $\mathcal{A}$ and $c \in \{0, 1..., M_c, \infty\}$ is an associated clock, and M_c is the maximum constant appearing in φ. ρ is accepted by $\mathcal{A}$ 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 $\langle \mathcal{N}, c \rangle$, where $\mathcal{N}$ is a node of a *metric alternating automata* and c is an associated clock.

Definition 3.13. For a *metric alternating automaton* $\mathcal{A}$, and an associated clock c, the function $\textbf{init}(A, c)$ that computes a set S of configurations is defined as follows:

$$
\begin{array}{lcl}
\textbf{init}(\epsilon_{\mathcal{A}}, c) & = & \emptyset \\
\textbf{init}(\mathcal{N}, c) & = & \{\langle \mathcal{N}, c\rangle\} \\
\textbf{init}(\langle \mathcal{A}_0^d \rangle, c) & = & \textbf{init}(\mathcal{A}_0, d)) \\
\textbf{init}(\mathcal{A}_1 \vee \mathcal{A}_2, c) & = & \textbf{init}(\mathcal{A}_1, c) \cup \textbf{init}(\mathcal{A}_2, c) \\
\textbf{init}(\mathcal{A}_1 \wedge \mathcal{A}_2, c) & = & \textbf{init}(\mathcal{A}_1, c) \otimes \textbf{init}(\mathcal{A}_2, c)
\end{array}
$$

where, $\otimes$ denotes the following:

$\{C_1 ... C_n\} \otimes \{C'_1 ... C'_m\} \quad = \{C_i \uplus C'_j | i = 1...n, j = 1...m\}$

Procedure `BREADTH-FIRST(`$\mathcal{A}, \rho$`)`

Input : An automaton $\mathcal{A}$ and a *program trace* ρ
Output: A boolean
begin
  $S \leftarrow$ **init**$(\mathcal{A})$
  for *n=0 ...* $|\rho| - 2$ **do**
    $S' \leftarrow \emptyset$
    for *each* C *in* S **do**
      if ***state-satisfied(***C, ρ_n***)*** **then**
        $S' \leftarrow S' \cup$ **Successor**(C)
    $S \leftarrow S'$
  $S' \leftarrow \emptyset$
  for *each* C *in* S **do**
    if ***state-satisfied(*** C, ρ_{n-1}***)*** ***and final(***C***)*** **then**
      $S' \leftarrow S' \cup \{C\}$
  return $S' \leftarrow \emptyset$
end

Definition 3.14. For a configuration C , the function **Successor**(C) is define as follows:

$$\textbf{successor}(C) \quad = \bigotimes_{\langle\langle \mathcal{F}, \delta, acc\rangle, c\rangle \in C} \textbf{init}(\delta, c-1)$$

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

for every $X \in C$, **evaluate**(X, s) returns $true$.

Definition 3.16. For a configuration C , the function **final**(C) returns $true$ if

for every $\langle \mathcal{A}, c\rangle \in C$, $\mathcal{A}.acc = 1$.

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 $\mathcal{A}$, translated from BTL formula φ, the space requirement for *Breadth-First* is bounded by $2^{X*(M_c+2)}$, where $X = |\mathcal{A}|$ and M_c is the maximum constant appearing in φ.

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'.

Recall that without any optimization, that maximum possible size of the C is $X * (M_c + 2)$. 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* $\mathcal{A}$, shown in Figure 3.1, and the *execution trace* $\rho = [\langle a, \neg p, \neg q\rangle, \langle \neg a, p, \neg q\rangle, \langle \neg a, \neg p, q\rangle]$, the *Optimized Breadth-First* algorithm checks ρ against $\mathcal{A}$ by generating all the possible configurations at each system step. The program trace ρ 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 $\langle \mathcal{N}, c\rangle$ by $\mathcal{N}^c$ in this section.

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

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

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

Two configurations $\{n_1^\infty\}$ and $\{n_1^\infty, n_3^1, n_5^3\}$ 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:

$S_2 =$ **successor**$(\{n_0^\infty\}) \cup$ **successor**$(\{n_1^\infty, n_3^1, n_5^3\})$

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

Three configurations $\{n_0^\infty\}, \{n_4^0, n_5^2\}$ 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 **Successor** as follows:

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

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

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 sate-satisfied or not accepting. Thus, ρ is accepted by $\mathcal{A}$.

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 $\uplus$, 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 $\mathcal{A}$, as each element $\langle \mathcal{N}, c \rangle \in C$ has distinct $\mathcal{N}$. The size of S_c, as proved in Theorem 3.7, is bounded by $(M_c + 2)^{|\mathcal{A}|}$.

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

Proof. To prove our claim, we make the following case distinctions:
Case 1: $\mathcal{N} = \langle \nu, acc \rangle$
The evaluation of non-timer nodes does not depend on the value of the clock. Thus, $\langle \mathcal{N}, c \rangle \Leftrightarrow \langle \mathcal{N}, c' \rangle$ and any of the pair can be picked randomly.
Case 2:$\mathcal{N} = \langle \lambda x.x > 0, \delta, 1 \rangle$

- $c > 0$ 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 δ 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 λ 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 δ 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 λ 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.

$\square$

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

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 $\langle \mathcal{N}, c \rangle \in C$ has a distinct $\mathcal{N}$, where $\mathcal{N}$ is a node in $\mathcal{A}$ 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* ρ and a *metric alternating automaton* $\mathcal{A}$, `BREADTH-FIRST`$(\mathcal{A}, \rho)$ =**true**, if there exists an accepting run of ρ in $\mathcal{A}$.

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

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 $\langle \mathcal{N}, c \rangle \in C$ has a distinct $\mathcal{N}$. C is mapping from a set of nodes of $\mathcal{A}$ $\{\mathcal{N}_1 ... \mathcal{N}_X\}$ 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 $\mathcal{O}(X * (M_c + 2)^X)$

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 $X * 2^X * (M_c + 2)^X$ or $X * (M_c + 2)^{2X}$.

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

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

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

$\square$

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 ∞ 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* $\mathcal{A}$ into an *annotated metric alternating automaton* $\mathcal{A}_a$. The additional information associated with $\mathcal{A}_a$ guides the algorithm to apply either *breadth first* or *forward backward* techniques for each *metric subautomaton* $\mathcal{A}_{metric} \in \mathcal{A}_a$.

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 can not 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* $\mathcal{A}$ is defined as follows:

$$\begin{array}{rll} \mathcal{A} ::= & \epsilon_{\mathcal{A}} & \textit{empty automaton} \\ | & \mathcal{N} & \textit{an automaton node} \\ | & \mathcal{A} \wedge \mathcal{A} & \textit{conjunction of two automata} \\ | & \mathcal{A} \vee \mathcal{A} & \textit{disjunction of two automata} \\ | & \langle \mathcal{A}^d, k \rangle & \textit{metric sub-automaton}\ , \end{array}$$

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

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

$$\begin{array}{lcl} \textbf{cons-sum}(\epsilon_{\mathcal{A}}) & = & 0 \\ \textbf{cons-sum}(\mathcal{N}) & = & 0 \\ \textbf{cons-sum}(\mathcal{A}_0^d) & = & d + \textbf{cons-sum}(\mathcal{A}_0) \\ \textbf{cons-sum}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \textbf{cons-sum}(\mathcal{A}_1) + \textbf{cons-sum}(\mathcal{A}_2) \\ \textbf{cons-sum}(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & \textbf{cons-sum}(\mathcal{A}_1) + \textbf{cons-sum}(\mathcal{A}_2) \end{array}$$

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

$$\begin{array}{lcl} \textbf{annotate}(\epsilon_{\mathcal{A}}) & = & \emptyset \\ \textbf{annotate}(\mathcal{N}) & = & \mathcal{N} \\ \textbf{annotate}(\mathcal{A}_0^d) & = & \langle \textbf{annotate}(\mathcal{A}_0), d + \textbf{cons-sum}(\mathcal{A}_x) \rangle \\ \textbf{annotate}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \textbf{annotate}(\mathcal{A}_1) \vee \textbf{annotate}(\mathcal{A}_2) \\ \textbf{annotate}(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & \textbf{annotate}(\mathcal{A}_1) \wedge \textbf{annotate}(\mathcal{A}_2) \end{array}$$

Definition 4.4. (Configuration DAG) A *configuration DAG G*, generated by unrolling $\mathcal{A}$, is defined as follows:

$$
\begin{array}{rll}
G ::= & \epsilon_G & \textit{empty DAG} \\
| & \langle S_p, \langle \mathcal{N}, c\rangle, res\rangle & \textit{a leaf represents a node-clock pair} \\
| & \langle S_p, C, res\rangle & \textit{a leaf represents a set of node-clock pair} \\
| & \langle S_p, \langle \wedge, S\rangle, res\rangle & \textit{a conjunctive branching point with a set of G} \\
| & \langle S_p, \langle \vee, S\rangle, res\rangle & \textit{a disjunctive branching point with a set of G}
\end{array}
$$

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 $res \in \{-1, 0, 1\}$.

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 **translate** ($\mathcal{A}$,c) is defined as follows:

$$
\begin{array}{lll}
\textbf{translate}(\epsilon_{\mathcal{A}}, c) & = & \emptyset \\
\textbf{translate}(\mathcal{N}, c) & = & \langle \emptyset, \langle \mathcal{N}, c\rangle, 0\rangle \\
\textbf{translate}(\mathcal{A}_1 \vee \mathcal{A}_2, c) & = & \langle \emptyset, \langle \vee, \{\textbf{translate}(\mathcal{A}_1, c)\} \cup \{\textbf{translate}(\mathcal{A}_1, c)\}\rangle, 0\rangle \\
\textbf{translate}(\mathcal{A}_1 \wedge \mathcal{A}_2, c) & = & \langle \emptyset, \langle \wedge, \{\textbf{translate}(\mathcal{A}_1, c)\} \cup \{\textbf{translate}(\mathcal{A}_1, c)\}\rangle, 0\rangle \\
\textbf{translate}(\langle \mathcal{A}_0^d, k\rangle, c) & = & \textbf{translate}(\mathcal{A}_0, d)
\end{array}
$$

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

$$
\begin{array}{lll}
\textbf{init}(\epsilon_{\mathcal{A}}, c) & = & \emptyset \\
\textbf{init}(\mathcal{N}, c) & = & \{\langle \{\langle \mathcal{N}, c\rangle\}, \epsilon_G\rangle\} \\
\textbf{init}(\langle \mathcal{A}_0^d, k\rangle, c) & = & \{\langle \emptyset, \{\textbf{translate}(\mathcal{A}_0, d)\}\rangle\} \\
\textbf{init}(\langle \mathcal{A}_0^d, \infty\rangle, c) & = & \textbf{init}(\mathcal{A}_0, d) \\
\textbf{init}(\mathcal{A}_1 \vee \mathcal{A}_2, c) & = & \textbf{init}(\mathcal{A}_1, c) \cup \textbf{init}(\mathcal{A}_2, c) \\
\textbf{init}(\mathcal{A}_1 \wedge \mathcal{A}_2, c) & = & \textbf{init}(\mathcal{A}_1, c) \otimes \textbf{init}(\mathcal{A}_2, c)
\end{array}
$$

where, $\otimes$ denotes the following:

$$
\{\langle C_1, D_1\rangle ... \langle C_n, D_n\rangle\} \otimes \{\langle C_1', D_1'\rangle ... \langle C_m', D_m'\rangle\} \quad = \quad \begin{array}{l} \{\langle C_i \uplus C_j', D_i \cup D_j'\rangle \\ |i = 1...n, j = 1...m\} \end{array}
$$

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 $\mathcal{A}$ and c is an associated clock.
- D is a set of configuration DAGs.

For an alternating automaton $\mathcal{A}$ and an execution trace ρ, the algorithm generates a set S of possible system configurations at every system step, using the function **init**. The initial configuration set S_1 is computed by calling **init** with argument $\mathcal{A}$. ρ is accepted by $\mathcal{A}$ 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 `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**
    $\mathcal{X} \leftarrow \langle \emptyset, C, res \rangle$
    $G_c \leftarrow \langle \emptyset, \langle \wedge, \{\mathrm{X}\} \cup D \rangle, 0 \rangle$
    $D' \leftarrow D' \cup \{G_c\}$
  $G \leftarrow \langle \emptyset, \langle \vee, \{D'\} \rangle, 0 \rangle$
  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 CC in the configuration set S, generated by *Generic* using the function **init**, contains a set D of DAGs. The evaluation of CC can not be reduced to the evaluation of successor configurations unless each $d \in D$ is fully evaluated. Since *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* ($\mathcal{G}$). The translation from S to $\mathcal{G}$ is shown in the procedure `Generate-DAG`. $\mathcal{G}$ is a *disjunctive branching point* over a set D_{sub} of *configuration sub-DAGs* $\langle S_p, \langle \wedge, \{\langle \mathcal{G}, C, 0\rangle\} \cup D\rangle$ 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* $\mathcal{G}$.
Output: A set of leaves of the expanded $\mathcal{G}$ + expansion of $\mathcal{G}$ as a side effect.

begin
  $S'_t \leftarrow \emptyset$
  for *each* $\mathcal{X} \in S_t$ **do**
    switch $\mathcal{X}$ **do**
      case $\langle S_p, C, res\rangle$
        $S_p \leftarrow$ **remove-child** ($S_p, \mathcal{X}$)
        $S' \leftarrow \bigotimes_{\langle\langle \mathcal{F}, \delta, acc\rangle, c\rangle \in C}$ **init**$(\delta, c-1)$
        $\mathcal{G} \leftarrow$ `Generate-DAG`(S')
        $S_p \leftarrow$ **add-child** ($S_p, \mathcal{G}$)
        $S'_t \leftarrow S'_t \cup$ `Get-Leaves`$(S_p, \mathcal{G})$
      case $\langle S_p, \langle\langle \mathcal{F}, \delta, acc\rangle, c\rangle, 0\rangle$
        $S_p \leftarrow$ **remove-child** ($S_p, \mathcal{X}$)
        $\mathcal{G} \leftarrow$ **translate**$(\delta, c-1)$
        $S_p \leftarrow$ **add-child** ($S_p, \mathcal{G}$)
        $S'_t \leftarrow S'_t \cup$ `Get-Leaves`$(S_p, \mathcal{G})$
  return S'_t
end

4.2.2 Forward Expansion

A *configuration DAG* ($\mathcal{G}$) 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 $\mathcal{G}$ and expands $\mathcal{G}$ 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.

Procedure `Eval-DAG(`S_t, ***eval***, ρ_i`)`

Input : A set S_t of leaves of a *configuration DAG*, a function **eval** and the current system state ρ_i.
Output: A set of *configuration DAGs*.

```
begin
    S ← ∅
    for each ⟨S_p, X, res⟩ ∈ S_t do
        if X = ⟨N, c⟩ then
            res ← eval(X, ρ_i)
        else
            for each ⟨N, c⟩ ∈ X do
                res ← res and eval(⟨N, c⟩, ρ_i)
        if res ≠ 0 then
            S ← S ∪ Eval-Back(S_p, ⟨S_p, X, res⟩)
    return S.
end
```

4.2.3 Backward Evaluation

The *backward evaluation* of a *configuration DAG* $\mathcal{G}$ starts off from the leaves. The procedure `Eval-DAG` takes a set S_t of leaves of $\mathcal{G}$ and evaluates every element in S_t. The evaluation result is propagated upwards using the procedure `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 $\langle \mathcal{N}, c\rangle \in C$ is evaluated to -1, all $\langle \mathcal{N}, c\rangle \in C$ are evaluated to -1 or one of $\langle \mathcal{N}, c\rangle \in C$ is evaluated to 0 respectively.

4.2.4 How it works

The procedure `GENERIC` takes a program trace ρ of length n and a *metric alternating automaton* $\mathcal{A}$ and checks whether ρ is a model of $\mathcal{A}$. A set S of initial

```
Procedure GENERIC(𝒜, ρ)
    Input  :  An automaton 𝒜 and a program trace ρ.
    Output :  A Boolean.
    begin
        S ← init(𝒜, ∞)
        ℛ ← Generate-DAG(S)
        S_t ← Get-Leaves(∅, ℛ)
        for n=1 ... |ρ| − 1 do
            S_d ← Eval-DAG(S_t, evaluate, ρ_n)
            if ℛ ∈ S_d and result(ℛ) ≠ 0 then
                return result(ℛ)
            for each ⟨S_p, X, res⟩ ∈ S_d do
                S_p ← remove-child (S_p, ⟨S_p, X, res⟩)
            S_t ← Forward(S_t)
        return ℛ ∈ Eval-DAG(S_t, eval-final) and result (ℛ) = 1
    end
```

system configurations is computed and then translated into a *configuration DAG* $\mathcal{R}$. $\mathcal{R}$ 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 $\mathcal{R}$ backwards from the leaves.

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

3. Removes all the sub-DAGs in $\mathcal{R}$ that are evaluated to 1 or -1.

4. Calls the procedure `Forward` to expand $\mathcal{R}$ from the leaves, and computes a set of leaves of expanded $\mathcal{R}$.

5. Repeats step 1, 2, 3, 4 and 5 until ρ reaches its last state.

At the final position of the trace ρ, 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 $\mathcal{A}$. $\mathcal{A}$, shown in Figure 4.1, is constructed from the BTL formula $\Diamond_{[0,\infty]}(a \wedge ((\Diamond_{[0,1]}p)\mathcal{U}_{[0,\infty]}q))$.

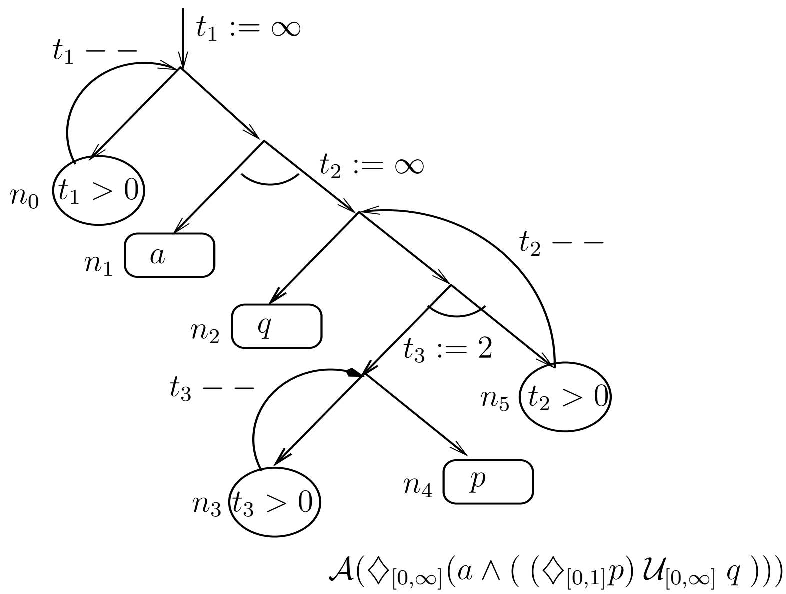

Figure 4.1: A Construction of a *metric alternating automaton* from the BTL formula $\Diamond_{[0,\infty]}(a \wedge ((\Diamond_{[0,1]}p) \,\mathcal{U}_{[0,\infty]}\, q))$.

As discussed in previous sections, the *Generic* algorithm stores system configurations in the form a *configuration DAG* $\mathcal{G}$. The algorithm tries to detect a path $\mathcal{G}$ that starts at the root of $\mathcal{G}$ and ends at accepting nodes.

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

To check ρ against $\mathcal{A}$ for acceptence, *Generic* works follow:

Step 1:

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

- Both the type A and the type B nodes of $\mathcal{G}$ are evaluated, and the procedure `Eval-Back` propagates the evaluation result upwards in $\mathcal{G}$, 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 $\mathcal{G}$ that are evaluated in the previous step are removed and $\mathcal{G}$ is reduced to a compact form, as shown in Figure 3.3(2c).

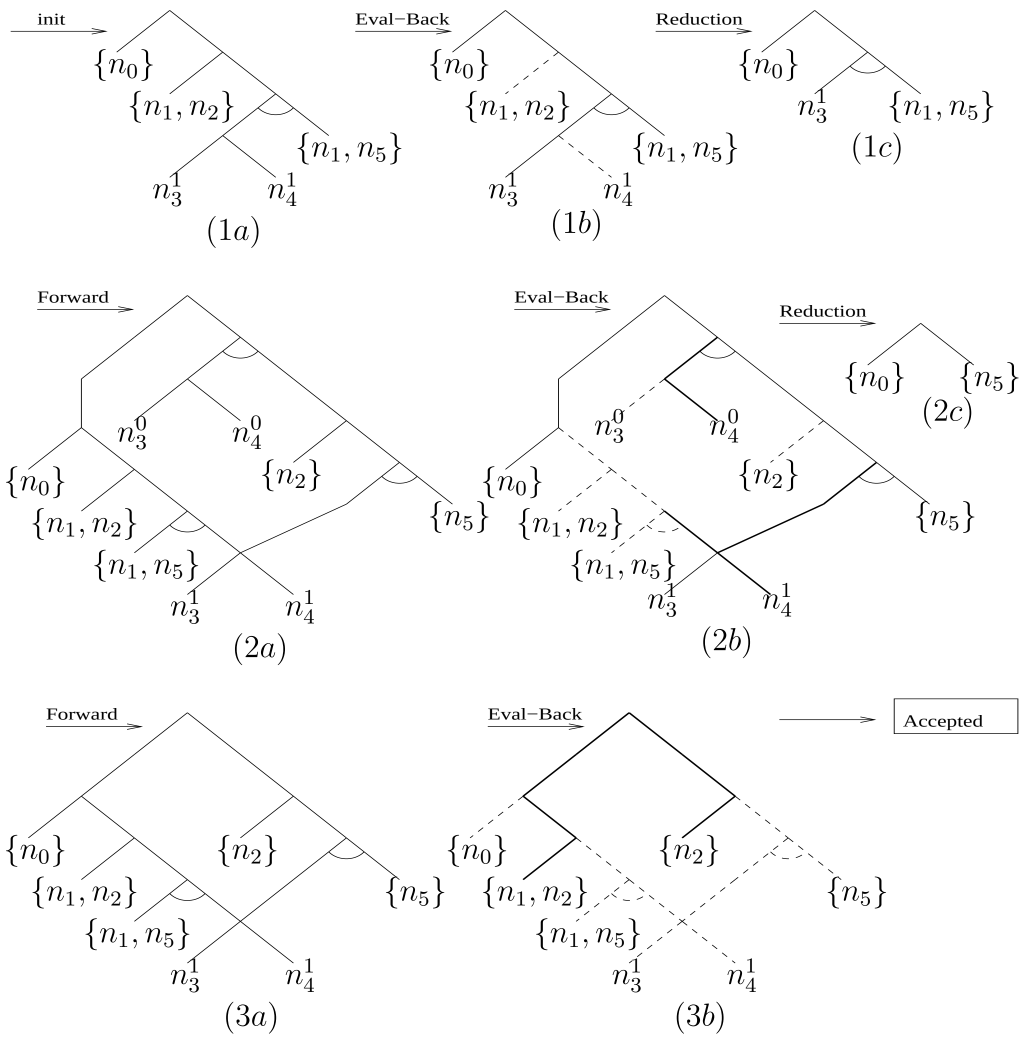

Figure 4.2: A stepwise construction of a *configuration DAG*

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).

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).
- ρ is accepted, as there exist two $\mathcal{T}$s, starting from the root of $\mathcal{G}$ 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 ρ and an *alternating automaton* $\mathcal{A}$, constructed from BTL formula φ, `GENERIC` runs in time $\mathcal{O}(|\rho| * ((M_c+2)^{2(X-Y)} + (M_c+1) * Y^2))$ and space $\mathcal{O}(X * (M_c+1)((M_c+2)^{2(X-Y)} + (M_c+1) * Y^2))$, where M_c is the largest constant appearing in φ (excluding ∞), $X = |\mathcal{A}|$ and Y is the sum of the sizes of subautomata representing FBTL subformulae in φ

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 ρ and an automaton $\mathcal{A}$, `GENERIC` $(\mathcal{A}, \rho)$ =**true**, if there exists an accepting run of ρ in $\mathcal{A}$.

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

Proof. Like the *Optimized Forward-Backward* algorithm, presented in Chapter 3, `GENERIC` also maintains a *configuration DAG* $\mathcal{G}$ during its execution. The space complexity of `GENERIC` is linear in the size of $\mathcal{G}$. At each system step, $\mathcal{G}$ 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 $|\mathcal{G}|$ at a given system step.

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

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

Every *configuration sub-DAG* generated at a given system step n is fully evaluated within $n+X*(M_c+1)$ system steps (Lemma 3.4). Lets take a set S of configurations, generated by the the function **init** at a system step n. Each $\langle C, D\rangle \in S$ is translated into a *configuration sub-DAG* $\mathcal{G}'$ = $\langle S_p, \langle\wedge, \{\langle\mathcal{G}', C, 0\rangle\}\cup\rangle, 0\rangle$, where S_p is the set of parent edges. At system step $n+1$, the procedure `Forward` replaces the leaf $\langle\mathcal{G}', C, 0\rangle \in \mathcal{G}'$ with the successor *configuration sub-DAG* $\mathcal{G}''$, generated by translating a set of successor configurations of C. At a system step $m \leq n+M_C*X$, every element $d \in D$ is fully evaluated and removed from D, resulting in $D=\emptyset$. The procedure `Eval-Back` reduces the evaluation of $\mathcal{G}$ to the evaluation of $\mathcal{G}''$ and replaces $\mathcal{G}'$ by $\mathcal{G}''$.

Thus, an increment in the size of $\mathcal{G}$ made at any system step is decremented from $\mathcal{G}$ within the next $X*(M_c+1)$ system steps. The overall size of $\mathcal{G}$ is always bounded by $X*(M_c+1)((M_c+2)^{2(X-Y)}+(M_c+1)*Y^2)$.

Hence, space complexity of `GENERIC` is $\mathcal{O}(X*(M_c+1)((M_c+2)^{2(X-Y)}+(M_c+1)*Y^2))$.

Overall running time of `GENERIC` is the increment in $|\mathcal{G}|$ at each system step times the number of system steps. Thus, the running time of `GENERIC` is $\mathcal{O}(|\rho|*((M_c+2)^{2(X-Y)}+(M_c+1)*Y^2))$

$\square$

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:

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

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:

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

Definition 4.8. Slightly-Restricted Bounded Temporal Logic Let ψ be a FBTL formula, a SBTL formula φ can be defined inductively as follows:

$$\begin{aligned} \varphi ::= \;& \psi \\ \mid\;& \varphi \wedge \varphi \\ \mid\;& \varphi \wedge \varphi \\ \mid\;& \Box_{[0,\infty]}\varphi \\ \mid\;& \Diamond_{[0,\infty]}\varphi \\ \mid\;& \varphi \mathcal{U}_{[0,\infty]}\varphi \end{aligned}$$

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

Proof. For a metric alternating automaton $\mathcal{A}$, translated from φ, the function **init** does not generate *configuration DAGs*. Each configuration $\langle C, \emptyset \rangle$ in the configuration set S, generated by **init**, is translated into a *configuration DAG* $\mathcal{G}' = \langle S_p, \langle \wedge, \{\langle \mathcal{G}', C, 0 \rangle\}\rangle, 0\rangle$ by the procedure `Generate-DAG`. In *backward evaluation*, $\mathcal{G}'$ 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, $\mathcal{G}$ 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 φ, is $\mathcal{O}(X * (M_c + 2)^X)$.

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

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

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

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

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

□

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* $\mathcal{A}$, translated from SBTL formula φ, `GENERIC` runs in time $\mathcal{O}(X * (M_c + 1) * (2^{2(X-Y)} + (M_c + 1) * Y^2))$ and space $\mathcal{O}(|\rho| * (2^{2(X-Y)} + (M_c + 1) * Y^2))$,where M_c is the largest constant appearing in $X = |\mathcal{A}|$ and Y is the sum of sizes of subautomata that corresponds to FBTL subformulae in φ .

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

In φ, we have FBTL subformulae of total size Y inside a top level LTL formula of size $X - Y$. For a *configuration DAG* $\mathcal{G}$ 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 $Y * (M_c + 1)$(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, $Y * (M_c + 1)$ leaves of type A increment the size of $\mathcal{G}$ by at most $Y^2 * (M_c + 1)$ and 2^{X-Y} leaves of type B increment the size of $\mathcal{G}$ by at most $2^{2(X-Y)}$. The total increment in the size of $\mathcal{G}$ is therefore bounded by $2^{(X-Y)} + (M_c + 1) * Y^2$.

Hence, for φ, `GENERIC` runs in space $\mathcal{O}(X * (M_c + 1) * (2^{2(X-Y)} + (M_c + 1) * Y^2))$ and in time $\mathcal{O}(|\rho| * (2^{(X-Y)} + (M_c + 1) * Y^2))$.

□

Theorem 4.7. For an execution trace ρ, and a *metric alternating* automaton $\mathcal{A}$, translated from BTL formula φ, the space complexity of `GENERIC` for different sub-logics of BTL is given below:

$\mathcal{O}(2^X)$	if φ is a LTL formula
$\mathcal{O}(X^3 * (M_c + 1)^2)$	if φ is a FBTL formula
$\mathcal{O}(X * (M_c + 2)(2^{2(X-Y)} + (M_c + 1) * Y^2))$	if φ is a SRBTL formula
$\mathcal{O}(X * (M_c + 1)((M_c + 2)^{2(X-Y)} + (M_c + 1) * Y^2))$	if φ is a BTL formula

where,

- $X = |\mathcal{A}|$.
- Y is the sum of the sizes of subautomata representing FBTL formulae.
- M_c is the maximum constant appearing in φ (excluding ∞).

Theorem 4.8. For an execution trace ρ, and a *metric alternating* automaton $\mathcal{A}$, translated from input formula φ, the time complexity of `GENERIC` for different sub-logics of BTL is given below:

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

where,

- $X = |\mathcal{A}|$.
- Y is the sum of the sizes of subautomata representing FBTL formula.
- M_c is the maximum constant appearing in φ (excluding ∞).

Chapter 5

Conclusion

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.

Bibliography

[1] Z. Manna and A. Pnueli, “Temporal Verification of Reactive Systems: Safety”, Springer-Verlag, New York, 1995.

[2] Z. Manna and A. Pnueli,“Temporal Verification of Reactive Systems: Progress”, Springer-Verlag, New York, 1996.

[3] M.Y. Vardi, “Alternating Automata and Program Verification”, in: J.Van Leeuwen, editors, Computer Science Today. Recent Trends and Developments LNCS 1000, Springer-Verlag, 1995 pp. 471-485.

[4] M.Y. Vardi, “An Automata-Theoretic Approach to Linear Temporal Logic”, in: F. Moller and G. Birtwistle, editors, Logics for Concurrency. Structure and Automata, LNCS 1043(1996), pp. 238-266.

[5] O. Kupferman and M.Y. Vardi, “Weak alternating automata are not that weak”, J.ACM , 2(3):408-429, 2001.

[6] B. Finkbeiner and H. Sipma, “Checking Finite Traces Using Alternating Automata”, in: K. Havelund and G. Rosu , editors, Runtime Verification 2001, Electronic Notes in Theoretical Computer Science 55(2001), pp 1-17.

[7] B. Finkbeiner, S. Sankaranarayanan and H. Sipma, “Collecting Statistics over Runtime Executions”, in: K. Havelund and G. Rosu , editors, Runtime Verification 2002, Electronic Notes in Theoretical Computer Science 70(2002), pp 1-17.

[8] B. Finkbeiner, “Verification Algorithms based on Alternating Automata”. PhD thesis, Stanford University, 2002.

[9] R. Alur and T.A. Henzinger, “Real-Time Logics: Complexity and Expressiveness”, In Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1990, pp. 390-401.

[10] R. Alur and T.A. Henzinger, “Logics and Models of Real Time: A Survey”, In: Real Time: Theory in Practice, Lecture Notes in Computer Science 600, Springer-Verlag, 1992, pp. 74-106.

[11] T.A. Henzinger , Z. Manna and A. Pnueli, “Temporal proof methodologies for real-time systems”, Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.353-366, January 21-23, 1991, Orlando, Florida, United States

[12] T.A. Henzinger. “The Temporal Specification and Verification of Real-Time Systems”. PhD thesis, Stanford University, 1991.

[13] R. Koymans. “Specifying Real-Time Properties with Metric Temporal Logic”. RealTime Systems, 2(4):255–299, 1990.

[14] P. Bellini , R. Mattolini and P. Nesi, “Temporal logics for real-time system specification”, ACM Computing Surveys (CSUR), v.32 n.1, p.12-42, March 2000.

[15] Z. Manna and A. Pnueli, “The Temporal Logic of Reactive and Concurrent Systems,” Springer-Verlag, New York, 1991.

[16] E.M. Clarke, O. Grumberg and D.A. Peled, “Model Checking,” The MIT Press, 1999.

[17] M. Abadi and L. Lamport, “The existence of refinement mappings”, Elsevier Science Publishers Ltd., Essex(UK), 1991.

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 $\mathcal{P}$ is instrumented with additional instructions to emits relevant events. $\mathcal{P}$,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 Υ from a text file and translates Υ to a *metric alternating automaton* $\mathcal{A}$. The translation from φ to $\mathcal{A}$ is done by the *Formula Translator* as follows:

- Parses φ to check against the syntax and semantics of the BTL.

- If Υ is syntactically a and semantically correct, *Formula Translator* produces a syntax tree $\mathcal{T}$ from Υ.

- $\mathcal{T}$ is then transformed into the *negation normal form*, such that negation are pushed to the proposition level.
- $\mathcal{T}$, in the negation normal form, is then translated into a *metric alternating automaton* $\mathcal{A}$ according to the translation rules defined in Chapter 2.
- Return the automaton $\mathcal{A}$ as output.

A.1.2 Program Instrumentation

The *instrumentation module* takes a specification script Υ and program segment $\mathcal{P}$,written in $C0$ (a subset of C langueage), and produces an instrumented code $\mathcal{P}'$ that can compiled with any standard $C0$ compiler. $\mathcal{P}'$ contains additional instructions to emit events, and also the specification script Υ' inserted, as comments, at the top of the program file. Υ' is produced after replacing each proposition in Υ with their respective index numbers. The boolean constants $true$ and $false$ are replaced with 1 and 0 respectively. Given $\mathcal{P}$ and Υ, the program instrumentation works as follows:

- Parses φ script to extract the set of predicates $\mathcal{S}_p$ that appears in φ. Predicates or boolean functions over program variables. Each Predicate $pred \in \mathcal{S}_p$ is assigned a unique integer value key and then put into a list $\mathcal{L}_p$ of pair $\langle pred, key \rangle$. Keys are assigned in increasing order staring with 2. 0 and 1 are reserved for the boolean constant $false$ and $true$ respectively.
- Each $pred \in \mathcal{S}_p$ in the specification script Υ is replaced by their respective key, and the resultant specification script Υ' is written at the top of instrumented program $\mathcal{P}'$ enclosed in the comments.
- Traverses $\mathcal{L}_p$ to compute a list $\mathcal{L}_{vp}$ of pair $\langle var, plist \rangle$, where var belongs to a set of program variables V that appears in Υ. The list $\mathcal{L}_{vp}$ maps each $var \in V$ to a set $\mathcal{S}' \subseteq \mathcal{S}_p$, such that, var appears in every element of $\mathcal{S}'$.
- 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 $var \in V$, computes the set $\mathcal{S}'$ of predicates , such that each $P \in \mathcal{S}'$ contains 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 $\mathcal{S}$ of predicates appearing in the input formula. The module, maintains a list $\mathcal{L}$ of pair $\langle id, val\rangle$, where val is a predicate's truth value and id uniquely identifies each predicate. The list $\mathcal{L}$ is computed by parsing the specification script Υ of a BTL formula . *Event Dispatcher* receives a pair $\langle id, val\rangle$, from the running program and updates $\mathcal{L}$. After updating $\mathcal{L}$, *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 Υ. The stepwise activities of *Runtime Monitor* are given below:

1. Parses the specification script Υ, and also initializes the list $\mathcal{L}$ 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 Υ 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 Υ, 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:

```
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 ::= ε
           | [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)
                  | →    (the boolean operator for implication)
                  | ↔    (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:

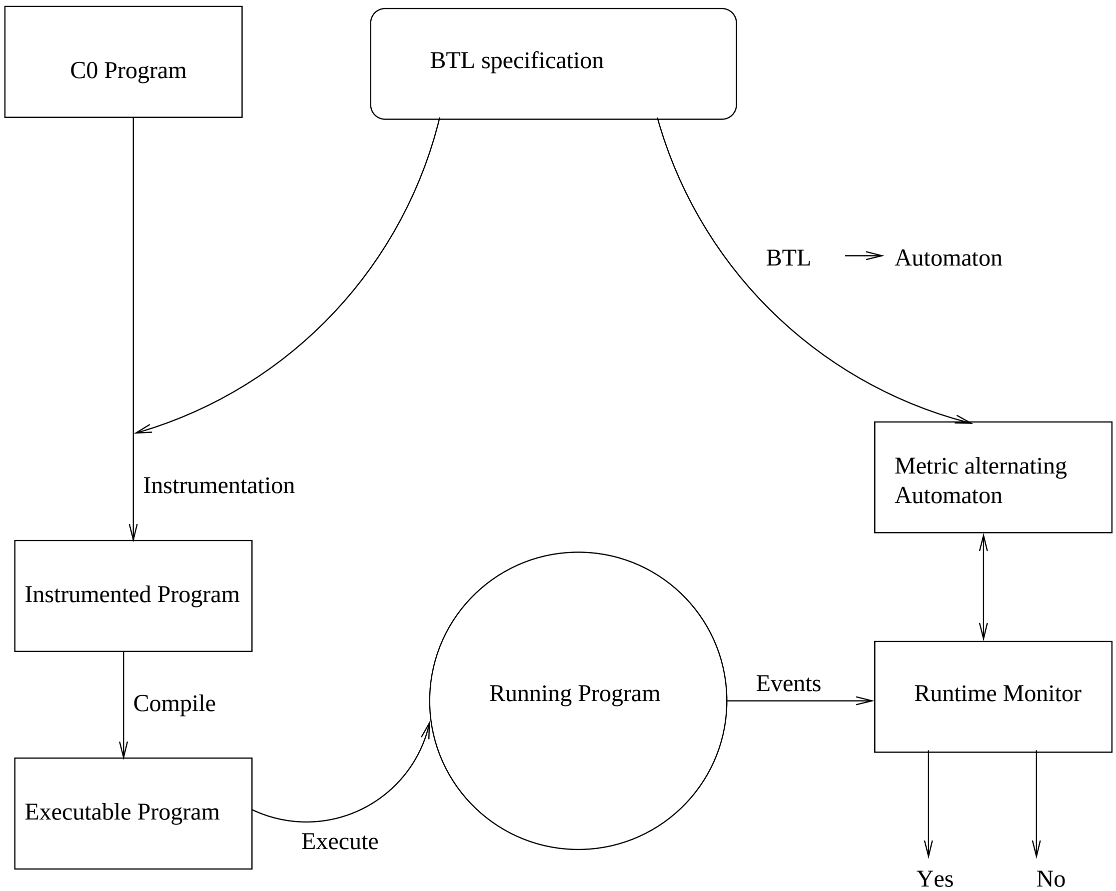

Figure A.1: The architecture of the software tool OPrA (Online Program Analyzer).

Precedence	Operators
0	!
1	G, F, X
2	and
3	or
4	xor
5	$\leftrightarrow$
6	$\rightarrow$
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 spftware takes the specification script Υ and a program $\mathcal{P}$ as inputs. Υ contains the script for the BTL specification φ, stating which system's property is to be monitored. $\mathcal{P}$ is instrumented in accordance with Υ, to produce the instrumented program $\mathcal{P}'$. At the same time φ is translated to a *metric alternating automaton* $\mathcal{A}$. $\mathcal{P}'$ is compiled with $C0$ compiler, and executed. During the execution $\mathcal{P}'$ emits events, which are received by the *Runtime Monitor*. *Runtime Monitor* checks whether a sequence of events emitted by $\mathcal{P}'$ are accepted by $\mathcal{A}$ or not.