Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Advanced Saturation-based Model Checking

Master's thesis

Author
Vince Molnár

Supervisors
dr. Tamás Bartha,
András Vörös

December 21, 2014

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Molnár Vince*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. december 21.

Molnár Vince
hallgató

Kivonat

A lineáris temporális logikai (LTL) specifikációk verifikációjára számos explicit és szimbolikus modellellenőrzési technikát dolgoztak ki az elmúlt évtizedekben. Manapság a kutatások meghatározó iránya a két algoritmuscsalád előnyeinek kombinálása, legfőképp az ún. “on-the-fly” (menet közbeni) modellellenőrzés megvalósítása szimbolikus kódolást használva. Az LTL modellellenőrzés megvalósításához két problémát kell megoldani: ki kell számolni a rendszer állapotterének és a követelményt leíró automatának a szinkron szorzatát, majd ebben ellenpéldákat, vagyis hibás lefutásokat keresni. Utóbbi visszavezethető körök vagy erősen összefüggő komponensek keresésére a szorzatot leíró gráfban. Konkurens rendszerek esetén az ún. szaturációs algoritmus nagyon hatékony megoldásnak bizonyult a szimbolikus állapottér-felderítés problémájának megoldására.

Jelen diplomamunka célja egy olyan új megoldás bemutatása, amely a szaturációs algoritmus előnyeit mind a szorzat közvetlen kiszámításában, mind egy erősen összefüggő komponenseket kereső inkrementális fixpont-számító algoritmusban kiaknázza. A keresést kiegészítve explicit megoldások és absztrakció segítségével a bemutatott algoritmus menet közben az ellenpélda hiányát is megpróbálja bebizonyítani. Az eredmény egy “on-the-fly” működésű, inkrementális LTL modellellenőrző algoritmus ami a Model Checking Contest modelljein végzett kísérletek alapján jól skálázódik a vizsgálandó modell méretével.

Abstract

Efficient symbolic and explicit model checking approaches have been developed for the verification of linear time temporal logic (LTL) properties. Nowadays, several attempts have been made to combine on-the-fly search with symbolic encoding. Model checking LTL properties usually pose two challenges: one must compute the product between the state space of the system and the automaton model of the desired property, then look for counterexamples that is reduced to searching for fair loops or strongly connected components (SCCs) in state space of the product. In case of concurrent systems, the so-called saturation algorithm proved to be an efficient symbolic state space generation approach.

This thesis proposes a new approach that leverages the saturation algorithm both as an iteration strategy used to compute the product directly, as well as in a new incremental fixed-point computation algorithm to compute strongly connected components on-the-fly. Complementing the search for SCCs, explicit techniques and abstraction will be used to prove the absence of counterexamples. The result is an on-the-fly, incremental LTL model checking algorithm that proved to scale well with the size of models, as evaluation on models of the Model Checking Contest suggests.

Introduction

In 1968, a revolutionary conference was held by NATO, a conference that signals the birth of “software-engineering” [1]. Computer scientists realized that while software is getting "mainstream", there are no such paradigms that can ensure quality and safety like in other areas of engineering. Few things have changed since then, but today we face an even greater challenge. Computer systems have invaded our house, software runs in our phone, our car, in the traffic control system, in airplanes we travel by, even in industrial control systems found in factories or nuclear power plants. While programmers found the way to build critical infrastructure with proper design architectures, much less progress has been made in ensuring, verifying and proving the correctness and safety of such systems.

Among the various techniques that tried to tackle the above problem, formal methods represent a mathematical perspective. Formal methods can be used on a number of levels in software engineering. First of all, formal specifications can precisely characterize the requirements towards a system and provide the basis of applying other techniques. Formal development and verification are then used to ensure that the produced system really satisfies its specifications, raising the overall quality of the product in terms of dependability and robustness. A more heavyweight approach in formal methods is theorem proving, where highly trained personnel provide fully formal mathematical proofs with the aid of automatic theorem provers.

In the early ’80s, *model checking*, a promising new technique of formal verification was developed by E. M. Clarke and E. A. Emerson [26] and by J. P. Quielle and J. Sifakis [48]. They used *temporal logics* to specify the formal requirements and enumerated the possible behaviors of the system, checking them against the specification. Since then, there has been an enormous leap in the field of model checking. Various kinds of algorithms have been implemented for different temporal logics and model formalisms, and a lot of optimizations and special methods were devised to broaden the range of systems that can be verified this way.

A great challenge of model checking – even to this day – is the so-called *state space explosion problem.* Especially when a system has independent components or many variables, the number of states of the system can grow extremely large. To address this problem, *symbolic model checking* was introduced in the early ’90s [9]. In symbolic model checking, states and state transitions are not simply enumerated, but represented symbolically by functions, encoded in efficient data structures such as *decision diagrams.* Although this encoding can usually represent state spaces larger by several orders of magnitudes, the traditional *explicit state model checkers* can sometimes outperform symbolic approaches using the many devised optimizations that could not be implemented in symbolic model checking so far. The reason behind this mostly lies in graph traversal algorithms, because symbolic methods usually have to use breadth-first search instead of depth-first search.

In the early 2000s, G. Ciardo developed a new iteration strategy specially tailored to decision diagrams [14]. He called it the *saturation* iteration strategy, in which the search was driven by the structure of the decision diagram, processing nodes in it instead of the states it represented. Saturation proved to be very successful in verifying concurrent, asynchronous systems. These systems are common when modeling the interaction of individual components, and they are especially prone to the state space explosion problem.

In terms of temporal logic specifications, two formalisms became widespread. *Linear temporal logic* (LTL) was the first formalism introduced in formal specification due to its intuitive interpretation. *Computational tree logic* (CTL) became popular with the first model checker algorithms, because the worst-case complexity of CTL model checking is linear in the size of the state space (cf. state space explosion) and the specification as well. LTL model checkers appeared later, with exponential complexity in the size of the specification [49]. However, the semantics of LTL provided an opportunity to design the algorithms to work in an on-the-fly manner, terminating immediately when a proof is found. CTL model checkers rarely possess such a feature, because many of them explore the state space first, then compute fixed points according to the subexpressions of the specification formula.

On the other hand, the latter approach made CTL model checking suitable for a symbolic implementation, since fixed point computations on decision diagrams were easy to perform. Symbolic LTL model checkers also appeared, but the on-the-fly manner of the explicit algorithms relied on depth-first search, which is cumbersome to implement in a symbolic way. This is what makes symbolic LTL model checking an interesting challenge, one that has been therefore tackled by many researchers in different ways.

This thesis aims to *1)* give the reader a deeper insight into the current state of symbolic LTL model checking, as well as to *2)* propose new algorithms based on the saturation iteration strategy and the presented related work.

The layout of the thesis is aligned according to the stated two goals. After setting the historical background and motivation here, the theoretical background of LTL model checking and symbolic techniques – including the saturation iteration strategy – are presented in Chapter 1. Chapter 2 presents the related work in LTL model checking, summarizing various approaches implemented in state-of-the-art tools. The main contributions are presented in Chapters 3 and 4 in the form of new algorithms solving the LTL model checking problem. The prototype implementation of the presented algorithm is evaluated in Chapter 5. Finally, Chapter 6 concludes the results.

Chapter 1

Background

This chapter will introduce the basic concepts behind the solutions presented in this thesis. Since the main emphasis is on symbolic LTL model checking, the reader should first get familiar with a few modeling formalisms (Section 1.1) and a family of specification languages called temporal logics (Section 1.2). Section 1.3 will present automata theory, which is closely related to linear time temporal logic and model checking. Based on the presented artifacts, the model checking problem itself will be characterized in Section 1.4. The description of the saturation iteration strategy closes the chapter by introducing symbolic representation and the algorithm to generate the state space of a system with saturation.

Common notations

Throughout this thesis, the following notations will be used for sets of numbers and Boolean symbols:

- $\mathbb{N}$ is the set of *natural numbers* $\{0, 1, 2, \ldots\}$;
- $\mathbb{Z}$ is the set of *integers* $\{\ldots, -1, 0, 1, \ldots\}$;
- $\mathbb{Z}^+$ is the set of *positive integers* $\{1, 2, \ldots\}$;
- $\mathbb{B}$ is the set of *Boolean values* consisting of two symbols: true ($\top$) and false ($\bot$).

A *sequence of elements* from a set Σ is a mapping $f : \{0, 1, 2, \ldots, |f|\} \to \Sigma$ where $|f|$ is the length of the sequence. An element at a certain position k is therefore $f(k)$. Subsequences of f starting from position k are denoted by $f[k]$, where $f[k](i) = f(k+i)$ for all $0 \leq i \leq |f| - k = |f[k]|$. A sequence is called infinite if it maps from $\mathbb{N}$. An infinite sequence has an infinite length $|f| = \infty$. The sets of possible sequences with a certain length are denoted by:

- Σ^n for the set of sequences with length n;
- Σ^* for the set of finite sequences;
- Σ^ω for the set of infinite sequences.

In this thesis, paths, runs and words are typically defined as sequences. In case of words, the set Σ is called an alphabet and its elements are letters. A word over an alphabet denotes a sequence of letters from the alphabet.

1.1 Modeling Formalisms

Formal verification of systems cannot be possible without defining a model of the system with mathematical precision. Such a model is called formal if it has a well-defined syntax and precise semantics. There are low-level formalisms, which are usually built on simple mathematical constructs such as sets, relations and functions. The most common model types are based on graphs with annotations or labels on either arcs, nodes or both. Low-level formalisms are easy to handle mathematically, while they are usually hard to comprehend for humans and are impractical for direct modeling of a system. *Kripke structures* are presented in Section 1.1.1 as a good example that is widely used in model checking.

Higher-level formalisms address the practical modeling difficulties by providing constructs that are closer to the modeled phenomena. These models also have formal semantics, usually given by a transformation to a lower-level formalism. A common higher-level formalism for modeling concurrent systems is Petri nets. Since many examples in this thesis will be modeled by Petri nets, Section 1.1.2 will introduce them briefly.

1.1.1 Kripkes Structures

Kripke structures [40] are directed graphs with labeled nodes. Nodes represent different states of the modeled system, while arcs denote state transitions. Each state is labeled with properties that hold in that state. This way, paths in the graph represent possible behaviors of the system. Labels along the paths give the opportunity to reason about sequences of valuations of Boolean propositions (see temporal logics in Section 1.2).

Definition 1 (Kripke structure). Given a set of atomic propositions $AP = \{p, q, \ldots\}$, a (finite) Kripke structure is a 4-tuple $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, L \rangle$, where:

- $\mathcal{S} = \{s_1, \ldots, s_n\}$ is the (finite) set of states;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation consisting of state pairs (s_i, s_j);
- $L : \mathcal{S} \to 2^{AP}$ is the labeling function that maps a set of atomic propositions to each state. ▪

When used in model checking, the transition relation is often defined as left-total, i. e., for all $s_i \in \mathcal{S}$, there exists $s_j \in \mathcal{S}$ such that $(s_i, s_j) \in \mathcal{R}$. In that case, a *path* in M can be defined as an infinite sequence $\rho \in \mathcal{S}^\omega$ with $\rho(0) \in \mathcal{I}$ and $(\rho(i), \rho(i+1)) \in \mathcal{R}$ for every $i \geq 0$. The infinite sequence of sets of atomic propositions assigned to the states in ρ by L is called a *word* on the path and is denoted by $L(\rho) \in (2^{AP})^\omega$. The language that the Kripke structure M can *produce* (i. e., the set of all possible words on every path of M) is $\mathcal{L}(M)$.

Without the labeling function, the structure is called a *transition system*. Since many algorithms ignore the labeling, sometimes transition systems are used in definitions instead of Kripke structures. Even then, the labeling function is usually implicitly there, in the form of different properties of a state or some related information.

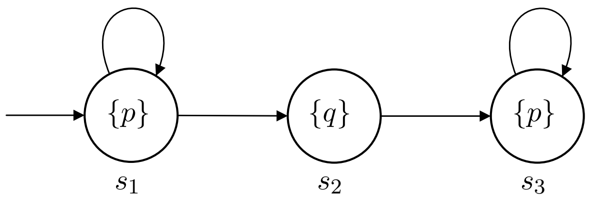


Figure 1.1. *A simple finite Kripke structure with a left-total transition relation.*

Example 1. *Figure 1.1 shows a simple finite Kripke structure with left-total transition relation. States are denoted by circles, while arrows between them are transitions of the Kripke structure. The initial state is s_1, this is denoted by the arc without a source. Labels are written inside the states.*

A path of the Kripke structure can be $\rho = s_1 s_1 s_2 s_3 s_3 \ldots$, its corresponding word is $w = \{p\}\{p\}\{q\}\{p\}\{p\}\ldots$. This Kripke structure produces words with infinitely many letters of $\{p\}$ and at most one $\{q\}$.

1.1.2 Petri nets

A Petri net [45] is a mathematical modeling language applicable to many kinds of systems. It is traditionally used to describe and study concurrent, asynchronous and nondeterministic systems. Besides the mathematical definition and semantics, Petri nets have a graphical representation similar to flow charts or block diagrams, with tokens to simulate dynamic behavior. Despite their simple definition and semantics presented below, Petri nets have a very deep mathematical theory that is out of the scope of this thesis.

A Petri net is a directed, weighted, bipartite graph with the partite sets called *places* and *transitions*. Places have a visual representation of a circle, while transitions are drawn as bars or boxes. *Arcs* are going from places to transitions or vice versa, labeled with a positive integer indicating their *weights*. The weight k of an arc can be interpreted as k parallel arcs (with weight 1) between the same place and transition. The state of a Petri net is called a *marking*, which assigns a nonnegative integer to each place of the net. State changes occur when transitions *fire*.

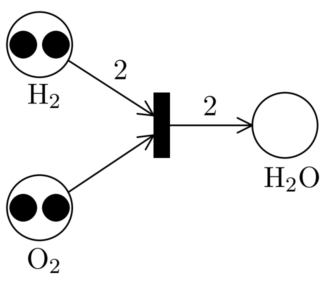
Definition 2 (Petri net). A Petri net is a 5-tuple $PN = \langle P, T, E, w, M_0 \rangle$, where:

- $P = \{p_1, \ldots, p_n\}$ is the finite set of places;
- $T = \{t_0, \ldots, t_m\}$ is the finite set of transitions;
- $E \subseteq (P \times T) \cup (T \times P)$ is the set of directed arcs;
- $w : E \to \mathbb{Z}^+$ is the weight function;
- $M_0 : P \to \mathbb{N}$ is the initial marking;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$ must also hold. ▪

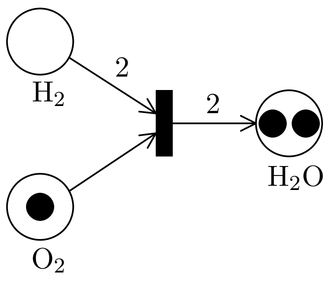
Input (output) places of a transition $t \in T$ is denoted by $\bullet t = \{p \mid (p, t) \in E\}$ (and $t\bullet = \{p \mid (t, p) \in E\}$ respectively). State changes are described by the following *firing rule*: *1)* A transition t is *enabled* if each input place $p \in \bullet t$ is marked with at least $w(p, t)$ tokens. *2)* An enabled transition may fire nondeterministically, or may not fire at all. *3)* The firing of an enabled transition t removes $w(p, t)$ tokens from each input place $p \in \bullet t$ and puts $w(t, p)$ tokens to each output place $p \in t\bullet$.

The semantics of Petri nets in terms of Kripke structures are given by the following transformation. Let the states of the Kripke structure be the (possibly infinite) set of possible markings $\mathcal{S} = \{M_0, M_1, \ldots\}$ with the initial marking as the initial state $\mathcal{I} = \{M_0\}$. A pair of markings (M_i, M_j) is in the transition relation of the Kripke structure iff there exists a transition in the Petri net that is enabled in M_i and its firing in M_i results in M_j.

A marking M is reachable if there exists a (possibly infinite) path in the Kripke structure that ends in M. A Petri net is bounded if there exists a nonnegative integer b such that there is no reachable marking in which any of the places $p \in P$ is marked with more than b tokens. Kripke structures of bounded Petri nets are always finite. The Kripke structure of a Petri net (or other higher-level model) is often referred to as its *state space*. In this context, the transformation to the Kripke structure (or transition system) is called *state space generation* or *state space exploration*.



(a) Before firing the transition.



(b) After firing the transition.

Figure 1.2. *Petri net modeling the chemical reaction of hydrogen and oxygen.*

Example 2. *Figure 1.2 shows a Petri net modeling the reaction of hydrogen and oxygen (before formalizing it, Petri initially used Petri nets to visualize chemical reactions). Firing the transition in the middle changes the marking according to the weights of the arcs.*

1.2 Temporal Logics

Temporal logics [20] are widely used to reason about paths of Kripke structures. In this sense, time is only represented by the order of states in a path, no explicit clock values appear in the logic itself.

In the family of temporal logics, the most general formalism for characterizing infinite traces of system behaviors is the μ-calculus. However, due to its cumbersome interpretation, it is rarely used as a real-life specification language. CTL* is a less general but more intuitive subset of the μ-calculus, using temporal operators and path quantifiers to reason about possible behaviors and temporal relations. Nevertheless, the most widespread temporal logics in formal specification are two of its subsets, linear temporal logic (LTL) and computational tree logic (CTL).

Historically, LTL and CTL were developed separately. CTL* was designed as a combination of the two, and later it was proved that both three can be translated into the μ-calculus. The following sections present CTL*, CTL and LTL both formally and informally. Since the classic scheme of LTL model checking involves a translation to automata, it is not strictly necessary to delve into the formal definitions in order to understand the presented model checking algorithms later.

1.2.1 CTL*

CTL* [28] is a branching-time logic: every nondeterministic choice branches the time model to form a tree structure of possible futures. The tree can be regarded as unfolded paths of a Kripke structure starting from a certain state. This model motivates the operators of CTL* – besides the operators of propositional logic, CTL* defines *temporal operators* and *path quantifiers* to reason about paths and branches, respectively.

Temporal operators are:

- $\mathsf{X}\ \varphi$: the *next state* operator, φ must hold in the next state along the path;
- $\mathsf{F}\ \varphi$: the *eventually* (or future) operator, φ must hold in some future state along the path;
- $\mathsf{G}\ \varphi$: the *globally* operator, φ must hold in every state of the path;
- $[\psi\ \mathsf{U}\ \varphi]$: the *until* operator, φ most hold in some future state, but until then, ψ must hold in every state;
- $[\psi\ \mathsf{R}\ \varphi]$: the *release* operator, φ must hold in every state along the path which is not after a state in which ψ was true.

Path quantifiers are:

- $\mathsf{A}\ \varphi$: the *universal* path quantifier, φ must hold for every paths starting in the current state;

- $\mathsf{E}\ \varphi$: the *existential* path quantifier, φ must hold for some path starting in the current state.

In CTL*, these operators can be mixed almost arbitrarily. For a graphical explanation of temporal operators, observe Figure 1.4 is Section 1.2.3.

Definition 3 (Syntax of CTL*). The formal syntax of CTL* is given by the following BNF grammar, where $p \in AP$ is an atomic proposition:

$$\Phi ::= \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \Rightarrow \Phi \mid \Phi \Leftrightarrow \Phi \mid \mathsf{A}\phi \mid \mathsf{E}\phi$$
$$\phi ::= \Phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \mathsf{X}\ \phi \mid \mathsf{F}\ \phi \mid \mathsf{G}\ \phi \mid [\phi\ \mathsf{U}\ \phi] \mid [\phi\ \mathsf{R}\ \phi]$$

A well-formed CTL* formula is generated by the symbol Φ. Formulae generated by this symbol are called *state formulae*, while those generated by ϕ are *path formulae.* ▪

Apart from the well-known equivalences of logical operators, the following equivalences apply to temporal operators and path quantifiers:

- $\neg\mathsf{X}\ \varphi \equiv \mathsf{X}\ \neg\varphi$
- $\neg\mathsf{F}\ \varphi \equiv \mathsf{G}\ \neg\varphi$
- $\neg[\psi\ \mathsf{U}\ \varphi] \equiv [\neg\psi\ \mathsf{R}\ \neg\varphi]$
- $\mathsf{F}\ \varphi \equiv [\top\ \mathsf{U}\ \varphi]$
- $\mathsf{G}\ \varphi \equiv [\bot\ \mathsf{R}\ \varphi]$
- $\mathsf{A}\varphi \equiv \neg\mathsf{E}\neg\varphi$

Using these equivalences, it is possible to construct a minimal set of connectives that can define the other operators. Such a set is $\{\top, \neg, \wedge, \mathsf{A}, \mathsf{X}, \mathsf{U}\}$.

Definition 4 (Semantics of CTL*). The formal semantics of CTL* is defined with respect to a Kripke structure M with a left-total transition relation (i. e., every path ρ of M is infinite).

A state s of M satisfies the state formula Φ, denoted by $(M, s) \models \Phi$ if its a valid model of the formula. This is defined inductively:

1. $(M, s) \models \top$;
2. $(M, s) \models p$ iff $p \in L(s)$;
3. $(M, s) \models \neg\Phi$ iff $(M, s) \not\models \Phi$;
4. $(M, s) \models \Phi_1 \wedge \Phi_2$ iff $(M, s) \models \Phi_1$ and $(M, s) \models \Phi_2$;
5. $(M, s) \models \mathsf{A}\ \phi$ iff $\rho \models \phi$ for all paths ρ of M starting in s.

Let ρ be a path of M, and let $\rho[k]$ be a sub-path of ρ. The satisfaction of ϕ by ρ, denoted by $\rho \models \phi$ is also defined inductively:

1. $\rho \models \Phi$ iff $(M, s_0) \models \Phi$;
2. $\rho \models \neg\phi$ iff $\rho \not\models \phi$;
3. $\rho \models \phi_1 \wedge \phi_2$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$;
4. $\rho \models \mathsf{X}\ \phi$ iff $\rho[1] \models \phi$;
5. $\rho \models [\phi_1\ \mathsf{U}\ \phi_2]$ iff for some $k \geq 0$, $\rho[k] \models \phi_2$ and for all $0 \leq i < k$, $\rho[i] \models \phi_1$. ▪

1.2.2 CTL

Computational tree logic (often called branching-time logic) differs from CTL* in a single restriction of the syntax: valid operators consist of a single path quantifier followed by a single temporal operator [34]. This way, the possible operators (apart from the logical operators) are AX, AF, AG, AU, AR, EX, EF, EG, EU and ER[1]. For a graphical explanation of the meaning of these operators, observe Figure 1.3.

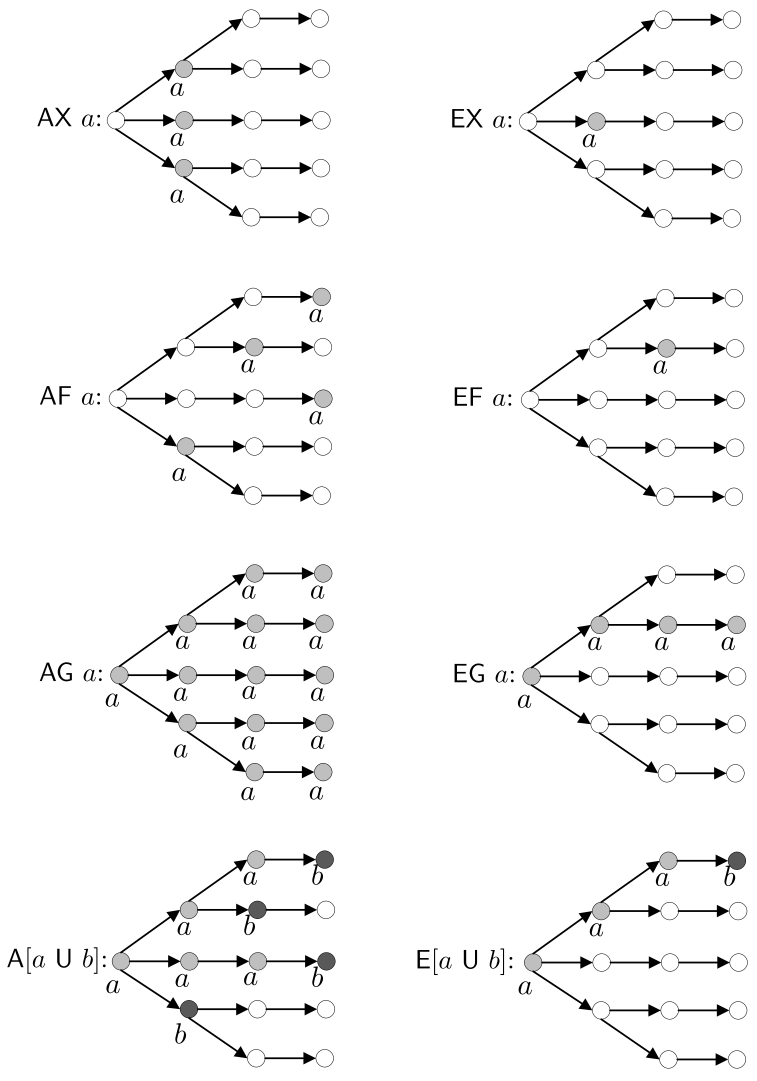


Figure 1.3. *Illustration of CTL operators.*

[1]The release operator is rarely used in CTL.

The main advantage of restricting the syntax is that this way, every possible subformula is a state formula. A way of model checking CTL properties is assigning new labels to the states by iteratively computing the sets of states in which the subfromulae hold. Since this approach processes only the states of a Kripke structure and not paths in it, such an algorithm can be linear in the number of states of the Kripke structure.

Definition 5 (Syntax of CTL). The formal syntax of CTL is given by the following BNF grammar, where $p \in AP$ is an atomic proposition:

$$\begin{aligned}\Phi ::= \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \Rightarrow \Phi \mid \Phi \Leftrightarrow \Phi \mid \\ \mathsf{AX}\ \Phi \mid \mathsf{AF}\ \Phi \mid \mathsf{AG}\ \Phi \mid \mathsf{A}[\Phi\ \mathsf{U}\ \phi] \mid \mathsf{EX}\ \Phi \mid \mathsf{EF}\ \Phi \mid \mathsf{EG}\ \Phi \mid \mathsf{E}[\Phi\ \mathsf{U}\ \Phi]\end{aligned}$$

A well-formed CTL formula is a generated by the symbol Φ. ▪

A possible minimal set of connectives in CTL is $\{\top, \neg, \wedge, \mathsf{EX}, \mathsf{EG}, \mathsf{EU}\}$. The other operators can be defined by the following equivalences (and the well-known logical equivalences):

- $\mathsf{AX}\ \varphi \equiv \neg\mathsf{EX}\ \neg\varphi$
- $\mathsf{AF}\ \varphi \equiv \neg\mathsf{EG}\ \neg\varphi$
- $\mathsf{AG}\ \varphi \equiv \neg\mathsf{EF}\ \neg\varphi$
- $\mathsf{EF}\ \varphi \equiv \mathsf{E}[\top\ \mathsf{U}\ \varphi]$
- $\mathsf{A}[\psi\ \mathsf{U}\ \varphi] \equiv \neg\mathsf{E}[\neg\varphi\ \mathsf{U}\ \neg\psi \wedge \neg\varphi] \wedge \mathsf{AF}\ \varphi$

The last equivalence states that there must always come a state in which φ holds, and no path can lead to a state in which neither φ nor ψ holds when there has not been a state on the path before in which φ held.

Definition 6 (Semantics of CTL). The formal semantics of CTL is defined with respect to a Kripke structure M with a left-total transition relation (i. e., every state s_i of M has at least one successor s_j such that $(s_i, s_j) \in \mathcal{R}$). The satisfaction relation $(M, s) \models \Phi$ is defined inductively:

1. $(M, s) \models \top$;
2. $(M, s) \models p$ iff $p \in L(s)$;
3. $(M, s) \models \neg\Phi$ iff $(M, s) \not\models \Phi$;
4. $(M, s) \models \Phi_1 \wedge \Phi_2$ iff $(M, s) \models \Phi_1$ and $(M, s) \models \Phi_2$;
5. $(M, s_0) \models \mathsf{EX}\ \phi$ iff for some path $\rho = s_0 s_1 s_2 \ldots$ of M, $(M, s_1) \models \phi$;
6. $(M, s_0) \models \mathsf{EG}\ \phi$ iff for some path $\rho = s_0 s_1 s_2 \ldots$ of M, $(M, s_k) \models \phi$ for every $k \geq 0$;
7. $(M, s_0) \models \mathsf{E}[\phi_1\ \mathsf{U}\ \phi_2]$ iff for some path $\rho = s_0 s_1 s_2 \ldots$ of M, there exists some $k \geq 0$ such that $(M, s_k) \models \phi_2$ and for all $0 \leq i < k$, $(M, s_i) \models \phi_1$. ▪

Example 3. *An example for a CTL formula is $\mathsf{AG}\ \mathsf{EF}\ (init)$. This is the so-called reset property, requiring that the system can always return to the initial state.*

1.2.3 LTL

At first glance, linear temporal logic [47] is very similar to path formulae in CTL*. In LTL, a formula cannot contain path quantifiers, and this is not only a syntactic restriction: the time model of LTL is linear, meaning that it considers a single realized future behavior of a system, that is, a single path in a Kripke structure. The possible oprators in LTL are the same as the temporal operators of CTL*. For a graphical explanation, observe Figure 1.4.

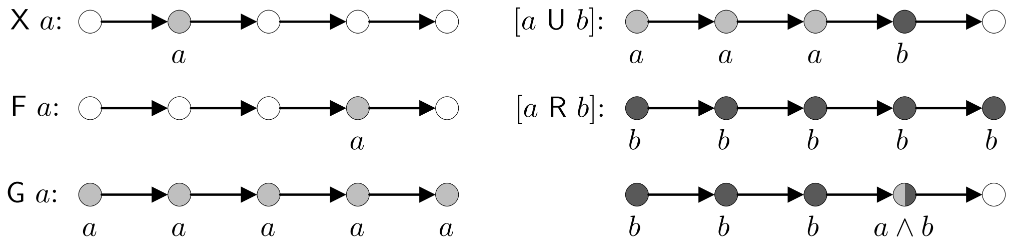


Figure 1.4. *Illustration of LTL operators.*

An LTL formula φ is said to be *valid* with regard to a Kripke structure M, if it holds for all paths of M. It is *satisfiable* if it holds for some path is M. As a specification language, usually validity is desired. A CTL* formula equivalent to φ is $\mathsf{A}\varphi$ in case of validity, while $\mathsf{E}\varphi$ in case of satisfiability.

Definition 7 (Syntax of LTL). The formal syntax of LTL is given by the following BNF grammar, where $p \in AP$ is an atomic proposition:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \mathsf{X}\ \phi \mid \mathsf{F}\ \phi \mid \mathsf{G}\ \phi \mid [\phi\ \mathsf{U}\ \phi] \mid [\phi\ \mathsf{R}\ \phi]$$

A well-formed LTL formula is generated by ϕ. ▪

In LTL, a minimal set of connectives can be $\{\top, \neg, \wedge, \mathsf{X}, \mathsf{U}\}$. The CTL* equivalences listed in Section 1.2.1 can be used to define the remaining temporal operators.

Definition 8 (Semantics of LTL). The formal semantics of LTL is defined with respect to a Kripke structure M with a left-total transition relation (i. e., every path ρ of M is infinite). Let ρ be a path of M, and let $\rho[k]$ be a sub-path of ρ. The relation $\rho \models \phi$ is again defined inductively:

1. $\rho \models \top$;
2. $\rho \models p$ iff $p \in L(s_0)$;
3. $\rho \models \neg\phi$ iff $\rho \not\models \phi$;
4. $\rho \models \phi_1 \wedge \phi_2$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$;
5. $\rho \models \mathsf{X}\ \phi$ iff $\rho[1] \models \phi$;
6. $\rho \models [\phi_1\ \mathsf{U}\ \phi_2]$ iff for some $k \geq 0$, $\rho[k] \models \phi_2$ and for all $0 \leq i < k$, $\rho[i] \models \phi_1$. ▪

As the definition of the semantics suggests, LTL can distinguish between words on paths of a Kripke structure, which are ω-regular words over 2^{AP} as discussed in Section 1.1.1. This way, it is possible to speak about such words satisfying an LTL formula φ, denoted by $w \models \varphi$, and the language of the formula $\mathcal{L}(\varphi) = \{w \mid w \models \varphi\}$, which is an ω-regular language. More specifically, the set of languages that can be described in LTL is the set of *star-free languages* that is a strict subset of ω-regular languages.

Example 4. *A typical example for an LTL property is* $\mathsf{G}\ \mathsf{F}$ *(step). This is a so-called fairness property that requires the system to step infinitely often.*

CTL*, CTL and LTL

CTL and LTL differs in expressive power mainly due to the different semantical models [57]. There are properties that can be expressed in both, but there are also some that only one can formulate. CTL* includes both of them.

An example for a CTL property that cannot be described in LTL is the so-called *reset property*: $\mathsf{AG}\ \mathsf{EF}\ \varphi$, i. e., there is always a possibility to reach φ. LTL cannot describe such possibility, only that something *has to happen*. On the other hand, a similar property in LTL called *stability property* is outside of the expressive power of CTL: $\mathsf{G}\ \mathsf{F}\ \varphi$, i. e., φ will eventually hold forever. CTL can express only stricter ($\mathsf{AF}\ \mathsf{AG}\ \varphi$) or more permissive ($\mathsf{AF}\ \mathsf{EG}\ \varphi$) properties. The latter is cleary no the stability property, since it only requires the possibility of stability. The former one is a bit harder to see.

Consider the Kripke structure of Figure 1.1 and the formula $\mathsf{AF}\ \mathsf{AG}\ p$. While $\mathsf{F}\ \mathsf{G}\ p$ hold for the model, the semantics of CTL causes the formula to be invalid here. It is clear that the rightmost and the middle state can be labeled with $\mathsf{AG}\ p$. The initial state, however, cannot receive the label $\mathsf{AG}\ \mathsf{AF}\ p$, because there is always a path that leads to the middle state with label q.

Properties only describable with CTL* are easily built by mixing state and path formulae, for example $\mathsf{EX}\ \varphi \land \mathsf{A}(\mathsf{G}\ \mathsf{F}\ \varphi)$ is not expressible in either LTL or CTL.

Negation normal form

The negation normal form of an LTL formula [31] is particularly useful when transforming it to an automaton (see Section 1.3.4). In negation normal form, negations can appear only before atomic propositions, and the usable logical and temporal operators are restricted to $\top$, $\bot$, $\land$, $\lor$, X, U and R.

All LTL formulae can be transformed into negation normal form by applying the following equivalences (replace left hand side to right hand side whenever possible):

1. $\psi \Rightarrow \varphi \equiv \neg\psi \vee \varphi$
2. $\psi \Leftrightarrow \varphi \equiv (\psi \wedge \varphi) \vee (\neg\psi \wedge \neg\varphi)$
3. $\mathsf{F}\ \varphi \equiv [\top\ \mathsf{U}\ \varphi]$
4. $\mathsf{G}\ \varphi \equiv [\bot\ \mathsf{R}\ \varphi]$
5. $\neg(\psi \wedge \varphi) \equiv \neg\psi \vee \neg\varphi$
6. $\neg(\psi \vee \varphi) \equiv \neg\psi \wedge \neg\varphi$
7. $\neg\mathsf{X}\ \varphi \equiv \mathsf{X}\ \neg\varphi$
8. $\neg[\psi\ \mathsf{U}\ \varphi] \equiv [\neg\psi\ \mathsf{R}\ \neg\varphi]$
9. $\neg[\psi\ \mathsf{R}\ \varphi] \equiv [\neg\psi\ \mathsf{U}\ \neg\varphi]$
10. $\neg\neg\varphi \equiv \varphi$

Example 5. *Consider a formula $\neg[p\ \mathsf{U}\ q \vee \mathsf{F}\ p \wedge q]$. Conversion to negation normal form involves the following steps:*

$$\begin{aligned}
\neg[p\ \mathsf{U}\ (q \vee \mathsf{F}\ p \wedge q)] &= \\
\neg[p\ \mathsf{U}\ (q \vee [\top\ \mathsf{U}\ p \wedge q])] &= \\
[\neg p\ \mathsf{R}\ \neg(q \vee [\top\ \mathsf{U}\ p \wedge q])] &= \\
[\neg p\ \mathsf{R}\ (\neg q \wedge \neg[\top\ \mathsf{U}\ p \wedge q])] &= \\
[\neg p\ \mathsf{R}\ (\neg q \wedge [\bot\ \mathsf{R}\ \neg(p \wedge q)])] &= \\
[\neg p\ \mathsf{R}\ (\neg q \wedge [\bot\ \mathsf{R}\ (\neg p \vee \neg q)])]
\end{aligned}$$

1.3 Automata Theory

An automaton is an abstract machine that models some kind of computation over a sequence of input symbols. In formal language theory, automata are mainly used as a finite representation of infinite languages. In that context, they are often classified by the class of language they are able to recognize.

The simplest class of automata is finite automata (also known as finite state machines). A finite automaton operates with a finite and constant amount of memory (independent of the length of the input). A finite automaton can operate on finite or infinite inputs. In formal language theory, inputs are called words, while elements of the input are called letters, the set of all possible letters constituting the alphabet.

Definition 9 (Finite automaton). A finite automaton is a 5-tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = \{\alpha_1, \alpha_2, \ldots\}$ is the finite alphabet, i. e., the set of possible letters;
- $\mathcal{Q} = \{q_1, q_2, \ldots\}$ is the finite set of states;
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states;
- $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation consisting of state–input–state triples (q, α, q');
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final or accepting states. ▪

Let $w \in \Sigma^*$ be a finite word over the alphabet Σ, with length $|w|$. A *run* of the finite automaton $\mathcal{A}$ *reading* the word w is a sequence of automaton states $\rho_w \in \mathcal{Q}^*$, where $\rho_w(0) \in \mathcal{I}$ and $(\rho_w(i), w(i), \rho_w(i+1)) \in \Delta$ for all i, that is, the first state is an initial state and state changes are permitted by the transition relation. The ultimate goal of automata in formal language theory is to distinguish between words that are part of a language and those that are not. This behavior is defined by the *acceptance condition*.

Definition 10 (Acceptance condition for finite words). A run ρ_w of a finite automaton $\mathcal{A}$ reading the finite word w is called accepting iff the last state is a final state, i. e., $\rho_w(|\rho_w|) \in \mathcal{F}$. A finite automaton $\mathcal{A}$ accepts a finite word w iff it has an accepting run reading w. ▪

The set of all finite words accepted by a finite automaton $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ and is called the language of $\mathcal{A}$. The class of languages that can be characterized by a finite automaton over finite words is called regular languages.

Regular expressions

Regular expressions are textual descriptions of regular languages. Given an alphabet Σ, three constants are defined as regular expressions:

- $\emptyset$ that characterizes an empty language;
- ε that characterizes the language consisting of only the empty word;
- $a \in \Sigma$ describing a language with a single-letter word a.

Given two regular expressions R and S, the following operators are defined to produce regular expressions:

- RS is the *concatenation* of the languages (words of S are concatenated to words of R in every combination);
- $R|S$ is the *alternation* of the languages (it represents the union of the sets of words represented by R and S);
- R^* is the *Kleene star* denoting the smallest superset of the set of words represented by R that contains ε and is closed under concatenation (this includes any word that can be obtained by concatenating a finite number of words from R, including zero).

There is an extension to regular expressions that can describe ω-regular languages called ω-regular expressions. In addition to the former construct, it contains another operator: R^ω means almost the same as R^*, but with an infinite number of words concatenated.

Example 6. *The regular expression $\varepsilon|((a|b)^*a)$ describes the every word that ends with an a and the empty word. The omega regular expression $(b^*a)^\omega$ describes infinite words that contain infinitely many a's.*

A finite automaton can be deterministic or nondeterministic. Definition 9 described the nondeterministic version where the structure describing the transitions is a relation. In the case of deterministic automata, transitions can be described as a function of the current state and the input letter $\Delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. In practice, this means that given two transitions (q, α, q') and (q, α, q''), the target states must be the same $(q' = q'')$. Another implication is that a specific word can have at most one corresponding run in a deterministic finite automaton. In the rest of this thesis, we will consider nondeterministic automata.

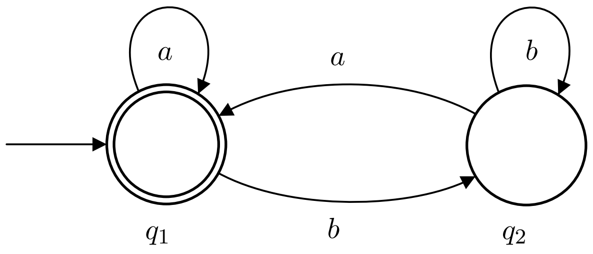


Figure 1.5. *A finite automaton.*

Example 7. *Figure 1.5 shows the graphical representation of an automaton. It is similar to that of a Kripke structure, but labels are written on the transitions, denoting the letter that they can read. Accepting states are marked with double circles.*

The automaton accepts the word abbaa, because there exists an accepting run $q_1q_1q_2q_2q_1q_1$ reading it. The language described by this automaton is $\varepsilon|((a|b)^*a)$, i. e., the empty word and those ending with an a.

1.3.1 Büchi automata

Büchi automata [8] are the simplest finite automata operating on infinite words. The structure of a Büchi automaton is the same as that of a finite automaton (defined by Definition 9). A run of a Büchi automaton reading an infinite word $w \in \Sigma^\omega$ is now an infinite sequence $\rho_w \in \mathcal{Q}^\omega$ with the same rules as finite runs of a finite automaton. Since the last state in a run can no longer be defined, the acceptance condition has to be modified to treat infinity.

Definition 11 (Büchi acceptance condition). Let $w \in \Sigma^\omega$ be an infinite word and $\rho_w \in \mathcal{Q}^\omega$ be an infinite run reading w. Let $inf(\rho_w)$ denote the set of states that appear infinitely often in ρ_w. The run ρ_w is called accepting iff it has at least one accepting state appearing infinitely often, i. e., $inf(\rho_w) \cap \mathcal{F} \neq \emptyset$. A Büchi automaton $\mathcal{A}$ accepts a word w iff it has an accepting run reading w. ▪

The class of languages that can be characterized by Büchi automata is called ω-regular languages.

Example 8. *Figure 1.5 can also be interpreted as a Büchi automaton. Then it accepts the language $(b^*a)^\omega$, i. e., every infinite word containing an infinite number of a's.*

Kripke structures, LTL and Büchi automata

Three structures were presented so far that are related to words and languages. In terms of terminology, Kripke structures *produced* words, while linear temporal logic and Büchi automata *characterized* or *recognized* them. These terms essentially mean the same – both three structures are associated with a family of languages.

Büchi automata are one of the standard means to characterize ω-regular languages, meaning that for every ω-regular language, there exists a Büchi automaton that recognizes exactly that language and vice versa.

Although Kripke structures are very similar in structure, they are less expressive, i. e., there are ω-regular languages that Kripke structures cannot produce. To see this, the following proposition presents a way to construct a Büchi automaton that accepts exactly the language of a Kripke structure.

Proposition 1 (Büchi automaton of Kripke structure). Given a Kripke structure $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, L \rangle$ with the set of atomic propositions AP, an equivalent Büchi automaton that accepts the language produced by M is $\mathcal{A}_M = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = 2^{AP}$, i. e., letter are sets of atomic propositions;
- $\mathcal{Q} = \mathcal{S} \cup \{init\}$, i. e., the states of the Kripke structure together with a special initial state;
- $\mathcal{I} = \{init\}$, i. e., the special initial state;
- $\Delta = \{(s, \alpha, s') \mid (s, s') \in \mathcal{R} \land \alpha = L(s')\} \cup \{(init, \alpha, s) \mid s \in \mathcal{I} \land \alpha = L(s)\}$, i. e., the automaton reads the labels of target states and additional transitions go from the special initial state to initial states of the Kripke structure;
- $\mathcal{F} = \mathcal{Q}$, i. e., every state is accepting. ▪

Defining accepting states as the entire set of states is indeed necessary, because every path in a Kripke structure produces a word, no matter what states it passes. In other words, there is no such thing as *fairness* in ordinary Kripke structures. Based on that, an example ω-regular language that cannot be produced by a Kripke structure is a^*b^ω (infinitely many b's after finitely many a's). Any Büchi automaton accepting this language must have a loop that reads the letter a arbitrary many times and another loop that reads b infinitely many times. Only the second loop can contain an accepting state to force runs out of first loop, which cannot be modeled in Kripke structures.

The set of languages that can be described by linear temporal formulae is the set of star-free languages, which is also a strict subset of ω-regular languages. An example ω-regular language that cannot be described by a finite LTL formula is $((a|b)b)^\omega$ (every second letter is a b, the others may be a or b).

It is also easy to see that Kripke structures and LTL has different expressive powers, since $((a|b)b)^\omega$ can be modeled by a Kripke structure, while a^*b^ω can be expressed by the LTL formula $[a \land \neg b \mathsf{~U~G~} b]$.

1.3.2 Synchronous product of Büchi automata

The synchronous product of two Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is another Büchi automaton $\mathcal{A}_1 \cap \mathcal{A}_2$ that accepts exactly those words that both $\mathcal{A}_1$ and $\mathcal{A}_2$ accept [20]. The language of the product automaton is therefore $\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Definition 12 (Synchronous product). Given two Büchi automata $\mathcal{A}_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{I}_1, \Delta_1, \mathcal{F}_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{I}_2, \Delta_2, \mathcal{F}_2 \rangle$, their synchronous product is $\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_\cap, \mathcal{I}_\cap, \Delta_\cap, \mathcal{F}_\cap \rangle$, where:

- $\mathcal{Q}_\cap = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \{0,1,2\}$, i. e., the pair of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ and a counter;
- $\mathcal{I}_\cap = \mathcal{I}_1 \times \mathcal{I}_2 \times \{0\}$, i. e., every state with counter value 0;
- $\mathcal{F}_\cap = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \{2\}$, i. e., every state with counter value 2.

A transition $((q, r, x), \alpha, (q', r', y))$ is part of the transition relation $\Delta_\cap$ iff:

- $(q, \alpha, q') \in \Delta_1$ and $(r, \alpha, r') \in \Delta_2$, i. e., both automata can progress with the input letter α;
- the third component satisfies the following rule (i. e., counts the accepting states encountered):

$$y = \begin{cases} 0 & \text{if } x = 2 \\ 1 & \text{if } x = 0 \text{ and } q' \in \mathcal{F}_1 \\ 2 & \text{if } x = 1 \text{ and } r' \in \mathcal{F}_2 \\ x & \text{otherwise.} \end{cases}$$

■

The third component ensures that an accepting path contains accepting states of both automata infinitely often. Simply using $\mathcal{F}_1 \times \mathcal{F}_2$ is not sufficient, since even though both $\mathcal{A}_1$ and $\mathcal{A}_2$ have to pass their own accepting states infinitely often, they are not required to do so at the same time.

The third component is a counter that can reach 2 (an accepting state by definition) after seeing at least one accepting state from $\mathcal{A}_1$ and then at least one from $\mathcal{A}_2$. The counter resets in the next state, so an infinite number of accepting states in the product automaton implies an infinite number of accepting states in both $\mathcal{A}_1$ and $\mathcal{A}_2$. If at any point, no more accepting state is encountered in one of the automata, that automaton will reject the word. In that case, the counter gets stuck and the product will not accept the word either. See Figure 1.6 for an example.

In the special case where all states of one of the automata are accepting ($\mathcal{F}_1 = \mathcal{Q}_1$), the definition of the product automaton is simpler. Since any infinite run in $\mathcal{A}_1$ will be accepting, the counter in not necessary – the product inherits the accepting states of $\mathcal{A}_2$. Such is the case when an automaton describes the possible behaviors of a Kripke structure (see Proposition 1).

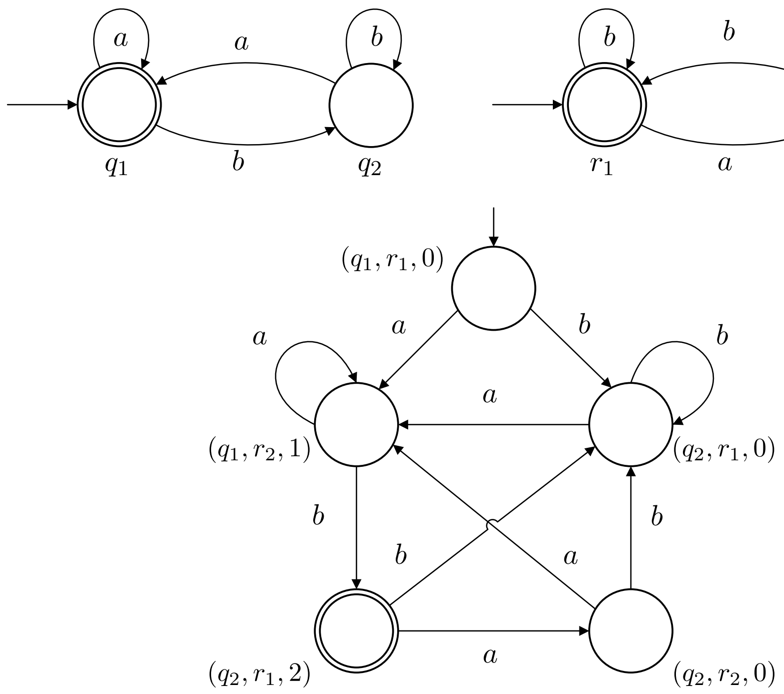


Figure 1.6. *Two automata and their synchronous product.*

Definition 13 (Synchronous product – special case). Given two Büchi automata $\mathcal{A}_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{I}_1, \Delta_1, \mathcal{F}_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{I}_2, \Delta_2, \mathcal{F}_2 \rangle$ with $\mathcal{F}_1 = \mathcal{Q}_1$, their synchronous product is $\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_\cap, \mathcal{I}_\cap, \Delta_\cap, \mathcal{F}_\cap \rangle$, where:

- $\mathcal{Q}_\cap = \mathcal{Q}_1 \times \mathcal{Q}_2$;
- $\mathcal{I}_\cap = \mathcal{I}_1 \times \mathcal{I}_2$;
- $\Delta_\cap = \{((q, r), \alpha, (q', r')) \mid (q, \alpha, q') \in \Delta_1 \wedge (r, \alpha, r') \in \Delta_2\}$;
- $\mathcal{F}_\cap = \mathcal{F}_1 \times \mathcal{F}_2 = \mathcal{Q}_1 \times \mathcal{F}_2$. ▪

1.3.3 Generalized Büchi automata

Sometimes – especially in model checking – it is more convenient to use a special variant of Büchi automata called *generalized Büchi automata* (GBA). A generalized Büchi automaton has multiple sets of accepting states (possibly even zero), a feature that enables more compact encoding of languages [20].

Definition 14 (Generalized Büchi automaton). A generalized Büchi automaton is a 5-tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where Σ, $\mathcal{Q}$, $\mathcal{I}$ an Δ denote the same as in Büchi automata, while $\mathcal{F} \subseteq 2^{\mathcal{Q}}$ is the set of acceptance sets instead of a single acceptance set. ▪

Example 9. *Figure 1.7 shows a generalized Büchi automaton. It has two acceptance sets: the two states belong to different ones (states of different acceptance sets are denoted by double circles and dashed circles). It accepts the language $(aa*bb*)^\omega|(bb*aa*)^\omega$, i. e., words that contain infinitely many a's and b's as well.*

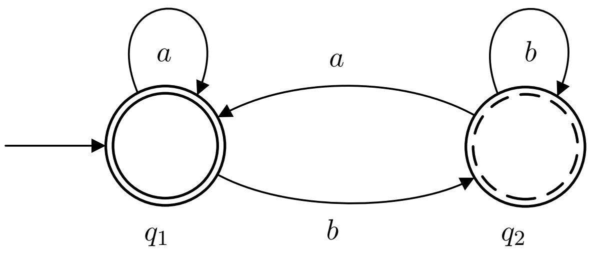


Figure 1.7. *A generalized Büchi automaton with two acceptance sets.*

The acceptance condition of a generalized Büchi automaton is similar to the Büchi acceptance condition, but it is extended to multiple acceptance sets.

Definition 15 (Generalized Büchi acceptance condition). Let $w \in \Sigma^\omega$ be an infinite word and $\rho_w \in \mathcal{Q}^\omega$ be an infinite run reading w. Let $\mathit{inf}(\rho_w)$ denote the set of states that appear infinitely often in ρ_w. The run ρ_w is called accepting iff it has at least one accepting state from all acceptance sets appearing infinitely often, i. e., for all $\mathcal{F}_i \in \mathcal{F}$, $\mathit{inf}(\rho_w) \cap \mathcal{F}_i \neq \emptyset$. A generalized Büchi automaton $\mathcal{A}$ accepts a word w iff it has an accepting run reading w. ▪

The expressive power of generalized Büchi automata is equivalent to that of Büchi automata, because there exists a language preserving transformation between them.

Proposition 2 (Equivalent Büchi automaton of generalized Büchi automaton). Given a generalized Büchi automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$ where $|\mathcal{F}| = n$, an equivalent Büchi automaton is $\mathcal{A}' = \langle \Sigma, \mathcal{Q}', \mathcal{I}', \Delta', \mathcal{F}' \rangle$, where:

- $\mathcal{Q}' = \mathcal{Q} \times \{0, \ldots, n\}$, i. e., the states of the GBA combined with a counter;
- $\mathcal{I}' = \mathcal{I} \times \{0\}$, i. e., the initial states of the GBA combined with counter state 0;
- $\mathcal{F}' = \mathcal{Q} \times \{n\}$, i. e., states with counter state n are accepting.

A transition $((q, x), \alpha, (q', y))$ is part of the transition relation Δ' iff:

- $(q, \alpha, q') \in \Delta$, i. e., , the GBA can progress with the input letter α.
- the third component satisfies the following rule (where $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$):

$$y = \begin{cases} 0 & \text{if } x = n \\ i & \text{if } x = i - 1 \text{ and } q' \in \mathcal{F}_i \\ x & \text{otherwise.} \end{cases}$$

▪

Just like when building the synchronous product, the counter ensures that accepting states from every acceptance set are visited infinitely often. To understand the working of the counter, one can imagine the structure of the generalized Büchi automaton copied $n + 1$ times, with instances having the different accepting sets. The run starts in instance 0. Whenever an accepting state is encountered in the current instance, the next transition takes the run into the following state, but in the next instance. Accepting sets of the resulting Büchi automaton are in the nth instance, where every transition goes back into instance 0. If the generalized Büchi automaton has no acceptance sets, every state will be accepting in the equivalent Büchi automaton.

1.3.4 Transforming LTL expressions

Since linear temporal logic formulae characterize a strict subset of ω-regular languages, there is an equivalent Büchi automaton for every LTL formula φ that accepts the same language that satisfies ϕ. The algorithm of Gerth, Peled, Vardi and Wolper [31] is able to transform an LTL formula in negation normal form into a generalized Büchi automaton. This section informally presents the main ideas of the algorithm.

The algorithm constructs a *tableau*, a tree-like structure that represents the evaluation of the formula with possible inputs. The base element of the construction is a *node*, which represents a state of the resulting generalized Büchi automaton, as well as it is an intermediate data structure of the construction. It has three sets of fomulae and a set of incoming transitions from other nodes.

The set of formulae *Old* contains subformulae that will be evaluated in the corresponding automaton state. Atomic propositions and their negations will be used to characterize the input read by incoming transitions, while other formulae will be used to determine the accepting states of the automaton.

The sets *New* and *Next* are used in the construction process. *New* contains formulae that have to be processed before a node is finished, while *Next* contains formulae that will be processed in the successor node.

When processing *New*, the algorithm removes formulae from it and may insert new ones in *Old* and *Next*, as well as split the node. The node is finished if *New* gets empty. At that point, the algorithm checks if there is an already finished node with the same *Old* and *Next* sets. If there is such a node, their *Incoming* sets are merged and the old node is replaced, otherwise the new node is inserted into the set. If at any time, *Old* contains φ and $\neg\varphi$, the node is discarded due to contradiction.

The algorithm starts with a single node, with *Old* and *Next* as empty sets, *New* containing only the full LTL formula, and *Incoming* consisting of a single element, the special *init* state. Whenever a formula φ is chosen, removed from *New* and put into *Old*, the following can happen based on the root operator of its expression tree.

- If φ is an atomic propositions (p) or its negation ($\neg p$) nothing more has to be done.
- If φ is a conjunction ($\psi \wedge \phi$), ϕ and ψ are put in *New*.
- If φ is a disjunctions ($\psi \vee \phi$), the node is split: the resulting two nodes inherit the *Old*, *Next* and *Incoming* sets, ψ is put in the *New* set of the first node, ϕ in that of the second. Processing continues recursively on the resulting nodes.

- If φ is $\mathsf{X}\ \psi$, ψ is put in the *Next* set.
- If φ is $[\psi\ \mathsf{U}\ \phi]$ or $[\psi\ \mathsf{R}\ \phi]$, it is reduced to disjunction, conjunction and the X operator by the following recursive definitions:
  - $[\psi\ \mathsf{U}\ \phi] = \phi \vee (\psi \wedge \mathsf{X}\ [\psi\ \mathsf{U}\ \phi])$
  - $[\psi\ \mathsf{R}\ \phi] = (\psi \wedge \phi) \vee (\phi \wedge \mathsf{X}\ [\psi\ \mathsf{R}\ \phi])$

When every node has been processed, a generalized Büchi automaton can be built based on the structure of nodes.

- As usual, the alphabet is 2^{AP}.
- States of the automaton are the set of finished nodes and the special *init* state.
- The only initial state is *init*.
- Transitions are obtained from the *Incoming* sets of the nodes. A transition (q, α, q') is in the transition relation iff the *Incoming* set of q' contains q' and α contains every ponated atomic proposition in *Old* and does not contain negated atomic propositions.
- The automaton has an acceptance set corresponding to every subformula in the form $[\psi\ \mathsf{U}\ \phi]$, because eventually ϕ has to be true. A state is in the acceptance set of subformula $[\psi\ \mathsf{U}\ \phi]$, if the *Old* set of the corresponding node contains ϕ or it does not contain $[\psi\ \mathsf{U}\ \phi]$ (i. e., the subformula is satisfied by entering the state or it is already satisfied and not considered anymore).

The size of the resulting automaton is exponential in the worst case, but experience shows that it is usually small [20].

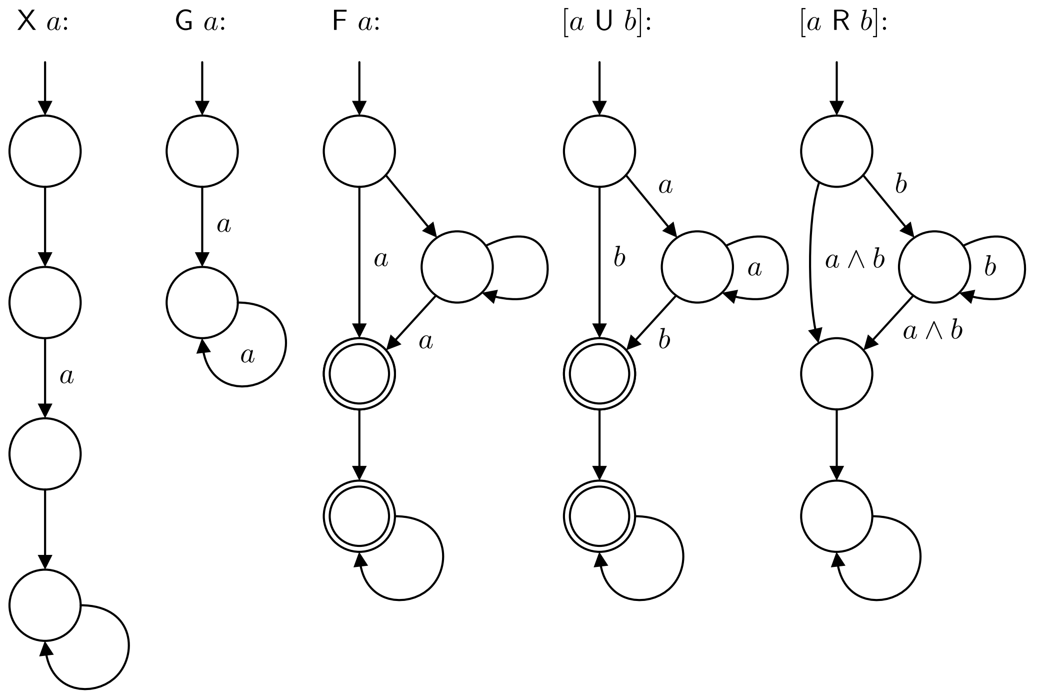


Figure 1.8. *Automata constructed by the algorithm of [31] for simple LTL formulae.*

Example 10. *Figure 1.8 shows automata corresponding to different operators of LTL constructed by this algorithm. These are not the simplest automata describing the formulae, but they all possess a property that will be described and exploited in Section 3.1.1.*

Other transformation algorithms have been proposed, a more complex and efficient one is for example [30].

1.3.5 Language emptiness of Büchi automata

The language emptiness problem for Büchi automata asks if given a Büchi automaton $\mathcal{A}$, is there any word that is accepted by $\mathcal{A}$, i. e., $\mathcal{L}(\mathcal{A}) \overset{?}{=} \emptyset$. The problem can be reduced to finding strongly connected components (SCC) in the structure of the automaton [20].

In order for a Büchi automaton to accept a word, it has to have an accepting run, i. e., an infinite path in the automaton that passes accepting states infinitely often. Since the size of a Büchi automaton is finite, infinite runs can occur only with eventually repeating sequences. This implies that the path has to enter a strongly connected component and never leave it.

An SCC is called accepting if it has at least one accepting state (from every acceptance sets in case of GBAs). If such an SCC is reachable from an initial state of the Büchi automaton, then there is an accepting run and a corresponding word that the automaton accepts. If no accepting SCCs are reachable from the initial states, then the language is empty.

Example 11. *Consider the product automaton on Figure 1.6. The states* $(q_1, r_2, 1)$*,* $(q_2, r_1, 2)$*,* $(q_2, r_2, 0)$ *and* $(q_2, r_1, 0)$ *constitute a reachable SCC, and* $(q_2, r_1, 2)$ *is an accepting run, so the language of the automaton is nonempty. Indeed, it accepts the word* $a(ba)^\omega$*.*

1.4 Model Checking

Model checking [18] is an automatic formal verification technique that aims to exhaustively analyze the possible behaviors of a model and check if they meet a given specification. A specific class of the model checking problem that received particularly much attention is the model checking of temporal logic properties. This class is also the focus of this thesis, so model checking will refer to model checking of temporal logic properties.

Formally, the problem of model checking is deciding $(M, s) \overset{?}{\models} \varphi$, where M is a Kripke structure, $s \in \mathcal{I}$ is an initial state, and φ is a temporal logic formula [26].

An important problem related to model checking is state space generation. Models are rarely given as Kripke structures, instead, various higher level formalisms are used to compactly describe a system. As seen in the case of Petri nets, the semantics of these models are usually defined in such a way that transformation to a Kripke structure is possible. This process is called state space generation or state space exploration.

One of the biggest challenges of model checking arises here: the so-called "state space explosion problem" denotes the usual combinatorial explosion in the size of the Kripke structure compared to the size of high-level models. In other words, even small high level models may result in huge Kripke structures, often exponential in size. There are many techniques to handle a huge state space, usually exploiting symmetries of the Kripke structure. Some of these will be presented in Section 1.6 and 2.1.

Depending on the class of temporal logic used in the specification, various algorithms have been developed to solve this problem automatically, most of them reducing the problem to graph algorithms.

1.4.1 CTL model checking

The first model checking algorithm was designed to check branching-time logic formulae [19]. In fact, CTL was specifically defined for efficient model checking. An important observation of Section 1.2.2 already noted that every possible CTL (sub)expression is a state formula, providing a way to process the states of the model individually (and avoiding the complexity of processing paths).

CTL model checking algorithms work by labeling the states of the system with subformulae of the CTL property iteratively, until a fixed-point is reached. Given an expression tree of the formula φ, nodes (operators) are processed starting from the leaves, which are atomic propositions already assigned by the labeling function of the Kripke structure. Using a minimal set of connectives, the subformulae are assigned to states with the following strategy:

- $\top$ is assigned to every state.
- $\neg\phi$ is assigned to every state not labeled with ϕ.
- $\psi \wedge \phi$ is assigned to every state labeled with ψ and ϕ.
- $\mathsf{EX}\ \phi$ is assigned to every state that is backwards reachable from states labeled with ϕ in one step.
- $\mathsf{EG}\ \phi$ is assigned to every state that is backwards reachable from a directed loop through states labeled with ϕ (both in the loop and in the path from the loop).
- $\mathsf{E}[\psi\ \mathsf{U}\ \phi]$ is assigned to every state that is backwards reachable from states labeled with ϕ though states labeled with ψ.

Whenever a subformula ϕ is processed, other (sub)formulae might be processed by treating ϕ as a simple label. The algorithm ends when states are labeled with the whole formula φ. If the initial state is labeled by φ, then the model meets the specification, otherwise not. The complexity of the algorithm [49] is linear in the size of the Kripke structure $|\mathcal{S}|$ and the size of the formula $|\varphi|$ measured by the number of operators: $\mathcal{O}(|\mathcal{S}| \cdot |\varphi|)$. For more details about the algorithm and common challenges and optimizations, the reader is referred to [20].

1.4.2 LTL model checking

Model checking of linear temporal logic formulae (or linear temporal model checking) is computationally harder than CTL model checking, because it has to process paths of the Kripke structure. The essential idea of LTL model checking is using Büchi automata to describe the property and the model and reducing the model checking problem to language emptiness [20].

Given a Kripke structure M and an LTL formula φ using the same atomic propositions AP, let $\mathcal{L}(M)$ and $\mathcal{L}(\varphi)$ denote the languages produced by the Kripke structure and characterized by the formula. The language $\mathcal{L}(M)$ contains every observable behavior of M in terms of AP (provided behaviors), while $\mathcal{L}(\varphi)$ contains valid behaviors. The model checking problem can be rephrased to the following: is the set of provided behaviors fully within the set of valid behaviors? This is called the language inclusion problem and is denoted by $\mathcal{L}(M) \overset{?}{\subseteq} \mathcal{L}(\varphi)$.

An equivalent formalization is $\mathcal{L}(M) \cap \overline{\mathcal{L}(\varphi)} \overset{?}{=} \emptyset$, where $\overline{\mathcal{L}(\varphi)}$ is the complement of the language of φ. Although both the language inclusion problem and the complementation of Büchi automata is hard, in case of LTL model checking, the complementation can be avoided. The key observation is that the complement of the language of an LTL formula is the language of the negated formula: $\overline{\mathcal{L}(\varphi)} \equiv \mathcal{L}(\neg\varphi)$. This way, the model checking problem is reduced to language intersection and language emptiness, both of which can be efficiently computed on Büchi automata.

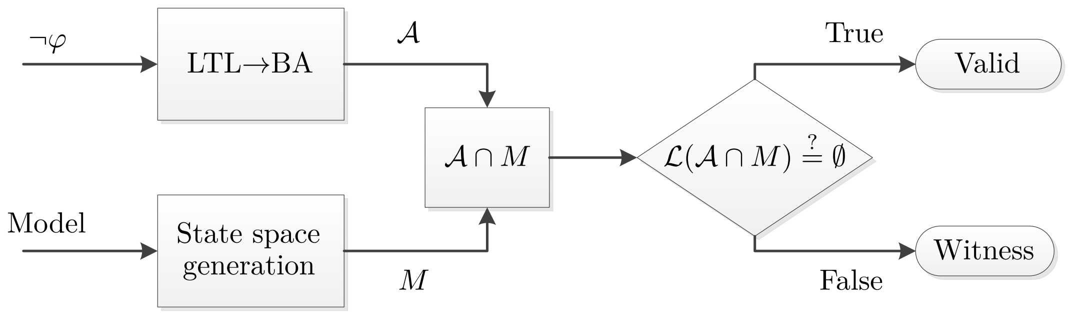


Figure 1.9. *Steps of ω-regular model checking.*

This approach is called *automata-theoretic model checking* [56]. Given a high-level model and a linear temporal logic specification, the following steps have to be realized (see Figure 1.9:

1. Compute the Kripke structure M of the high-level model (state space generation).

2. Transform the Kripke structure into a Büchi automaton $\mathcal{A}_M$ (Proposition 1).
3. Transform the negated LTL formula into a Büchi automaton $\mathcal{A}_{\neg\varphi}$ (Section 1.3.4).
4. Compute the synchronous product $\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}$.
5. Check language emptiness of the product: $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}) \overset{?}{=} \emptyset$.

If $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi}) = \emptyset$ then the model meets the specification. Otherwise words of $\mathcal{L}(\mathcal{A}_M \cap \mathcal{A}_{\neg\varphi})$ are counterexamples, i. e., provided behaviors that violate the specification. As seen in Section 1.3.4, the size of the Büchi automata corresponding to φ is exponential in the size of φ measured by the number of operators, while the size of the product is linear in the size of the operands. Thus, the complexity of the algorithm [49] is linear in the size of the Kripke structure $|\mathcal{S}|$ and exponential in the size of the LTL formula: $\mathcal{O}(|\mathcal{S}| * 2^{|\varphi|})$.

Sometimes, it is possible to design the algorithms such that some steps may overlap or can be executed together. For example, many algorithms compute the product automaton on-the-fly during state space generation, using the high-level model as an implicit representation of the Kripke structure. Language emptiness may sometimes also be checked continuously during the computation of the product. If both optimizations are present in an algorithm, it is said to perform the LTL model checking on the fly (during state space generation). For an example, see [31] that is the base of the SPIN explicit model checker.

1.4.3 Symbolic model checking

Symbolic model checking techniques have been devised to combat the state space explosion problem [43, 9]. While traditional *explicit-state model checkers* employ graph algorithms and enumerate states and transitions one-by-one, *symbolic model checking* algorithms use a special encoding to efficiently represent the state space and try to process large numbers of similar states together. The special encoding can be regarded as a compression of the sets and relations, but unlike traditional data compression methods, the encoded data can be manipulated without returning to the explicit representation.

In [17], where symbolic model checking was first introduced, states of hardware models were determined by binary variables. Similarly, states of a Kripke structure can also be encoded in $k = \lceil log(|\mathcal{S}|) \rceil$ Boolean variables. Sets of states can then be represented by Boolean functions $f_{\mathcal{S}} : \mathbb{B}^k \to \mathbb{B}$, returning true when a state is in the set. The transition relation can also be represented by functions mapping from $2k$ binary variables to true or false, half of them encoding the source state, the other half encoding the target: $f_{\mathcal{R}} : \mathbb{B}^{2k} \to \mathbb{B}$.

The functions are usually represented by Boolean formulae or binary decision diagrams. In case of Boolean formulae, the model checking problem is reduced to the Boolean satisfiability problem (see Section 2.2), while in case of decision diagrams, efficient algorithms are known to manipulate the sets directly in the encoded form [7].

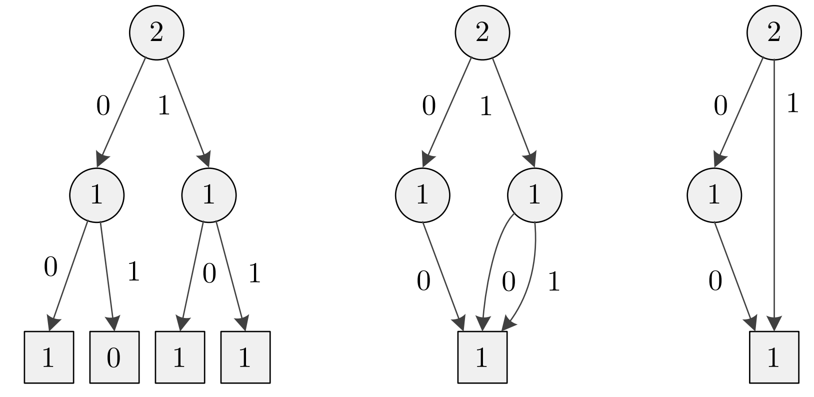
1.5 Decision Diagrams

A common encoding for binary functions is based on binary decision trees. In such a tree, every node corresponds to the evaluation of a variable, with arcs going to the next node based on the possible values. Leaf nodes are called terminal nodes and they provide the result of the function for a given input: the *terminal one* (denoted by **1**) means the result of the function is *true*, while the *terminal zero* (denoted by **0**) represents the result *false*. In other words, nodes mark *if-then-else* operations, and any input defines a path leading from the root node to a terminal node through a series of decisions.

A *binary decision diagram* is a more compact representation of a binary decision tree, where isomorphic subtrees are merged. Thus a decision diagram is a directed acyclic graph with a root node and two terminal nodes.

1.5.1 Reduced ordered binary decision diagrams.

A decision tree can be reduced into a decision diagram by merging the isomorphic subtrees. If in addition, the order of variable evaluation is fixed on every path from the root node to the terminal nodes, the result is an *ordered binary decision diagram.* Further reductions can be made by eliminating nodes that carry no information, that is, both arcs starting from them lead to the same node. If both of these reductions and restrictions are applied to a binary decision tree, we get a *reduced ordered binary decision diagram* (ROBDD) [7]. In the literature, the term binary decision diagram or BDD usually refers to an ROBDD.



(a) Ordered decision tree. (b) OBDD. (c) ROBDD.

Figure 1.10. *From binary decision trees to ROBDDs.*

Example 12. *Figure 1.10 shows how an ordered decision tree can be reduced to an ROBDD. All of the graphs encode the binary function* $b_1 \vee \neg b_2$*, i. e., the set* $\{(0,0),(1,0),(1,1)\}$.

The size of an ROBDD depends both on the function (or set of elements) it encodes and the chosen variable ordering. In the best case, BDDs can achieve exponential compression of certain sets with the proper ordering. However, there are some functions for which the size of the smallest BDD is linear in the size of the encoded set. All in all, the ordering alone can cause an exponential difference, so it is crucial to carefully choose it. Unfortunately, the problem of finding the best variable ordering in NP-hard [4], and even any multiplicative approximation is known to be NP-hard [51]. Nevertheless, some heuristics exist to tackle the problem [25, 11]

1.5.2 Multi-valued decision diagrams.

A *multi-valued decision diagram* (MDD) is an extension of an (ordered) binary decision diagram in the sense that the domain of input variables to the encoded function can be any finite set [14]. The domains of the variables can be different. The result of the function is still *true* or *false*. Nodes in the diagram can be interpreted as *switch-case* evaluations, so each arc starting from a node denotes a valuation of the corresponding variable. Reductions and ordering can also be applied to MDDs.

Relations and functions

As it is well-known, relations are sets of k-tuples, or equivalently, functions assigning truth values to k-tuples. Formally, given k sets $D_1, \ldots, D_k$, a relation over these *domains* is $R \subseteq D_1 \times \ldots \times D_k$. In this thesis, mainly *binary* ($R \subseteq D_1 \times D_2$) or *ternary* ($R \subseteq D_1 \times D_2 \times D_3$) relations are used.

An important and common subclass of binary relations is *endorelations*, where $R = D \times D$, i. e., the domains of components are the same. There are some interesting properties that such relations can have:

- *reflexive*, i. e., for all $x \in D$, $(x, x) \in R$;
- *symmetric*, i. e., for all $x, y \in D$, if $(x, y) \in R$ then $(y, x) \in R$.
- *transitive*, i. e., for all $x, y, z \in D$, if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$.

There are some basic and more complicated operations over relations that yield other relations. Let R and S be binary relations over D_1 and D_2. Let P be another relation over D_2 and D_3. Finally, let Q be an endorelation over D.

- $R \cup S = \{(x, y) \mid (x, y) \in R \vee (x, y) \in S\}$ is the *union* of the relations.
- $R \cap S = \{(x, y) \mid (x, y) \in R \wedge (x, y) \in S\}$ is the *intersection* of the relations.
- $R \circ P = \{(x, z) \mid \exists y \in D_2, (x, y) \in R \wedge (y, z) \in P\}$ is the *composition* of the relations (similar to relational join).
- $R^{-1} = \{(y, x) \mid (x, y) \in R\}$ is the *inverse* of a relation (yielding a relation over D_2 and D_1).
- $Q^{=} = I \cup Q$ is the *reflexive closure*, where $I = \{(x, x) \mid x \in D\}$ is the identity relation. The reflexive closure of Q is the smallest reflexive relation that contains Q.

- $Q^+ = Q \cup (Q \circ Q) \cup ((Q \circ Q) \circ Q) \cup \ldots$ is the *transitive closure* of Q, i. e., the smallest transitive relation that contains Q.
- $Q^* = I \cup Q^+$ is the *reflexive transitive closure* of Q, i. e., the smallest reflexive and transitive relation that contains Q.

A relation R is called *functional* if it can be described with a function $f_R : D_1 \times \ldots \times D_i \times D_{i+1} \times \ldots \to D_k$, i. e., if $(x, \ldots, y, z_1 \ldots, w_1) \in R$ and $(x, \ldots, y, z_2, \ldots, w_2) \in R$, then $(z_1 = z_2) \wedge \ldots \wedge (w_1 = w_2)$ also holds (assume without loss of generality that $D_1 \ldots D_k$ is a permutation of the domains of R). For a simpler example, the binary relation $R \subseteq D_1 \times D_2$ is functional if it is equivalent to the function $f_R : D_1 \to D_2$, i. e., if $(x, y) \in R$ and $(x, z) \in R$ then $y = z$.

Any relations can be interpreted as functions mapping from some domains to the power set of the Cartesian product of the remaining domains. Again, a simple example can be the binary relation $R \subseteq D_1 \times D_2$ that is not functional, but describes the function $f_R : D_1 \to 2^{D_2}$. With the common notation of using the symbol of the relation itself as the symbol of the function, $R(x) = \{y \mid (x, y) \in R\}$. This definition can be extended to map from the power sets of D_1 as well (i. e., the parameter of the function is a set of elements): $R(X) = \{y \mid \exists x \in X, (x, y) \in R\}$. This definition is identical to that of the *relational product* $X \circ R$. As a well-known analogy, one can consider linear algebra, where matrices are relations over the set of vectors, and vector-matrix multiplication is similar to the relational product, while matrix-matrix multiplication is analogous to relation composition.

1.5.3 Operations and notations

From a more abstract point of view, decision diagrams can represent sets: encode a function that returns *true* if the given element is in the set and *false* otherwise. Input variables can be obtained by logarithmic (binary) encoding, or by representing the elements naturally by vectors. In this context, it is very useful to define operations on decision diagrams to manipulate the encoded data.

Common operations include the union and the intersection of two sets represented by decision diagrams, complementation, and relational product. The latter is defined for a set A and a relation $R \subseteq A \times B$ as follows: $A \circ R = \{y \mid \exists x \in B, (x, y) \in R\}$. For the description of recursive algorithms realizing union, intersection and complementation with decision diagrams, the reader is referred to [7]. Efficient computation of the relational product is discussed in Section 1.6. A common property of these algorithms is aggressive caching. Since decision diagrams are obtained by merging isomorphic subtrees of decision trees, many paths in the diagram run into the same subdiagram. Recursive algorithms can cache the results of processing subdiagrams to reuse them when the subdiagram is reached again.

Throughout this thesis, decision diagrams will be at least ordered (but often not reduced). For this reason, nodes of the diagram will be grouped by their *levels*, which in turn correspond to one of the input variables (or vector components). Terminal nodes are on level 0, while the root node is on level K. A node on level k is denoted by n_k. The domain of the input variable corresponding to level k is denoted by D_k. The child of n_k corresponding to valuation $i \in D_k$ is $n_k[i]$.

A single node n_k of a decision diagram inherently represents a part of the encoded set. Every path starting in n_k and going to terminal $\mathbf{1}$ represents a partial assignment of some input variables (or vector components). The set of all these *subelements* is denoted by $\mathcal{B}(n_k) \subseteq D_1 \times \ldots D_k$ [14]. Paths starting from the root node and ending in n_k also represent a set of subelements denoted by $\mathcal{A}(n_k) \subseteq D_{k+1} \times \ldots \times D_K$. The subset of elements that a node n_k encodes in a decision diagram is then $\mathcal{S}(n_k) = \mathcal{B}(n_k) \times \mathcal{A}(n_k)$, corresponding to the set of paths going through n_k and ending in the terminal $\mathbf{1}$. Using the above notations, it is possible to define $\mathcal{B}(n_k)$ recursively:

$$\mathcal{B}(n_k) = \begin{cases} \{i \mid n_k[i] = \mathbf{1}\} & \text{if } k = 1 \\ \bigcup_{i \in D_k} \mathcal{B}(n_k[i]) \times \{i\} & \text{otherwise.} \end{cases}$$

For the sake of convenience, the notation $\mathcal{B}_{[i]}(n_k) = \mathcal{B}(n_k[i]) \times \{i\}$ is introduced to reason about sets of subelements constituting $\mathcal{B}(n_k)$.

1.6 Saturation

Saturation [12] is a symbolic iteration strategy specifically designed to work with decision diagrams. It was originally used as a state space generation algorithm [15] to answer reachability queries on concurrent systems, but applications in branching-time model checking [59] and SCC computation [60] also proved to be successful.

1.6.1 Input model

Saturation works best if it can exploit the structure of high-level models. Therefore, it defines the input model on a finer level of granularity, introducing the concept of components and events into traditional transition systems.

A component is a part of the model that has its own *local state*. A *global state* is the combination of the local states of every component in the system. In practice, each component i may define a state variable s_i containing its local state, while global states are tuples or vectors of these values denoted by $\mathbf{s} = (s_1, \ldots, s_K)$ (where K is the number of components). A straightforward example for a component in Petri nets can be a single place, with its local state being the number of tokens on it. The global state – the marking of the Petri net – is the combination of all token counts.

An event is a set of transitions somehow related in the high-level model. These transitions typically rely on and change the state of the same components. Again, a transition of a Petri net is a good example of an event. Petri net transitions have well-defined input and output places and they don't rely on or affect any other parts of the net. Depending on the current marking, a Petri net transition can represent various transitions of the underlying Kripke structure.

Definition 16 (The input model of saturation). Given a system with K components, the input model of saturation is a 4-tuple $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{E}, \mathcal{N} \rangle$, where:

- $\mathcal{S} \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_K$ is the set of global states with $\mathcal{S}_k$ being the set of possible local states of the kth component;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{E}$ is the set of events;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the *next-state relation*, partitioned by events: $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$. ▪

The following constructs will often be used throughout the thesis: the inverse of the next-state relation $\mathcal{N}^{-1}$, the *next-state function* in the forms $\mathcal{N}(\mathbf{s}) : \mathcal{S} \to 2^{\mathcal{S}}$ and $\mathcal{N}(S) : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, and the reflexive transitive closure of the next-state relation $\mathcal{N}^*$. The next-state function is the function representation of the next-state relation. Considering the next-state relation as a function is often closer to reality, because it is usually given only implicitly by the higher-level model.

1.6.2 Symbolic encoding

Saturation works directly on ordered decision diagrams. It is therefore necessary to define an ordering for the components of the model. Without loss of generality, assume that component k is the kth component in the ordering (i. e., components are identified by their indices).

By introducing components and events, saturation is able to build on a common property of concurrent systems. *Locality* is the empirical assumption that high-level transitions of a concurrent model usually affect only a small number of components. Locality is exploited both in building a more compact symbolic encoding and in an efficient iteration strategy.

An event $\varepsilon \in \mathcal{E}$ is *independent* from component k if *1)* its firing does not change the state of the component, and *2)* it is enabled independently of the state of the component. If ε depends on component k, then k is called a *supporting* component: $k \in supp(\varepsilon)$. Let $Top(\varepsilon) = k$ denote the supporting component of ε with the highest index. Along the value of Top, events can be grouped by the highest supporting component: $\mathcal{E}_k = \{\varepsilon \in \mathcal{E} | \, Top(\varepsilon) = k\}$. For the sake of convenience, $\mathcal{N}_k = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$ is used to represent the next-state relation of all such events. The self-explanatory notations $\mathcal{E}_{\leq k}$ and $\mathcal{E}_{<k}$ will also be used as well as the corresponding next-state relations $\mathcal{N}_{\leq k}$ and $\mathcal{N}_{<k}$.

Due to the locality property, the next-state relations of events can be decomposed into two parts: *1)* the actual description of local state changes for the supporting variables and *2)* the identity relation for independent variables. Saturation employs a more relaxed decomposition, splitting the relation by the *Top* value: for an event $\varepsilon \in \mathcal{E}_k$, the next state function is $\mathcal{N}_\varepsilon(s_1, \ldots, s_k, s_{k+1}, \ldots, s_K) \equiv \mathcal{N}_\varepsilon(s_1, \ldots, s_k) \times (s_{k+1}, \ldots, s_K)$, where $\mathcal{N}_\varepsilon(s_1, \ldots, s_k)$ is the projection of the next-state function to components $1 \ldots k$. Thus, it is sufficient to encode the first part, since the event only depends on components lower than the *Top* value.

1.6.3 Iteration strategy

The goal of saturation as a state space generation algorithm is to compute the set of reachable states $\mathcal{S}_{rch} = \mathcal{N}^*(\mathcal{I})$ of model M. In other words, the goal is to compute the least fixed point of the next-state function $\mathcal{S}_{rch} = \mathcal{N}(\mathcal{S}_{rch})$ that contains the initial states $\mathcal{I}$. There are many strategies to do this, from simple depth-first or breadth-first explorations to complex iteration strategies such as that of saturation.

The basic idea of the iteration strategy of saturation is also built on the locality property. Instead of computing the fixed point by repeatedly applying the whole next-state function starting from the initial states, the computation is divided into smaller parts. First, only events with the lowest *Top* value are considered, and a *local fixed point* is computed. After that, events with the lowest and the second lowest *Top* values are applied to obtain another fixed point. This process goes on until the global fixed point is reached. Using the notations introduced so far, the difference between classic breadth-first style fixed point computation and saturation's decomposed strategy is shown by the following two (informal) schemes, respectively:

$$\begin{aligned}
\text{Breadth-first style:} \quad & \mathcal{S}_{rch} = \mathcal{I} \circ \mathcal{N}^* \\
\text{Saturation:} \quad & \mathcal{S}_{rch} = ((\ldots((\mathcal{I} \circ \mathcal{N}^*_{\leq 1}) \circ \mathcal{N}^*_{\leq 2}) \circ \ldots) \circ \mathcal{N}^*_{\leq K})
\end{aligned}$$

Saturation performs these operations on decision diagrams, a fact that allows the definition of the fixed points in terms of decision diagram nodes. A node n_k is *saturated* iff it represents a least fixed point of $\mathcal{N}_{\leq k}$, that is, $\mathcal{B}(n_k) = \mathcal{N}_{\leq k}(\mathcal{B}(n_k))$. An equivalent, recursive definition requires the node to represent the fixed point of $\mathcal{N}_k$ *and* all of its children be saturated.

The iteration strategy follows from the recursive definition: to saturate a node n_k, first saturate all of its children, then apply the next-state function $\mathcal{N}_k$ (i. e., compute the relational product $\mathcal{B}(n_k) \circ \mathcal{N}_k$) iteratively until the fixed point is reached. If new children are created during the latter step, saturate them immediately. The process starts with the root node of the decision diagram representing the set of initial states and recursively calls saturation until it reaches the bottom. Terminal nodes are saturated by definition,

Algorithm 1.1: Saturate

```
input   : $s_k$ : node
//$s_k$: node to be saturated,
output  : node
$n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram;
Return result from cache if possible;
$t_k \leftarrow$ new node;
foreach $i \in \mathcal{S}_k : s_k[i] \neq 0$ do
    $t_k[i] \leftarrow Saturate(s_k[i])$;
repeat
    foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
        $t_k \leftarrow (t_k[i'] \cup RelProd(t_k[i], n_{2k}[i][i']))$;
until $t_k$ unchanged;
$t_k = CheckUnique(t_k)$;
Put inputs and results in cache;
return $t_k$
```

Algorithm 1.2: RelProd

```
input   : $s_k, n_{2k}$ : node
//$s_k$: node to be saturated,
//$n_{2k}$: next state node,
output  : node
if $s_k = 1 \wedge n_{2k} = 1$ then
    return 1;
Return result from cache if possible;
$t_k \leftarrow$ new node;
foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
    $t_k[i'] \leftarrow (t_k[i'] \cup RelProd(s_k[i], n_{2k}[i][i'])$;
$t_k \leftarrow Saturate(CheckUnique(t_k))$;
Put inputs and results in cache;
return $t_k$;
```

so the recursion stops at level 1. By the time the root node is saturated, it represents the set of reachable states.

The power of saturation comes from two sources. First, the algorithm works with smaller next-state relations and decision diagrams, making the local fixed point computations lightweight compared to breadth-first style global fixed point computations. Secondly, like in any decision diagram algorithms, results of the individual computations can be cached to avoid redundant operations.

The detailed pseudocode of saturation along with the implementation of the relational product operator can be seen in Algorithms 1.1 and 1.2.

1.6.4 Constrained saturation

Constrained saturation is a variation of the saturation iteration strategy introduced in [59]. For a motivation, recall CTL model checking, where operators EG and EU required checking backwards reachability through a certain type of states. To formalize the problem, consider a set of states called the *constraint*: $\mathcal{C} \subseteq \mathcal{S}$. The goal of constrained saturation is to compute the states of $\mathcal{C}$ that are reachable from $\mathcal{I}$ (where $\mathcal{I} \cap \mathcal{C} \neq \emptyset$) *through states of* $\mathcal{C}$.

The solution is not trivial, since $\mathcal{S} \cap \mathcal{C}$ may contain states that are reachable only through paths not entirely in $\mathcal{C}$. For this reason, the steps of the exploration have to be modified to consider only those target states that are allowed by the constraint: $\mathcal{N}(\mathcal{S}) \cap \mathcal{C}$. Performing an intersection after each step is expensive, so constrained saturation employs a *pre-checking* phase, elements of the next-state relation are sorted out based on whether the target state is in $\mathcal{C}$ or not. Formally, the idea is based on the following equivalence: $\mathcal{N}(\mathcal{S}) \cap \mathcal{C} = \{\mathbf{s}' \mid \mathbf{s}' \in \mathcal{N}(\mathcal{S}), \mathbf{s}' \in \mathcal{C}\}$.

By using another decision diagram to encode the constraint, the algorithm can locally decide if a single target local state is allowed or not. This can be regarded as the evaluation

Algorithm 1.3: ConsSaturate

```
     input  : $s_k$, $c$ : node
   1 //$s_k$:  node to be saturated,
   2 //$c_l$:  constraint node
     output : node
   3 $n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram;
   4 Return result from cache if possible;
   5 $t_k \leftarrow$ new node;
   6 foreach $i \in \mathcal{S}_k : s_k[i] \neq 0$ do
*  7     $c' \leftarrow c[i]$;
   8     if $c' \neq 0$ then
   9         $t_k[i] \leftarrow ConsSaturate(s_k[i], c')$;
  10     else
  11         $t_k[i] \leftarrow s_k[i]$;        //no steps allowed
  12 repeat
  13     foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
* 14         $c' \leftarrow c[i']$;
  15         if $c' \neq 0$ then
  16             $t_k \leftarrow$
                 $(t_k[i'] \cup ConsRelProd(t_k[i], c', n_{2k}[i][i']))$;
  17 until $t_k$ unchanged;
  18 $t_k = CheckUnique(t_k)$;
  19 Put inputs and results in cache;
  20 return $t_k$
```

Algorithm 1.4: ConsRelProd

```
     input  : $s_k, c, n_{2k}$ : node
   1 //$s_k$:  node to be saturated,
   2 //$c$:  constraint node,
   3 //$n_{2k}$:  next state node,
     output : node
   4 if $s_k = 1 \wedge n_{2k} = 1$ then
   5     return 1;
   6 Return result from cache if possible;
   7 $t_k \leftarrow$ new node;
   8 foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
*  9     $c' \leftarrow c[i']$;
  10     if $c' \neq 0$ then
  11         $t_k[i'] \leftarrow$
             $(t_k[i'] \cup ConsRelProd(s_k[i], c', n_{2k}[i][i']))$;
  12 $t_k \leftarrow ConsSaturate(CheckUnique(t_k), c)$;
  13 Put inputs and results in cache;
  14 return $t_k$;
```

of of the binary function that the decision diagram encodes. The constraint is traversed simultaneously with the decision diagram encoding the set of discovered states, and if it does not contain a path corresponding to the target state, the recursion is stopped on that branch. The pseudocode of the extended algorithm is shown on Algorithms 1.3 and 1.4. Differences are marked with an asterisk.

In an abstract view, the constraint is a series of predicate evaluations on local states – this is the original purpose of decision diagrams (or decision trees). It has a "memory" that can keep track of previous results along the recursive call chain that led the algorithm to the considered decision diagram node. This aspect will be exploited in algorithms of Chapter 3.

Chapter 2

Related Work

This chapter briefly summarizes related results in the domain of model checking, escpecially those related to LTL model checking, SCC computation, abstraction and saturation. Section 2.1 will present some traditional explicit techniques and algorithms, including SCC detection algorithms and partial order reduction. Section 2.2 introduces methods to reduce model checking to the Boolean satisfiability problem (SAT). Section 2.3 will show how traditional BDD-based symbolic approaches can be used to perform LTL model checking. Section 2.4 points the reader to some abstraction techniques that proved to be successful in hybrid approaches. Finally, Section 2.5 will give some examples of using saturation in model checking.

2.1 Explicit Techniques in Model Checking

Model checking (both CTL and LTL) can be fully realized by graph algorithms – these solutions are called explicit model checkers. Implicit representation of transitions given by the high-level source model can be used to generate the states of the system one by one. State space generation can then use breadth-first search or depth-first search or some of their variants to explore the full state graph.

On-the-fly exploration of the product automaton in LTL model checking is also easy to implement, because transitions can be computed one by one as well. On-the-fly SCC detection, however, requires some specialized algorithms.

2.1.1 Explicit on-the-fly SCC computation

One such algorithm is *Tarjan's algorithm* [54], which works by exploiting the exploration strategy of DFS. The algorithm uses an additional stack. Every time a node is explored, it is put on the stack. Each node is assigned a number *index* in the order they are traversed, and also another one *lowlink* (initially equals *index*) that keeps track of the node with the lowest *index* reachable from the current node. This number is recursively calculated by the time DFS backtracks to the node. If *lowlink* is smaller than *index*, the node is in a strongly connected component and is left on the stack. If they are equal, the node is the "root" of the SCC and every state left on the stack above it will be part of the same SCC (they are all popped). Tarjan's algorithm aims to compute every SCC of a directed graph. Accepting SCCs can then be filtered by checking if the SCCs contain an accepting state or not.

Another similar algorithm is *nested depth-first search* [37]. While Tarjan's algorithm computes every SCC, nested DFS will look for a single accepting loop in the graph (i. e., a counterexample in LTL model checking). It employs two depth-first searches to do this. The first DFS explores the graph, and every time it backtracks to an accepting state, the second DFS will try to find a way back to a state in the stack of the first DFS. If it manages to do so, the two stacks together give a counterexample.

2.1.2 Partial order reduction

Partial order reduction (POR) is a technique used to exploit the symmetries of a state space usually caused by interleaving semantics in concurrent systems [55, 46, 32]. When several processes can advance independently until a point of synchronization, the order in which parallel local steps are interleaved may be irrelevant. These steps are only partially ordered, but with interleaving semantics, every possible total orders will be explored by the model checking algorithm. In such cases, it may be sufficient to choose a single total order and aggregate the independent steps, greatly reducing the size of the state space.

There have been some attempts to combine partial order reduction and symbolic model checking [2], but it is still an open problem that is not trivial to solve.

2.2 SAT-based Model Checking

SAT-based bounded model checking encodes an instance of the bounded model checking problem as a SAT instance [3]. A path of length k is described in term of the Kripke structure and the property, and SAT-solvers are used to find an assignment representing a violating path.

Bounded model checking works by iteratively incrementing the depth value k, and is only complete if k reaches the diameter of the state space. Nevertheless, due to the boundedness of each iteration, this approach can be applied to infinite models as a semi-decision procedure.

For safety properties, k-induction – a generalized form of mathematical induction – is able to prove unreachability without exploring the full state space by providing inductive proofs [50].

A major breakthrough in this area was the application of interpolants to over-approximate the state space [42]. This allowed the algorithm to prove unsatisfiability much earlier than traditional approaches.

A recent approach called IC3 [5] works by constructing a series of small intermediate and inductive lemmas. Its novelty and power lies in focusing on small proofs instead of constructing large monolithic interpolants. By dividing the proof into smaller subgoals, SAT-solvers have to process smaller queries. As it turned out, proving more smaller lemmas can be much easier than solving less, but large and monolithic ones.

IC3 was also applied to look for SCCs in the state space, making the approach able to check liveness properties [6]. The approach identifies one-way walls in the state space to divide it into SCC-closed sets (parts of the state space that can fully contain an SCC) or prove the absence of SCCs. In this sense, one of the main contributions of this thesis (presented in Section 4.3.2) is based on similar ideas.

2.3 Decision Diagram-based LTL Model Checking

Decision diagram-based model checking implements symbolic model checking (described in Section 1.4.3) by encoding states and transitions in decision diagrams (see Section 1.5). Symbolic steps are then reduced to set operations, which can be performed directly on the digram representation providing an efficient way to handle large state spaces.

Different properties of the state space can be characterized using fixed points [27]. Symbolic model checking works by computing these fixed points. In case of CTL and LTL model checking, least and greatest fixed points are used to characterize the set of reachable states and strongly connected components, respectively.

2.3.1 Fixed point algorithms on decision diagrams

Using the notations of Section 1.6.1, the set of reachable states is characterized by the least fixed point of $\mathcal{S}_{rch} = \mathcal{N}(\mathcal{S}_{rch})$ including the set of initial states ($\mathcal{I} \subseteq \mathcal{S}_{rch}$).

SCCs are characterized by greatest fixed points of some functions where the fixed point is part of the reachable states or transitions. The selected function greatly affects the performance of the algorithm and the exact meaning of the fixed point.

Instead of computing SCCs, it is common to compute *SCC-hulls*, parts of the state space that are sure to contain at least one SCC. A family of algorithms implementing this approach are instances of the Generalized SCC-hull (GSH) algorithm [52]. The most famous one is the algorithm of Emerson and Lei, which have optimal complexity.

As an illustration, a simple function $f : \mathcal{S} \to \mathcal{S}$ that characterizes an SCC-hull is one that removes “dead-end” states:

$$f(S) = \{\mathbf{s} \mid \exists \mathbf{s}', (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$$

The greatest fixed point of f contained in the set of reachable states is a reachable SCC. In LTL, such an SCC containing at least one accepting state indicates the presence of a counterexample.

Another main contribution of this thesis presented in Section 4.2 is also a variant of the SCC-hull algorithms.

2.4 Abstraction in LTL Model Checking

Approximation of a state space by abstraction is a widely used technique to prove properties without exploring the full state space (as also seen in Section 2.2). In LTL model checking, SCC detection can sometimes be realized on (small) abstractions. With this technique, it is possible to implement on-the-fly symbolic LTL model checking and efficient SCC detection.

2.4.1 Abstraction based on the automaton

A common way of abstracting the state space based on the automaton describing the desired property. Such techniques include symbolic observation graph (SOG) [33, 39], its extension, symbolic observation product (SOP) [22] and self-loop aggregation product (SLAP) [23].

Symbolic observation graphs are aggregated Kripke structures: each state of the SOG is a set of states of the original model. Consecutive states are grouped by observable atomic propositions. An improvement of symbolic observation product is based on the observation that that as the automaton of the property progresses, the set of relevant atomic propositions decreases, allowing a more aggressive grouping. Furthermore, SOP can substitute the product, because it is in itself a tableau representing the system in terms of the property. This approach, however, assumes a globally stuttering property (i. e., operator X cannot be used).

Self-loop aggregation product works with every LTL formula. SLAP aggregates consecutive states by observing self loops of the Büchi automaton: states are aggregated if their labels satisfy the condition of self-loops, i. e., the automaton does not change its state when the system does.

2.4.2 Abstraction based on components

An extensive approach to using abstraction in SCC computation has been proposed in [58]. By defining a lattice of abstractions based on one or more components of the model, the paper presents strategies of using some of the abstractions to discard uninteresting parts of the state space and search in relevant components.

Each abstraction is obtained by projecting the state space to some of the components, keeping only those transitions that does not affect other components (in this sense, they are similar to the must abstraction of Section 4.3.2). Searching for SCCs in the abstract graphs can prove that no SCCs exist within the selected components. If an SCC is found, the algorithm also looks for accepting states, which offers another way of discarding irrelevant components. Completeness is achieved by refining the abstraction until an SCC is found or the full state space is searched.

2.5 Saturation in Model Checking

Saturation was first applied in CTL model checking in [13]. It was used to compute labels of states according to the CTL operators. While this approach worked, the algorithm became really efficient with the introduction of constrained saturation [59].

LTL model checking based on saturation was introduced in [35].

2.5.1 Fixed point algorithms using saturation

Saturation-based SCC computation has also been proposed by [60]. The two implemented algorithms are that of Xie and Beerel (XB) and the Transitive Closure (TC) algorithm. Both of them differ from SCC-hull algorithms, because they aim to compute exactly those states that belong to an SCC.

The main idea of the XC algorithm is to compute the set of forward and backward reachable states from a randomly picked seed state – their intersection gives an SCC. After removing the states of this SCC, the procedure is repeated until no states are left.

The TC method works by building the relation of reachability between reachable states. This relation can then be used to identify SCCs in a fully symbolic way.

While the XB algorithm is usually much faster, it does not scale well with the number of SCCs in the state space. TC does not enumerate the strongly connected components, it encodes them symbolically instead.

Due to the ability to reuse the caches of other runs of saturation, these algorithms can be very efficient in computing an exact counterexample detected by the algorithm presented in Section 4.2 algorithm.

Chapter 3

Symbolic Computation of the Synchronous Product

As discussed in Section 1.4.2, a crucial point of optimization in LTL model checking is the computation of the synchronous product on-the-fly during state space generation. In this chapter, two new algorithms will be presented to efficiently perform the on-the-fly computation symbolically. Both algorithms are based on saturation as a state space generation method and exploit the properties of constrained saturation.

These algorithms are the base of the complex symbolic on-the-fly model checking algorithm that is the main contribution of this thesis. Any of them can be used together with algorithms of Chapter 4 depending on the representation of the specification.

Formally, given a model M defined in Definition 16 and a (generalized) Büchi automaton $\mathcal{A}$, the task is to compute $\mathcal{A}_M \cap \mathcal{A}$ on-the-fly using saturation, where $\mathcal{A}_M$ is the Büchi automaton accepting the language produced by M as a Kripke structure (described in Proposition 1). The result is not exactly an automaton, since inputs are omitted from the representation – they are only used to synchronize the two automata, but are irrelevant in the actual model checking process.

Although the following sections assume that the inputs are given as non-generalized Büchi automata, the computation of the product is almost identical if a generalized Büchi automaton is used to represent the specification.

3.1 Tableau-based Product Computation

The first variant of the synchronous product computation algorithm directly uses constrained saturation. Exploiting a special form of Büchi automata, the main idea of the algorithm is to constrain the set of states rather than to precisely encode the synchronization in the transition relation.

To do this, Section 3.1.1 defines a special form of Büchi automata, then Section 3.1.2 presents the encoding of the product and the special constraint that drives saturation to explore the product state space. Finally, Section 3.1.3 provides a formal view on the correctness and efficiency of the algorithm.

Boolean functions and sets of Boolean variables

In this chapter, atomic propositions will play a key role. It is therefore necessary to define the precise meaning of some terminology.

Given a set of atomic propositions or variables AP, a valuation is a function f assigning truth values to every variable: $\mu : AP \rightarrow \mathbb{B}$. Such a valuation can also be represented by a subset of the atomic propositions $\alpha \subseteq AP$, where each variable in α takes the value *true*, while those not in α are *false*.

A set of possible valuations can be represented by a Boolean function, similarly to the representation of sets of binary vectors discussed in Section 1.4.3. An equivalent representation again using sets is a subset of 2^{AP}, where 2^{AP} is the set of all possible valuations.

Sometimes an alternative representation will also be used if the set of valuations can be described by fixed valuations of *some* variables. Two sets of atomic propositions can describe which variables need to be *true* and *false*, the remaining propositions are free variables. An equivalent Boolean formula can be written as the conjunction of some ponated and negated atomic propositions.

3.1.1 Tableau automata

The first step in automata theoretic LTL model checking is the translation of the temporal expression into a Büchi automaton. Observing the output automaton of widely used tableau-based conversion algorithms (such as the one presented in Section 1.3.4 or those of [38, 31, 30]) a common structural property can be observed that can be exploited to efficiently encode and compute the product. Let these kinds of Büchi automata be called *tableau automata*.

Definition 17 (Tableau automaton). Given a set of atomic propositions AP, a tableau automaton is a tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F}, L^+, L^- \rangle$, where:

- $\Sigma = 2^{AP}$ is the set of all possible valuations of atomic propositions;
- $\mathcal{Q}$ is the set of states;
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states;
- $\Delta \subseteq \mathcal{Q} \times \mathcal{Q}$ is the transition relation;
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states;
- $L^+ : \mathcal{Q} \rightarrow 2^{AP}$ and $L^- : \mathcal{Q} \rightarrow 2^{AP}$ are labeling functions, L^+ assigning propositions that must hold in the given state, while L^- assigning those that must not. ▪

A run of a tableau automaton over an input word w – just like in the case of Büchi automata – is an infinite sequence of states $\rho \in \mathcal{Q}^*$ starting with an initial state $\rho(0) \in \mathcal{I}$, but unlike simple Büchi automata, there is an additional requirement beyond satisfying the (simpler) transition relation. For every i, the input letter $w(i) \in 2^{AP}$ of the word representing a valuation of the atomic propositions must contain every proposition assigned to $\rho_w(i+1)$ by L^+ and must not contain any assigned by L^-, formally: $L^+(\rho_w(i+1)) \subseteq w(i)$ and $L^-(\rho_w(i+1)) \cap w(i) = \emptyset$. The acceptance condition of tableau automata is the same as that of Büchi automata (defined in Definition 11).

An alternative notation of the labels is a binary formula $\Phi_L(q) = p_1 \wedge \ldots \wedge p_m \wedge \neg q_1 \wedge \ldots \wedge \neg q_n$ where $p_i \in L^+(q)$ and $q_i \in L^-(q)$, i. e., atomic propositions assigned by L^+ are ponated in the formula, while those assigned by L^- are negated. The criterion of entering a state q is then that the valuation $\mathbf{p} \in \mathbb{B}^{|AP|}$ of atomic propositions in the input letter must satisfy $\Phi_L(q)$, denoted by $\mathbf{p} \models \Phi_L$.

It is important to emphasize that tableau automata are only a special form of Büchi automata, with the same expressive power. An equivalent Büchi automaton can be obtained using the following proposition.

Proposition 3 (Equivalent Büchi automaton of tableau automaton). Given a tableau automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F}, L^+, L^- \rangle$, an equivalent Büchi automaton is $\mathcal{A}' = \langle \Sigma, \mathcal{Q}', \mathcal{I}', \Delta', \mathcal{F}' \rangle$, where:

- $\mathcal{Q}' = \mathcal{Q}$, i. e., the states of the tableau automaton;
- $\mathcal{I}' = \mathcal{I}$, i. e., the initial states of the tableau automaton;
- $\mathcal{F}' = \mathcal{F}$, i. e., the accepting states of the tableau automaton;
- $\Delta' = \{(q, \alpha, q') \mid (q, q') \in \Delta, L^+(q') \subseteq \alpha, L^-(q') \cap \alpha = \emptyset\}$, i. e., instances of the transitions of the tableau automaton with every valuation of the atomic propositions defined by L^+ and L^-. ▪

Because of this equivalence, a Büchi automaton that can be directly obtained from a tableau automaton using the above transformation is said to be in *tableau form.* In Section 3.1.3, the proof of Theorem 2 will show a language preserving transformation of any Büchi automata into tableau form.

It is interesting to note that automata obtained from a Kripke structure are also in tableau form, since the labeling of a state is converted into the input of incoming transitions.

Example 13. *Figure 3.1 shows different representations of the LTL expression* $[a \mathsf{R} b]$. *On Figure 3.1(a), the corresponding Büchi automaton produced by the algorithm of Section 1.3.4 is shown exactly as it is described by the definition: each transition in the transition relation gets an own arc. Labels of the arcs are the sets representing valuations of the atomic propositions a and b. Figure 3.1(b) shows the same automaton in a more compact graphical representation, merging arcs and characterizing their labels with a conjunctive expression. This automaton is in tableau form, since all of the arcs targeting the same*

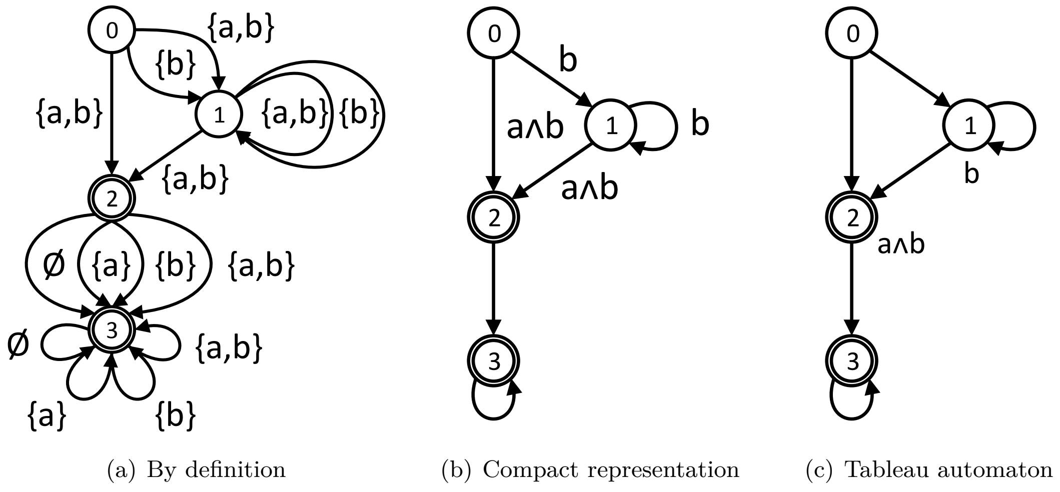


Figure 3.1. *Three forms of a Büchi automaton corresponding to the LTL property* $[a\ \mathsf{R}\ b]$.

state are labeled by the same conjunction (see Section 3.1.3). Moving these labels to the state itself results in a tableau automaton shown on Figure 3.1(c). Let ϕ_q *denote the conjunction on the arcs targeting* q*. Then the labeling functions* L^+ *and* L^+ *are defined such that every atomic proposition that is positive in* ϕ_q *is in* $L^+(q)$*, while those that are negative are returned by* $L^-(q)$*.*

3.1.2 Encoding the product automaton

To use saturation, the product automaton has to be represented by a model acceptable by saturation (introduced in Section 1.6.1). By the implicit definition of such a structure, saturation can be driven to compute the set of reachable states in the product automaton. Formally, a model M_P has to be defined in the form $M_P = \langle \mathcal{S}_\mathcal{P}, \mathcal{I}_\mathcal{P}, \mathcal{E}_\mathcal{P}, \mathcal{N}_\mathcal{P} \rangle$. This section discusses the opportunities in case the specification is given as a tableau automaton.

Saturation and variable ordering

Saturation is very sensitive to variable ordering, but the relation between the overall performance (runtime and memory usage) and the ordering is very complex and hard to determine.

It is usually advised to place related variables close to each other to enhance locality in the structure of decision diagrams as well. This strategy usually reduces the size of the decision diagram encoding of the set of states. In addition, the representation of events also tend to be smaller.

Another thing to consider is the *Top* values of events. A great deal of the power of saturation comes from the ability to apply the partitioned next-state relation locally, thus dividing the fixed point computation into smaller parts. This ability is even more enhanced by caching.

Although it is hard to say how much these values affect performance, one corner case is certainly undesirable: Setting every *Top* value to the index of the highest component, K. In this case, saturation would degrade to a somewhat breadth-first style iteration strategy, trying to apply every event on the top level of the decision diagram, effectively removing recursion and caching from the *Saturate* function.

States

Encoding of the states of the product is quite natural in the sense that they are pairs $(\mathbf{s}, q)$, which can be represented as a vector, thus it is possible to encode them in a decision diagram. The crucial part is the variable ordering.

Encoding the states of the tableau automaton can be done in any way from using a single variable to binary encoding. For now, assume that the state of the tableau automaton is encoded in a single variable s_a (i. e., it behaves as an additional component in M_P). Also assume that the original model M had a "proper" variable ordering. To keep the advantages of that ordering, the following heuristic is used.

Since state transitions of M and $\mathcal{A}$ are synchronized in M_P, s_a will be a supporting variable of every event. If the *Top* values of events are not to be raised at all (which is only the most straightforward, but not necessarily the best heuristic), putting s_a to the lowest level is an ideal choice.

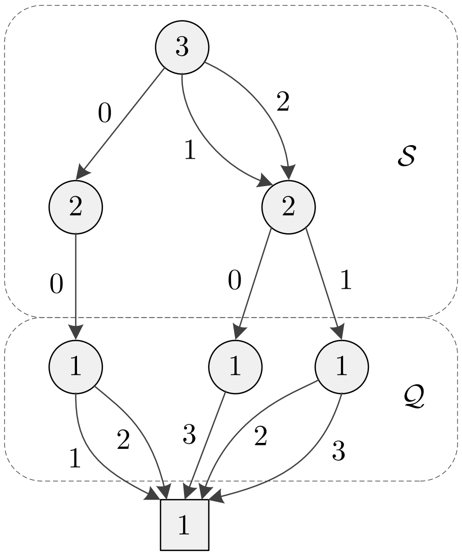


Figure 3.2. *Example of a decision diagram encoding of a product state space.*

Transitions

A similar consideration can be done in the case of transitions. There are two types of synchronization between steps of the model and steps of the automaton: *1)* they have to step together, and *2)* they have to read the same letter. The latter synchronization can be rephrased due to Proposition 1 (conversion of a Kripke structure to a Büchi automaton): $\mathcal{A}$ has to read the labels of the target state of M. This, in turn, means that the whole global state is required to decide if a transition of the automaton can be fired or not. Even if atomic propositions are assigned to local states, every local state that is a *subject* of a proposition must be known when the step of the automaton is selected. It is possible to include the corresponding variables in the support of events, but that would again raise the *Top* value of them.

The main idea of the algorithm is to omit the second synchronization constraint and fix the transitions by constraining the states. This way, the *relaxed next-state relation* of the product can be defined as the Cartesian product of the transition relations of M and $\mathcal{A}$: $\mathcal{N}_\mathcal{P} = \mathcal{N} \times \Delta$. To match the encoding of $\mathcal{S}_\mathcal{P}$, $\mathcal{N}_\mathcal{P}$ is also reordered to have the signature $\mathcal{N}_\mathcal{P} \subseteq (\mathcal{S} \times \mathcal{Q}) \times (\mathcal{S} \times \mathcal{Q})$. Events of the original system are kept, i. e., $\mathcal{E}_\mathcal{P} = \mathcal{E}$. The individual next-state relation of an event ε is $\mathcal{N}_\mathcal{P} = \mathcal{N}_\varepsilon \times \Delta$.

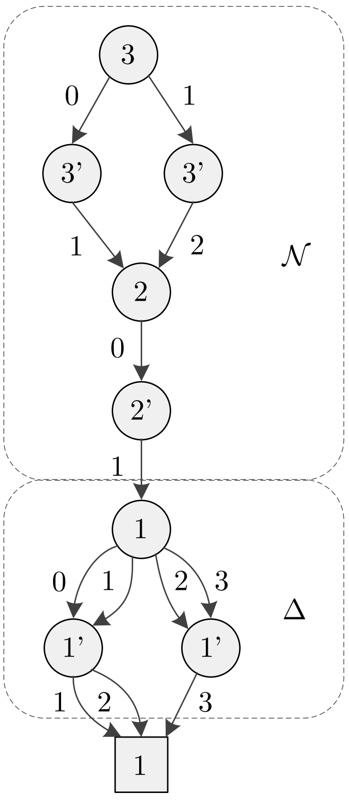


Figure 3.3. *Example of a decision diagram encoding of relaxed synchronous transitions.*

Algorithm 3.1: StepConstraint

```
input   : $c$ : node $i$, $l$: index
1 //$c$:  constraint node,
2 //$i$:  index of local state
3 //$k$:  level of local variable
output  : node
4 if $k = 1$ then
5     return $c[i]$;
6 foreach $p \in AP, Subject(p) = k$ do
7     $c \leftarrow c[p(i)]$ ;        //evaluate $p$ on $i$
8 return $c$;
```

Constraint

To fix the synchronization omitted in the encoding of the transitions, tableau automata provide a way to constrain the set of target states.

Definition 18 (Legal states of the product). A state $(\mathbf{s}, q) \in \mathcal{S}_{\mathcal{P}}$ of the product of a model M and a tableau automaton $\mathcal{A}$ is *legal* if the state of the automaton q can read the label of the model state $\mathbf{s}$, i. e., $L^+(q) \subseteq L(\mathbf{s})$ and $L^-(q) \cap L(\mathbf{s}) = \emptyset$.

With the set of legal states $\mathcal{C}_L = \{(\mathbf{s}, q) \mid L^+(q) \subseteq L(\mathbf{s}), L^-(q) \cap L(\mathbf{s}) = \emptyset\}$ as a constraint, constrained saturation will explore the state space of the product (see Section 3.1.3 for a proof).

The above construct assumed that the set of states of the model is known in advance. Even this limitation can be overcome by on-the-fly evaluation of states. Recall that decision diagrams are graphical representations of a *series of decisions* – this is exactly what is necessary to evaluate a state. Where local states are used to traverse the constraint in constrained saturation, an indirection layer can be included that *1)* determines the atomic proposition(s) assigned to the local state (except on the level of the automaton), *2)* evaluates node(s) of the constraint decision diagram, then *3)* returns the appropriate node just like the child operator ($n_k[i]$) would. This way, the constraint has to encode the set $\{(\mathbf{p}, q) \mid q \in \mathcal{Q}, \mathbf{p} \in \mathbb{B}^{|AP|}, \mathbf{p} \models \Phi_L(q)\}$, i. e., pairs of a valuation of the atomic propositions and an automaton state that can read it. The function that implements the indirection layer can be seen on Algorithm 3.1. Its is important to note that the ordering of atomic propositions in $\mathbf{p}$ has to be fixed and also has to match the order of components that are subjects of propositions.

3.1.3 Investigation of correctness and efficiency

In order to prove the correctness of the algorithm, it is necessary to show that every transition of the product can be simulated by the relaxed transitions and the constraint, and no new transitions are introduced by the construct.

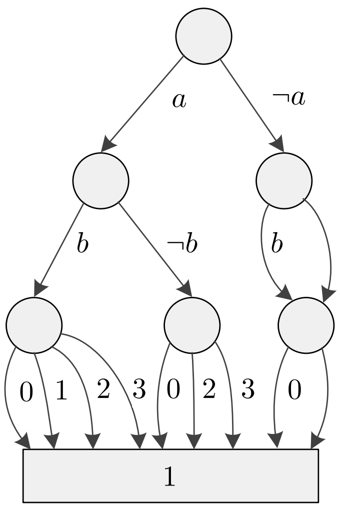


Figure 3.4. *Example of constraint corresponding to the tableau automaton of Figure 3.1(c).*

Theorem 1 (Correctness of tableau-based product computation). Given the transition relation $\Delta_\cap$ of the product automaton $\mathcal{A}_M \cap \mathcal{A}$ and the relaxed next-state relation $\mathcal{N}_\mathcal{P}$ with the set of legal states $\mathcal{C}_L$, every transition of $\Delta_\cap$ can be simulated by a transition allowed by constrained saturation with $\mathcal{N}_\mathcal{P}$ and $\mathcal{C}_L$ and vice versa. ▪

Proof. *Consider a transition of the product automaton $((\mathbf{s}, q), \alpha, (\mathbf{s}', q')) \in \Delta_\cap$. An equivalent transition in the relaxed next-state relation is $((\mathbf{s}, q), (\mathbf{s}', q')) \in \mathcal{N}_\mathcal{P}$ (see Definition 12). The target state $(\mathbf{s}, q)$ is a legal state because $\alpha = L(\mathbf{s}')$ and q could read it according to the product automaton ($L^+(q') \subseteq L(\mathbf{s}')$ and $L^-(q') \cap L(\mathbf{s}') = \emptyset$).*

In the other direction, consider a relaxed transition that is allowed by the constraint: $((\mathbf{s}, q), (\mathbf{s}', q')) \in \mathcal{N}_\mathcal{P}$ and $(\mathbf{s}', q') \in \mathcal{C}_L$. A corresponding transition in the product automaton is $((\mathbf{s}, q), \alpha, (\mathbf{s}', q')) \in \Delta_\cap$ where $\alpha = L(\mathbf{s}')$. If $(\mathbf{s}', q')$ is a legal state, then every transition of the automaton have to be able to read the labels of $\mathbf{s}'$ (due to Definition 18 and Proposition 2). □

Although the algorithm takes tableau automata as input, it is possible to transform any Büchi automata into tableau form. The following theorem gives an upper bound on the size of the equivalent tableau automaton, with a constructive proof providing a way to perform the transformation.

Theorem 2. Given a Büchi automaton $\mathcal{A}$ with a state count of $|\mathcal{Q}| = n$ over 2^{AP} as an alphabet, a tableau automaton accepting the same language can be constructed with a state count of at most $n \cdot \mathcal{O}\left(2^{|AP|}\right)$. ▪

Proof. *Constructive proof. Let q' be a state of $\mathcal{A}$ with incoming transitions $\delta \subseteq \Delta$. Denote the set of letters appearing in these transitions by $\sigma = \{\alpha \mid \exists (q, \alpha, q') \in \delta\}$. Let ϕ_σ denote a logical function in minimal disjunctive normal form that characterizes the letters in σ (such an expression always exists). Finally, denote the conjunctive clauses of ϕ_σ by $c(\phi_\sigma)$.*

If $|c(\phi_\sigma)| = 1$, replace the parallel transitions of δ (i. e., those with the same source and target states) with a single one of the tableau automaton, then label q' according to ϕ_σ (i. e., ponated propositions go into L^+, negated ones to L^-). Otherwise, split q' into $|c(\phi_\sigma)|$ states $\{q'_i\}$ according to the conjunctive clauses, each of them with the same outgoing transitions and input transitions whose α satisfies the corresponding clause. Repeat the step until every (potentially new) state is processed, then the result is a tableau automaton.

To examine the number of resulting states, consider that every state will be split only once. The number of states created when a state is split depends on the disjunction ϕ_σ, since every conjunctive clause in it will yield a new state. It is well known that the upper limit on the number of conjunctive clauses in a minimal disjunctive normal form of any Boolean formula over b binary variables is $\mathcal{O}\left(2^b\right)$. As a result, every state will be split into at most $\mathcal{O}\left(2^{|AP|}\right)$ new states. □

Although this exponential blowup may sound very disappointing, the emphasis is on that every automata built with widely-used tableau-based LTL translation algorithms are inherently tableau automata. The only restriction that this approach implies is a lesser ability to simplify the automata. Many simplification methods keep the tableau form, but the more powerful approaches will not be applicable (for an example, see [29] where every approach is applicable except simulation-based optimizations).

3.2 Decompositions-based Product Computation

This section presents a more general approach to compute the synchronous product $\mathcal{A}_M \cap \mathcal{A}$ with any kind of Büchi automata. Although the restrictions implied by using a tableau automaton are not too grave, saving a preprocessing step or achieving better optimization motivates the design of an improved algorithm. While the former approach used constrained saturation as is, the algorithm presented here uses a slightly modified version.

The main idea is the decomposition of the synchronous steps and the modification of constrained saturation to compute the possible combinations. The constraint will serve as a function instead of a set of allowed states, and mechanisms of constrained saturation will be used to evaluate it on states on the model. This approach is presented in Section 3.2.1), then Section 3.2.2 will investigate correctness from a formal point of view.

3.2.1 Composing the transition relation of the product automaton

Decomposing the synchronized transitions means that instead of building a single next-state relation encoding the state changes of both M and $\mathcal{A}$, transitions of the model and the automaton are stored and handled separately. The synchronization will be handled by the state space traversal algorithm, this is why constrained saturation has to be modified.

The encoding of states is the same as in the previous algorithm, the discussion about preserving the *Top* values also applies here.

To understand the motivation of the following construct, recall that constrained saturation evaluates a binary function on the states of the system and allows only those that make the function *true*. Also recall that relations can be interpreted as functions, and they can also be encoded in decision diagrams.

The next-state relation of M and the transition relation of $\mathcal{A}$ are kept separately, the only specific part of their encoding is that the transition relation of $\mathcal{A}$ is reordered to have the signature $\Delta \subseteq 2^{AP} \times \mathcal{Q} \times \mathcal{Q}$, and elements of 2^{AP} are represented by binary vectors like in the previous algorithm. Transitions of the model will serve as one half of the synchronous transitions, while $\Delta : 2^{AP} \to \mathcal{Q} \times \mathcal{Q}$ as a function will compute the other half. The input of the function will come from the same indirection layer that computed the labeling of global states in Section 3.1.2. Constrained saturation will be used to evaluate the function, and the resulting slice of the transition relation (denoted by $\mathcal{N}_\alpha$) will be used for stepping the automaton.

More precisely, as saturation is recursively calling itself and traverses the decision diagram, one of the following can happen. Assume that n is the number of levels in the decision diagram, while k is the highest of those that encode the automaton.

- If the current level belongs to M, i. e., it is above k, the next local transition of $\mathcal{N}$ is used and the constraint evaluates the atomic propositions corresponding to the target local state.
- If the current level is the kth, the current constraint node encodes $\mathcal{N}_\alpha$, the result of the function Δ. For this level and others below it, this relation is used instead of $\mathcal{N}$ to determine the next local transition. The constraint is not considered under this level.
- If the current level is below k (so it belongs to $\mathcal{A}$), the relation $\mathcal{N}_\alpha$ determined on level k is used to get the next local transition.

This evaluation step has to be included in both *Saturate* and *RelProd*. The modified version of these functions can be seen on Algorithms 3.2 and 3.3.

Formally, the algorithm applies the following set of transitions: $\{((\mathbf{s}, q), (\mathbf{s}', q')) \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}, (q, q') \in \Delta\left(L(\mathbf{s}')\right)\}$, i. e., the part of the transition affecting the model takes it into a state whose labeling is read by the transition of the automaton. $\Delta\left(L(\mathbf{s}')\right)$ can be seen as a partitioning of the transition relation based on the letters they read.

3.2.2 Investigation of correctness and efficiency

The correctness of the algorithm can be characterized in the same way as in Section 3.1.3.

Algorithm 3.2: ProdSaturate

```
  input   : $s_k$, $c$ : node
  //$s_k$:  node to be saturated,
  //$c_l$:  constraint node
  output  : node
* if $k = 1$ then
  | $n_{2k} \leftarrow c$;   //transitions of automaton
  else
  | $n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram;
  Return result from cache if possible;
  $t_k \leftarrow$ new node;
  foreach $i \in \mathcal{S}_k : s_k[i] \neq 0$ do
  | $c' \leftarrow c[i]$;
* | if  $c' \neq 0 \vee k = 1$  then
  | | $t_k[i] \leftarrow ProdSaturate(s_k[i], c')$;
  | else
  | | $t_k[i] \leftarrow s_k[i]$;        //no steps allowed
  repeat
  | foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
  | | $c' \leftarrow c[i']$;
* | | if $c' \neq 0 \vee k = 1$ then
  | | | $t_k \leftarrow$
  | | | $(t_k[i'] \cup ProdRelProd(t_k[i], c', n_{2k}[i][i']))$;
  until $t_k$ unchanged;
  $t_k = CheckUnique(t_k)$;
  Put inputs and results in cache;
  return $t_k$
```

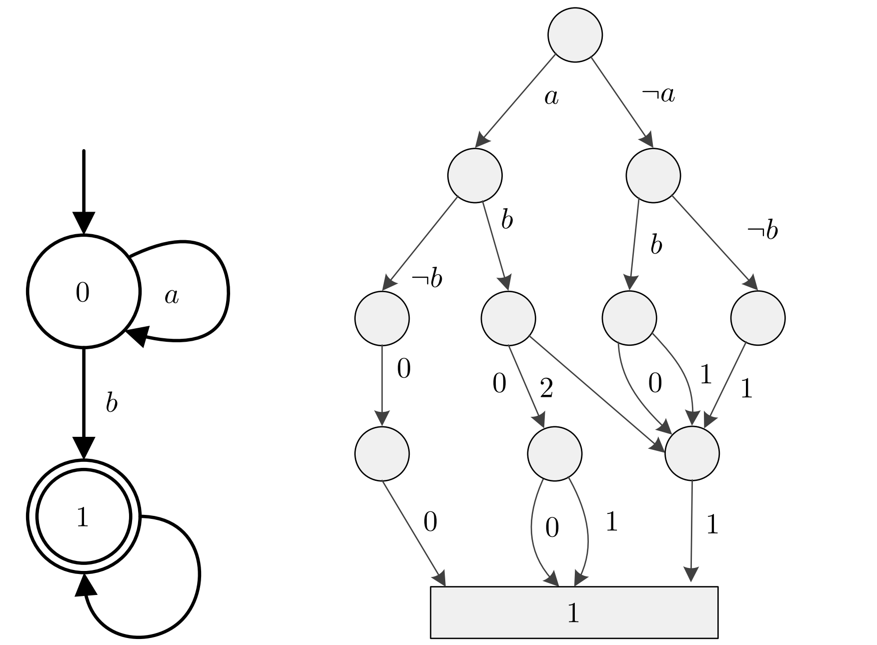
Algorithm 3.3: ProdRelProd

```
  input   : $s_k, c, n_{2k}$ : node
  //$s_k$:  node to be saturated,
  //$c$:  constraint node,
  //$n_{2k}$:  next state node,
  output  : node
  if $s_k = 1 \wedge n_{2k} = 1$ then
  | return 1;
  Return result from cache if possible;
* if $k = 1$ then
  | $n_{2k} \leftarrow c$;        //transitions of automaton
  $t_k \leftarrow$ new node;
  foreach $s_k[i] \neq 0 \wedge n_{2k}[i][i'] \neq 0$ do
* | $c' \leftarrow c[i']$;
  | if $c' \neq 0 \vee k = 1$ then
  | | $t_k[i'] \leftarrow$
  | | $(t_k[i'] \cup ProdRelProd(s_k[i], c', n_{2k}[i][i'])$;
  $t_k \leftarrow ProdSaturate(CheckUnique(t_k), c)$;
  Put inputs and results in cache;
  return $t_k$;
```

Theorem 3 (Correctness of decomposition-based product computation). Given the transition relation $\Delta_\cap$ of the product automaton $\mathcal{A}_M \cap \mathcal{A}$, the next-state relation $\mathcal{N}$ of M and the transition relation Δ of $\mathcal{A}$, every transition of $\Delta_\cap$ can be simulated by the decomposition-based product computation algorithm and vice versa. ▪

Proof. *The definition of the transition relation of the product (without the input letters) is* $\Delta_\cap = \{((\mathbf{s}, q), (\mathbf{s}', q')) \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}, (q, \alpha, q') \in \Delta, \alpha = L(\mathbf{s}')\}$. *The modified constrained saturation applies the transitions* $\{((\mathbf{s}, q), (\mathbf{s}', q')) \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}, (q, q') \in \Delta\,(L(\mathbf{s}'))\}$. *These sets are equivalent if* $(q, \alpha, q') \in \Delta \wedge \alpha = L(\mathbf{s}')$ *and* $(q, q') \in \Delta\,(L(\mathbf{s}'))$ *are equivalent, which follows from the interpretation of functions as relations.* □

As the proof suggests, the algorithm directly computes the synchronous transitions on-the-fly without actually storing them. This way, there is no limitation on the type of the Büchi automaton used. Furthermore, no additional storage is required above the representation of the separate relations, and the computational overhead is also negligible compared to traditional constrained saturation.



(a) Büchi automaton of [a U b].

(b) Constraint encoding the transition relation of the automaton.

Figure 3.5. *Minimal Büchi automaton for the LTL formula [a U b] and its encoding as a constraint.*

Chapter 4

Checking language emptiness

In this chapter, the most essential component of the contributed model checking algorithm is presented: algorithms for the detection of strongly connected components (SCCs). As seen in Section 1.3.5, the language emptiness problem can be solved by looking for SCCs in the state space, as well as Section 1.4.2 discussed the reduction of LTL model checking to language emptiness. This is why devising efficient algorithms is especially important here.

There are simple and powerful algorithms for SCC detection in the explicit case, where the state space is represented by a graph that can be traversed freely (see Section 2.1). In a symbolic setting, set operations can be used to compute an SCC-hull as a greatest fixed point in the state space (see Section 2.3.1). To introduce the advantage of on-the-fly SCC detection and early termination of explicit techniques into symbolic algorithms, hybrid approaches using abstraction have also spread (see Section 2.4). To further enhance the power of the latter approach, both symbolic and explicit methods will be introduced in this chapter, carefully designed to work together in a symbolic setting.

The symbolic algorithm will be a variant of traditional SCC-hull computation schemes, but optimized to process the state space *incrementally* during the exploration. On the verge of symbolic and explicit, saturation will be enhanced to collect recurring states, states that are visited more than once during the exploration and thus can indicate SCCs. Finally, a cheap abstraction considering decision diagram nodes will also be employed to use explicit algorithms to quickly reason about SCCs in the state space without actually trying to find them. The last two techniques are capable of making the overhead of on-the-fly model checking to almost disappear.

Algorithms of this chapter are orthogonal to algorithms of Chapter 3. Every one of them assumes that a model is given in the input format of saturation, no matter if it is a product automaton or anything else. Consequently, the devised methods can be used in settings other than LTL model checking, although in the thesis, the focus is on this particular application.

A conference paper [44] presenting the content of this chapter is accepted to be published in the TACAS (Tools and Algorithms for the Construction and Analysis of Systems) proceedings in 2015.

4.1 Lemma of Saturated State Spaces

Before presenting the algorithms, a seemingly trivial but fundamental lemma has to be declared. This observation will serve as the basis of the incremental symbolic SCC detection algorithm as well as the explicit and abstraction techniques.

Lemma 1 (Lemma of saturated state spaces). Given a saturated decision diagram node n_k, any path of $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ that is not present in the subspaces $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ (encoded by the children of n_k) contains at least one transition from $\mathcal{N}_k$.

Proof. *The lemma trivially follows from the definition of saturation. A set of states represented by a saturated node n_k is closed with regard to $\mathcal{N}_{\leq k}$, even if an additional component is appended to each state as $\mathcal{B}_{[i]}(n_k)$ is defined.* □

An implication of the lemma is that there is no way to traverse the boundaries of $\mathcal{B}_{[i]}(n_k)$ without using at least one transition from $\mathcal{N}_k$, since the state spaces $\langle \mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k} \rangle$ are *disjoint* (they differ in the local state of component k).

4.2 Incremental Symbolic SCC Computation

In the presented model checking algorithm, on-the-fly SCC detection is intended to be achieved by running SCC computations frequently during the state space generation. Certainly, this strategy would mean a massive overhead if repetition of work would not be addressed, so the main requirement towards the design of this algorithm was incrementality.

The context of the algorithm is the following. When saturation processes the state space (see Chapter 3), SCC detection phases will be inserted whenever a node n_k becomes saturated. Processing saturated nodes has the advantage of handling a set of (sub)states $\mathcal{B}(n_k)$ that is closed with regard to events independent from higher levels ($\mathcal{N}_{\leq k}$). This means that the set will not change anymore during the exploration, i. e., each closed set has to be processed only once.

Even though a set with its related events will be processed only once, the recursive definition of saturation will cause such sets to appear again as part of larger sets encoded by the parent node in the decision diagram (see Section 1.5.3). This means that the algorithm has to be able to distinguish parts of the state space already processed ($\mathcal{B}_{[i]}(n_k)$ and $\mathcal{N}_{<k}$) and focus only on new opportunities gained by considering new events. This is exactly the base idea of the algorithm: in each run, look for only those SCCs that contain at least one transition form $\mathcal{N}_k$.

Elementary steps of the fixed point computation

The idea can be implemented by discarding transitions from a set of "new" transitions $\mathcal{N}_{new}$ that cannot be closed to form a loop. In other words, a transition is discarded if its *source state* cannot be reached from its *target state.* In SCC-hull algorithms, sets of states are processed iteratively to eventually get rid of "bad" states (dead-end states, for example) by computing a greatest fixed point of some function in the original set. Now, a set of transitions have to be processed, and the function is the following ($\mathcal{N}$ is the set of all transitions, i. e., both "old" and "new" transitions):

$$f(N) = \{(\mathbf{s}_1, \mathbf{s}_1') \mid (\mathbf{s}_1, \mathbf{s}_1') \in N, \exists (\mathbf{s}_2, \mathbf{s}_2'), \mathbf{s}_1 \in \mathcal{N}^*(\mathbf{s}_2')\}$$

Function f in words

The function returns a subset of N. Transitions of this subset can be characterized by a few equivalent descriptions.

- The source state of the returned transitions can be reached through transitions of $\mathcal{N}$ from the target state of at least one transition in N.
- There is at least one transition in N after which a returned transition can become enabled on some path consisting of transitions in $\mathcal{N}$.
- The returned transitions start from a set of states reachable through $\mathcal{N}$ from target states of transitions in N.

The formal goal of the proposed SCC detection algorithm is to compute the greatest fixed point of f as $N_\Theta = f(N_\Theta)$ where $N_\Theta \subseteq \mathcal{N}_{new}$. The following lemma justifies the definition of function f and will prove the correctness and completeness of the incremental SCC detection algorithm.

Lemma 2 (Correctness and completeness). Given a transition system $\langle \mathcal{S}, \mathcal{N} \rangle$ and a set of "new" transitions $\mathcal{N}_{new} \subseteq \mathcal{N}$, the fixed point N_Θ is *empty* iff $\langle \mathcal{S}, \mathcal{N} \rangle$ does not contain any SCC with transitions from $\mathcal{N}_{new}$.

Proof. *The two directions are proven separately.*

($\leftarrow$)*: Indirect proof. Suppose there is a strongly connected component* Θ *in the transitions system that contains at least one transition from* $\mathcal{N}_{new}$*, let this be* $(\mathbf{s}, \mathbf{s}')$*. Also suppose that the fixed point* N_Θ *is empty. Consider* $f(\{(\mathbf{s}, \mathbf{s}')\})$*. By the definition of a strongly connected component, every state of* Θ *is reachable from every other state of* Θ*, so* $\mathbf{s}$ *is also reachable from* $\mathbf{s}'$*. Reachability in terms of* $\mathcal{N}$ *is defined by inclusion in the set of states* $\mathcal{N}^*(\mathbf{s}')$*, so* $(\mathbf{s}, \mathbf{s}') \in f(\{(\mathbf{s}, \mathbf{s}')\}$ *can be concluded that contradicts the assumption of* N_Θ *being empty. Note that because of the transition* $(\mathbf{s}, \mathbf{s}')$*,* Θ *is a real (but not necessarily nontrivial) SCC even if* $\mathbf{s} = \mathbf{s}'$ *because of the self-loop.*

($\rightarrow$)*: The other direction is also proved indirectly. Suppose there is no strongly connected component in the transitions system that contains at least one transition from $\mathcal{N}_{new}$. Also suppose that the fixed point is nonempty. Take a transition $\nu_1 = (\mathbf{s}_1, \mathbf{s}'_1)$ from N_Θ. Since it is in the fixed point, its source state must be reachable from the target state of some other transition $\nu_2 = (\mathbf{s}_2, \mathbf{s}'_2)$. Now consider this transition and repeat the process. Since N_Θ is finite, at some point, the transition ν_i will be the same as some transition before: $\nu_i = \nu_j$, where $j < i$. However, since $\mathbf{s}_i$ is reachable from $\mathbf{s}'_j$ and $\mathbf{s}_j$ is reachable from $\mathbf{s}'_i$, they must be in an SCC, which leads to a contradiction.* □

Incremental steps of the fixed point computation

An incremental step has to compute the fixed point N_Θ. This is implemented by the function *DetectSCC* that can be seen on Algorithm 4.1. Checking reachability is performed using saturation. The main advantage of this is that caches can be reused in the SCC detection phase as well, just like the decision diagram structures built during the state space exploration.

Compared to the definition of f, the implementation is slightly different. Instead of handling a set of transitions, the implementation reduces the problem to sets of states by considering the source states and target states of every transition in $\mathcal{N}_{new}$. The sets of source and target states are denoted by $\mathcal{S}^-$ and $\mathcal{S}^+$ respectively. Furthermore, a set of states $\mathcal{S}$ (typically the set of states discovered so far) is also input to *DetectSCC* to constrain SCCs – during the state space generation (especially when saturation is used), $\mathcal{N}$ may contain transitions that are not in $\mathcal{S} \times \mathcal{S}$.

The core of *DetectSCC* is the *filtering cycle*, where function f is implemented and performed iteratively until no more changes occur. In each iteration, the sets $\mathcal{S}^-$ and $\mathcal{S}^+$ are filtered:

- Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{S}^+$ are removed.
- Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

Lemma 2 still holds if N_Θ is substituted by $\mathcal{S}^-$ and $\mathcal{S}^+$. Note that these two will always be both empty or both nonempty at the fixed point, because a transition has to have a source *and* a target state. However, the iteration can be stopped one step before – if any of them becomes empty, the next step will discard every state from the other one as well.

The number of iterations in the filtering cycled has an upper bound of $\mathcal{O}(|\mathcal{N}_{new}|)$, since in every step, at least one transition is discarded from the set. Methods to make the initial set of "new" transitions smaller and thus reduce the number of required steps are discussed in Sections 4.4.1 and 4.4.3.

An incremental step will always assume that there is no SCC in the transition system that *does not contain* a transition from $\mathcal{N}_{new}$ (otherwise the on-the-fly model checking algorithm would have already been terminated). With this, the completeness of the algorithm depends on the strategy of applying the incremental steps.

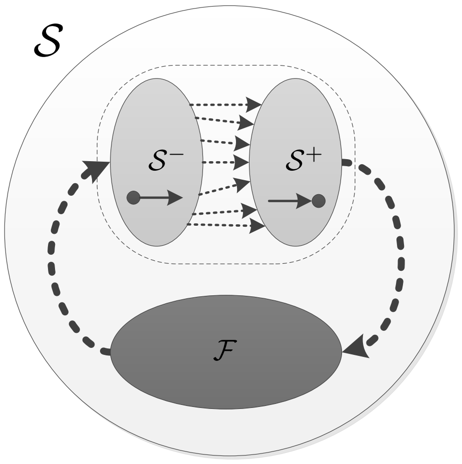


Figure 4.1. *Illustration of Algorithm 4.1.*

On-the-fly search using the incremental steps

The last design question is how and when to call *DetectSCC*. The chosen design is to call *DetectSCC* whenever a node n_k becomes saturated. The set of states to constrain the search is $\mathcal{B}(n_k)$, transitions are $\mathcal{N}_{\leq k}$ and new transitions are $\mathcal{N}_k$ (it is trivial that $\mathcal{N}_k \subseteq \mathcal{N}_{\leq k}$).

In the general case (and in breadth-first style strategies), the whole next-state relation of a model can be partitioned by the traversal strategy, for example, a breadth-first exploration would partition the transitions into "layers" based on their distance from the initial state. If *DetectSCC* is called on each partition as new transitions, Lemma 2 will imply that the algorithm is correct and complete, i. e., it finds an SCC exactly if there exists one.

In saturation, however, the exploration is recursive, so calling *DetectSCC* after a node is saturated does not fall into the above case. Proving that the algorithm is complete can thus be performed inductively.

Theorem 4 (Completeness of incremental SCC detection). Calling *DetectSCC* during state space generation every time after a node is saturated yields a complete algorithm, i. e., in at least one call, the fixed point will not be empty iff there exists an SCC in the state space.

Proof. *Inductive proof. It is trivial that the terminal nodes do not encode any SCCs, because no transitions are associated with level* 0*. Assume the children of a decision diagram node n_k together with their related transitions $\mathcal{N}_{<k}$ (that is, $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k}\rangle$) do not contain SCCs either. Proving that the fixed point will be nonempty iff there exists an SCC*

Algorithm 4.1: DetectSCC

```
   input  : $\mathcal{S}, \mathcal{N}, \mathcal{N}_{new}$ : set
 1 //$\mathcal{S}$: set of states,
 2 //$\mathcal{N}, \mathcal{N}_{new}$: set of transitions
   output : bool
 3 $\mathcal{S}^- \leftarrow \mathcal{N}_{new}^{-1}(\mathcal{S})$;    $\mathcal{S}^+ \leftarrow \mathcal{N}_{new}(\mathcal{S}^-)$;
 4 if $\mathcal{S}^+ = \emptyset$ then return false;
 5 repeat
 6     $\mathcal{S}^- \leftarrow \mathcal{S}^- \cap \mathcal{N}^*(\mathcal{S}^+)$;
 7     $\mathcal{S}^+ \leftarrow \mathcal{S}^+ \cap \mathcal{N}_{new}(\mathcal{S}^-)$;
 8 until $\mathcal{S}^+$ and $\mathcal{S}^-$ unchanged;
 9 return $\mathcal{S}^- \neq \emptyset \wedge \mathcal{S}^+ \neq \emptyset$
```

in $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k}\rangle$ would imply that when the root node is saturated and the algorithm stops, the statement of the theorem would hold. The inductive hypothesis directly follows from Lemmas 1 and 2. If there exists an SCC, it must contain a path that is not present in $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k}\rangle$ (otherwise $\langle \mathcal{B}(n_k[i]), \mathcal{N}_{<k}\rangle$ would also contain the SCC), so according to Lemma 1 it will contain at least one transition from $\mathcal{N}_k$. This would cause the fixed point to be nonempty (Lemma 2). □

4.2.1 Extensions to support fair SCCs

When looking for fair SCCs, i. e., SCCs containing at least one state from a set of states $\mathcal{F}$, the algorithm can be extended to involve $\mathcal{F}$ as the third set in the filtering cycle. Operations performed in the cycle are then the following:

- Elements of $\mathcal{F}$ that are not reachable from $\mathcal{S}^+$ are removed.
- Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{F}$ are removed.
- Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

The first two operations ensure that if a transition of $\mathcal{N}_{new}$ is in an unfair SCC (i. e., does not contain any states from $\mathcal{F}$), it is removed from the fixed point. Note that even accepting trivial SCCs (a single state of $\mathcal{F}$ with a transitions of $\mathcal{N}_{new}$ as a self loop) are found this way, because a state is by definition reachable from itself (i. e., reachability as a relation is reflexive).

When looking for accepting SCCs during LTL model checking, $\mathcal{F}$ is the set of accepting states. Supporting multiple acceptance sets to directly use generalized Büchi automata in the model checking algorithm is possible, but it is the subject of the author's future work.

4.3 Indicators of SCCs in the Product Automaton

After presenting an incremental way to detect strongly connected components during state space generation, this section will introduce methods to prove the absence of SCCs without performing symbolic fixed-point computations. These methods will be used to decide if a symbolic check should be performed when a node is saturated or it can be safely omitted.

When looking for accepting SCCs, checking the absence of accepting states is a usual optimization in similar algorithms, for example in the abstraction refinement approach presented in [58]. The contribution of this thesis goes two steps further. Section 4.3.1 introduces the use of recurring states and a specialized algorithm to compute them, while Section 4.3.2 presents new abstraction techniques tailored to decision diagrams that allow the use of explicit algorithms directly to reason about the presence or absence of SCCs.

4.3.1 Recurring states during state space exploration

Recurring states are those that have already been discovered before reaching them again during state space exploration. To precisely define them, let a concrete *exploration* ϵ of a fully reachable, connected state space $\langle \mathcal{S}, \mathcal{N} \rangle$ be a sequence of subsets of $\mathcal{S}$, where each element of the sequence contains states discovered in that step: $\epsilon \in (2^{\mathcal{S}})^*$ such that $\epsilon(i+1) \subseteq \mathcal{N}(\mathcal{S}_i)$ for every $0 < i \leq |\epsilon|$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$. An exploration is *full* if $\mathcal{S}_{|\epsilon|} = \mathcal{S}$ and the exploration algorithm considered every enabled transition in $\mathcal{N}$.

Definition 19 (Recurring states). Given an exploration $\epsilon \in (2^{\mathcal{S}})^*$, the set of recurring states in each step is $\mathcal{R}_i = \mathcal{S}_i \cap \epsilon(i+1)$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$. ▪

Many explicit algorithms rely on recurring states as indicators of SCCs (for example [54] and [37]). Indeed, they are candidates of being in an SCC, since there are only two cases in which they can appear: the exploration reached them again on a parallel path, or a previous state of a path is reached again, i. e., a loop is found. Using the following proposition, recurring states can be used to reason about SCCs.

Proposition 4. Given a state space containing an SCC, any full exploration will yield at least one recurring state. ▪

According to the proposition, recurring states offer a cheap way to distinguish situations where there is no chance of finding an SCC – situations that often arise during an on-the-fly algorithm.

On-the-fly collection of recurring states

Since a basic step in saturation is performed by applying some $\mathcal{N}_\varepsilon$ (i. e., computing the relational product of a set of states and $\mathcal{N}_\varepsilon$), recurring states have to be collected in this

Algorithm 4.2: SaturateRec

```
    input  : $s_k$ : node      //to saturate
    output : node
  1 Return result from cache if possible;
  2 $n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram;
  3 $t_k \leftarrow$ new node;
  4 $\mathcal{A}^{\exists}_{t_k} \leftarrow (\mathcal{S}_k, \emptyset)$;
  5 foreach $i \in \mathcal{S}_k : s_k[i] \neq \mathit{0}$ do
  6     $t_k[i] \leftarrow SaturateRec(s_k[i])$;
* 7 $r_k \leftarrow$ new node;      //recurring states
  8 repeat
  9     foreach $s_k[i] \neq \mathit{0} \wedge n_{2k}[i][i'] \neq \mathit{0}$ do
*10         $r'_{k-1} \leftarrow$ new node;
 11         $u_k \leftarrow$
            $RelProdRec(t_k[i], n_{2k}[i][i'], t_k[i'], r'_{k-1})$;
 12         $t_k[i'] \leftarrow (t_k[i'] \cup u_k)$;
*13         $r_k[i'] \leftarrow (r_k[i'] \cup r'_{k-1})$;
 14 until $t_k$ unchanged;
 15 Put inputs and results in cache;
 16 return $CheckUnique(t_k)$;
```

Algorithm 4.3: RelProdRec

```
    input  : $s_k, n_{2k}, o_k$ : node
  1 //$s_k$:  node to be saturated,
  2 //$n_{2k}$:  next state node,
  3 //$o_k$:  old node
    in-out : $r_k$ : node //recurring states
    output : node
  4 Return result from cache if possible;
  5 if $s_k = \mathit{1} \wedge n_{2k} = \mathit{1}$ then
* 6     if $o_k = \mathit{1}$ then $r_k \leftarrow s_k$;
  7     return 1;
  8 $t_k \leftarrow$ new node;
  9 foreach $s_k[i] \neq \mathit{0} \wedge n_{2k}[i][i'] \neq \mathit{0}$ do
*10     $r'_{k-1} \leftarrow$ new node;
 11     $t_k[i'] \leftarrow (t_k[i'] \cup$
        $RelProdRec(s_k[i], n_{2k}[i][i'], o_k[i'], r'_{k-1})$;
*12     $r_k[i'] \leftarrow (r_k[i'] \cup r'_{k-1})$;
 13 $t_k \leftarrow SaturateRec(CheckUnique(t_k))$;
 14 Put inputs and results in cache;
 15 return $t_k$;
```

granularity as well. In this context, the desired output of the function *RelProd* would be two sets of states: the result of the relational product and the set of recurring states.

Instead of performing the costly intersection at the end of the function, the collection of recurring states can be done on the fly. In general, constrained saturation did exactly the same, as it only allowed steps that stay in a certain set of states. This set is now the set of "old" states, i. e., those that were passed to *RelProd*.

Algorithm 4.3 shows the implementation in *RelProdRec*. The function basically performs a constrained and an unconstrained saturation simultaneously, gathering the results to two separate sets (decision diagrams). An important note is that contrary to normal constrained saturation, the recursion occurs even if the constraint won't allow the step, because the main functionality is the computation of the relational product.

In each *SaturateRec*(the version of *Saturate* seen on Algorithm 4.2), all recurring states produced by transitions of $\mathcal{N}_k$ are collected separately as the union of sets returned by *RelProdRec*– recurring states encountered by applying lower level events during recursive *SaturateRec*calls are processed there. The arguments of *RelProdRec* are the original node, the next-state node and the node that represents the relevant part of the old states (with which the results of the relational product will be merged).

It is important to note that this mechanism is useful only if the inputs and results of *RelProdRec* are cached. Otherwise, repeating a previous call would by definition recognize every target state as a recurring state, since they have already been reached by the previous call.

4.3.2 Using abstractions to reason about emptiness

Hybrid model checking algorithms usually use symbolic encoding to process huge state spaces, while introducing clever abstraction techniques to produce an abstract model on which explicit graph algorithms can be used. In this context, the goal of abstraction is to reduce the size of a system's state space while preserving certain properties, such as the presence or absence of SCCs. The purpose of abstractions is no different is this thesis wither, they are used to reason about SCCs. However, unlike in most approaches in this domain, multiple abstractions are used, ordered in a hierarchy matching the structure of the underlying decision diagram to build an inductive proof about strongly connected components of the state space.

In a symbolic setting, components of the model provide a convenient basis for abstraction. In LTL model checking, it is usual to use the Büchi automaton or its observable language to group states and build an abstraction from these aggregates. The abstraction framework presented in [58] goes beyond using only one kind of abstraction and explores strategies on a tableau of possible abstractions based on one or more components. For a more detailed summary of using abstraction in LTL model checking, refer to Section 2.4.

In addition to selecting the basis, there are multiple ways to define an abstraction in terms of a component. To illustrate this, two simple abstractions are presented before introducing a new approach of using the structure of a decision diagram to define a more powerful abstraction.

Abstraction of graphs

Using abstraction on graphs (such as states spaces) is the process of obtaining a new graph that is usually smaller, and nodes and arcs of the original graph can be mapped into nodes and arcs of the abstract graph. The term abstraction is often used to name the abstract graph as well as the process that constructs it.

Formally, given two graphs $G = \langle V, E \rangle$ and $A = \langle V_A, E_A \rangle$, A is an abstraction of G if there is a *surjective* abstraction function that maps V onto V_A and E onto E_A. It is usual to define A in terms of G – this way, the function is part of the definition.

If nodes of the graph are tuples, such as in the case of concurrent models, components can be the basis of the abstraction. A selection of components can be omitted from every state and identical states can be merged to obtain an abstraction based on a few components only. The set of abstract transitions can be defined in many ways, depending on the application of the abstraction.

Simple abstractions

Using abstractions to answer binary decisions has two potential goals. One can create an over-approximating abstraction that can say a definite *yes* (these are called *must ab-*

stractions), or an under-approximating one that can say a definite *no* (these are *may abstractions*). To construct an abstraction based on a single component of the system, the definition of an abstraction function is required for both the states and the transitions in the global state space. Abstracting states is straightforward, as the set of local states $\mathcal{S}_k$ of component k can be used directly. Regarding may and must abstractions, different transformations have to be defined for the transitions of the state space.

Definition 20 (May abstraction). May abstraction of state space $\langle \mathcal{S}, \mathcal{N} \rangle$ based on component k is $\mathcal{A}_k^\exists = \langle \mathcal{S}_k, \mathcal{N}_k^\exists \rangle$, where $\mathcal{S}_k$ is the set of reachable local states of component k, and $\mathcal{N}_k^\exists = \{(s_k, s'_k) \mid \exists \varepsilon \in \mathcal{E}, \mathit{supp}(\varepsilon) = \{k\}, \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_\varepsilon\}$, i. e., the projections of events to component k. ▪

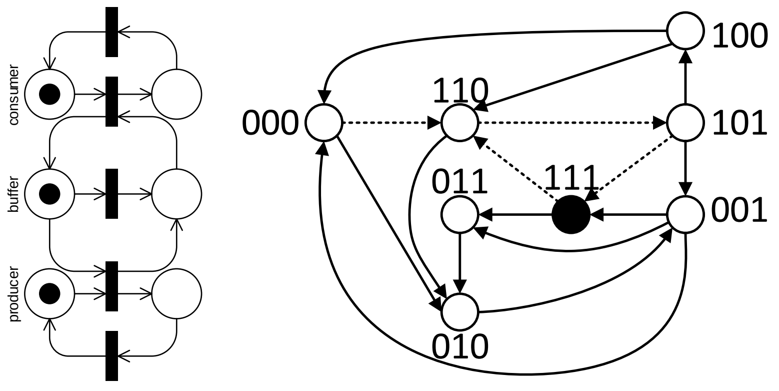
Definition 21 (Must abstraction). Must abstraction of state space $\langle \mathcal{S}, \mathcal{N} \rangle$ based on component k is $\mathcal{A}_k^\forall = \langle \mathcal{S}_k, \mathcal{N}_k^\forall \rangle$, where $\mathcal{S}_k$ is the set of reachable local states of component k, and $\mathcal{N}_k^\forall = \{(s_k, s'_k) \mid \exists \varepsilon \in \mathcal{E}, k \in \mathit{supp}(\varepsilon), \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_\varepsilon\}$, i. e., transitions that affect only component k. ▪

The must abstraction of transitions is defined to keep only those transitions that correspond to events fully within the support of the chosen component. May abstraction preserves every local transition, but omits the synchronization constraints (i. e., assumes that if a transition is enabled in component k, it is globally enabled).

Due to this construction, it is sometimes possible to reason about the presence or absence of global SCCs. If there is an SCC in a single must abstraction, it is the direct representation of one or more SCCs of the global state space. Complementary, if there is no SCC in the may abstraction of *any* component, then the global state space cannot contain any SCCs either.

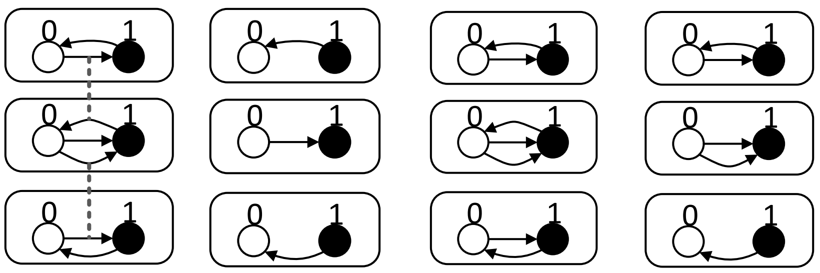
These abstractions usually yield small state graphs that can be represented explicitly. Running linear-time explicit algorithms on them gives a very cheap opportunity to possibly prove or refute the presence of SCCs before symbolic methods are used. Moreover, the definition of may and must abstractions implies $\mathcal{N}_k^\forall \subseteq \mathcal{N}_k^\exists$, so running the explicit SCC computation on a may abstraction and checking if every transition of a possible SCC is in $\mathcal{N}_k^\forall$ effectively considers both cases at the same time.

Example 14. *As an example, observe Figure 4.2 that illustrates the Petri net model of a producer-consumer system, also showing the explicit state graph. Transitions of the system are shown on Figure 4.3(a), with connected arcs representing a single transition affecting multiple components. In this case, every transition belongs to a separate event (events are related to transitions of the Petri net). Events affecting multiple components can be regarded as synchronization constraints between* local transitions. *Abstractions can be acquired by removing synchronizations and local transitions. Figure 4.3(c) and 4.3(b) depict the transitions transformed by may and must abstractions. If the goal is to find an SCC containing the state where only the places at the left of the Petri net are marked* *(depicted as a black state on Figure 4.2(b)), none of the simple abstractions can give an exact answer.*



(a) Petri net model. (b) State space.

Figure 4.2. *Producer-consumer model with non-deterministic buffer.*



(a) Original (b) Must (c) May (d) Nodewise

Figure 4.3. *The effect of the abstractions to the transitions*

Node-wise abstraction.

Among the main algorithmic contributions of this thesis, the last one is a specialized abstraction that fits saturation and the presented incremental symbolic SCC detection algorithm, as well as it complements Proposition 4 (about recurring states) as a cheap way to prove the absence of SCCs. The goal of the following construct is to match the order in which events are processed during saturation, as well as the structure of the underlying decision diagram.

Definition 22 (Node-wise abstraction). Node-wise abstraction of state space $\langle \mathcal{S}, \mathcal{N} \rangle$ with regard to node n_k is $\mathcal{A}^{\exists}_{n_k} = \langle \mathcal{S}_{n_k}, \mathcal{N}^{\exists}_{n_k} \rangle$, where $\mathcal{S}_{n_k} = \{i \mid n_k[i] \neq \mathbf{0}\}$, i. e., local states encoded by the arcs of n_k, and $\mathcal{N}^{\exists}_{n_k} = \{(s_k, s'_k) \mid s_k, s'_k \in \mathcal{S}_{n_k}, \exists((\ldots, s_k, \ldots), (\ldots, s'_k, \ldots)) \in \mathcal{N}_k\}$, i. e., the projections of events $\mathcal{E}_k$ to component k.

Node-wise abstraction can be regarded as a may abstraction of the states space $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ based on component k. Just like may abstractions, a series of node-wise abstractions can also be used to reason about global SCCs. Moreover, this can be done inductively during the state space generation. The following theorem gives the basis for this inductive method of using node-wise abstractions to prove the absence of SCCs.

Theorem 5 (Node-wise abstraction and SCCs). Given a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ with regard to a saturated node n_k, the state space $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ does not contain any SCCs if *1)* neither $\mathcal{A}^{\exists}_{n_k}$ *2)* nor the state spaces $(\mathcal{B}_{[i]}(n_k), \mathcal{N}_{<k})$ belonging to the children of n_k contain an SCC. ▪

Proof. *Indirect proof. Suppose that $\langle \mathcal{B}(n_k), \mathcal{N}_{\leq k} \rangle$ contains an SCC with a state in $\mathcal{B}_{[i]}(n_k)$ (i. e., , component k is in local state i). There are three possible cases to realize this. The SCC:*

1. *is fully within $\mathcal{B}_{[i]}(n_k)$ and contains transitions only from $\mathcal{N}_{<k}$;*
2. *is fully within $\mathcal{B}_{[i]}(n_k)$ and contains transitions from $\mathcal{N}_k$, but they do not change the local state of component k;*
3. *contains a state from at least one other $\mathcal{B}_{[j]}(n_k)$ $(i \neq j)$ as well;*

Case 1 contradicts assumption 2, while case 2 is also a contradiction with assumption 1, since not changing the local state of component k would mean a self loop in $\mathcal{A}^{\exists}_{n_k}$ constituting a trivial SCC. Proving that case 3 also yields a contradiction is based on Lemma 1. To reach the state in $\mathcal{B}_{[j]}(n_k)$ from the state in $\mathcal{B}_{[i]}(n_k)$, a path has to use at least one transition from $\mathcal{N}_k$. These transitions will also form a path in $\mathcal{A}^{\exists}_{n_k}$ from i to j. To be in an SCC, there has to be a path from the state in $\mathcal{B}_{[i]}(n_k)$ to the state in $\mathcal{B}_{[j]}(n_k)$ also using at least one transition from $\mathcal{N}_k$, which in turn also forms a path in $\mathcal{A}^{\exists}_{n_k}$ from j to i. Since i and j are reachable from each other, they are in an SCC of $\mathcal{A}^{\exists}_{n_k}$, which is a contradiction. □

Explanation of the proof

The main idea of the proof is that node-wise abstraction represents the effects of the events $\mathcal{E}_k$ exactly on the level of their *Top* value. At the time a node n_k becomes saturated, the only transitions that can change the local states of component k are in $\mathcal{N}_k$. Node-wise abstractions contain the images of exactly these transitions, thus they describe the possible transitions between sets of substates encoded by the children of n_k. This is why they can be used to identify *one-way walls* that separate the possible spaces for SCCs.

Note that the theorem did not specify how to prove assumption 2, meaning that even if the corresponding node-wise abstraction did contain an SCC (which only implies the *possible* presence of a global SCC), the symbolic fixed point computation algorithm of Section 4.2 can check if the abstract SCC candidate is realizable in the global state space. This way, the series of saturated nodes can inductively prove the absence of SCCs by the end of the state space generation.

Example 15. *Observing Figure 4.3 again, node-wise abstractions of the state space can be seem on Figure 4.3(d). As Theorem 5 suggests, it is unnecessary to start symbolic SCC detection until the top level becomes saturated.*

By the time a node is saturated, its node-wise abstraction will not change anymore. This way, a single abstraction has to be built and analyzed only once. The computation of node-wise abstractions is very simple and cheap. It can be done on demand by projecting the next-state relation of corresponding events to the *Top* component, or on-the-fly during saturation by adding vertices and arcs each time a new local state is discovered or a new transition of the corresponding events is fired, respectively. A simple must abstraction can also be examined as part of computing SCCs of the node-wise abstraction by checking if every local transitions used in the SCC belong to events having only the current component as a supporting one. If this is the case, that SCC is inherently realizable and can be returned as a counterexample.

4.4 On-the-fly Incremental Hybrid Model Checking of LTL Properties

In this section, the building blocks presented so far are assembled into an on-the-fly, hybrid and incremental LTL model checking algorithm. Section 4.4.1 will show a way to use recurring states and the modified relational product operator in the symbolic fixed point computation algorithm. Section 4.4.2 shows how to consider recurring and accepting states in the explicit search, while Section 4.4.3 will integrate the symbolic and explicit searches. Section 4.4.4 will present the full hybrid SCC detection algorithm.

4.4.1 Recurring states in the fixed point computation

In addition to indicating the absence of SCCs, recurring states can also help in finding them. Since each *SaturateRec*builds its own set of recurring states, the set of all gathered recurring states on level k will be the actual target states of transitions in $\mathcal{N}_k$ (only these transitions are fired on level k). This way, the subset of $\mathcal{S}^+$ actually reached is available at the end of a *SaturateRec*call and *DetectSCC* can be initialized with the recurring states instead of $\mathcal{B}(n_k)$.

Corollary 1. (Accelerating the fixed point computation) During an SCC detection phase after the saturation of node n_k, restricting the set of "new" transitions $\mathcal{N}_k$ to those that end in recurring states (that is, $\{(\mathbf{s}, \mathbf{s}') \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}_k, \mathbf{s}' \in \mathcal{R}\}$) will still find any SCCs that would be found otherwise.

The corollary follows from the fixed point computation strategy of saturation and its caching mechanisms. It can be used to accelerate the fixed point computation by using a smaller input set, since a smaller set of "new" transitions makes *DetectSCC* finish in fewer iterations.

Another application of *RelProdRec* is the computation of the filtered sets in the filtering cycle of *DetectSCC*. Since *RelProdRec* can be used to compute the intersection of the relational product and any other set of states on the fly, it is suitable to substitute the intersection used in the computation of reachable states.

4.4.2 Constraining the explicit search of abstractions

Recurring states together with accepting states (if any) are useful in the explicit search on node-wise abstractions as well. According to Proposition 4, every SCC has to contain at least one recurring state. If fair SCCs are sought, then at least one state of an acceptance set $\mathcal{F}$ also have to be included in the SCC.

In a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$, every node i represents the set of states $\mathcal{B}_{[i]}(n_k)$. If the SCC candidate is realizable, it will contain some states from this set. A necessary property of abstract SCC candidates is described by the following corollary that is a direct consequence of the above considerations.

Corollary 2 (Necessary condition for abstract SCC realizability). An abstract candidate SCC $\Theta_{\mathcal{A}} = \langle \mathcal{S}_\Theta, \mathcal{N}_\Theta \rangle$ of a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ is not realizable if

- $\left(\bigcup_{i \in \mathcal{S}_\Theta} \mathcal{B}_{[i]}(n_k)\right) \cap \mathcal{R} = \emptyset$, or
- $\left(\bigcup_{i \in \mathcal{S}_\Theta} \mathcal{B}_{[i]}(n_k)\right) \cap \mathcal{F} = \emptyset$ if a fair SCC is sought. ▪

Since every $\mathcal{B}_{[i]}(n_k)$ as well as $\mathcal{R}$, and even the set of accepting states (formally $\{(\mathbf{s}, q) \mid q \in \mathcal{F}\}$) is known by the time of the explicit search, candidate SCCs can be evaluated and sorted out of the necessary condition does not hold. If no potentially realizable candidate SCCs remain, the state space does not contain any global SCCs either.

4.4.3 SCC candidates in the fixed point computation

The incremental symbolic fixed point computation algorithm and node-wise abstraction are strongly related. Node-wise abstraction contains exactly those transitions that are considered "new" in the fixed point algorithm, so the latter can be regarded as a method to check if a candidate SCC is realizable or not.

According to the definition of node-wise abstraction (Definition 22) and Theorem 5, arcs of candidate SCCs represent the transitions that *may* be part of an SCC – if an arc is not part of any candidate SCCs, its corresponding transitions will not be the part of a global SCC either.

Corollary 3 (Realizing abstract SCCs). Given a node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ and an abstract SCC candidate $\Theta = \langle \mathcal{S}_\Theta, \mathcal{N}_\Theta \rangle$, the only transitions of $\mathcal{N}_k$ that can be part of the global SCC are those corresponding to abstract arcs $\mathcal{N}_\Theta$, formally $\{(\mathbf{s}, \mathbf{s}') \mid (\mathbf{s}, \mathbf{s}') \subseteq \mathcal{N}_k, (s_k, s'_k) \in \mathcal{N}_\Theta\}$, where s_k and s'_k are the local states of component k in the global states $\mathbf{s}$ and $\mathbf{s}'$.

The corollary can be exploited in the symbolic fixed point algorithm by considering only those transitions as “new” that are part of a candidate SCC instead of the full relation $\mathcal{N}_k$.

4.4.4 Assembling the pieces – the full SCC detection algorithm

After getting familiar with different aspects and components of the incremental hybrid SCC detection algorithm presented in this chapter, the last section summarizes the algorithm as a whole. The input model is assumed to be in any form saturation can process. SCC detection relies only on a base algorithm employing the iteration strategy of saturation, including, but not limited to traditional saturation, constrained saturation or any product computation algorithm of Chapter 3.

Whenever a node n_k becomes saturated, the following steps have to be executed:

1. The set of encoded states $\mathcal{B}_{[i]}(n_k)$ and transitions $\mathcal{N}_k$ are checked against emptiness – if any of them is empty, SCC detection reports no SCC.
2. The set of collected recurring states $\mathcal{R}$ and – if applicable – accepting states $\mathcal{F}$ are checked against emptiness – if no recurring states were found, SCC detection reports no SCC (see Proposition 4).
3. An explicit SCC computation algorithm is run on the current node-wise abstraction $\mathcal{A}^{\exists}_{n_k}$ to obtain an SCC candidate – if no candidate is found, SCC detection reports no SCC (see Theorem 5).
4. Candidate SCCs are checked according to Corollary 2 – if none of them is realizable, SCC detection reports no SCC.
5. A fixed point computation is started with $\mathcal{S}^+ = \mathcal{R}$ and transitions corresponding to arcs of the candidate SCCs as “new” transitions – the result of the computation is returned as a final answer (see Theorems 2 and 4 and Corollaries 1 and 3).

The algorithm terminates as soon as *DetectSCC* finds a nonempty fixed point, or if the root node is saturated. If a nonempty fixed point is found, algorithms of Section 2.5.1 can be used to extract a counterexample (probably aided by the contents of the fixed point). Otherwise there is no counterexample and the model M is valid in terms of the specification φ.

Algorithms 4.4 and 4.5 show the pseudocodes of the modified *Saturate* and *RelProd* functions.

Algorithm 4.4: SaturateScc

```
    input  : $s_k$ : node      //to saturate
    output : node
  1 Return result from cache if possible;
  2 $n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram;
  3 $t_k \leftarrow$ new node;
  4 $\mathcal{A}^{\exists}_{t_k} \leftarrow (\mathcal{S}_k, \emptyset)$;
  5 foreach $i \in \mathcal{S}_k : s_k[i] \neq \mathbf{0}$ do
  6     $t_k[i] \leftarrow SaturateScc(s_k[i])$;
  7 $r_k \leftarrow$ new node;        //recurring states
  8 repeat
  9     foreach $s_k[i] \neq \mathbf{0} \wedge n_{2k}[i][i'] \neq \mathbf{0}$ do
 10         $r'_{k-1} \leftarrow$ new node;
 11         $u_k \leftarrow$
            $RelProdScc(t_k[i], n_{2k}[i][i'], t_k[i'], r'_{k-1})$;
* 12        if $u_k \neq \mathbf{0}$ then add arc $(i, i')$ to $\mathcal{A}^{\exists}_{t_k}$;
 13         $t_k[i'] \leftarrow (t_k[i'] \cup u_k)$;
 14         $r_k[i'] \leftarrow (r_k[i'] \cup r'_{k-1})$;
 15 until $t_k$ unchanged;
* 16 $\mathcal{N}_\Theta \leftarrow TransitionsInSCC(\mathcal{A}^{\exists}_{t_k})$;
* 17 if $DetectSCC(\mathcal{B}(r_k), \mathcal{N}_{\leq k}, \mathcal{N}_\Theta)$ then
 18     terminate with counterexample;
 19 Put inputs and results in cache;
 20 return $CheckUnique(t_k)$;
```

Algorithm 4.5: RelProdScc

```
    input  : $s_k, n_{2k}, o_k$ : node
  1 //$s_k$:  node to be saturated,
  2 //$n_{2k}$:  next state node,
  3 //$o_k$:  old node
    in-out : $r_k$ : node //recurring states
    output : node
  4 Return result from cache if possible;
  5 if $s_k = \mathbf{1} \wedge n_{2k} = \mathbf{1}$ then
  6     if $o_k = \mathbf{1}$ then $r_k \leftarrow s_k$;
  7     return 1;
  8 $t_k \leftarrow$ new node;
  9 foreach $s_k[i] \neq \mathbf{0} \wedge n_{2k}[i][i'] \neq \mathbf{0}$ do
 10     $r'_{k-1} \leftarrow$ new node;
 11     $t_k[i'] \leftarrow (t_k[i'] \cup$
        $RelProdScc(s_k[i], n_{2k}[i][i'], o_k[i'], r'_{k-1})$;
 12     $r_k[i'] \leftarrow (r_k[i'] \cup r'_{k-1})$;
 13 $t_k \leftarrow SaturateScc(CheckUnique(t_k))$;
 14 Put inputs and results in cache;
 15 return $t_k$;
```

Chapter 5

Evaluation

To demonstrate the efficiency of the presented new algorithm (referred to as *Hyb-MC*), models of the Model Checking Contest[1] has been used to compare it to three competitive tools. NuSMV2 [16] is a BDD-based model checker implementing traditional SCC-hull algorithms and is well-established in the industrial and academical community. Its successor, nuXmv [10] implements the IC3 algorithm for LTL model checking. ITS-LTL [23] is a powerful tool based on saturation that implements various optimizations both for the symbolic encoding and on-the-fly SCC detection.

5.1 Implementation

The algorithms presented in Chapters 3 and 4 were implemented in the PetriDotNet[2] framework. PetriDotNet is a modeling and model checking tool written in C# supporting (colored) Petri nets. It is developed by students of Fault Tolerant Systems Research Group of Budapest University of Technology and Economics, including the author himself. Currently (in 2014), it supports basic analysis of Petri nets, and a handful of saturation-based model checking techniques including state space generation and reachability analysis, CTL model checking, bounded CTL model checking and on-the-fly LTL model checking (presented in this thesis).

The presented LTL model checking algorithm was implemented on top of existing algorithms and data structures in order to reuse the results of previous research and development. It also reuses tools of the SPOT toolset [24] to parse LTL formulae and transform them to Büchi automata.

5.2 Benchmark Cases

The Model Checking Contest offers Petri net models of various artificial and real-world problems. The models are given in PNML format [36], usually both as colored Petri nets

[1]`http://mcc.lip6.fr/`
[2]`http://petridotnet.inf.mit.bme.hu/en/`

and unfolded P/T nets (described in Section 1.1.2) as well. Due to the supported input formats of the selected tools, only P/T nets could be used.

Excluding the "surprise models" of the contest of 2014 (which were released after conducting the measurements), a total of 27 scalable models were used in the benchmark, with instances of different size, resulting in 157 model instances. The majority of the models expose concurrent and asynchronous behavior. Except the "planning" model, state spaces of the nets are finite. Even the infinite model was kept in order to demonstrate that on-the-fly LTL model checking can sometime bear with infinite models: if the product is finite or a counterexample is found in a finite subspace.

As for the specifications, a tool of the SPOT toolset was used to generate true (i. e., no pure Boolean) LTL formulae with predefined atomic propositions. Atomic propositions were generated based on the models: for every place of the smallest instance (those that appear in instances of any size) two propositions requiring zero and nonzero token counts were defined. SPOT generated 50 properties for every model class – instances of every size were checked against these properties.

All in all, the 157 model instances and 50-50 properties gave a total of 7 850 benchmark cases.

Generated LTL formulae

There are many categorizations of LTL formulae based on syntactic or semantic considerations [41]. The generated LTL properties were also cathegorized by a tool of SPOT. The following categories were used in the measurements.

- *Safety properties* (1 466 formulae): Specifies that something "bad" never happens. In general, counterexamples of these properties consists of a finite prefix that cannot be "fixed" with any suffix, i. e., the violation occurs in the prefix itself. In LTL, they are usually in the form $\mathsf{G}\ \varphi$.
- *Guarantee properties* (2 112 formulae): Specifies that something "good" is guaranteed to happen. Counterexamples for guarantees are "lasso" shaped, since they have to describe an infinite behavior that fails to expose the desired property. In LTL, they are usually in the form $\mathsf{F}\ \varphi$.
- *Obligation properties* (4 975 formulae): Combination of safety and guarantee properties. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \vee \mathsf{F}\ \varphi_2$.
- *Pure eventuality formulae* (1 620 formulae): If φ is a pure eventuality formula and the path ρ models φ, then $\mu\rho$ also models φ, where $\mu \in \mathcal{S}^*$ is any finite prefix. In other words, pure eventualities describe *left-append closed* languages.
- *Pure universality formulae* (1 620 formulae): If φ is a pure universality formula and the path $\mu\rho$ models φ, then ρ also models φ, where $\mu \in \mathcal{S}^*$ is any finite prefix. In other words, pure universalities describe *suffix closed* languages.

There some more complex and interesting categories that cannot be automatically recognized by SPOT. For this reason, the category "not obligation" (2 875 formulae) is used for such properties, including the following categories.

- *Progress properties*: Specifies that something "good" will keep happening again and again. In LTL, they are usually in the form $\mathsf{G}\ \mathsf{F}\ \varphi$.
- *Response properties*: Specifies that a response will eventually be given to a certain event whenever it occurs. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \Rightarrow \mathsf{F}\ \varphi_2$.
- *Stability properties*: Specifies that a certain property will stabilize eventually. In LTL, they are usually in the form $\mathsf{F}\ \mathsf{G}\ \varphi$.

As it turned out, the presented algorithm is not sensitive to the category of the checked LTL property.

5.2.1 Tools and Inputs

The model checking algorithm presented in this thesis will be referred to as Hyb-MC. Due to the optimized automata produced by SPOT, decomposition-based product computation was used for the comparison.

As mentioned, the tools selected for comparison are NuSMV, nuXmv and ITS-LTL. NuSMV is a traditional and well-established tool that has been used in the industry for a long time. It has many options – in this evaluation, BDD-based symbolic model checking were used (see Section 2.3 for algorithms). Although NuSMV tends to be better for synchronous models, it has been heavily used in various domains, hence its importance. Its successor, nuXmv came out in early 2014 and has even more features. The one used here is based on the IC3 algorithm (see Section 2.2). ITS-LTL is part of the ITS toolset and uses saturation-based on-the-fly algorithms together with hierarchical set decision diagrams (see Section 2.4.1 and [21]).

Unfortunately, PNML is not supported in any of the tools. Both NuSMV and nuXmv consume models in their native SMV format, while ITS supports many types of models and formats other than PNML. In case of NuSMV and nuXmv, PNML models were translated to SMV by PetriDotNet, similarly to the approach discussed in [53]. Since ITS accepts models in CAMI format (the native format of the tool CPN-AMI) and there exists a standardized tool to convert between PNML and CAMI, this tool was used to generate the input files to ITS.

In terms of the properties, both Hyb-MC and ITS-LTL uses SPOT to parse the formula and transform it into a Büchi automaton. A small tool was implemented to transform these formulae into the format of NuSMV and nuXmv.

5.2.2 Benchmark settings

Measurements were done on identical server machines with Intel Xeon processors (4 cores, 2.2GHz) and 8 GB of RAM. Hyb-MC and ITS-LTL were run on Windows 7, while NuSMV and nuXmv were run on CentOS 6.5.

Timeout was set to 600 seconds. Runtimes were measured internally by every tool. In case of Hyb-MC, the runtime includes the processing time of SPOT (transformation to automaton) and the total runtime of the algorithm including its initialization, but not the loading time of PetriDotNet. ITS-LTL also measured its runtime internally, also including the runtime of SPOT and the algorithm. NuSMV and nuXmv can measure time by placing special command in the input script. In case of these tools, only the actual runtime of model checking was measured, omitting the time of loading the model and constructing the binary model from the SMV input.

The decision diagram based tools (Hyb-MC, NuSMV and ITS-LTL) used the same variable ordering produced by heuristics of the ITS toolset. Script 11 was used to produce a flat ordering with each place encoded in a separate variable – this produced orderings that every tool could parse and use. This way, the hierarchical features of ITS might not have been fully exploited, but saturation-based algorithms in PetriDotNet can also be faster if a variable can encode multiple places.

Approximately 40 days of processing time was spent on the evaluation.

5.3 Results

The following sections present some aspects of the results of the evaluation. Overall, 5 megabytes of data was collected and analyzed. Out of the successfully checked cases, properties were fulfilled 2 811 times, while 3 565 cases gave counterexamples. In 1 474 cases, all the tools exceeded the time limit.

The whole benchmark and the analyzed results can be downloaded from the website[3] of PetriDotNet dedicated to [44].

5.3.1 Comparison of runtime results

The main emphasis of preliminary analysis was on the runtime results of each tool. The runtime of Hyb-MC was compared to the corresponding runtime of the other tools. In addition, simple state space generation with saturation was also done for each model instance to prove that saturation is not the single reason of high performance – on-the-fly model checking can finish much earlier than a simple state space generation.

[3]`http://inf.mit.bme.hu/en/tacas15`.

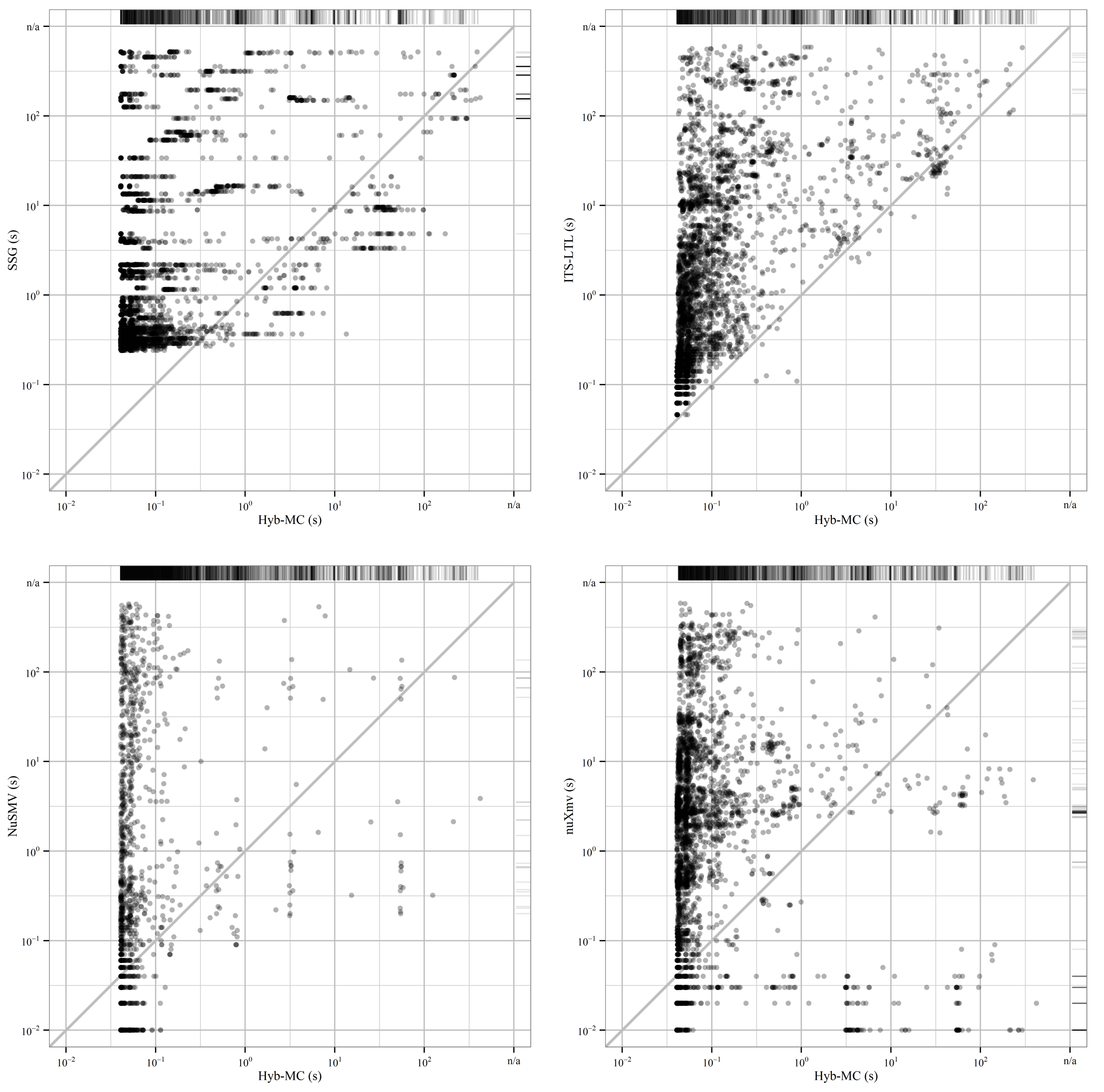
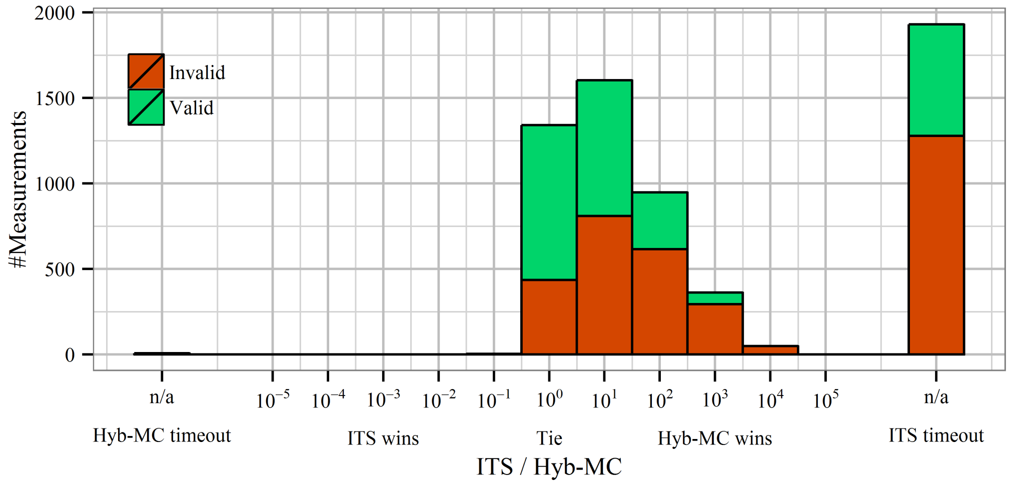
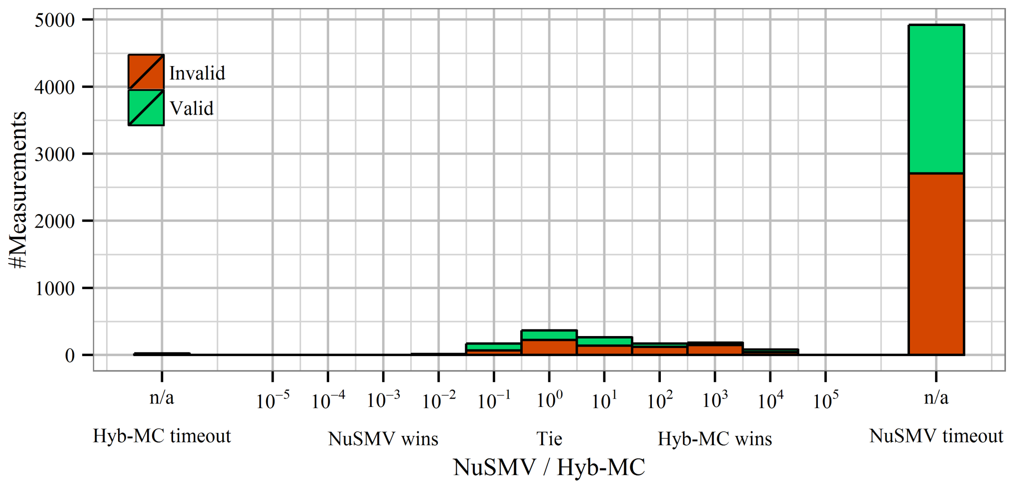


Figure 5.1. *Scatter plots comparing the runtime of tools and state space generation on individual benchmark cases.*

Results can be seen on Figure 5.1. The four scatter plots show the comparison of Hyb-MC to simple state space generation (SSG) with saturation and the three chosen tool. Each dot represents a benchmark case. The horizontal and vertical axes measure the runtime of Hyb-MC and the other tool, respectively. A dot above the diagonal line is a benchmark case that Hyb-MC solved faster than the competitor. The borders of the diagrams represent cases where one of the tools did not finish under the time limit. Cases in which neither of the tools finished are not shown on this diagram.

As the top-left plot suggest, model checking with Hyb-MC is usually faster than simple state space generation. The main cause of this is the algorithm's on-the-fly operation and efficient incremental operation (i. e., , a low degree of redundancy). There are also some cases where state space generation could be finished, but model checking was unsuccessful. These cases show that incremental operation and the presented optimizations cannot always compensate the overhead of model checking complex properties.





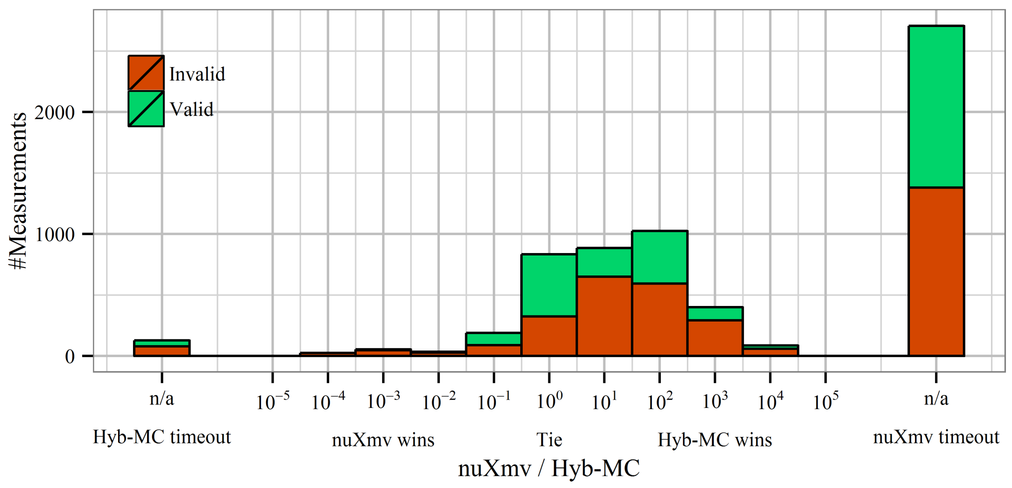


Figure 5.2. *Bar charts showing the relative performance of the selected model checking tools.*

Comparing to the other model checking tools, the vast majority of cases show the competitiveness of the presented algorithm. Since the scales of the axes are logarithmic, the distance of a dot from the diagonal indicate an exponential difference in the runtime of the tools. This difference is visualized on Figure 5.2. The bar chart shows the quantity t_A/t_B with a logarithmic horizontal axis, where t_A and t_B are the runtime of the compared algorithms.

Figure 5.2 is colored to show the distribution of valid and invalid properties. Hyb-MC is better than the other tools more often in refuting an invalid property than in proving valid ones. This shows the efficiency of SCC detection algorithm compared to the other approaches, but even valid cases show a decent speedup compared to any of the other tools, justifying the product computation algorithm and the use of abstraction and recurring states as a means to prove the absence of counterexamples.

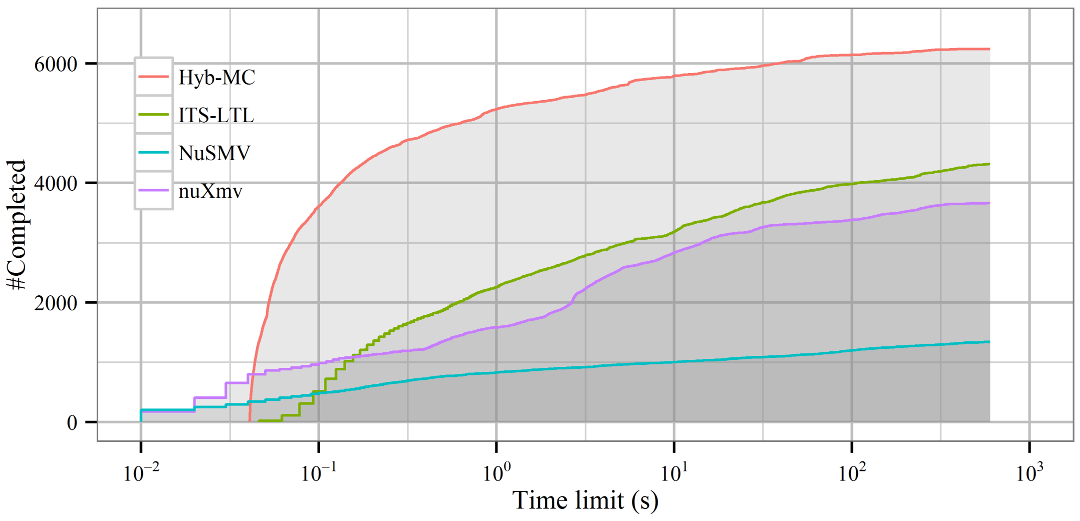


Figure 5.3. *Cumulative frequency diagram of the runtime results of each tool.*

Another diagram on Figure 5.3 show the cumulative frequency of runtime results for each tool. A point of the diagram shows how many of the cases finished with runtime under the value of the horizontal axis, i. e., how many cases would have finished if the timeout was set to a specific value. This diagram indirectly suggests the scalability of the algorithms. The initial delay of Hyb-MC and ITS-LTL is due to SPOT building the Büchi automaton from the LTL property.

It is interesting to note some specialities of the tools. ITS-LTL performed the best among the chosen competitors, but it could outperform Hyb-MC in very few cases only. Since it is also based on saturation and uses abstraction to perform on-the-fly model checking, these results are the most significant among all the others.

Although NuSMV performed the worst of all competitors, it could beat Hyb-MC in more cases and also more significantly than ITS-LTL. NuSMV is the oldest of all the tools and is supposed to be better with synchronous models, so these cases deserve future attention.

Results of nuXmv are very different from results of the other tools. This is not surprising, since it uses a SAT-based algorithm with different techniques and strengths. It is also the one that outperformed Hyb-MC most of the times, it even managed to finished with many cases that caused a timeout in Hyb-MC. It is also interesting to note that nuXmv proved to be the only tool sensitive to the category of the checked property.

Analysis of runtime results measured on model families showed other interesting differences in the scalability of the tools. NuSMV and nuXmv performed significantly better on models where instances of the model scaled in the domain of state variables. On the other hand, saturation-based tools Hyb-MC and ITS-LTL were much better on models that scaled in the number of variables (components).

5.3.2 Efficiency of the presented techniques

During the measurements, Hyb-MC spent only 17% of the time computing SCCs. Overall, 359 084 symbolic fixed point computations were started, while abstraction and explicit algorithms prevented $1.22 \cdot 10^8$ runs of symbolic SCC computation, 99.7% of all the cases. 90% of these cases were prevented by the absence of recurring states (as a first check), while the remaining 10% were the cases where explicit runs on node-wise abstractions managed to find even more evidence.

Cases	Time spent with SCC detection	Prevented symbolic runs	Prevented by	
			Recurring	Abstraction
All	16.9%	99.7%	89.7%	10.3%
Valid	13.5%	99.7%	89.0%	11.0%
Invalid	19.6%	99.7%	91.3%	8.7%
Obligation	13.7%	99.7%	86.3%	13.7%
Not obligation	23.2%	99.7%	93.0%	7.0%
Echo (2D)	2.7%	100.0%	100.0%	0.0%
Erathostenes	65.9%	62.9%	19.0%	81.0%

Table 5.1. *SCC detection statistics in different groups of cases.*

Table 5.1 shows the discussed statistics in some interesting subsets of the benchmark cases. Not surprisingly, Hyb-MC spends more time looking for SCCs in case of invalid properties. The efficiency of recurring states and explicit search slightly varies, but not significantly. The difference is greater between obligation and more complex formulae: complex properties (such as progress properties) require more time spent on SCC detection. The share of recurring states and explicit search also varies a bit more.

Echo (2-dimensional instances) and Erathostenes are two extreme models in terms of time spent on SCC detection. Echo was one of the hardest model for Hyb-MC, only 26% of all the cases were solved in the 2-dimensional family, all of which were valid. The fact that symbolic SCC detection was never called in these cases suggests that Hyb-MC fails here in SCC detection. Erathosthenes, on the other hand, had all of its cases solved, and they also were valid properties. In spite of this, almost two thirds of the time was spent on computing SCCs. Explicit search also shows extremely high efficiency here, although the percentage of prevented symbolic runs is the only value among the models that is under 90%.

Chapter 6

Summary

LTL model checking of asynchronous systems is a computationally difficult problem. Various techniques and algorithms were developed in the history to tackle the state space explosion problem which is inherent in these systems, and to combat the complexity of LTL model checking. This is also the context of this work: in this thesis, a new approach for LTL model checking of asynchronous systems was presented. The new approach combines recent advances from this field as saturation is used for traversing the possible states of the system, while a modified version of the constrained saturation algorithm constructs the synchronous product on the fly. A new incremental fixed-point algorithm computes local model checking results from which the algorithm is able to decide the model checking question. In order to decrease the number of symbolic fixed point computations, the thesis introduces a scalable abstraction framework, which efficiently filters SCC computations by using local model checking runs.

The thesis presented the following new algorithms:

- Two efficient on-the-fly synchronous product generation algorithms based on saturation;
- An incremental fixed point computation algorithm for SCC detection;
- An abstraction technique to support inductive reasoning on the absence of SCCs;
- A unique hybrid model checking algorithm combining explicit state traversal with symbolic state space representation.

As a theoretical result, the thesis also contains the proofs for the correctness of the presented algorithms.

The new algorithms were implemented in the PetriDotNet framework and extensive measurements were conducted to examine efficiency. The tool was compared to industrial and academic model checking tools. Although the implementation is only in the prototype phase, it turned to be quite competitive and could solve more of the benchmarks cases than any of its competitors.

The presented algorithm has a huge potential for future development. Following the idea of driving the symbolic algorithm with explicit runs, a promising direction is to combine partial order reduction with symbolic model checking. In addition, advanced representations of the properties can also be used to further improve the speed of model checking.

List of Figures

List of Algorithms

Bibliography

[1] *Software engineering: report on a conference sponsored by the NATO Science Committee, Garmish, Germany, 7th to 11th October 1968.* Scientific Affairs Division, NATO, Brussels, Belgium, 1969.

[2] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18(2):97–116, 2001.

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[4] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, September 1996.

[5] Aaron Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing – SAT 2012*, number 7317 in LNCS, pages 1–14. Springer, 2012.

[6] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. of the International Conference on Formal Methods in Computer-Aided Design*, pages 144–153, Austin, Texas, 2011. FMCAD Inc.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[8] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[10] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Alberto Griggio, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. Technical report, Fondazione Bruno Kessler, 2014.

[11] G. Ciardo, G. Lüttgen, and A. Yu. Improving static variable orders via invariants. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume 4546 of *Lecture Notes in Computer Science*, pages 83–103. Springer, 2007.

[12] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 379–393. Springer, 2003.

[13] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 40–53. Springer-Verlag, 2003.

[14] Gianfranco Ciardo, Gerald LĂźttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 328–342. Springer Berlin Heidelberg, January 2001.

[15] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *Int. J. on Softw. Tools for Technology Transfer*, 8(1):4–25, February 2006.

[16] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In Ed Brinksma and KimGuldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[17] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, volume 1102 of *LNCS*. 1996.

[18] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking: History, Achievements, Perspectives*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[19] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[20] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[21] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, number 3731 in Lecture Notes in Computer Science, pages 443–457. Springer Berlin Heidelberg, January 2005.

[22] Alexandre Duret-Lutz, Kais Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. *arXiv:1106.5700 [cs]*, June 2011. arXiv: 1106.5700.

[23] Alexandre Duret-Lutz, Kaïs Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Self-loop aggregation product – a new hybrid approach to on-the-fly LTL model checking. In *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2011.

[24] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: An extensible model checking library using transition-based generalized bĂźchi automata. In *Proc. of the IEEE Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 76–83, 2004.

[25] Rüdiger Ebendt, Görschwin Fey, and Rolf Drechsler. *Advanced BDD optimization*, volume 282. Springer, 2005.

[26] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, number 85 in Lecture Notes in Computer Science, pages 169–181. Springer Berlin Heidelberg, January 1980.

[27] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, UK, 1980. Springer-Verlag.

[28] E. Allen Emerson and Joseph Y. Halpern. “sometimes” and “not never” revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[29] Kousha Etessami and Gerard J. Holzmann. Optimizing bĂźchi automata. In *CONCUR 2000 – Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2000.

[30] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV ’01, pages 53–65, London, UK, UK, 2001. Springer-Verlag.

[31] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the Int. Symp. on Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, Ltd., 1995.

[32] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[33] Serge Haddad, Jean-Michel IliĂŠ, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In Farn Wang, editor, *Automated Technology for Verification and Analysis*, number 3299 in Lecture Notes in Computer Science, pages 196–210. Springer Berlin Heidelberg, January 2004.

[34] Thilo Hafer and Wolfgang Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In Thomas Ottmann, editor, *Automata, Languages and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 269–279. Springer Berlin Heidelberg, 1987.

[35] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In Kees M. van Hee and RĂźdiger Valk, editors, *Applications and Theory of Petri Nets*, number 5062 in Lecture Notes in Computer Science, pages 211–230. Springer Berlin Heidelberg, January 2008.

[36] Lom M. Hillah, Ekkart Kindler, Fabrice Kordon, Laure Petrucci, and Nicolas Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, 2009.

[37] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proceedings of the Second SPIN Workshop*, DIMACS Ser. in Discrete Mathematics and Theoretical Computer Science, pages 81–89. AMS, 1997.

[38] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1993.

[39] Kais Klai and Denis Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In *Applications and Theory of Petri Nets*, volume 5062 of *LNCS*, pages 288–306. Springer, 2008.

[40] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[41] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[42] K. L. McMillan. Interpolation and SAT-based model checking. In *Lecture Notes in Computer Science*, volume 2725, pages 1–13, 2003.

[43] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[44] Vince Molnár, Dániel Darvas, András Vörös, and Tamás Batha. Saturation-based incremental LTL model checking with inductive proofs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. 2015.

[45] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[46] Doron Peled. All from one, one for all: On model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 409–423, London, UK, UK, 1993. Springer-Verlag.

[47] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[48] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, number 137 in Lecture Notes in Computer Science, pages 337–351. Springer Berlin Heidelberg, January 1982.

[49] Philippe Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic*, volume 4, pages 1–44. King's College Publications, 2003.

[50] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[51] Detlef Sieling. The nonapproximability of obdd minimization. *Inf. Comput.*, 172(2):103–138, February 2002.

[52] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In Mark D. Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, number 2517 in Lecture Notes in Computer Science, pages 88–105. Springer Berlin Heidelberg, January 2002.

[53] Marcin Szpyrka, Agnieszka Biernacka, and Jerzy Biernacki. Methods of translation of petri nets to nusmv language. In *CS&P*, pages 245–256, 2014.

[54] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[55] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.

[56] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin Heidelberg, 1996.

[57] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 1–22, London, UK, UK, 2001. Springer-Verlag.

[58] Chao Wang, Roderick Bloem, Gary D. Hachtel, Kavita Ravi, and Fabio Somenzi. Compositional SCC analysis for language emptiness. *Formal Methods in System Design*, 28(1):5–36, 2006.

[59] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, number 5799 in Lecture Notes in Computer Science, pages 368–381. Springer Berlin Heidelberg, January 2009.

[60] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7(2):141–150, 2011.