Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Efficient Translation of Linear Temporal Logic to Deterministic Automata

Master's Thesis

by

Flavia Tego

April 2024

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Klaus Schneider
Daniel Theis

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Efficient Translation of Linear Temporal Logic to Deterministic Automata“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 29.4.2024

Flavia Tego

Acknowledgement

I would like to express my deepest gratitude to Prof. Dr. Klaus Schneider, my thesis supervisor, for the invaluable guidance, support and encouragement throughout the entire thesis. Their expertise, immense patience, and insightful feedback was fundamental for the end of this process.
Special thanks are due to my dear friends, Vishnu, Devika, Shalini, Chinmaya, Karolin, Marie and my PROAURIS colleagues who provided moral support throughout this journey.
Lastly but not least, I would like to thank my family for their encouragement and understanding, even when both were tested multiple times, for being with me on this endeavor.

Abstract

The development of computer system in our daily lives has brought to attention the need of more complex systems, which interact continuously with different stimuli from the environment. However, the user still expects a specific and defined behavior from the system. Considering reactive synthesis procedure, we can construct a system based on a high-level specification. The specification not only should describe the behavior of the system regarding its inputs and outputs, but also take into account the temporal relationships, in other words what happens to the system over time.

The temporal behavior can be specified by using various specification logics, among which we will focus on linear temporal logics (LTL). LTL is a temporal logic which is quite popular for verification and specification of reactive systems. In order to facilitate the verification problem, LTL formulas have to be translated to a nondeterministic automata, which typically is a Büchi automata. While nondeterministic automata has found widespread application, there are cases where a deterministic automata is needed. To this end, algorithms which translate a nondeterministic automata to deterministic automata are in demand. Regrettably, known determinization procedures such as Safra's procedure are difficult to implement or the implementation is able to handle only small examples.

Based on the findings of the paper 'Generating Deterministic ω-Automata for most LTL Formulas by the Breakpoint Construction', Morgenstern, Schneider, and Lamberti propose a method for generating deterministic ω-automata for most LTL formulas. Experiments of the translation of different LTL formulas have been evaluated to see the time efficiency of the proposed method and how does it differ from the explicit enumerative determinization methods.

Contents

1 Introduction

1.1 General Problem Setting

Over the past few decades, the impact of computer systems on our daily lives has steadily increased. The demand of such systems is needed, not only in known fields such as healthcare systems, supply chain management or energy grid optimization, but also in new areas such as autonomous vehicles, cybersecurity or natural language understanding.
However, we are faced with the problem of having complex systems which interact continuously with the environment, reacting to different stimuli and expected to produce a specific behaviour. Since the goal is to create a system that meets the behavioral requirements defined by the user, what comes to mind is reactive synthesis. Reactive synthesis [KC87] refers to a computational procedure employed in computer science and formal methods to autonomously construct a system based on a high-level specification. The synthesized system can be often referred to as "implementation", while the high-level specification is commonly expressed in a formal language. The specification should describe the desired behavior of the system regarding its inputs, outputs and their temporal relationships.
To specify the temporal behavior of systems, various specification logics, such as μ-calculus [Koz83], ω-automata and temporal logics have been proposed (see [SST04] for an overview). In order to ensure the system meets certain behavioral properties that are specified, various techniques and tools, such as model checking [CGP94; HR04], have been proposed. A widely used technique in formal verification is linear temporal logic (LTL) model checking due to its expressiveness, formalism, automation, scalability, and tool support.
Various algorithms have been introduced to generate automata from given specification languages [Büc60; CE82; VW86], usually generating *nondeterministic* automata. However, there are cases where we want to simplify the analysis and verification processes, which require the adoption of *deterministic* automata instead of *nondeterministic* automata. Therefore, the necessity arises for algorithms, that convert the equivalent *nondeterministic* automata attained from the temporal logic specification, into *deterministic* automata.
In many applications, precise control and predictability is desirable, hence *deterministic* automata are more preferable. They often require less memory and computational resources compared to their *nondeterministic* counterparts leading to faster execution times. Even though, they may become significantly larger than their equivalent *nondeterministic* automaton, *deterministic* automata are usually easier to understand.
As previously mentioned, we want to verify if a system satisfies a property,

such as safety, liveness or persistence. Hence, taking into consideration the properties, an automaton can have different acceptance conditions, depending on the way a word is accepted by it.
Various acceptance criteria necessitate the creation of distinct determinization procedures, like the subset construction [RS59] or breakpoint construction [MH84]. Nonetheless, these algorithms differ in their complexity. Even the simplest acceptance condition [AL04] can have an exponential worst-case complexity and the most complex ones may even become doubly exponential $O(2^{2^{|n|}})$ even though this is generally not necessary.

1.2 New Contributions

Automata, whether operating on finite or infinite words, have been extensively examined. Particularly, automata dealing with infinite words, known as ω-automata, play a crucial role in reactive systems [SST04]. One example includes the task of model checking the temporal logic LTL [Pnu77]. The given problem can scale down to checking the nonemptiness of ω-automata: To check if a system $\mathcal{M}$ satisfies a property, in our case a LTL property φ, firstly we translate the negation of φ ($\neg\varphi$) to an equivalent *nondeterministic* ω-automaton $\mathcal{A}_{\neg\varphi}$ so we can continue on the second step of checking the emptiness of the product $\mathcal{M} \times \mathcal{A}_{\neg\varphi}$.
For model checking, an important technique to keep in mind is symbolic model checking [Bur+92]. Its aim is to verify the properties of a system by representing its state space symbolically. Instead of exploring individual states, we can now operate on symbolic representation of sets of states. The system model $\mathcal{M}$ and the property φ can be expressed using propositional logic (Boolean expressions), often in the form of binary decision diagrams (BDDs) or other symbolic data structures. The model checker will perform operations on the symbolically represented sets of states to check if the property φ holds for all states or if there exists a counterexample.
Leveraging symbolic set representations, algorithms that translate LTL formulas into *symbolically* represented ω-automata have been introduced [CGH97]. As mentioned in 1.1, in practice, these algorithms yield generally a *nondeterministic* Büchi automaton. Afterwards, to achieve a *deterministic* automata, the Safra [Saf92] determinization procedure is applied to get a *deterministic* (Rabin) automata. As Safra's algorithm poses limitations in its implementation [KB06] and the inherent data structures are not suitable for the utilization of symbolic set representations, the available tools are subsequently constrained to handling small LTL formulas, limiting their practical use.
The thesis will focus on how to replace the Safra's determinization procedure to simpler ones, more amenable to symbolic representation. Namely, we will concentrate on subset construction and breakpoint construction, which can be used to determinize automata with different acceptance conditions, such as safety, liveness and co-Büchi. The subset construction can generate a deterministic automaton with $O(2^n)$ many states, while the breakpoint construction can scale up to $O(3^n)$ many states, thus it is crucial to create more efficient

implementations of the aforementioned algorithms.
We want to make use of an implementation to generate a symbolic encoding for the deterministic automata where for given n state variables $q_1, \ldots, q_n$, we can possibly encode 2^n states. For each state variable q_i, the transition relation has a transition equation of the form $\mathsf{next}(q_i) \leftrightarrow \Phi(q_1, \ldots, q_n, x_1, \ldots, x_m)$. In this way, we can determine its value at the next point of time based on the current values of the state variables $q_1, \ldots, q_n$ and the input variables $x_1, \ldots, x_m$.
The main point of the thesis is on the implementation of the algorithms published in [MSL08] and their evaluation.

1.3 Outline of the Thesis

We want to start the outline by having an understanding of the basic theoratical knowledge in Section 2. The section offers an introduction to the data structures of binary decision diagrams which are important for the symbolic representation of ω-automata. The connection between BDDs and ω-automata is further addressed in 2.3.4. Following, we talk about the representation of the ω-automata and the different acceptance conditions. In 2.4, we describe the main determinization procedures of subset and breakpoint construction, continuing with other known determinization algorithms and an overview of the symbolic determinization procedures. Chapter 3 covers the Averest framework used in the implementation phase, transitioning on the implementation of the symbolic determinization algorithms and the experimental results achieved. To conclude, in chapter 4, we include a summary of the main findings and contributions of the research.

2 Preliminaries

In this chapter, we will define a part of the theoretical background of the thesis. We start off with a brief introduction to binary decision diagrams (BDDs), the basis data structure for the symbolic representation further discussed in sections 2.3.4 and 2.4.4. The BDD section (2.1) includes information about the structure of the BDDs, as well as an example, and what operations we can apply to them, which are useful later in the implementation phase. Afterwards, a short overview of linear temporal logic (2.2), containing explanation about the LTL syntax and the various temporal operators and how we can express a temporal operator using another temporal operator, which can be of assistance in translating to different temporal logic classes mentioned in 2.3.3. Following is the ω-automata section (2.3), where we specify the ω-automata representation, the different acceptance conditions, utilizing the ω-automata and temporal logic hierarchy for translation from temporal logic formula to nondeterministic automata and lastly, how to use BDDs for a symbolic representation. Section 2.4 dives into the different determinization procedures, focusing on the two main procedures used in this thesis (Subset Construction and Breakpoint Construction) and following with further algorithms. We conclude the section by outlining the symbolic determinization procedures.

2.1 Binary Decision Diagrams (BDDs)

BDD [Bry86] is the abbreviation for binary decision diagram. They are data structures used in computer science for efficiently representing boolean expressions. BDDs are especially useful for tasks such as formal verification of digital circuits and symbolic model checking. The structure of a BDD consists of a single root node v_0 and we have two leaf nodes, L_0 and L_1. Each node, $v \notin L_0, L_1$ has two successors, also known as cofactors, $high(v)$ and $low(v)$.
Each vertex (node) $v \notin L_0, L_1$ has a label $label(v)$, where $label(L_0) \coloneqq 0$ and $label(L_1) \coloneqq 1$. Edges will respect a particular variable ordering.
Fig. 2.1 represents an example of a BDD, where we start with the root node with $label(v_r) \coloneqq \mathsf{d}$. We can observe, that the BDD has a specific variable ordering of $a \prec b \prec c \prec d$, meaning that d has a higher variable ordering than c, hence it represents also the root node, c has a higher variable ordering than b and so on. Another aspect to keep in mind for visualizing a BDD, is that often, we depict the $low(v)$ with a dashed line, while $high(v)$ is shown with a solid line.

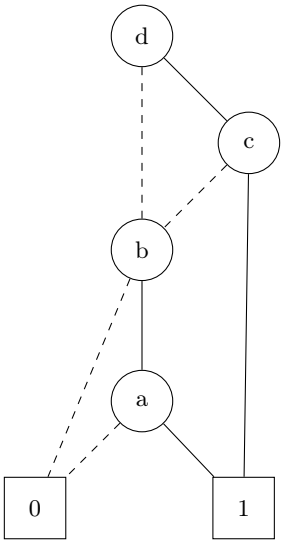


Figure 2.1: *Example of a BDD*

2.1.1 Operations on BDDs

BDDs support several algorithms [Bry86; MT96] for the manipulation of boolean expressions, making them efficient for symbolic representation. Some of the following algorithms are used for the implementation of the determinization procedures. Important algorithms include:

Apply Algorithm

Using Apply algorithm, we can do recursive computation of Boolean operations, such as conjunction, disjunction or implication, on BDDs.

Algorithm 1 Implementation of Apply Algorithm

```
function APPLY(⊙, BddNode a, b)
    int m; BddNode h, l;
    if isLeaf(a)&isLead(b) then
        return Eval(⊙, label(a), label(b));
    end if
    else
        m := max {D(label(a)), D(label(b))};
        (a0, a1) := Ops(a, m);
        (b0, b1) := Ops(b, m);
        h := Apply(⊙, a1, b1);
        l := Apply(⊙, a0, b0);
    return CreateNode(m,h,l);
end function
```

The *Ops* algorithm is implemented in Algorithm 2.

Compose Algorithm

Compose Algorithm is used to substitute all the occurrences of a variable in BDD1 with BDD2.

Algorithm 2 Implementation of Ops Algorithm

```
function OPS(v,m)
    x := label(v);
    if m = D(x) then
        return low(v), high(v);
    end if
    else return (v,v);
end function
```

Algorithm 3 Implementation of Compose Algorithm

```
function COMPOSE(x, BddNode α, φ)
    int m; BddNode h,l;
    if D(x) > D(label(φ)) then
        return φ;
    else if x = label(φ) then
        return ITE(α, high(φ), low(φ));
    else
        m := max {D(label(φ)), D(label(α))};
        (α0, α1) := Ops(α, m);
        (φ0, φ1) := Ops(φ, m);
        h := Compose(x, α1, φ1);
        l := Apply(⊙, α0, φ0);
        return CreateNode(m,h,l);
    end if
end function
```

The implementation for ITE algorithm is in Algorithm 4.

Constrain and Restrict Algorithm

Constrain and Restrict are the two algorithms that are supposed to shrink the BDDs for the formulas that are partially defined. In contrast to Constrain, the BDD computed by Restrict will never depend on new variables.

Boolean Quantification

Boolean Quantification includes Existential Quantification algorithm (disjunction of cofactors) and Universal Quantification algorithm (conjunction of cofactors). In a more general description, boolean quantification means removing one or more variables from a BDD by existential or universal quantification, depending on the result we want to achieve.

Algorithm 4 Implementation of ITE Algorithm

```
function ITE(BddNode i, j, k)
    int m; BddNode h,l;
    if i = BDD(0) then
        return k;
    else if i = BDD(1) then
        return j;
    else if j = k then
        return j;
    else
        m := max {D(label(i)), D(label(j)), D(label(k))};
        (i0, i1) := Ops(i, m);
        (j0, j1) := Ops(j, m);
        (k0, k1) := Ops(k, m);
        l := ITE(i0, j0, k0);
        h := ITE(i1, j1, k1);
        return CreateNode(m,h,l);
    end if
end function
```

Algorithm 5 Implementation of Exists Algorithm

```
function EXISTS(BddNode e, φ)
    BddNode h,l;
    if isLeaf(φ) ∨ isLeaf(e) then
        return φ;
    else if D(label(e)) > D(label(φ))) then
        return Exists(high(e), φ);
    else if label(e) = label(φ) then
        h := Exists(high(e), high(φ));
        l := Exists(high(e), low(φ));
        return Apply(∨, l, h);
    else
        h := Exists(e, high(φ));
        l := Exists(e, low(φ));
        return CreateNode(label(φ),h,l);
    end if
end function
```

2.2 Linear Temporal Logic

Temporal logic is essential if we want to reason and make specifications about reactive systems. To express temporal formulas, we have to keep in mind propositional logic and extend it by temporal operators, which talk about steps in the transition systems, and path quantifiers which quantify over the

existence of paths. Among the diverse temporal logics [CE82; EH86; SS97], our focus is on Linear Temporal Logic (LTL) [Pnu77].
The formulas of a logic consist of the set of available variables and operators. Operators can be classified in boolean operators of negation, conjunction and disjunction, as well as future and past temporal operators.
Given a set of boolean variables V_Σ, we can define a set of LTL formulas as follows [MS10]: $\varphi \coloneqq V_\Sigma \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid [\varphi \,\underline{\mathsf{U}}\, \varphi] \mid \overleftarrow{\mathsf{X}}\varphi \mid [\varphi \,\underline{\overleftarrow{\mathsf{U}}}\, \varphi]$.
As considered previously, the syntax of LTL formulas consists of future and past temporal operators, with a weak and strong variant. Apart from the operators in the definition, future operators include $[\phi \,\mathsf{U}\, \psi]$ (*weak until*), $[\phi \,\underline{\mathsf{B}}\, \psi]$ (*strong before*), $[\phi \,\mathsf{B}\, \psi]$ (*weak before*), $[\phi \,\underline{\mathsf{W}}\, \psi]$ (*strong when*) and $[\phi \,\mathsf{W}\, \psi]$ (*weak when*). To understand the difference between the strong and week operator, we define an example for the *before* operator. $[\phi \,\underline{\mathsf{B}}\, \psi]$ means that ϕ must be true before ψ becomes true. So ϕ must be true in some time in future and once ψ becomes true, we give up. ϕ weak before ψ ($[\phi \,\mathsf{B}\, \psi]$), means almost the same as strong before but here ϕ doesn't need to be true in future but ψ must be always false. So if both ϕ and ψ are always false, then $[\phi \,\mathsf{B}\, \psi]$ is true. We can explicitly write the previous definition with temporal operators as $[\phi \,\mathsf{B}\, \psi] = [\phi \,\underline{\mathsf{B}}\, \psi] \vee \mathsf{G}\neg\psi$. If eventually ϕ holds, then $[\phi \,\underline{\mathsf{B}}\, \psi]$ and $[\phi \,\mathsf{B}\, \psi]$ are equivalent, otherwise $[\phi \,\underline{\mathsf{B}}\, \psi]$ is false and $[\phi \,\mathsf{B}\, \psi]$ is equivalent to $\mathsf{G}\neg\psi$. As seen from the definition, the underlined operators are the strong operators while the weak operators aren't. Apart from the discussed operators, there are also the corresponding past temporal operators, defined in the same way as the future operator with the difference being the reversed time flow. In practice, we may need to express one operator in terms of other operators as follows [Mor10]:

$\mathsf{G}\varphi = [0 \,\mathsf{B}\, \neg\varphi]$	$\overleftarrow{\mathsf{G}}\varphi = [0 \,\underline{\overleftarrow{\mathsf{B}}}\, \neg\varphi]$
$\mathsf{F}\varphi = [1 \,\underline{\mathsf{U}}\, \varphi]$	$\overleftarrow{\mathsf{F}}\varphi = [1 \,\underline{\overleftarrow{\mathsf{U}}}\, \varphi]$
$[\varphi \,\mathsf{B}\, \psi] = \neg\,[\neg\varphi \,\underline{\mathsf{U}}\, \psi]$	$[\varphi \,\underline{\overleftarrow{\mathsf{B}}}\, \psi] = \neg[\neg\varphi \,\underline{\overleftarrow{\mathsf{U}}}\, \psi]$
$[\varphi \,\mathsf{U}\, \psi] = [\psi \,\mathsf{B}\, (\neg\varphi \wedge \neg\psi)]$	$[\varphi \,\underline{\overleftarrow{\mathsf{U}}}\, \psi] = [\psi \,\underline{\overleftarrow{\mathsf{B}}}\, (\neg\varphi \wedge \neg\psi)]$
$[\varphi \,\underline{\mathsf{B}}\, \psi] = [\neg\psi \,\underline{\mathsf{U}}\, (\varphi \wedge \neg\psi)]$	$[\varphi \,\underline{\overleftarrow{\mathsf{B}}}\, \psi] = [\neg\psi \,\underline{\overleftarrow{\mathsf{U}}}\, (\varphi \wedge \neg\psi)]$
$[\varphi \,\mathsf{W}\, \psi] = [(\varphi \wedge \psi) \,\mathsf{B}\, (\neg\varphi \wedge \psi)]$	$[\varphi \,\underline{\overleftarrow{\mathsf{W}}}\, \psi] = [(\varphi \wedge \psi) \,\underline{\overleftarrow{\mathsf{B}}}\, (\neg\varphi \wedge \psi)]$
$[\varphi \,\underline{\mathsf{W}}\, \psi] = [\neg\psi \,\underline{\mathsf{U}}\, (\varphi \wedge \psi)]$	$[\varphi \,\underline{\overleftarrow{\mathsf{W}}}\, \psi] = [\neg\psi \,\underline{\overleftarrow{\mathsf{U}}}\, (\varphi \wedge \psi)]$

Regarding the semantics definition of an LTL formula φ, we have to consider if the formula holds in a path of a structure (i.e a Kripke Structure).

2.3 ω-Automata

2.3.1 ω-Automata representation

Definition 1. A *nondeterministic ω-automaton* is a tuple $\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$, where Σ is a finite alphabet, $\mathcal{Q}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states, $\mathcal{R} \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation and $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states.
Following the previous definition, an automaton is *deterministic* if there exists

only one initial state and for all $q \in \mathcal{Q}$ and $\sigma \in \Sigma$ with $(q, \sigma, q') \in \mathcal{R}$ is uniquely defined.
A word is accepted by the automaton depending on the set of runs. According to [MS08], a run ξ is defined as:
Definition 2. A run ξ of $\mathcal{A}$ on an infinite word $\alpha = \alpha^{(0)}\alpha^{(1)} \in \Sigma^\omega$ is an infinite sequence of states $\xi = \xi^{(0)}\xi^{(1)}... \in \mathcal{Q}^\omega$ where $\xi^{(0)} \in \mathcal{I}$ and $\forall i \geq 0$, $\xi^{(i+1)} \in \delta(\xi^{(i)}, \alpha^{(i)})$. For a run $\xi = \xi^{(0)}\xi^{(1)}...$, we consider $\inf \xi := \{q \in \mathcal{Q} \mid \big|\{i \in \mathbb{N} \mid q = \xi(i)\}\big| = \infty\}$ to be the set of all states that occur infinitely often on the run.
After considering the definition of a run, an automaton $\mathcal{A}$ is *deterministic*, if every word $\alpha \in \Sigma^\omega$ has exactly one run ξ through $\mathcal{A}$.

2.3.2 Acceptance Conditions

Depending on the way a word is accepted by the automaton, different kinds of acceptance conditions $\mathcal{F}$ have been proposed and studied [Wag79; Tho90; CMP92; SST04]:

- **Safety Condition:** $G\varphi$
- **Liveness Condition:** $F\varphi$
- **Büchi Condition:** $GF\varphi$
- **Persistence (Co-Büchi) Condition:** $FG\varphi$

The mentioned acceptance conditions were formalized as LTL-formulas, hence, the following definitions hold:

- A run is accepted by a safety condition $G\varphi$ with a state set φ if the run exclusively runs through the set φ.
- A run is accepted by a liveness condition $F\varphi$ with a state set φ if the run visits at least one state of the set φ at least once.
- A run is accepted by a Büchi condition $GF\varphi$ with a state set φ if the run visits at least one state of the set φ infinitely often.
- A run is accepted by a Co-Büchi condition $FG\varphi$ with a state set φ if the run visits only states of the set φ infinitely often.

Other special acceptance conditions include:

- **Rabin Condition** [Rab72]**:** $\bigvee_{j=0}^{f} GF\varphi_j \wedge FG\psi_j$
- **Strett Condition** [Str82]**:** $\bigwedge_{j=0}^{f} FG\varphi_j \vee GF\psi_j$
- **Prefix Conditions** [SW74]**:** $\bigwedge_{j=0}^{f} G\varphi_j \vee F\psi_j$

2.3.3 ω-Automata Hierarchy and LTL Hierarchy

To achieve an equivalent *deterministic* automaton from a given *nondeterministic* automaton with acceptance conditions mentioned in 2.3.2, it is worth keeping in mind the hierarchy of different ω-automata. The acceptance conditions $G\varphi$, $F\varphi$, $GF\varphi$ and $FG\varphi$, define respectively the following automaton classes $(\mathsf{N})\mathsf{Det}_G$, $(\mathsf{N})\mathsf{Det}_F$, $(\mathsf{N})\mathsf{Det}_{GF}$ and $(\mathsf{N})\mathsf{Det}_{FG}$. $(\mathsf{N})\mathsf{Det}_{\mathsf{Prefix}}$ automata have acceptance conditions of the form $\bigwedge_{j=0}^{f} G\varphi_j \vee F\psi_j$, while $(\mathsf{N})\mathsf{Det}_{\mathsf{Strett}}$ have acceptance conditions as $\bigwedge_{j=0}^{f} FG\varphi_j \vee GF\psi_j$.
The figure 2.2 illustrates the expressiveness of the automata classes. $\mathcal{C}_1 \precsim \mathcal{C}_2$ means that for any automaton in class $\mathcal{C}_1$, there is an equivalent one in class $\mathcal{C}_2$. We can define $\mathcal{C}_1 \approx \mathcal{C}_2 \coloneqq \mathcal{C}_1 \precsim \mathcal{C}_2 \wedge \mathcal{C}_2 \precsim \mathcal{C}_1$ and $\mathcal{C}_1 \precnsim \mathcal{C}_2 \coloneqq \mathcal{C}_1 \precsim \mathcal{C}_2 \wedge \neg(\mathcal{C}_2 \approx \mathcal{C}_1)$ [MSL08]. Furthermore, we can observe that the ω-automata hierarchy is expressed by six classes, where each class has a deterministic class representative.

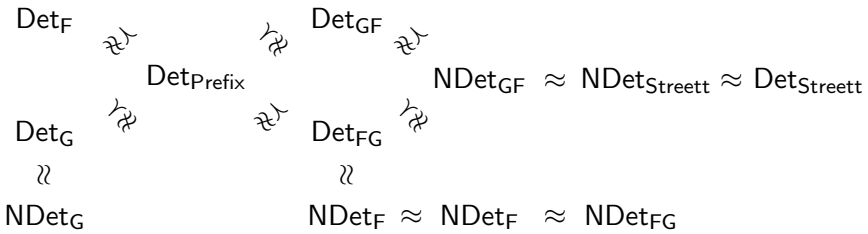


Figure 2.2: *(Borel) Hierarchy of ω-Automata [MSL08]*

Considering the ω-automata hierarchy, a temporal logic hierarchy [CMP92; Sch01; SST04] was defined, consisting of six temporal logics $\mathsf{TL}_\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{GF}, \mathsf{FG}, \mathsf{Streett}\}$, which correspond to the six automaton classes $(\mathsf{N})\mathsf{Det}_\kappa$. A representation of the temporal logic hierarchy is shown in fig. 2.3. The same definition for the classes from the ω-automata hierarchy, applies also here. The hierarchy of the temporal logics can be defined syntactically by the grammar rules in table 2.1. In case, there are formulas which don't belong to any of the six classes, we can try to rewrite them into an equivalent formula which will belong to one of the classes.
Using Temporal Logic and Automa hierarchy, we can translate formulas from $\mathsf{TL}\kappa$ to equivalent $(\mathsf{N})\mathsf{Det}_\kappa$ automata, based on the results of [SST04]. Moreover, according to the findings in [SST04], automata derived from the classes TL_G and TL_F can be determinized using the Rabin-Scott construction, while automata derived from the translation of TL_FG and TL_GF formulas can be determinized using the Miyano-Hayashi construction. $\mathsf{TL}_{\mathsf{Prefix}}$ is expressed as $\mathsf{TL}_\mathsf{G} \cup \mathsf{TL}_\mathsf{F}$ and $\mathsf{TL}_{\mathsf{Streett}}$ as $\mathsf{TL}_\mathsf{FG} \cup \mathsf{TL}_\mathsf{GF}$, hence the deterministic automaton achieved is a boolean combination of $\mathsf{Det}_\mathsf{G} \backslash \mathsf{Det}_\mathsf{F}$ for translation of $\mathsf{TL}_{\mathsf{Prefix}}$ formulas and boolean combination of $\mathsf{Det}_\mathsf{GF} \backslash \mathsf{Det}_\mathsf{FG}$ for $\mathsf{TL}_{\mathsf{Streett}}$.

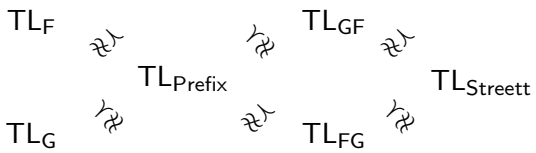


Figure 2.3: *Hierarchy of temporal logic*

$P_{\mathsf{G}} ::= \mathsf{V}_{\Sigma} \mid \neg P_{\mathsf{F}} \mid P_{\mathsf{G}} \wedge P_{\mathsf{G}} \mid P_{\mathsf{G}} \vee P_{\mathsf{G}} \mid \mathsf{X}P_{\mathsf{G}} \mid \mathsf{G}P_{\mathsf{G}} \mid [P_{\mathsf{G}}\ \mathsf{U}\ \mathsf{P_G}]$	$P_{\mathsf{F}} ::= \mathsf{V}_{\Sigma} \mid \neg P_{\mathsf{G}} \mid P_{\mathsf{F}} \wedge P_{\mathsf{F}} \mid P_{\mathsf{F}} \vee P_{\mathsf{F}} \mid \mathsf{X}P_{\mathsf{F}} \mid \mathsf{F}P_{\mathsf{F}} \mid [P_{\mathsf{F}}\ \underline{\mathsf{U}}\ P_{\mathsf{F}}]$
$P_{\mathsf{Prefix}} ::= P_{\mathsf{G}} \mid P_{\mathsf{F}} \mid \neg P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \wedge P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \vee P_{\mathsf{Prefix}}$	
$P_{\mathsf{GF}} ::= P_{\mathsf{Prefix}} \mid \neg P_{\mathsf{FG}} \mid P_{\mathsf{GF}} \wedge P_{\mathsf{GF}} \mid P_{\mathsf{GF}} \vee P_{\mathsf{GF}} \mid \mathsf{X}P_{\mathsf{GF}} \mid \mathsf{G}P_{\mathsf{GF}} \mid [P_{\mathsf{GF}}\ \mathsf{U}\ \mathsf{P_{GF}}] \mid [\mathsf{P_{GF}}\ \underline{\mathsf{U}}\ \mathsf{P_F}]$	$P_{\mathsf{FG}} ::= P_{\mathsf{Prefix}} \mid \neg P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \wedge P_{\mathsf{FG}} \mid P_{\mathsf{FG}} \vee P_{\mathsf{FG}} \mid \mathsf{X}P_{\mathsf{FG}} \mid \mathsf{F}P_{\mathsf{F}} \mid [P_{\mathsf{FG}}\ \underline{\mathsf{U}}\ P_{\mathsf{FG}}] \mid [P_{\mathsf{G}}\ \mathsf{U}\ \mathsf{P_{FG}}]$
$P_{\mathsf{Streett}} ::= P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \mid \neg P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \wedge P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \vee P_{\mathsf{Streett}}$	

Table 2.1: *Syntactic Characterizations of Temporal Logic Hierarchy [MS10]*

2.3.4 Representation of ω-automata symbolically

Using BDDs structures from section 2.1, we can express an ω-automata on infinite words. Considering the definition of an automata from 2.3.1, we assume the finite alphabet Σ is encoded by boolean variables V_{Σ}. The finite set of states $\mathcal{Q}$ is also encoded by a state set of boolean variables. To represent the transition relation, apart from the state set $\mathcal{Q}$ which shows the current point of time, we introduce another state set of boolean variables to show the next point of time, i.e the transition from the current state to the next state. Taking into consideration the propositional (boolean) variables mentioned, we can encode the initial states, the transition relation and the acceptance condition. Afterwards, we can use the BDDs for encoding the propositional formulas.
As it was stated, using boolean encodings, an automata can be defined by propositional formulas. Formulas $\varphi_{\mathcal{I}}$, $\varphi_{\mathcal{R}}$, $\varphi_{\mathcal{F}}$, and the set $\mathcal{V}_{state}$, representing respectively the initial states, the transition relation, the acceptance condition formulas and the set of state variables, completely describes an automaton. Therefore, we can use *automaton formulas* $\mathcal{A}_{\exists}(\mathcal{V}_{state}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{R}}, \varphi_{\mathcal{F}})$, where

- $\mathcal{V}_{state}$ is the set of state variables
- $\varphi_{\mathcal{I}}$ and $\varphi_{\mathcal{F}}$ are propositional formulas over $\mathcal{V} \cup \mathcal{V}_{state}$
- $\varphi_{\mathcal{R}}$ is a propositional formula $\varphi_{\mathcal{R}}$ with variables $\mathcal{V}_{state} \cup \mathcal{V}_{in} \cup \{q' \mid q \in \mathcal{V}_{state}\}$

Additionally, automaton formulas allow boolean combinations and nesting. A

specification logic $\mathcal{L}_\omega$ defines the automaton formulas. The syntax of automata logic $\mathcal{L}_\omega$ over variables $\mathcal{V}$ is stated as follows

- $\mathcal{V} \subseteq \mathcal{L}_\omega$
- $\neg\varphi,\ \varphi \wedge \psi,\ \varphi \vee \psi \in \mathcal{L}_\omega$, provided that $\varphi,\ \psi \in \mathcal{L}_\omega$
- $\mathsf{G}\varphi,\ \mathsf{F}\varphi \in \mathcal{L}_\omega$, provided that $\varphi,\ \psi \in \mathcal{L}_\omega$
- $\mathcal{A}_\Theta\ (\mathcal{V}_{state},\ \varphi_\mathcal{I},\ \varphi_\mathcal{R},\ \varphi_\mathcal{F})\ \in\ \mathcal{L}_\omega$ with
  - $\Theta\ \in\ \{\exists,\ \forall\}$
  - $\mathcal{V}_{state} \cap \mathcal{V} = \{\}$ is the set of state variables
  - $\varphi_\mathcal{I}$ is a propositional formula over $\mathcal{V} \cup \mathcal{V}_{state}$
  - $\varphi_\mathcal{R}$ is a propositional formula $\varphi_\mathcal{R}$ with variables $\mathcal{V}_{state}\ \cup\ \mathcal{V}_{in}\ \cup\ \{q' \mid q \in \mathcal{V}_{state}\}$
  - $\varphi_\mathcal{F} \in \mathcal{L}_\omega$

Automata logic $\mathcal{L}_\omega$ considers paths, especially paths of Kripke structures. Kripke structures [Kri63] provide a formal framework for representing all the possible states and transitions of a system. The states and transitions between states are encoded using propositional logic. Apart from propositional logic, to illustrate the behavior of a system, we can use also temporal operators such as *next* X, *until* U or *eventually* F. Now we can also reason about the temporal properties of a sequence of states. The Kripke structure can be subsequently defined as $(\mathcal{K}, \pi, t) \models \varphi$, where

- $\mathcal{K}$ is the Kripke structure with states $\mathcal{S}$
- $\pi : \mathbb{N} \to \mathcal{S}$ is a path through structure $\mathcal{K}$
- $t \in \mathbb{N}$ is a position on the path
- $\varphi \in \mathcal{L}_\omega$

Different acceptance conditions will yield different kinds of automata according to the temporal logic and ω-automata hierarchy. We may have temporal formulas that consist of combinations of temporal formulas with boolean operators or nested temporal logic formulas. As a result, from the translation of temporal logic formulas to nondeterministic automata, the derived automata can follow the same behavior. As a result, we have to be aware of the boolean operations of ω-automata and the nesting of ω-automata:

- complement
$$\neg\mathcal{A}_\forall(Q, \mathcal{I}, \mathcal{R}, \mathcal{F}) = \mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \neg\mathcal{F})$$
$$\neg\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F}) = \mathcal{A}_\forall(Q, \mathcal{I}, \mathcal{R}, \neg\mathcal{F})$$
- conjunction
$$(\mathcal{A}_\exists(Q_1, \mathcal{I}_1, \mathcal{R}_1, \mathcal{F}_1) \wedge \mathcal{A}_\exists(Q_2, \mathcal{I}_2, \mathcal{R}_2, \mathcal{F}_2))$$
$$= \mathcal{A}_\exists(Q_1 \cup Q_2, \mathcal{I}_1 \wedge \mathcal{I}_2, \mathcal{R}_1 \wedge \mathcal{I}_2, \mathcal{F}_1 \wedge \mathcal{F}_2)$$

- disjunction: With a new variable q, we have the following:

$$\begin{aligned}&(\mathcal{A}_\exists(Q_1,\mathcal{I}_1,\mathcal{R}_1,\mathcal{F}_1)\vee\mathcal{A}_\exists(Q_2,\mathcal{I}_2,\mathcal{R}_2,\mathcal{F}_2))\\&=\mathcal{A}_\exists\begin{pmatrix}Q_1\cup Q_2\cup\{q\},\\\neg q\wedge\mathcal{I}_1\vee q\wedge\mathcal{I}_2,\\\neg q\wedge\mathcal{R}_1\wedge\neg q'\vee q\wedge\mathcal{R}_2\wedge q',\\(\neg q\wedge\mathcal{F}_1)\vee(q\wedge\mathcal{F}_2)\end{pmatrix}\end{aligned}$$

- reducing propositional formulas to automaton formulas

$$\begin{aligned}\varphi&=\mathcal{A}_\exists(\{\},1,1,\varphi)\\\varphi&=\mathcal{A}_\forall(\{\},1,1,\varphi)\end{aligned}$$

- eliminating nested automaton formulas

$$\begin{aligned}&\mathcal{A}_\exists(Q^1,\mathcal{I}_1^1,\mathcal{R}_1^1,\mathcal{A}_\exists(Q^2,\mathcal{I}_1^2,\mathcal{R}_1^2,\mathcal{F}_1))\\&=\mathcal{A}_\exists(Q^1\cup Q^2,\mathcal{I}_1^1\wedge\mathcal{I}_1^2,\mathcal{R}_1^1\wedge\mathcal{R}_1^2,\mathcal{F}_1)\end{aligned}$$

- eliminating nesting of different automaton types require determinization

2.4 Determinization

As we are faced with the existence of different acceptance conditions, utilizing the ω-automata and temporal logic hierarchy in 2.3.3, we can translate formulas from TL_κ to an equivalent $(\mathsf{N})\mathsf{Det}_\kappa$, where $\kappa\in\{\mathsf{G},\mathsf{F},\mathsf{FG},\mathsf{GF},\mathsf{Prefix},\mathsf{Streett}\}$. In this regard, different determinization procedures can be employed depending on their efficiency and field or purpose of usage. For our thesis, we will take into consideration two determinization algorithms: Rabin-Scott construction [RS59], also known as subset construction, and Miyano-Hayashi construction, otherwise known as breakpoint construction [MH84]. Later in the section, we will give a brief overview for other determinization procedures such as Antichain-Based Determinization, Parikh's Theorem-Based Determinization, Zone-Based Determinization, the well-known Safra's Determinization Construction and Piterman's Determinization. We expand our determinization procedures by including also the symbolic determinization procedures in 2.4.4, where we mention BDD Based Determinization, Symbolic Transition-Based Determinization, SAT-Based Determinization and Bounded Model Checking-Based Determinization.

2.4.1 Subset Construction

For a nondeterministic automaton $\mathcal{A}=\langle\Sigma_{in},\mathcal{S},\mathcal{R},\mathcal{F}\rangle$, there is an equivalent deterministic one $\mathcal{A}_{det}=\langle\Sigma_{in},\mathcal{S}_{det},\mathcal{R}_{det},\mathcal{F}_{det}\rangle$ as follows

- The set of states for the deterministic automaton is 2 to the power of the set of states from the nondeterministic automaton: $\mathcal{S}_{det}\coloneqq 2^{\mathcal{S}}$
- The $\mathcal{I}_{det}$ includes all the initial states from $\mathcal{I}$ and there will be only one initial state in the deterministic automaton: $\mathcal{I}_{det}\coloneqq\{\mathcal{I}\}$

- The transition $\mathcal{R}_{det} \subseteq 2^{\mathcal{S}} \times \Sigma_{in} \times 2^{\mathcal{S}}$ is defined as follows

$$\mathcal{R}_{det} \coloneqq \{(Q, \sigma, suc^{\sigma}_{\exists}(Q)) \mid Q \subseteq \mathcal{S}, \sigma \in \Sigma_{in}\}$$

  and with existential successors as

$$suc^{\sigma}_{\exists}(Q) \coloneqq \{s' \in \mathcal{S} \mid \exists s \in \mathcal{Q}.(s, \sigma, s') \in \mathcal{R}\}$$

- The accepted set of states are determined $\mathcal{F}_{det} \coloneqq \{Q \subseteq \mathcal{S} \mid Q \cap \mathcal{F} \neq \{\}\}$

In order to understand how the subset construction works in practice, we consider the following example of a nondeterministic automata in fig. 2.4 and its equivalent deterministic automata in fig. 2.5 achieved by using the subset construction:

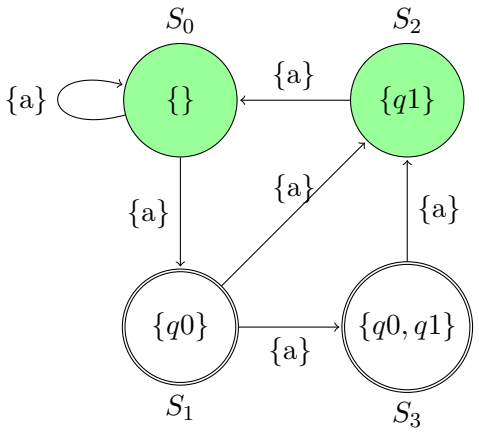


Figure 2.4: *Example of nondeterministic automaton*

- Set of inputs: $\Sigma = \{\{a\}, \{\}\}$
- State variables: $\mathcal{V}_{state} = \{q0, q1\}$
- *Note:* The states depicted with double lines are the initial states, while the states in green are the accepting states of the automata.

The subset construction gathers the sets of reachable states of the nodeterministic automaton $\mathcal{A} = \langle \Sigma_{in}, \mathcal{S}, \mathcal{R}, \mathcal{F} \rangle$ from one of its initial states $\mathcal{I}$. Hence, according to the above definition every state of the deterministic automaton is a set of states from Q. The accepting states of the deterministic automaton need to have as part of the set, at least one of the accepting states from the original automaton $\mathcal{A}$.

2.4.2 Breakpoint Construction

The breakpoint construction of $\mathcal{A} = \langle \Sigma_{in}, \mathcal{S}, \mathcal{R}, \mathcal{F} \rangle$ gives a deterministic automaton $\mathcal{A}_{det} = \langle \Sigma_{in}, \mathcal{S}_{det}, \mathcal{R}_{det}, \mathcal{F}_{det} \rangle$ with the following definitions

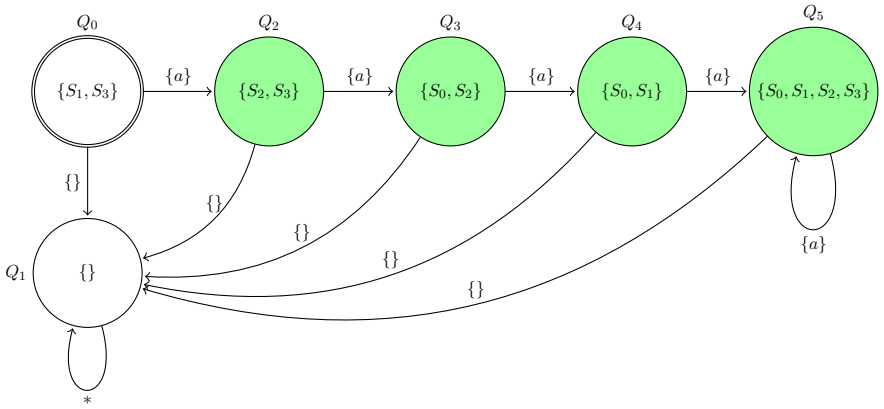


Figure 2.5: *Example of Subset Construction of automaton in 2.4*

- The set of states $\mathcal{S}_{det} \coloneqq 2^{\mathcal{S}} \times 2^{\mathcal{S}}$
- The initial state $\mathcal{I}_{det}$ is $(\mathcal{I}, \{\})$
- The successor state (Q, Q_f) with input σ is encoded as

$$\begin{cases} (suc^{\sigma}_{\exists}(Q), suc^{\sigma}_{\exists}(Q) \cap \mathcal{F}) & : if\ Q_f = \{\} \\ (suc^{\sigma}_{\exists}(Q), suc^{\sigma}_{\exists}(Q_f) \cap \mathcal{F}) & : otherwise \end{cases}$$

- The accepted states $\mathcal{F}_{det}$ are

$$\mathcal{F}_{det} = \{(Q, Q_f) \in \mathcal{S}_{det} \mid Q_f \neq \{\}\}$$

To understand the application of breakpoint construction, we consider the following example of a nondeterministic automata in fig. 2.6 and its equivalent deterministic automata in fig. 2.7 using the breakpoint construction:

- Set of inputs: $\Sigma = \{\{a\}, \{\}\}$
- State variables: $\mathcal{V}_{state} = \{s0, s1\}$
- *Note:* As in 2.4, the double lines states are the initial states, while the green states are the accepting states of the automata.
- From the example we can see that there is state $Q_2 = \{\{\}, \{\}\}$. This state is called a sink state, also known as a dead-end state. The moment a system enters the sink state, it remains there indefinitely. Hence, sink states are often used to indicate an unreachable state or undesirable condition in the system.

2.4.3 Further Algorithms for Determinization

Apart from the determinization algorithms we discussed in 2.4.1 and 2.4.2, there are further algorithms aiming to convert non-deterministic ω-automata into deterministic ones. Subsequently, we outline some of these procedures:

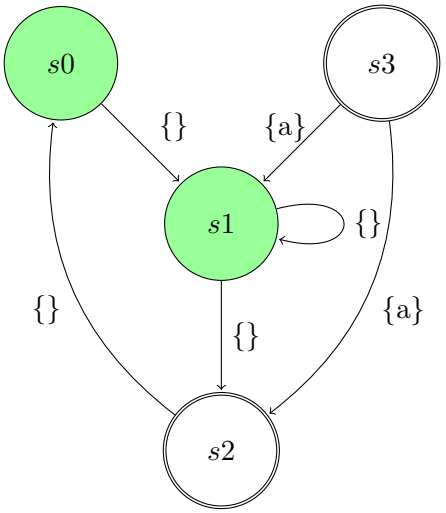


Figure 2.6: *Example of nondeterministic automaton*

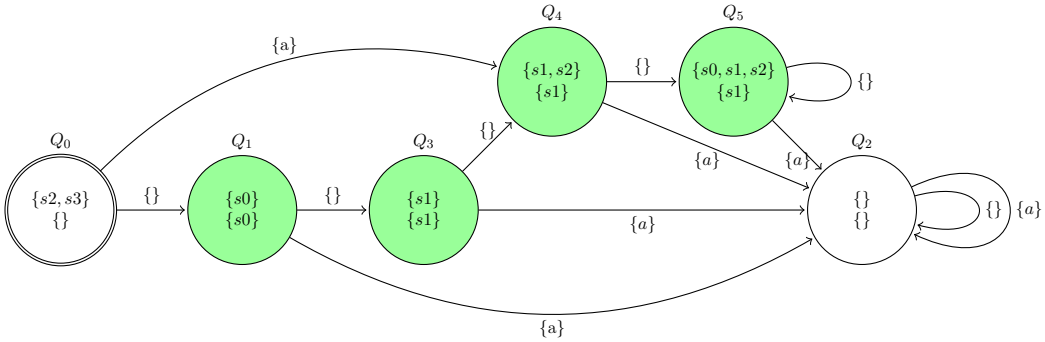


Figure 2.7: *Example of Breakpoint Construction of automaton in 2.6*

Antichain-Based Determinization [HKK95; Som16]

This approach uses antichains to represent sets of states. Given a nondeterministic Büchi automaton $\mathcal{A} = (\mathcal{Q}, \Sigma, q_0, \delta, \mathcal{F})$, where Σ is the input alphabet, $\mathcal{Q}$ is the set of states, q_0 is the initial state, $\delta : \mathcal{Q} \times \Sigma \to 2^{\mathcal{Q}}$ is the transition relation and $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states, we want to build an equivalent deterministic Büchi automaton $\mathcal{A}_{det} = (2^{\mathcal{Q}}, \Sigma, \delta', \mathcal{Q}_0, \mathcal{F}')$ with

- $2^{\mathcal{Q}}$: the power set of $\mathcal{Q}$
- δ': the transition relation of the deterministic automaton
- $\mathcal{Q}_0$: the initial state of the deterministic automaton
- $\mathcal{F}' \subseteq 2^{\mathcal{Q}}$: the set of accepting states of the deterministic automaton

Antichain-Based determinization algorithm can be utilized as in the successive steps:

1. Initialize an empty set A of antichains.

2. Initialize A with the set of antichains corresponding to the initial state.
3. While there exists an unprocessed antichain in A, do:
   a. Select an unprocessed antichain C from A.
   b. For each input symbol $\sigma \in \Sigma$, compute the set of successor states $\delta(\mathsf{C}, \sigma)$ using the transition relation δ.
   c. If $\delta(\mathsf{C}, \sigma)$ is not already in A, add it to A.
   d. Update the transition relation δ' and the set of accepting states $\mathcal{F}'$ based on the computed successor states.
4. The resulting set of antichains in A represents the states of the deterministic automaton $\mathcal{A}_{det}$.

Parikh's Theorem-Based Determinization [Sal85]

This method is based on Parikh's theorem, which relates the language accepted by a Büchi automaton to the occurrences of symbols in its accepted infinite words. Parikh's theorem states that any language recognized by a nondeterministic Büchi automaton N, the set of vector representations of word lengths (also known as Parikh vectors) of accepted words forms a semilinear set. The determinization algorithm takes into consideration the theorem to construct a deterministic automaton simulating the behaviour of the original nondeterministic Büchi automaton by keeping track of the Parikh vectors.

Zone-Based Determinization [Dix+13]

This algorithm represents the states of the ω-automaton as zones in a multidimensional space, allowing for a more concise representation of the state space. The sets of states represent a combination of control and clock valuations. Given a nondeterministic Büchi automaton N, the determinization procedure wants to construct a deterministic automaton D which simulates the behavior of the original nondeterministic Büchi automaton by partitioning the state space into zones, with each zone representing a set of states and clock valuations. To get the deterministic automaton D, we have to mantain the set of zones and update them based on the transition function of the original nondetermistic automaton and the passage of time. The acceptance condition of the deterministic automaton will be defined in terms of zones that contain at least one accepting state from N and satisfies certain clock constraints.

Safra's Determinization Construction [Saf92]

Another well-known method for determinization is the Safra's procedure. The main idea behind this procedure is to represent all the possible accepting runs of the original Büchi automaton using a compact data structure called a Safra tree. Given a nondeterministic Büchi automaton, for each state of it, the Safra's construction builds a corresponding node in the Safra tree. Thus, each node represents a set of states from the original automaton. The algorithm

recursively explores all possible transitions of the original automaton and for each possible transition, a child node is created in the Safra tree. In this way, the Safra tree capturess all the available runs of the initial nondeterministic Büchi automaton. Once the tree has been establish, the subsequent process is to achieve a deterministic automaton. In the case of Safra determinization procedure, the deterministic automaton is a deterministic Rabin automaton with states corresponding to the nodes in the Safra tree and the transition between the states based on the transition relation of the initial automaton. Along the same lines, the acceptance condition is derived from the acceptance condition of the original nondeterministic Büchi automaton.

Piterman's Determinization [Pit06]

Piterman's procedure offers an alternative to the Safra's construction and provides certain efficiency advantages in terms of both time and space complexity. The set of states of a given nondeterministic Büchi automaton are encoded using binary decision diagrams (BDDs) or similar data structures. Each set of states is uniquely encoded typically with a BDD. This type of encoding is helpful in the determinization process for an efficient manipulation and comparison of state sets. Similar to other determinization methods, Piterman's procedure constructs the transitions between the state sets based on the transition relation of the given nondeterministic automaton. Furthermore, the procedure has to ensure that the original acceptance condition is preserved. Piterman's method aims to minimize the number of transitions and states in the resulting deterministic automaton while preserving language equivalence.

2.4.4 Symbolic Determinization Algorithms

Symbolic determinization is a process used in model checking to convert a nondeterministic automaton to a deterministic one, while taking advantage of the symbolic representation of states and transition relation. Instead of explicitly enumerating all states and transitions, symbolic determinization represents sets of states and transitions symbolically, often using data structures such as binary decision diagrams (BDDs) or boolean formulas. By using symbolic representation, the determinization procedure can handle large state spaces more effectively and can potentially decrease the computational burden compared to explicit enumerative methods [Bur+92].
Symbolic determinization algorithms are widely used in formal verification, model checking, and synthesis of reactive systems, where they enable the analysis of systems with complex temporal properties. They offer advantages in terms of efficiency, scalability, and the representation of transition systems.

Binary Decision Diagrams (BDD) Based Determinization

In 2.3.4, we already mentioned how we can represent an ω-automata by using BDDs. We asume that the finite alphabet, the set of states, the transition relation and the initial states are encoded with propositional logic. Given the re-

sulting propositional logic expressions for each representation of the automata, we can use BDDs for the encoding and then continue with the determinization procedure.

Symbolic Transition-Based Determinization

Symbolic transition-based determinization primarily deals with transitions between states. Instead of explicitly representing individual states, the algorithm focuses on symbolically representing transitions between states. This involves encoding transition conditions using symbolic expressions. Starting from the initial state of the given nondeterministic automaton, the procedure explores possible transitions based on the inputs and the transition function. At each step, it uses symbolic operations, such as conjunction, disjunction or negation to determine the set of possible next states.

SAT-Based Determinization

SAT-based determinization algorithms make use of propositional satisfiability (SAT) solvers to construct transitions between states. Thus, the transition relation and the acceptance condition are encoded as propositional formulas. The problem of determining the existence of an accepting run in the determinized automaton is formulated as a SAT instance, which is then solved using SAT solvers. SAT-based approaches can be effective to determinize an automata with a complex acceptance condition.
As mentioned previously, the transition relation of a given nondeterministic automaton is depicted using boolean expressions. Each transition from a current state to the next state depending on a specific input represents a clause in the formula. SAT solvers have to determine the existence of an assignment of truth values for the formula variables in a way that satisfies all clauses, indicating a valid transition. Depending on the acceptance conditions of the original automaton, these formulas have to capture also the acceptance criteria for paths, which also have to be included in the SAT problem. After combining the boolean expressions for the transition relation and acceptance conditions, SAT solvers have to find a satisfying assignment that meets all conditions.

Bounded Model Checking (BMC)-Based Determinization

Another method used to determinize nondeterministic automata is BMC-Based Determinization which makes use of Bounded Model Checking [Bie99] for the determinization.
Bounded Model Checking verifies whether a system satisfies a given property within a bounded number of steps. All possible execution paths up to a certain length (bounded depth) are checked to see if any path violates a specified property.
The procedure will iteratively apply the bounded model checking to bounded execution paths (sequence of states with a specified depth) and encode the

transition relation with boolean formulas. By verifying the satisfiability of the aforementioned formulas, we determinize a given nondeterministic automaton.

3 Evaluation of Symbolic Determinization Algorithms

This chapter will include an introduction to the framework used for this work. Furthermore, following is a brief overview of the different packages used in the implementation phase. Section 3.2 expands on the implementation of the symbolic determinization algorithms, such as the subset construction and breakpoint construction. In conclusion, the experimental results are shown in 3.3.

3.1 Averest Framework

Averest[1] is a framework containing functions for modeling, simulation, synthesis and analysis of reactive embedded systems [SS05; SS06]. Offering a variety of modules for the different objectives mentioned previously, we will focus mostly on the listed modules:

- Averest.Core
- Averest.Quartz
- Averest.Analysis

3.1.1 Averest.Analysis

`Averest.Analysis` includes the modules `BDD` and `TemporalLogic`. The implementation of `BDD` module is a simple BDD package where the BDD nodes indicate boolean expressions. In order to use the package, we have to initialize the BDD through the `Initialize` function. We have to specify all the available variables in a distinct order. `BoolExpr2Bdd` converts a boolean expression to a BDD after the initialization of the BDD manager. Afterwards, the typical BDD functions, such as *Apply algorithm*, *Constrain* or *Restrict* algorithm, can be used.
`TemporalLogic` encompasses functions for the translations of different logics, i.e LTL / LeftCTL* / CTL into symbolically encoded existential ω-automata.

3.1.2 Averest.Core

`Averest.Core` offers, among other modules, `Expressions`, `Names` and `Specifications` modules. `Expressions` implements the different types of the Quartz language with our focus mostly on the functions related to boolean expressions.

[1] See `http://www.averest.org/`.

`Names` includes the qualified names used by identifiers, while `Specifications` is reserved for temporal logic specifications.

3.1.3 Averest.Quartz

`Averest.Quartz` implements data types for untyped expressions. For the implementation of our work, using the functions offered in this module, we can create a boolean parser which takes a string of the boolean expression and converts it to the data types of the Quartz language.

3.2 Implementing Symbolic Determinization Algorithms

Based on the temporal logic and ω-automata hierarchy discussed in 2.3.3, we can translate any $\varphi \in \mathsf{TL}_\kappa$ into an equivalent ω-automaton $\mathcal{A}_\kappa \in \mathsf{Det}_\kappa$. According to [SST04], the determinization procedures of subset and breakpoint construction are sufficient for translation of the formulas $\mathsf{TL_G} \backslash \mathsf{TL_F}$ and $\mathsf{TL_{GG}} \backslash \mathsf{TL_{FG}}$, respectively. These findings allow us to eliminate the use of Safra's determinization procedure. Nonetheless, studies have shown [AL04] that even a small temporal logic formula can become doubly exponential $O(2^{2^{|n|}})$ when the translation to deterministic automata occurs. Even though symbolic representation can eliminate the exponential blow up as shown in [SST04], this benefit falls through as all determinization procedures take into consideration explicitly represented automata. Since the deterministic automata is given in explicit form, where all the states are enumerated, then simple implementations undergo the double exponential complexity.
Consequently, the experimental results achieved by using the symbolic representation procedures introduced by [MSL08] show a considerable improvement in reducing the computational complexity. Esentially, the procedure is implemented as follows: Given a symbolic represented nondeterministic automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with reachable states $\{\vartheta_1, \ldots \vartheta_n\}$, we can symbolically construct the deterministic automaton obtained by either the subset or breakpoint construction. Apart from the step of enumerating the reachable states, the symbolic deterministic representation of the deterministic automaton is attained without the need of state enumeration. As a result, we can obtain deterministic automata of large LTL formulas or from worst case scenarios mentioned in [AL04]. Depending on the definitions of subset construction in 2.4.1 and breakpoint construction in 2.4.2, the following section will present the symbolic implementation of these two procedures.

3.2.1 Preliminary Implementation

As stated already in [MSL08], the implemented algorithms expect a *symbolic* representation of a nondeterministic automaton as $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F}')$, where $\mathcal{I}$ is a propositional formula over the states variable Q and the acceptance condition

$\mathcal{F}'$ is based on a propositional formula $\mathcal{F}$ over the state variables Q which can describe a liveness, safety, fairness or persistence property. We assume that the state variable $Q = \{q_1, \ldots, q_m\}$ and the input variable is $V_\Sigma = \{x_1, \ldots, x_k\}$.
Before implementing the determinization algorithms, we have to *compute the reachable states* of the automaton. As a first step, we eliminate any variable that is not a state variable, i.e input variables. The transition relation will therefore be defined as $\mathcal{R}_\exists :\equiv \exists x_1, \ldots x_k.\ \mathcal{R}$. Afterwards, we can compute the reachable states, which is nothing else than computing the existential successor of a given state. It can be achieved using the fixpoint $\mu x.\ \mathcal{I} \vee \overleftarrow{\Diamond}$. Additionally, we want to find reachable states that have at least one infinite path and remove the deadend states, hence the reachable states are computed as $\mathcal{S}_{reach} :\equiv (\nu y.\Diamond y) \wedge (\mu x.\mathcal{I} \vee \overleftarrow{\Diamond})$. The result at the end is a propositional formula over the state variables Q.
The following step is to perform a one-hot encoding of the reachable states of the original nondeterministic automaton. We explicitly enumerate the reachable states $\{\vartheta_1, \ldots, \vartheta_n\} \subseteq Q$ in such a way that we identify a reachable state ϑ_i with state variable p_i, where p_i is a new state variable for the representation of the deterministic automaton. Therefore, the new state variables are defined as $Q_{det} = \{p_1, \ldots, p_n\}$. Furthermore, to be able to enumerate all the reachable states $\vartheta \subseteq Q$, we have to define the minterms as:

$$minterm_Q(\vartheta) :\equiv \left(\bigwedge_{x \in \vartheta} x\right) \wedge \left(\bigwedge_{x \in Q \setminus \vartheta} \neg x\right)$$

Using the Averest package, we create a function `ComputeReachableStates` to compute the reachable states of a nondeterministic automaton, given by the set of its state variables, its initial condition on the state variables, and the transition relation where the state variables, input variables (atoms) occur as well as the next occurrences of the state variables (but not of the inputs). The next occurrences of the state variable are encoded using `BoolNext` type, to represent the transition of a state variable to the next point of time. The initial condition, the transition relation and the set of state variables are propositional formulas.
The procedure will make use of BDDs to represent the propositional formulas, therefore the transition relation will be constructed with a variable ordering $\mathsf{X}q_1 \prec \cdots \prec \mathsf{X}q_N \prec q_1 \prec \cdots \prec q_N \prec a_1 \prec \cdots \prec a_m$. As mentioned previously, the inputs need to be eliminated from the transition relation to compute the reachable states, which is why we existentially quantify them, in that the disjunction of the subtrees that start with (next) state variables is computed. Then, the reachable states are computed in a fixpoint computation where the conjunction with the modified transition relation and the so far computed reachable states is computed. The next states are then obtained by the disjunction of the subtrees starting with next state variables. At this point, the BDD must be relabeled since what were next states variables have become current state variables. Afterwards, we continue the computation recursively until we gather all the reachable states. Finally, the satisfying cubes are computed

from the BDD of the reachable states, and these cubes are then extended to minterms which are encoded as sets of sets of state variables, so that a minterm corresponds with the state variables that have positive occurrences in the minterm. The function makes use of several computed tables to avoid computations on the same shared subtrees of the BDDs.

3.2.2 Subset Construction

Since we computed the reachable states and used the one-hot encoding, the symbolic subset construction can be defined as follows:
Definition 3. [MSL08] Given a nondeterministic automaton $\mathcal{A}_\exists(Q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with reachable states $\{\vartheta_1, \ldots, \vartheta_2\}$ and state variables $Q_{det} = \{p_1, \ldots, p_n\}$, a deterministic automaton $\mathcal{A}_\exists(Q_{det}, \mathcal{I}_{det}, \mathcal{R}_{det}, \mathcal{F}_{det})$ is defined as follows:

- $\mathcal{H} :\equiv \bigvee_{j=1}^{n} p_j \wedge minterm_{Q_{det}}(\vartheta_j)$
- $\mathcal{I}_{det} :\equiv \bigwedge_{i=1}^{n}(p_i \leftrightarrow \exists q_1 \ldots q_m.\ minterm_{Q_{det}}(\vartheta_i) \wedge \mathcal{I})$
- $\mathcal{F}_{det} :\equiv \exists q_1 \ldots q_m.\ \mathcal{H} \wedge \mathcal{F}$
- $\mathcal{R}_{det} :\equiv \bigwedge_{i=1}^{n}(p_i' \leftrightarrow \eta_i)$ *with* $\eta_i :\equiv \exists q_1 \ldots q_m\ q_1' \ldots q_m'.\mathcal{H} \wedge \mathcal{R} \wedge (minterm_{Q_{det}}(\vartheta_i))'$

The correctness of the above definition has already been proven in [MSL08]. According to it, we give the following remarks:

- $\mathcal{F}_{det} :\equiv \bigvee_{j=1}^{n} p_j \wedge \exists q_1 \ldots q_m.minterm_{Q_{det}}(\vartheta_j) \wedge \mathcal{F}$. If ϑ_j doesn't satisfy $\mathcal{F}$, then the subformula $minterm_{Q_{det}}(\vartheta_j) \wedge \mathcal{F}$ is false. In the case ϑ_j satisfies $\mathcal{F}$, $minterm_{Q_{det}}(\vartheta_j) \wedge \mathcal{F}$ becomes equal to $minterm_{Q_{det}}(\vartheta_j)$ since $\mathcal{F}$ is now true. Now let's consider the existential quantification in the $\mathcal{F}_{det}$ definition. The existential quantification will return `true` when $minterm_{Q_{det}}(\vartheta_j)$ satisfies $\mathcal{F}$ and `false` otherwise. Therefore, following the rewritten formula at the beginning, $\mathcal{F}_{det} \Leftrightarrow \bigvee_{j=1}^{n} p_j$.
- $\mathcal{I}_{det}$ is the superstate encompasing all the initial states of the nondeterministic automata. Following the same lines, as we did for the $\mathcal{F}_{det}$, $\exists q_1 \ldots q_m.\ minterm_{Q_{det}}(\vartheta_i) \wedge \mathcal{I}$ can be `true` or `false`. The subformula yields true iff $\vartheta_i \in \mathcal{I}$.
- In order to find the next superstate transition from a current superstate, then all the state sets, part of the current superstate, need to have a transition to the set of states part of next superstate with a given input condition. This is shown and can be proven symbolically by the above definition of $\mathcal{R}_{det}$.

Now, we can use the symbolic representation of the automaton to determinize formulas from the temporal classes $\mathsf{TL_G}$, $\mathsf{TL_F}$ and $\mathsf{TL_{Prefix}}$.
In practice, we create a function `SymbolicRabinScott` which computes a symbolic representation of a deterministic automaton for a given nondeterministic automaton. For the nondeterministic automaton, we have to consider its state

variables, its initial condition, its transition relation, its acceptance condition, and the explicitly enumerated set of its reachable states in terms of minterms which are sets of state variables that contain those state variables that have positive occurrences in the minterm, i.e. which hold on the state. For the latter, we can already use the function in 3.2.1.
The determinization function associates then each reachable state (minterm, set of state variables) with a new state variable of the deterministic automaton whose name starts with the prefix "qDet". This way, a minterm on the new state variables corresponds with a set of sets of states of the nondeterministic automaton, since a minterm on the new state variables corresponds with a set of new state variables which is therefore a set of set of minterms of the old state variables, i.e., a set of set of states of the nondeterministic automaton.
Note that the transition equations are of the form $\mathsf{X}(p_i) \leftrightarrow p_1 \wedge \phi[i,1] \vee \cdots \vee p_N \wedge \phi[i,N]$ where $\phi[i,j]$ is the set of inputs that trigger a transition from state p_j to state p_i in the nondeterministic automaton. This corresponds with the existential successor computation where p_i is added to a successor state if one state p_j can reach p_i with input $\phi[i,j]$.

Algorithm 6 Algorithm to compute the symbolic subset construction

function SymbolicRabinScott
  $qDet \leftarrow$ new state string
  $stateVars \leftarrow$ state variable set
  $initCond \leftarrow$ initial condition
  $transRel \leftarrow$ transition relation
  $reachStates \leftarrow$ reachable states set
  for all $s \in reachStates$ **do**
    $newEnc \leftarrow qDet + string\ i$
    $i \leftarrow i + 1$
  end for
  return stateList ($newEnc$, s)
  for all $(newEnc, s) \in stateList$ **do**
    if s is initial condition **then**
      initial condition set $\leftarrow s$
    end if
    compute equations $next(s_i) \leftrightarrow s_i\ \&\ \phi[i,1] \mid \ldots \mid s_i\ \&\ \phi[i,N]$
  end for
  equation list $\leftarrow$ $disjunctions\ of\ next(s_i)$
  for all $(newEnc, s) \in stateList$ **do**
    if s is accepting state **then**
      acceptance state $\leftarrow s$
    end if
  end for
  acceptance list $\leftarrow$ $disjunction\ of\ acceptance\ state$
  fairness constrain list $\leftarrow$ [ ]
end function
return DetAuto (equation list, fairness constrain, acceptance list)

3.2.3 Breakpoint Construction

There are still temporal classes, which can't be determinized with the subset construction. Using the breakpoint construction we can determinize the classes $\mathsf{TL}_{\mathsf{GF}}$, $\mathsf{TL}_{\mathsf{FG}}$ and $\mathsf{TL}_{\mathsf{Streett}}$.
Following again the findings in [MSL08], the symbolic representation of the breakpoint construction follows: Given $\mathcal{A} = \mathcal{A}_{\exists}(q, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with reachable states $\{\varrho_1, \ldots, \varrho_n\}$ such that $\mathcal{F}$ is the set $\{\varrho_{n+1-l}, \ldots, \varrho_n\}$. With new state variables $Q_{bpt} = \{p_1, \ldots, p_n, b_1, \ldots, b_l\}$ and taking into consideration the definitions of $\mathcal{I}_{det}$ and $\mathcal{R}_{det}$ from the subset construction, $\mathcal{A}_{bpt}$ is defined:

- $\mathcal{H} :\equiv \bigvee_{j=1}^{n} p_j \wedge minterm_{Q_{det}}(\vartheta_j)$
- $\mathcal{I}_{bp} :\equiv \bigwedge_{i=1}^{l} b_i \leftrightarrow 0$
- $\mathcal{F}_{bp} :\equiv \bigvee_{i=1}^{l} b_i$
- $\mathcal{R}_{bp} :\equiv \bigwedge_{i=1}^{l} b_i' \leftrightarrow \neg\mathcal{F}_{bp} \wedge \eta_i \vee \mathcal{F} \wedge [\eta_i]_{\varrho}$, where
  $\eta_i :\equiv \exists q_1 \ldots q_m \; q_1' \ldots q_m'.\mathcal{H} \wedge \mathcal{R} \wedge (minterm_{Q_{det}}(\vartheta_i))'$ and ϱ is the substitution that maps each $p_1, \ldots, p_{n-l}$ to 0 and $p_{n-l}, \ldots, p_n$ to $b1, \ldots, b_l$ respectively.

The breakpoint construction yields pairs of sets of states. The first component is computed by the subset construction in 3.2.2, while the second component represents the set of states that has never left the set of designated states since the last breakpoint. A *breakpoint* is a state that has its second component empty.
States of deterministic automaton from a breakpoint construction determinization are a subset of Q_{bpt} and are considered a pair of subsets of $\{p_1, \ldots, p_n\}$ and $\{b_1, \ldots, b_l\}$. The moment we reach a breakpoint, the empty set is filled with successors from the first set. To find the successors we can just use the formula for the transition relation from the subset construction.

3.3 Experimental Results

Utilizing the symbolic representation of the determinization algorithms in 3.2.2 and 3.2.3, we have taken into consideration the translation of different LTL formulas, where the nondeterministic automaton has an exponential number of states. We want to prove with the results from the experiments that the symbolic representation of the algorithms is more efficient than the explicit enumeration procedures.

3.3.1 Experiment 1

In [AL04], the formulas $\bigwedge_{i=0}^{n-1} \mathsf{F}a_i$ and $\bigvee_{i=0}^{n-1} \mathsf{G}a_i$ were discussed as examples where the nondeterministic automaton has an exponential number of states.

The two formulas are dual: $\bigwedge_{i=0}^{n-1} \mathsf{F} a_i$ is a liveness property, and its dual formula $\bigvee_{i=0}^{n-1} \mathsf{G} a_i$ is a safety property.

Algorithm 7 Algorithm to compute the symbolic breakpoint construction

function SYMBOLICBREAKPOINT
  $qDet \leftarrow$ new state string for first component
  $bDet \leftarrow$ new state string for second component
  $stateVars \leftarrow$ state variable set
  $initCond \leftarrow$ initial condition
  $transRel \leftarrow$ transition relation
  $reachStates \leftarrow$ reachable states set
  for all $s \in reachStates$ **do**
    $newEnc1 \leftarrow qDet + string\ i$
    $newEnc2 \leftarrow bDet + string\ i$
    if s is accepting state **then**
      acceptance state $\leftarrow s$
    end if
    $i \leftarrow i + 1$
  end for
  return stateList (*newEnc1, newEnc2, acceptance state, s*)
  for all $(newEnc1, newEnc2, acceptancestate, s) \in stateList$ **do**
    if *acceptance state* is true **then**
      return newEnc2
    end if
    breakpoint state set $\leftarrow$ disjunction $newEnc2$
  end for
  for all $(newEnc1, newEnc2, acceptancestate, s) \in stateList$ **do**
    if s is initial condition **then**
      initial condition set $\leftarrow s$
    end if
    compute equations $next(s_i) \leftrightarrow s_i\ \&\ \phi[i,1] \mid \ldots \mid s_i\ \&\ \phi[i,N]$
    equation list $\leftarrow$ $disjunctions\ of\ next(s_i)$
    if s is previous accepting state **then**
      repeat previous forall for next(accepting state)
    end if
    breakpoint list $\leftarrow$ $disjunction\ of\ next(acceptance\ state)$
    $next1 \leftarrow$ conjunction of negation of breakpoint list and equation list
    $next2 \leftarrow$ conjunction of breakpoint list
    $bpnext \leftarrow$ disjunction of next1, next 2
  end for
  fairness constrain list $\leftarrow$ [ ]
end function(equation list, fairness constrain, acceptance list)

Both formulas can therefore be generated to nondeterministic ω-automata that can be determinized by the subset construction. In the following, we focus on

the safety formula $\bigvee_{i=0}^{n-1} \mathsf{G}a_i$. The symbolically represented automaton that is obtained by the standard translation of LTL formulas is the following one:

$$\mathcal{A}_\exists \left(\{q_0, \ldots, q_{n-1}\}, \bigvee_{i=0}^{n-1} q_i, \bigwedge_{i=0}^{n-1} (q_i \leftrightarrow a_i \wedge \mathsf{X}q_i), \mathsf{true} \right)$$

For illustrating, we take as an example $n = 2$ and the obtained automaton is

$$\mathcal{A}_\exists \left(\{q_0, q_1\}, q_0 \vee q_1, (q_0 \leftrightarrow a_0 \wedge \mathsf{X}q_0) \wedge (q_1 \leftrightarrow a_1 \wedge \mathsf{X}q_1), \mathsf{true} \right)$$

Fig. 3.1 represents the aforementioned example. *Note:* We annotate the input variables as a_i, a_j, which means that the input for the transition is $a_i \wedge a_j$ and also the negation of an input variable is shown with an exclamation mark. The deterministic automata of fig. 3.1 can be seen in fig. 3.2.

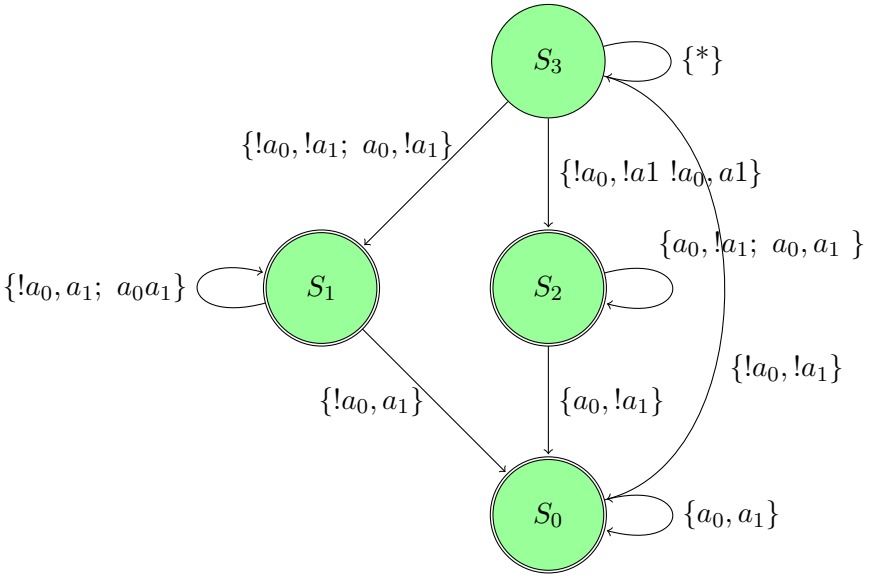


Figure 3.1: *Example of nondeterministic automata for* $\mathsf{G}a_0 \vee \mathsf{G}a_1$

For these type of automata, we obtain the following results in table 3.1.

Note that the nondeterministic automaton for parameter n has n state variables and $2^n - 1$ reachable states, so that the deterministic automaton generated by the symbolic subset construction has $2^n - 1$ state variables, and therefore possible 2^{2^n-1} many states. In this example, the deterministic automaton has however 'only' 2^n reachable states.

While the number of states seems not to be too big, it becomes quickly a problem for the explicit subset construction since for each superstate, we have to consider all of the 2^n inputs one after the other which becomes for the 2^n reachable states 2^{2n} transitions.

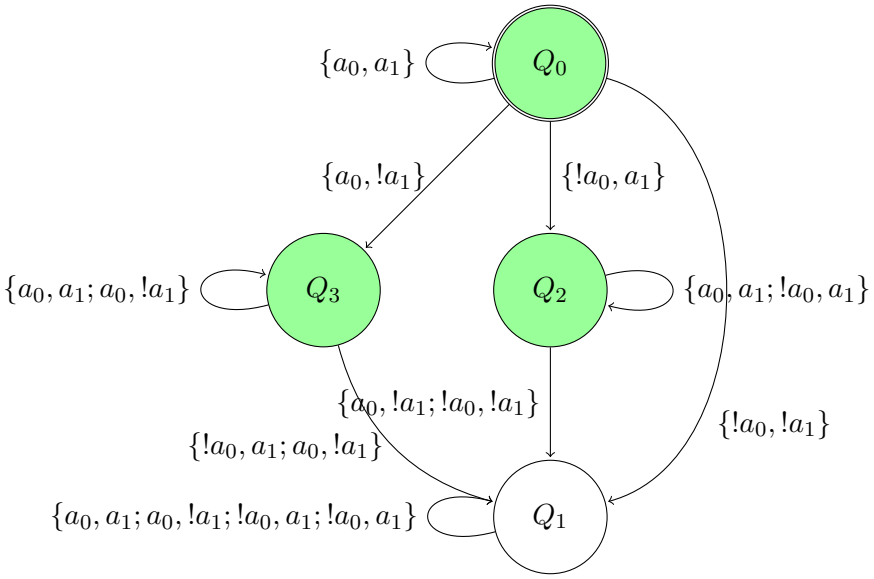


Figure 3.2: *Deterministic automata of Fig. 3.1*

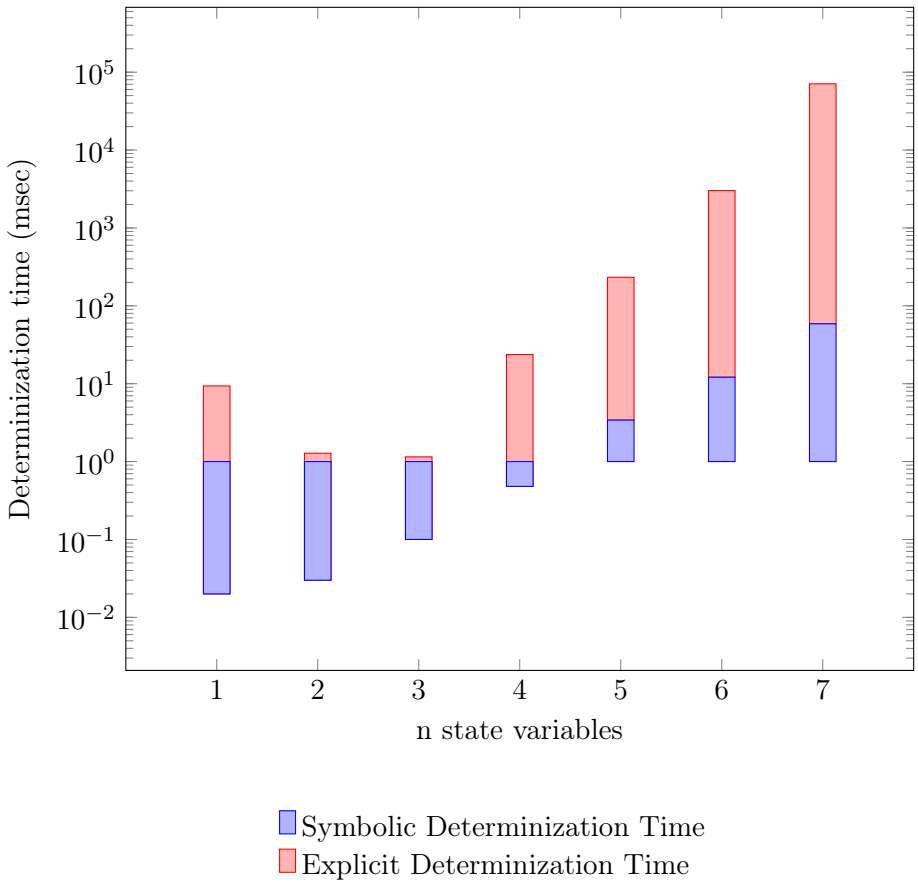


Figure 3.3: *Histogram for table 3.1*

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	1	21.87	0.02	2	9.34
2	4	3	15.17	0.03	4	1.25
3	8	7	19.83	0.10	8	1.05
4	16	15	30.60	0.48	16	23.21
5	32	31	122.14	3.41	32	228.96
6	64	63	260.81	12.11	64	3007.20
7	128	127	366.95	58.73	128	70792.49
8	256	255	1312.09	335.51	256	–
9	512	511	3686.26	1368.96	512	–
10	1024	1023	10484.80	6442.02	1024	–
11	2048	2047	33305.79	29567.14	2048	–
12	4096	4095	107224.29	131749.84	4096	–

Table 3.1: *Experimental results for obtained automata from* $\bigvee_{i=0}^{n-1} \mathsf{G}a_i$

As a variant of the above automaton, we may also work only with implication instead of equivalences, and consider therefore the following equivalent automaton:

$$\mathcal{A}_{\exists}\left(\{q_0,\ldots,q_{n-1}\}, \bigvee_{i=0}^{n-1} q_i, \bigwedge_{i=0}^{n-1} \left(q_i \rightarrow a_i \wedge \mathsf{X}q_i\right), \mathsf{true}\right)$$

This has only minor influences on the runtimes and on the number of states of the automata:

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	1	24.84	0.02	2	0.09
2	4	3	14.28	0.03	4	0.08
3	8	7	15.36	0.19	8	0.77
4	16	15	18.78	0.55	16	11.23
5	32	31	83.96	2.22	32	520.42
6	64	63	37.12	12.25	64	8546.83
7	128	127	79.75	56.91	128	601151.00
8	256	255	172.71	267.57	256	–
9	512	511	413.71	1415.23	512	–
10	1024	1023	1319.43	5946.10	1024	–
11	2048	2047	3506.00	27195.13	2048	–
12	4096	4095	10808.09	126400.19	4096	–

Table 3.2: *Experimental results of the variation automata with implication*

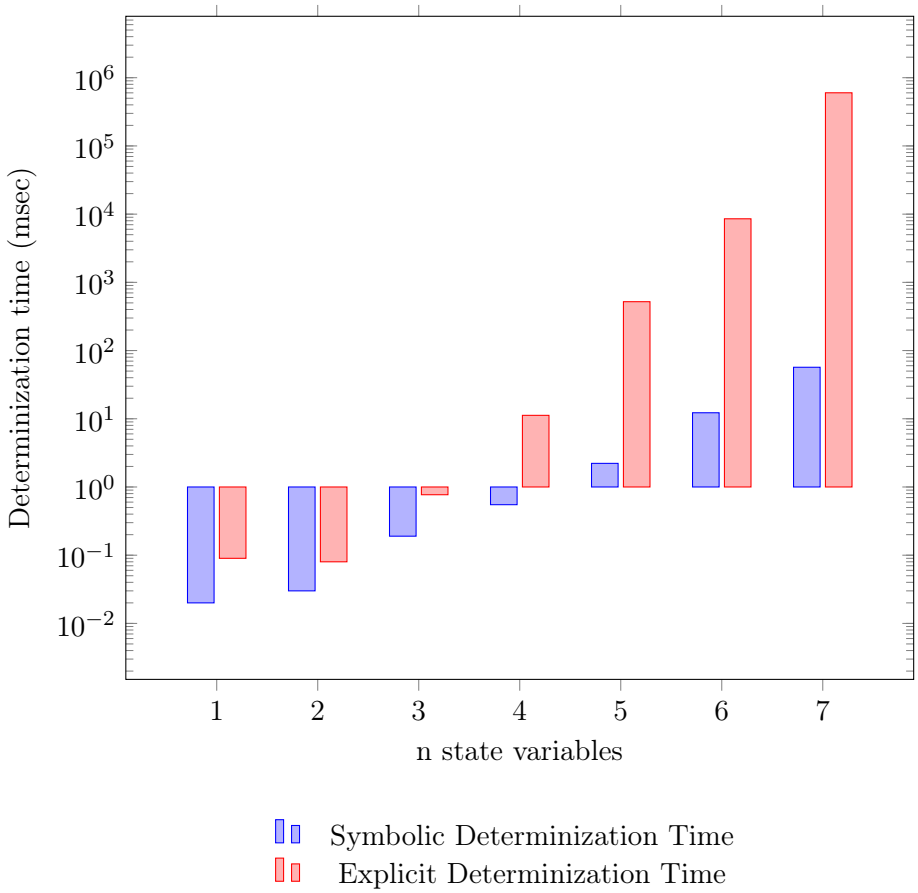


Figure 3.4: *Histogram for table 3.2*

As yet another variant of the above automaton, we may also use a combination of watchdogs and therefore the following equivalent automaton:

$$\mathcal{A}_{1c}(n) := \mathcal{A}_{\exists}\left(\{q_0,\ldots,q_{n-1}\}, \bigwedge_{i=0}^{n-1} q_i, \left(\begin{array}{l}\left(\bigvee_{i=0}^{n-1} q_i\right) \wedge \\ \bigwedge_{i=0}^{n-1} (\mathsf{X}q_i \leftrightarrow a_i \wedge q_i)\end{array}\right), \mathsf{true}\right)$$

Note that each state variable q_i is initially true, and remains true until its corresponding input variable a_i is true. If a_i is false, then q_i switches to false and remains false forever. Hence, at least one of the input variables a_i is always true if and only if at least one of the state variables q_i is always true. If that is not the case, i.e., if all state variables q_i are false, then the above automaton has no transition, which is the only reason why this automaton is not deterministic.
The above automata have n state variables and 2^n reachable states. The subset construction just has to add a sink state to add the missing transitions from the state where all state variables q_i are false. The deterministic automaton has therefore $2^n + 1$ states in this example. In particular, we get the following results in table 3.3.

As can be seen, the explicit subset construction can handle this example much better than the other two variants before, but the symbolic implementation is still without compare.

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	15.59	0.03	3	0.07
2	4	4	24.74	0.05	5	0.07
3	8	8	16.90	0.15	9	0.23
4	16	16	26.36	0.69	17	1.22
5	32	32	52.81	3.89	33	6.14
6	64	64	104.98	19.22	65	123.80
7	128	128	499.13	112.85	129	993.57
8	256	256	745.31	419.87	257	17267.47
9	512	512	2033.24	1896.41	–	–
10	1024	1024	6576.45	9303.78	–	–
11	2048	2048	21293.58	42444.72	–	–
12	4096	4096	72906.73	188371.20	–	–

Table 3.3: *Experimental results of the variation automata with implication*

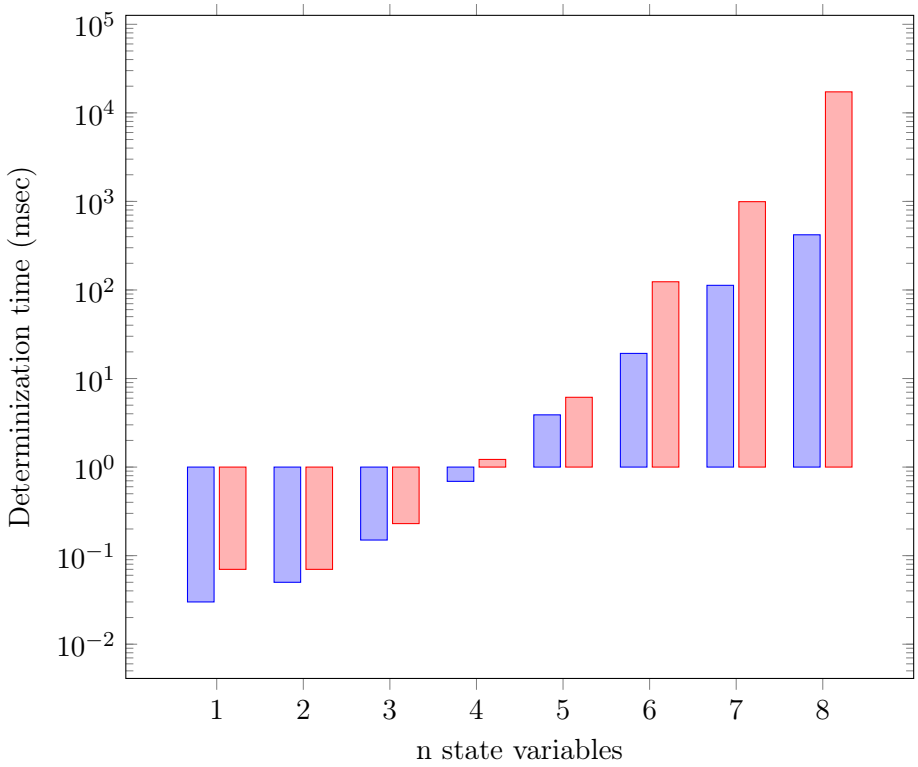


Figure 3.5: *Histogram for table 3.3*

3.3.2 Experiment 2

In [AL04], the formula $\mathsf{F}\bigwedge_{i=0}^{n-1}(a_i \vee \mathsf{F}b_i)$ was discussed as an example that requires a doubly exponential number of states for any equivalent deterministic automaton. Clearly, the same must then also hold for the dual safety property $\mathsf{G}\bigvee_{i=0}^{n-1}(a_i \wedge \mathsf{G}b_i)$ that we consider in this experiment. The standard translation of LTL formulas generates the following automaton for this formula:

$$\mathcal{A}_\exists\left(\{q_0,\ldots,q_n\}, q_n, \left(q_n \leftrightarrow \left(\bigvee_{i=0}^{n-1}(a_i \wedge q_i)\right) \wedge \mathsf{X}q_n\right) \wedge \bigwedge_{i=0}^{n-1}(q_i \leftrightarrow b_i \wedge \mathsf{X}q_i), \mathsf{true}\right)$$

For this class of automata, we obtain the following results:

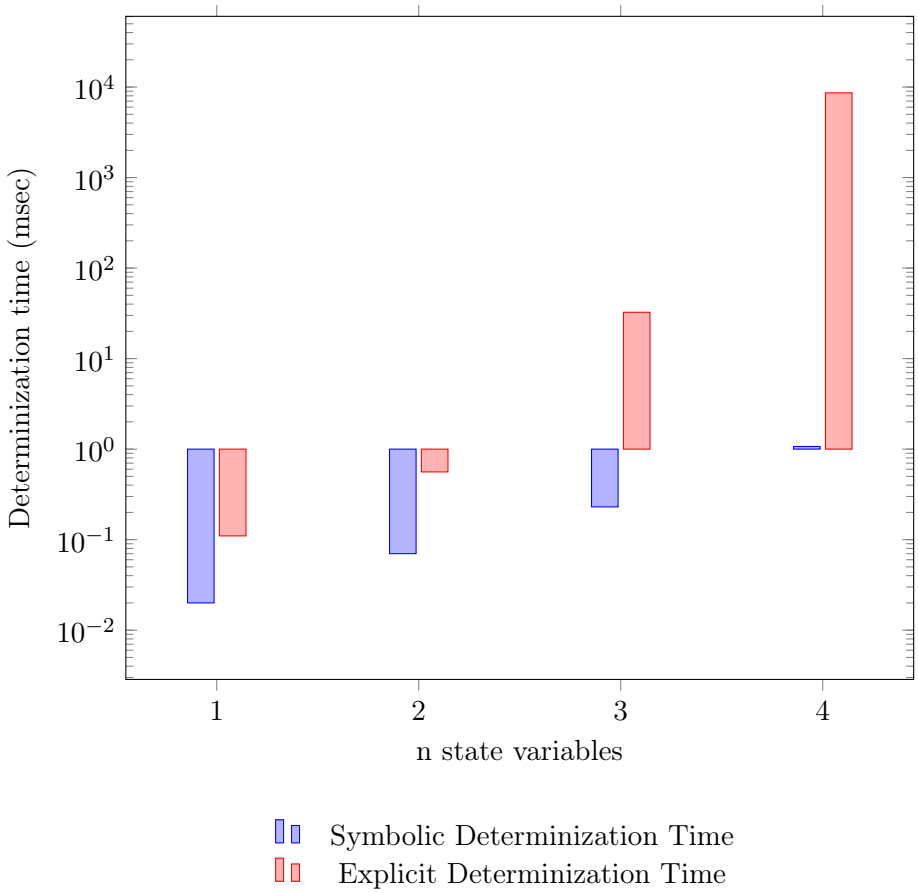


Figure 3.6: *Histogram for table 3.4*

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	22.05	0.02	3	0.11
2	4	4	21.13	0.07	6	0.56
3	8	8	55.44	0.23	20	32.51
4	16	16	92.43	1.07	168	8633.91
5	32	32	348.53	5.28	–	–
6	64	64	1690.04	87.70	–	–
7	128	128	6793.90	114.42	–	–
8	256	256	43764.68	645.26	–	–
9	512	512	263447.23	2968.07	–	–

Table 3.4: *Experimental results of obtained automata for* $\mathsf{G}\bigvee_{i=0}^{n-1}(a_i \wedge \mathsf{G}b_i)$

Note that the nondeterministic automaton for parameter n has $n+1$ state variables, but only 2^n reachable states. Hence, the deterministic automaton generated by the symbolic subset construction has 2^n state variables, and therefore possible 2^{2^n} many states. The symbolic subset construction can generate the deterministic automata even for $n = 9$ which would then have possibly 2^{2^9} many states. Clearly, this number of states may not be reachable, but recall that [AL04] proved that a doubly exponential number of states is required for any deterministic automaton. The number of reachable states of the generated deterministic automata starts with 3,6,20,168 which could therefore be the sequence of Dedekind numbers, i.e., the number of monotone Boolean functions of n variables which has a double exponential asymptotic growth $O(2^{2^n})$[2]

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	28.21	0.03	3	0.14
2	4	4	16.26	0.05	6	0.45
3	8	8	33.88	0.22	20	70.85
4	16	16	93.52	1.50	168	17206.42
5	32	32	198.48	5.27	–	–
6	64	64	598.97	27.16	–	–
7	128	128	2990.58	112.50	–	–
8	256	256	15162.93	562.14	–	–
9	512	512	90695.13	2720.06	–	–
10	1024	1024	606154.43	14317.74	–	–

Table 3.5: *Variation automata with implication for* $\mathsf{G}\bigvee_{i=0}^{n-1}(a_i \wedge \mathsf{G}b_i)$

[2] See `https://en.wikipedia.org/wiki/Dedekind_number` and `https://en.wikipedia.org/wiki/Central_binomial_coefficient`.

If we use implications instead of equivalences in the transition relation of the automaton, we obtain the following similar results presented in table 3.5.

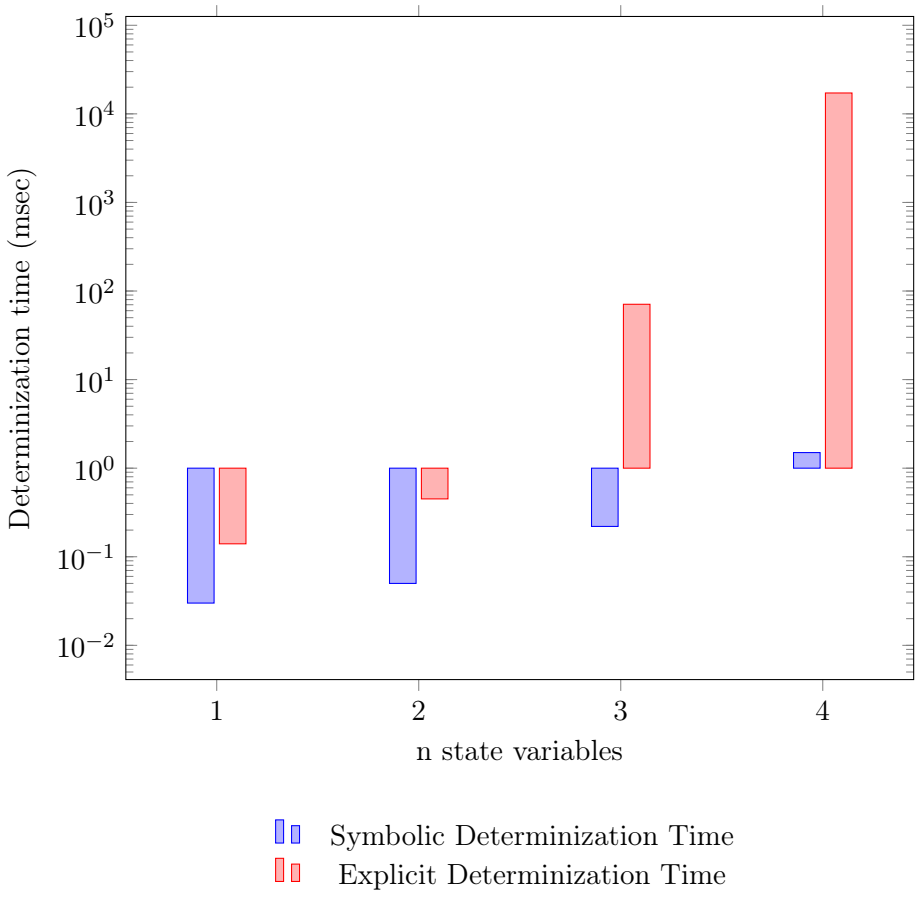


Figure 3.7: *Histogram for table 3.5*

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	29.42	0.02	3	0.06
2	4	4	32.17	0.05	6	0.56
3	8	8	37.36	0.18	20	22.48
4	16	16	58.64	0.89	168	2711.17
5	32	32	159.22	4.15	–	–
6	64	64	421.84	20.78	–	–
7	128	128	2121.25	122.92	–	–
8	256	256	10260.90	526.28	–	–
9	512	512	52041.00	2677.29	–	–
10	1024	1024	284437.58	11745.27	–	–

Table 3.6: *Variation automata with omitted state variable q_n*

Looking closer at the above automata, it becomes clear that we can omit state variable q_n in that we use the following automaton instead:

$$\mathcal{A}_{\exists}\left(\{q_0,\ldots,q_{n-1}\},\mathsf{true},\left(\bigvee_{i=0}^{n-1}(a_i\wedge q_i)\right)\wedge\bigwedge_{i=0}^{n-1}(q_i\leftrightarrow b_i\wedge \mathsf{X}q_i),\mathsf{true}\right)$$

The results are presented in table 3.6. Apart from the result, fig. 3.9 presents an example of the above automata with with $n = 2$. The inputs have been encoded differently for the limitations on the visualization part. Therefore, the definitions for the input variables will be given below the diagram of the automata.

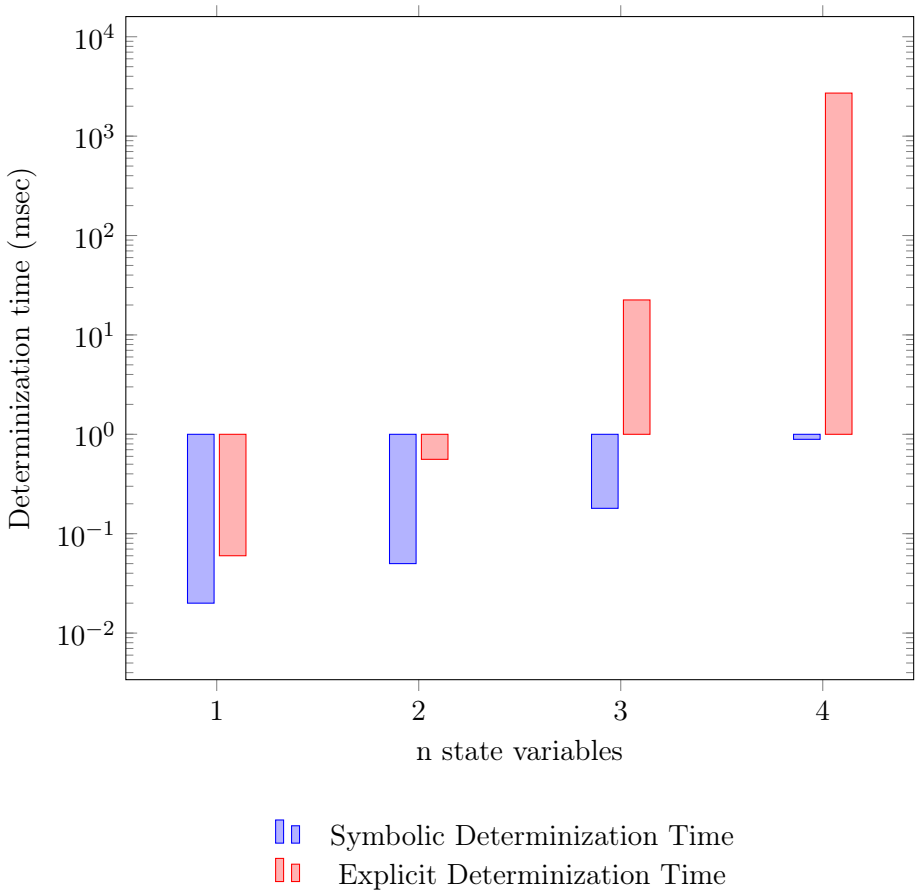


Figure 3.8: *Histogram for table 3.6*

The computations of both the symbolic as well as the explicit subset construction are faster, but the number of states of the generated deterministic automaton are the same.
Using watchdogs which already give us an almost deterministic automaton from the beginning is not straightforward for this example. The reason for this is that the watchdogs for the inner temporal operators have to be started not yet at initial time but at later time that is determined by the outer temporal

operator. The use of deterministic watchdogs would contradict to the need of a double exponential number of states for any deterministic automaton.

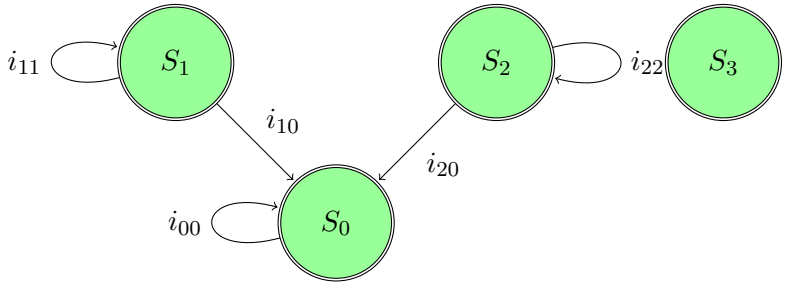


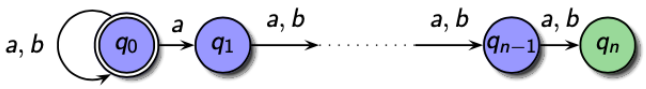
Figure 3.9: *Example of nondeterministic automaton*

Given input variables a_0, a_1, b_0, b_1, the input encodings shown on 3.9 define

- i_{00} Transition from S_0 to S_0 with input $\{a_0, a_1, b_0, b_1; a_0, !a_1, b_0, b_1; !a_0, a_1, b_0, b_1\}$
- i_{11} Transition from S_1 to S_1 with input $\{a_0, a_1, b_0, b_1; a_0, a_1, !b_0, b_1; !a_0, a_1, b_0, b_1; !a_0, a_1, !b_0, b_1\}$
- i_{10} Transition from S_2 to S_2 with input $\{\ a_0, a_1, !b_0, b_1; !a_0, a_1, !b_0, b_1\}$
- i_{22} Transition from S_1 to S_0 with input $\{a_0, a_1, b_0, b_1; a_0, a_1, b_0, !b_1; a_0, !a_1, b_0, b_1; a_0, !a_1, b_0, !b_1\}$
- i_{20} Transition from S_2 to S_0 with input $\{a_0, a_1, b_0, !b_1; a_0, !a_1, b_0, !b_1\}$

3.3.3 Experiment 3

A well-known worst case example for the subset construction on automata on finite words is the following automaton:



This automaton accepts a word if and only if the n-th last letter of the word is a letter a. As the automaton reads the words from left to right, it does however not know whether there are still further $n-1$ inputs when reading a letter a in the initial state. The nondeterministic automaton therefore has to guess this, and if the guess was right, it can accept the word which is sufficient for a nondeterministic automaton. The corresponding deterministic automaton must consider both the transition to stay in the initial state and the transition to move forward to state q_1. Since this repeats n times, the deterministic automaton obtains an exponential number of states that remember all words $a\{a,b\}^k$ for $k = 0, \ldots, n-1$.

To convert this example to an ω-automaton, we may consider the formulas $\mathsf{F}(\neg a \wedge \mathsf{X}^n b)$ and $\mathsf{G}(a \rightarrow \mathsf{X}^n b)$. For the safety formula, we generate the following ω-automaton:

$$\mathcal{A}_\exists \left(\{q_0, \ldots, q_n\}, q_n, \left(\begin{array}{l} (q_0 \leftrightarrow b) \wedge \\ \left(\bigwedge_{i=1}^{n-1} (q_i \leftrightarrow \mathsf{X} q_{i-1})\right) \wedge \\ (q_n \leftrightarrow (a \rightarrow q_{n-1}) \wedge \mathsf{X} q_n) \end{array} \right), \mathsf{true} \right)$$

For these automata, we obtain the following results for the symbolic and explicit subset constructions:

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	23.14	0.02	2	0.03
2	4	4	15.51	0.04	3	0.06
3	8	8	31.69	0.13	5	0.13
4	16	16	21.10	0.87	9	0.36
5	32	32	27.67	2.82	17	1.24
6	64	64	40.62	12.34	33	4.06
7	128	128	64.25	54.86	65	17.02
8	256	256	115.65	303.06	129	148.36
9	512	512	218.33	1245.18	257	735.50
10	1024	1024	447.27	5601.58	513	3493.95
11	2048	2048	870.47	23997.53	1025	11876.19

Table 3.7: *Experimental results for experiment 3*

As can be seen, the explicit subset construction is more efficient in this example, since it scales essentially with the number of states since we have for all n only two inputs a, b while the number of inputs was growing also exponentially with n in the previous examples. The generated deterministic automaton has even only half as many states as the given nondeterministic automaton.
The automaton can also be implemented more efficiently in that we implement with n state variables a counter for n-bit binary words that can therefore represent numbers up to 2^n-1. We also use only one boolean input variable a to denote the two input letters. We therefore consider automata for the formulas $\mathsf{F}(a \wedge \mathsf{X}^{2^n-2} a)$ and $\mathsf{G}(a \rightarrow \mathsf{X}^{2^n-2} \neg a)$. Using state variables $q_0, \ldots, q_{n-1}$, we interpret the values of the state variables are binary numbers $(q_{n-1 \ldots q_0})$ and start in the initial state $(q_{n-1 \ldots q_0}) = 0 \ldots 0$, i.e., with the initial condition $\bigwedge_{i=0}^{n-1} \neg q_i$.

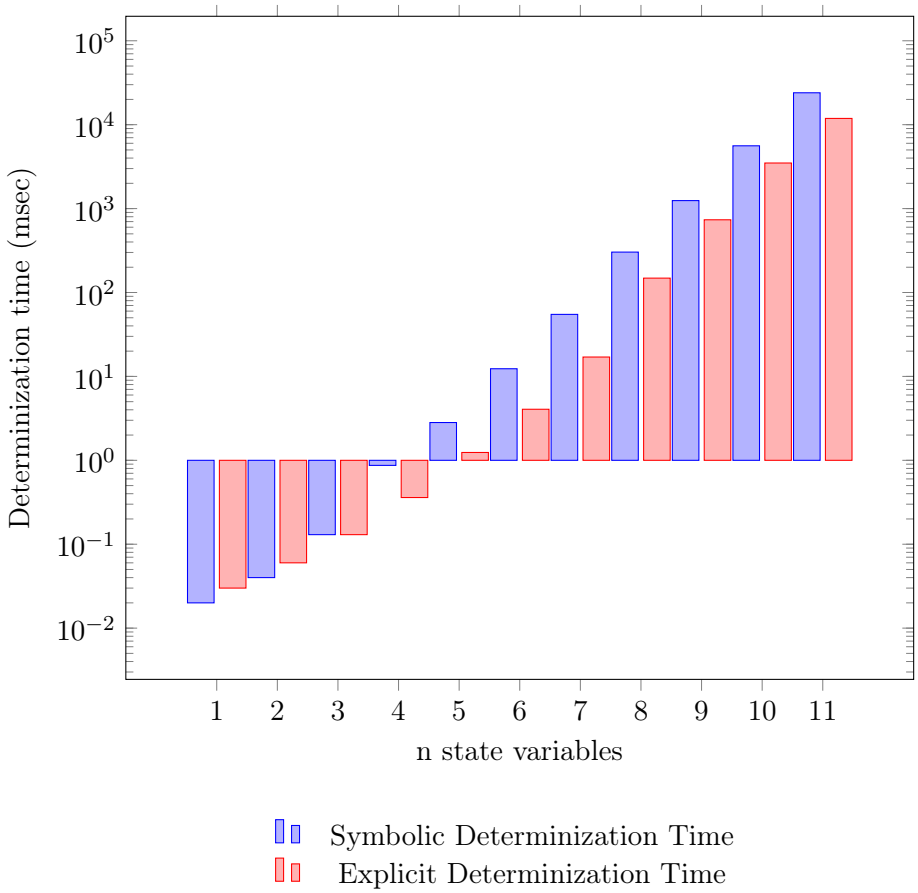


Figure 3.10: *Histogram for table 3.7*

The transition relation consists of the following parts that are combined in a disjunction to implement a large sequence of states counting from $0\ldots0$ to $1\ldots1$ in binary:

- $\left(\bigwedge_{i=0}^{n-1} \neg q_i\right) \wedge \left(\bigwedge_{i=0}^{n-1} \neg \mathsf{X} q_i\right)$ implements a self-loop on the initial state for any input
- $\left(\bigwedge_{i=0}^{n-1} \neg q_i\right) \wedge a \wedge \mathsf{X} q_0 \wedge \left(\bigwedge_{i=1}^{n-1} \neg \mathsf{X} q_i\right)$ implements a transition from the initial state $\bigwedge_{i=0}^{n-1} \neg q_i$ (representing number 0) to state $q_0 \wedge \bigwedge_{i=0}^{n-1} \neg q_i$ (representing number 1)
- $\neg q_0 \wedge \left(\bigwedge_{i=1}^{n-1} q_i\right) \wedge a \wedge \left(\bigwedge_{i=0}^{n-1} \mathsf{X} q_i\right)$ implements the transition from state $1\ldots10$ to the final state $1\ldots1$ which can only be taken with input a
- $\left(\bigwedge_{i=0}^{n-1} q_i\right) \wedge \left(\bigwedge_{i=0}^{n-1} \mathsf{X} q_i\right)$ implements a self-loop on the final state $1\ldots1$ for any input
- for all states that are neither the initial state $0\ldots0$, nor the state right before the final state $1\ldots10$, nor the final state $1\ldots1$, we add $\bigwedge_{i=0}^{n-1} (\mathsf{X} q_i \leftrightarrow q_i \oplus \bigwedge_{j=0}^{i-1} q_j)$ for incrementing the bits of the current state

For these automata, we obtain the following results in table 3.8.

n	2^n	#States (NDet)	NDetReach time[msec]	SymDet time[msec]	#States (Det)	ExpDet time[msec]
1	2	2	27.37	0.02	2	0.10
2	4	4	15.10	0.07	8	0.09
3	8	8	16.64	0.53	128	4.57
4	16	16	23.23	2.31	32768	4024.72
5	32	32	90.98	11.55	–	–
6	64	64	50.60	62.77	–	–
7	128	128	108.57	516.33	–	–
8	256	256	237.37	1834.90	–	–
9	512	512	533.12	8760.91	–	–
10	1024	1024	1442.29	40462.79	–	–
11	2048	2048	5793.02	189076.15	–	–

Table 3.8: *Experimental results*

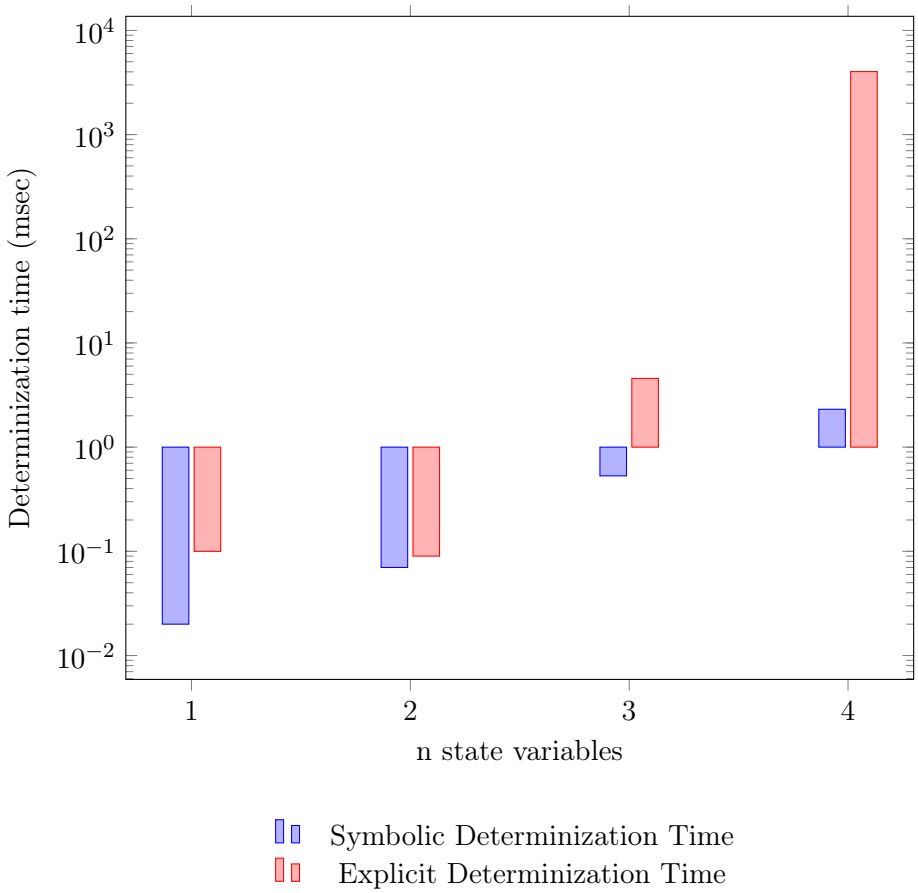


Figure 3.11: *Historgram for table 3.8*

For parameter n, we use n state variables, and all possible 2^n states of the generated nondeterministic automaton are indeed reachable which is clear by the construction of the automaton. The deterministic automaton generated by the symbolic subset construction has therefore 2^n state variables and is therefore able to encode 2^{2^n} possible states as in the previous examples.
The experimental results indicate that the generated deterministic automaton has $2^{(2^n-1)}$ reachable states, so that we have another example whose deterministic automaton has a doubly exponential growth. For example, for $n = 10$, the generated deterministic automaton has an incredible number of 2^{1023} many reachable states encoded by 1024 state variables.
The following figures are examples of the two automatas obtained from translating $\mathsf{G}(a \rightarrow \mathsf{X}^n b)$ and $\mathsf{G}(a \rightarrow \mathsf{X}^{2^n-2}\neg a)$.

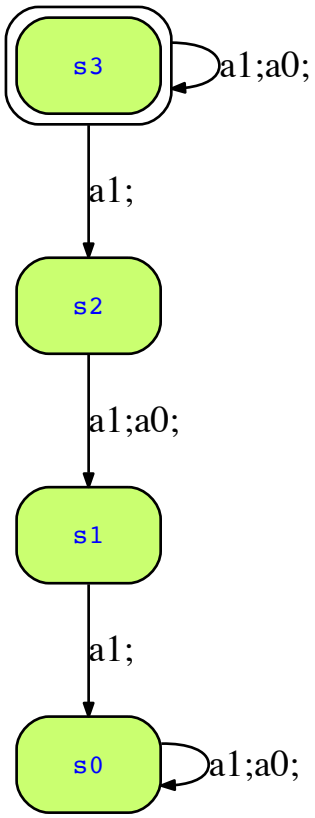


Figure 3.12: *Nondeterministic automata from* $\mathsf{G}(a \rightarrow \mathsf{X}^{2^n-2}\neg a)$

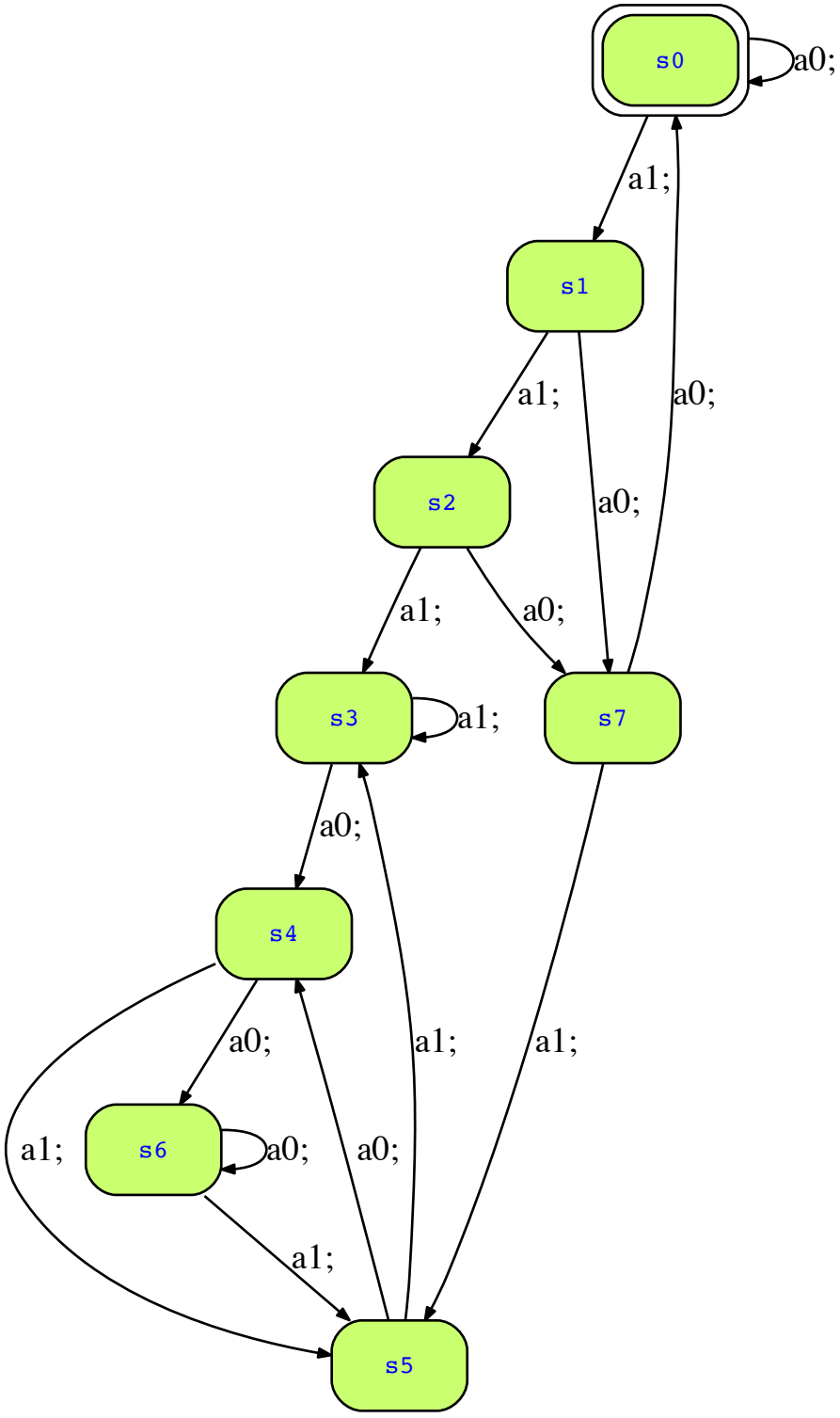


Figure 3.13: *Deterministic automaton of nondeterministic automata in 3.12*

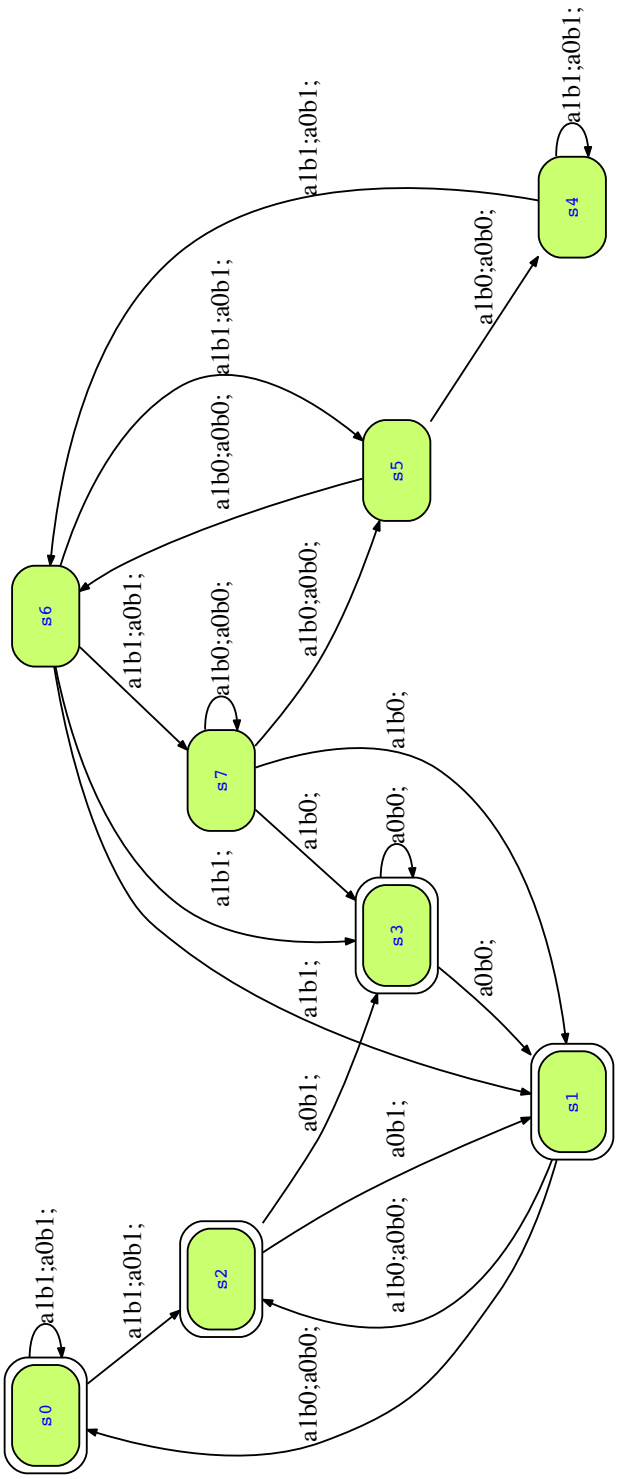


Figure 3.14: *Nondeterministic automaton from* $\mathsf{G}(a \rightarrow \mathsf{X}^n b)$

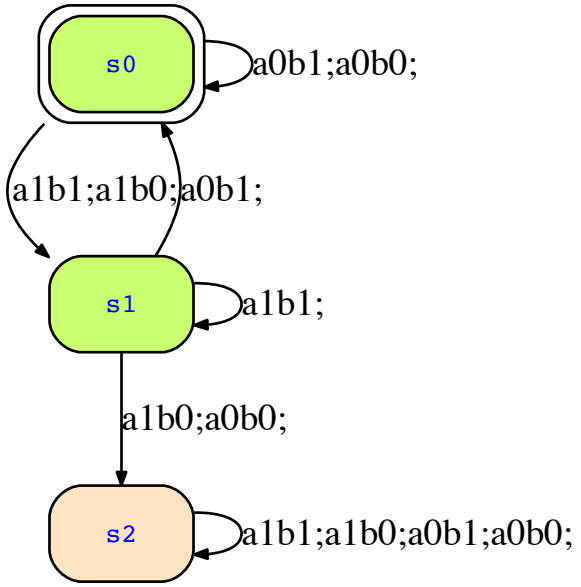


Figure 3.15: *Deterministic automaton of nondeterministic automata in 3.14*

4 Conclusions

This chapter summarizes the work presented in the thesis, limitations faced and what future improvements we can bring.

4.1 Summary

A reactive system faces repeatedly stimuli input from the environment. Notwithstanding, the user expects out of this system a specific behavior. Using various procedures, we can build a system from a higher-level specification. In doing so, we are faced with the problem of finding a language which can be expressive in regard to the temporal behaviors of the system. Among the proposed specification logics, linear temporal logic ($\mathsf{LTL}_)$ is a widely popular technique for formal verification.
To facilitate the verification problem of LTL formulas, they have to be translated to a nondeterministic automata. However, there are cases where a deterministic automata is needed. Hence, various procedures have been proposed for the determinization process mentioned extensively in section 2.4. Among them, our focus in this work, was on the subset construction and breakpoint construction.
The known algorithms, either have an explicit determinization procedure, which means that the procedure enumerates all the states of the deterministic automaton, or are difficult to implement such as the case of Safra's procedure. Such implementation brings a runtime blow-up, resulting in a not efficient determinization procedure and not able to be implemented in practice or can handle only small formulas. Based on the findings of the paper 'Generating Deterministic ω-Automata for most LTL formulas by the Breakpoint Construction', Morgenstern, Schneider,and Lamberti propose a method for generating deterministic ω-automata for most LTL formulas. Leveraging the temporal logic hierarchy, most of LTL formulas can be translated in deterministic automata by using only the subset or breakpoint construction as findings have shown in [SST04].
The method we exploited, based also on the research by [MSL08], is implementing the subset and breakpoint construction by symbolic representation. Making use of propositional logic, BDDs, temporal logic and ω-automata hierarchy, we implemented the two determinization algorithms with the Averest framework.
The goal of the work was to evaluate the symbolic representation of the determinization procedures with examples where a translation of the LTL formula to a deterministic automaton has a double exponential runtime. For the evaluation, as mentioned previously, we implemented the two determinization

algorithms and compared the determinization procedures of different scenarios with the runtime of explicit determinization procedure. The experiments have shown that the explicit determinization procedure has an exponential growth in runtime, even double exponential as seen in 3.3.3, while the symbolic determinization procedure is significantly more efficient. The efficiency of the symbolic determinization procedure was in a way "expected" since different from the other procedure we don't have to explicitly enumerate the number of states for the deterministic automaton. Another remark regarding the results of the various tables is that after a specific number of state variables given, the explicit determinization procedures terminates without computing anything for lack of memory in the personal system used in this work to compute the experiments. Faced with these situation, it raises the question on how powerful the system needs to be to make such computations.

4.2 Challenges and Work Limitations

One of the main challenges for this work was finding a way to better showcase the differences between the explicit determinization procedure and the symbolic determinization procedure. Firstly, we thought to compute the average runtime of the symbolic determinization procedure by translating randomly generated LTL formulas. However, we are faced with the issue that some generated formulas are not part of the temporal logic hierarchy. The solution would have been to either try to translate them in equivalent formulas which are part of the hierarchy or ignore them during the computation. Nevertheless, both of these solutions are limitations for the work. We can't claim that the symbolic determinization works for all LTL formulas since following the second statement of ignoring them, would not support the claim, while for the first statement we could limit the generation of formulas only to LTL formulas from the hierarchy classes. Trying to implement the limiting behavior also has its own concerns. In the case of safety properties, we can try to translate the formulas by making the operators from strong to weak, while for co-Büchi properties, even if we try to use the same method, the resulting property may not even be part of the co-Büchi class, which is not a desired result. Furthermore, the average runtime for the determinization procedure would not be such an "interesting" finding in the scope of the thesis.

4.3 Future Work

The previous section gave us an idea about the limitations of the thesis, but at the same time it offered an opportunity to expand on further research in this topic area. We mentioned that the translation of randomly generated LTL formulas was a bit of a problem. An algorithm which tackles the translation from an LTL formula to another LTL formula, part of the hierarchy classes, can be of interest for other areas of implementation.
Coming to the variety of experiments, more studies have to be conducted for more LTL formulas to see the behavior of the symbolic determinization

procedures for the various temporal logic classes. As for the system limitation, more experiments can be conducted for more state variables given to see in a way the limitations of the procedures.

Bibliography

[AL04] Rajeev Alur and Salvatore La Torre. "Deterministic generators and games for LTL fragments". In: *ACM Transactions on Computational Logic* 5.1 (2004), pp. 1–15.

[Bie99] Armin Biere. "Symbolic model checking without BDDs". In: *Tools and algorithms for the construction and analysis of systems*. Springer. 1999, pp. 193–207.

[Bry86] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[Büc60] J.R. Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: (1960).

[Bur+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. "Symbolic model checking: 1020 states and beyond". In: *Information and computation* 98.2 (1992), pp. 142–170.

[CE82] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3.

[CGH97] Edmund M Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. "Another look at LTL model checking". In: *Formal Methods in System Design* 10 (1997), pp. 47–71.

[CGP94] Edmund M Clarke, Orna Grumberg, and Doron A Peled. "Model checking". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542.

[CMP92] Edward Chang, Zohar Manna, and Amir Pnueli. "Characterization of temporal property classes". In: *Automata, Languages and Programming: 19th International Colloquium Wien, Austria, July 13–17, 1992 Proceedings 19*. Springer. 1992, pp. 474–486.

[Dix+13] Clare Dixon, Javier Esparza, Matthew Florian, and Verena Wolf. "Determinization via zone-based subsets". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 607–622.

[EH86] E. Allen Emerson and Joseph Y. Halpern. ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic". In: *J. ACM* 33.1 (Jan. 1986), pp. 151–178. ISSN: 0004-5411. DOI: 10.1145/4904.4999. URL: https://doi.org/10.1145/4904.4999.

[HKK95] Thomas A Henzinger, Christoph M Kirsch, and S Rao Kosaraju. "Antichains: A new algorithm for checking universality of finite automata". In: *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. ACM. 1995, pp. 403–412.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2nd. Cambridge University Press, 2004.

[KB06] Joachim Klein and Christel Baier. "Experiments with deterministic ω-automata for formulas of linear temporal logic". In: *Theoretical Computer Science* 363.2 (2006), pp. 182–195.

[KC87] Robert P Kurshan and Edmund M Clarke. "Automatic Synthesis of Controllers for Safety Specifications". In: *Proceedings of the IEEE* 75.8 (1987), pp. 1098–1124.

[Koz83] Dexter Kozen. "Results on the propositional µ-calculus". In: *Theoretical Computer Science* 27.3 (1983), pp. 333–354.

[Kri63] Saul A Kripke. "Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi". In: *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 9.5-6 (1963), pp. 67–96.

[MH84] Satoru Miyano and Takeshi Hayashi. "Alternating finite automata on ω-words". In: *Theoretical Computer Science* 32.3 (1984), pp. 321–330.

[Mor10] Andreas Morgenstern. "Symbolic Controller Synthesis for LTL Specifications". doctoralthesis. Technische Universität Kaiserslautern, 2010. URL: https://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-25721.

[MS08] Andreas Morgenstern and Klaus Schneider. "From LTL to symbolically represented deterministic automata". In: *Verification, Model Checking, and Abstract Interpretation: 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008. Proceedings 9*. Springer. 2008, pp. 279–293.

[MS10] Andreas Morgenstern and Klaus Schneider. "Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis". In: *arXiv preprint arXiv:1006.1408* (2010).

[MSL08] Andreas Morgenstern, Klaus Schneider, and Sven Lamberti. "Generating Deterministic ω-Automata for most LTL Formulas by the Breakpoint Construction". In: 2008. URL: https://api.semanticscholar.org/CorpusID:13180276.

[MT96] Christoph Meinel and Thorsten Theobald. "Local encoding transformations for optimizing OBDD-representations of finite state machines". In: *Formal Methods in Computer-Aided Design*. Ed. by Mandayam Srivas and Albert Camilleri. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 404–418. ISBN: 978-3-540-49567-3.

[Pit06] Nir Piterman. "From Nondeterministic Buchi and Streett Automata to Deterministic Parity Automata". In: *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)* (2006), pp. 255–264. URL: https://api.semanticscholar.org/CorpusID:14122357.

[Pnu77] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[Rab72] Michael Oser Rabin. *Automata on infinite objects and Church's problem*. Vol. 13. American Mathematical Soc., 1972.

[RS59] Michael O. Rabin and Dana S. Scott. "Finite Automata and Their Decision Problems". In: *IBM J. Res. Dev.* 3 (1959), pp. 114–125. URL: https://api.semanticscholar.org/CorpusID:3160330.

[Saf92] Shmuel Safra. "Exponential determinization for ω-automata with strong-fairness acceptance condition". In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. 1992, pp. 275–282.

[Sal85] Arto Salomaa. "Deterministic automata on infinite objects". In: *Bulletin of the European Association for Theoretical Computer Science* 26 (1985), pp. 67–74.

[Sch01] Klaus Schneider. "Improving Automata Generation for Linear Temporal Logic by Considering the Automaton Hierarchy". In: *Logic Programming and Automated Reasoning*. 2001. URL: https://api.semanticscholar.org/CorpusID:6475556.

[Som16] Fabio Somenzi. "Antichain-Based Determinization Revisited". In: *ACM Transactions on Computational Logic (TOCL)* 18.1 (2016), pp. 1–23. DOI: 10.1145/2745801.

[SS05] Klaus Schneider and Tobias Schuele. "Averest: Specification, verification, and implementation of reactive systems". In: *Conference on Application of Concurrency to System Design (ACSD)*. Citeseer. 2005.

[SS06] Klaus Schneider and Tobias Schuele. "A Framework for Verifying and Implementing Embedded Systems." In: *MBMV*. 2006, pp. 242–247.

[SS97] Klaus Schneider and D. Schmid. "CTL and Equivalent Sublanguages of CTL". In: *Hardware Description Languages and their Applications: Specification, modelling, verification and synthesis of microelectronic systems IFIP TC10 WG10.5 International Conference on Computer Hardware Description Languages and their Applications, 20–25 April 1997, Toledo, Spain*. Ed. by Carlos Delgado Kloos and Eduard Cerny. Boston, MA: Springer US, 1997, pp. 40–59. ISBN: 978-0-387-35064-6. DOI: 10.1007/978-0-387-35064-6_4. URL: https://doi.org/10.1007/978-0-387-35064-6_4.

[SST04] Klaus Schneider, Jimmy Shabolt, and John G Taylor. *Verification of reactive systems: formal methods and algorithms*. Springer, 2004.

[Str82] Robert S Streett. "Propositional dynamic logic of looping and converse is elementarily decidable". In: *Information and control* 54.1-2 (1982), pp. 121–141.

[SW74] Ludwig Staiger and Klaus Wagner. "Automatentheoretische und Automatenfreie Charakterisierungen Topologischer Klassen Regulärer Folgenmengen." In: (1974).

[Tho90] Wolfgang Thomas. "Automata on infinite objects". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 133–191.

[VW86] M.Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In: *Symposium on Logic in Computer Science*. 1986.

[Wag79] Klaus Wagner. "On ω-regular sets". In: *Information and control* 43.2 (1979), pp. 123–177.