On-the-fly Model Checking with Guided Abstraction

Gal Sade

On-the-fly Model Checking with Guided Abstraction

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Gal Sade

Submitted to the Senate
of the Technion — Israel Institute of Technology
Adar 5781 Haifa February 2021

This research was carried out under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Orna Grumberg. Our weekly meetings were an indispensable source of inspiration for me. Through these meetings I have learnt what research is, and how fun it might be. Our joint work has given me the opportunity to acquire skills and experience, learning from a world renowned and prominent researcher. Orna – thank you for believing in me, for supporting me and for your friendship.

I would like to thank Dr. Yakir Vizel for a fruitful and a successful collaboration. I enjoyed our mutual work, and I am sure that the knowledge I have acquired through it would greatly influence me in the future. I will also like to thank my peers and fellow researchers, who inspired and guided me along the way.

Finally, I would like to thank my beloved family for believing in me and helping me all throughout my studies. I cherish the motivation and curiosity, bequeathed to me by them. I am grateful for their support, both in happy moments and in challenging ones.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Model checking is an automatic verification method that gets a system model and a specification, and checks whether the model satisfies the specification.

CTL is a branching time temporal logic suitable for specifying behaviors of both software and hardware systems. It enables specifying properties that cannot be expressed in linear time logics, such as LTL. An example of such a property is restartability, which means that in every reachable state, the system may return to its initial state, due to a reset or a recovery. Further, in many cases, CTL model checking algorithms can be easily extended to handle the alternation-free fragment of the powerful μ-calculus logic.

In this work, we present a novel approach, OMG, that combines on-the-fly verification with abstraction in order to obtain an efficient CTL model checking algorithm.

On-the-fly verification ensures that only parts that are needed for determining the satisfaction of the specification are developed. The abstraction is used to form inductive invariants, allowing OMG to determine satisfaction of the CTL specification without traversing the entire state-space.

We formalize the correctness of OMG, and present both an explicit version of the algorithm and a symbolic one.

We implemented our algorithm on top of a combination of explicit and symbolic representations, where symbolic representations are handled with SAT/SMT solvers. Our experiments show that on a few examples, our algorithm outperforms a state-of-the-art SAT-based algorithm for CTL.

Chapter 1

Introduction

In this work we present a novel *CTL* model checking algorithm, called *On-the-fly Model checking with Guided abstraction* (OMG). OMG combines an on-the-fly algorithm with abstraction. Given a model and a specification, *model checking* [CGK+18] determines if the model satisfies the specification. *CTL* [BPM83, CGK+18] is a significant branching time specification language. It enables specifying properties that cannot be expressed in linear time logics, such as *LTL* [Pnu77]. An example of such a property is *restartability*, which means that in every reachable state, the system may return to its initial state, due to a reset or a recovery. Further, in many cases, *CTL* model checking algorithms can be extended to handle the alternation-free fragment of the powerful μ-calculus logic [Koz83].

OMG is an on-the-fly algorithm. This means that it only analyzes parts of the model, required in order to determine satisfaction of the examined property. Abstraction is used to enable termination (for infinite-state systems) or to obtain speedups (for finite-state systems).

As an intuition, consider the case of *reachability analysis*, expressed in *CTL* by formulas of the form AGp. Given a state s in a model M, checking $M, s \models AGp$ amounts to checking that all states reachable from s in M satisfy the property p. OMG explores the states reachable from s in an on-the-fly manner (i.e. lazily), checking if they all satisfy p. If it finds a state that does not satisfy p, a counterexample is found and the algorithm terminates with a negative answer.

Terminating with a positive answer, on the other hand, requires visiting all reachable states. We note, however, that if all successors of a state s have been visited, and all satisfy p, then no further analysis of s is required. If this property holds for all visited states, then OMG terminates with a positive answer. However, analyzing all reachable states may be infeasible. To overcome this issue, OMG uses abstraction. It associates an abstract state $\hat{s}$ with each visited state s, and uses the abstract states when forming an *inductive invariant*. This may accelerate convergence or even make it feasible, for the infinite-state case.

To further explain OMG, consider formulas of the form EGp. Checking $M, s \models EGp$

amounts to searching for an infinite path π from s such that every state along π satisfies p. When analyzing states reachable from s, if OMG encounters a state t that does not satisfy p, it determines that no further analysis is required for t. This is because t cannot be a part of a path that satisfies Gp. If an infinite path is not found and there are no more states to analyze, OMG terminates with a negative answer.

For a positive answer, OMG seeks again an inductive invariant. However, now a set of states D forms an *inductive invariant for EGp*, if every state in D, in addition to satisfying p, *has* a successor in D. This is in contrast with the inductive invariant for AGp, in which for each state, *all* successors need to remain in the invariant.

For both AGp and EGp, we use abstraction to form inductive invariants. When an inductive invariant cannot be formed, we may refine the abstraction or further explore parts of the concrete model.

The abstraction used by OMG defines both *may* and *must* transitions between abstract states. May transitions over-approximate the set of concrete transitions. Therefore, they are used in the inductive invariant for AGp, to show that *all* paths in the model satisfy Gp. Must transitions under-approximate the set of concrete transitions. Hence, they are used in the inductive invariant for EGp, to show that there *exists* a path satisfying Gp.

These approximations allow OMG to handle general CTL formulas with nested universal and existential path quantifiers. OMG analyzes a general CTL formula recursively. For example, to determine if $s \models p \wedge AXEGq$, it checks if $s \models p$ and, in addition, if all successors of s satisfy EGq.

OMG maintains an important feature that allows it to handle recursively any nesting of CTL operators: For a state s, which is associated with an abstract state $\hat{s}$, if $s \models \varphi$ has been determined, then all states associated with $\hat{s}$ agree with s on the satisfaction of φ.

Our method is suitable for model checking systems that have a finite branching degree, with either a finite or an infinite number of states. However, it may not terminate for infinite state systems.

We formalize and prove the correctness of our algorithm and present both an explicit version of OMG and a symbolic one.

We implemented our algorithm on top of a combination of explicit and symbolic representations, where symbolic representations are handled with SAT/SMT solvers. Our experiments show that on a few examples, OMG outperforms IICTL [HBS12], a state-of-the-art SAT-based algorithm for CTL.

Chapter 2

Preliminaries

A Kripke structure is a tuple $M = (S, S_0, R, L)$, defined over a set of atomic propositions AP. S is a set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{AP}$ is a labeling function. Note that, R is a total relation. Namely, for every $s \in S$ there exists $t \in S$ such that $(s, t) \in R$. From now on, the notations M, S, S_0, R, L refer to a Kripke structure $M = (S, S_0, R, L)$ (a *structure*). We sometimes refer to a Kripke structure as a *Kripke model*, or just a *model*. We use the term *cstate* to abbreviate the term concrete state, which refers to states in S. A path $\pi = s_0, s_1, s_2, \ldots$ is an infinite sequence of states (that is, $s_i \in S$ for every $i \in \mathbb{N}$), such that $(s_j, s_{j+1}) \in R$ for every $j \in \mathbb{N}$. We say that π is from s if $s = s_0$. Given a cstate s, a sequence of cstates $s_0, s_1, \ldots, s_k$ is a *consecutive sequence from s* if $s = s_0$ and $(s_i, s_{i+1}) \in R$ for every $0 \leq i < k$.
We next define CTL.

Definition 2.0.1. (Computation Tree Logic)
We refer to the set $\mathcal{P} = \{A, E\}$ as path quantifiers, and to the temporal operators X, V, U *next*, *until* and *release*, respectively. Let AP be a set of atomic propositions. CTL formulas are defined in the following manner:

- $\top$ and $\bot$ are CTL formulas.
- For every $p \in AP$, p is a CTL formula.
- if φ_1, φ_2 are CTL formulas, then $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$ and $\neg\varphi_1$ are CTL formulas.
- if φ_1, φ_2 are CTL formulas and $P \in \mathcal{P}$, then $PX\varphi_1, P\varphi_1 U \varphi_2$ and $P\varphi_1 V \varphi_2$ are CTL formulas.

We identify formulas of the form $\neg\neg\varphi$ with φ.

CTL formulas are interpreted over Kripke structures. Given a Kripke structure $M = (S, S_0, R, L)$ and a state $s \in S$, the semantics of CTL (that is, the satisfaction relation $\models$) are defined in the following manner [CGK$^+$18]:

- $M, s \models \top$

- $M, s \not\models \bot$
- for $p \in AP$, $M, s \models p \iff p \in L(s)$.
- $M, s \models \neg\varphi \iff M, s \not\models \varphi$.
- $M, s \models \varphi_1 \wedge \varphi_2 \iff M, s \models \varphi_1$ and $M, s \models \varphi_2$.
- $M, s \models \varphi_1 \vee \varphi_2 \iff M, s \models \varphi_1$ or $M, s \models \varphi_2$.
- $M, s \models EX\varphi \iff$ there exists a path $\pi = s_0, s_1, \ldots$ from s s.t. $M, s_1 \models \varphi$.
- $M, s \models AX\varphi \iff$ all paths $\pi = s_0, s_1, \ldots$ from s satisfy $M, s_1 \models \varphi$.
- $M, s \models E\varphi_1 U \varphi_2 \iff$ there exists a path $\pi = s_0, s_1, \ldots$ from s s.t. $\exists k \geq 0$, such that $M, s_k \models \varphi_2$ and $\forall 0 \leq i < k$, $M, s_i \models \varphi_1$.
- $M, s \models A\varphi_1 U \varphi_2 \iff$ for all paths $\pi = s_0, s_1, \ldots$ from s it holds that $\exists k \geq 0$ s.t. $M, s_k \models \varphi_2$ and $\forall 0 \leq i < k$, $M, s_i \models \varphi_1$.
- $M, s \models E\varphi_1 V \varphi_2 \iff$ there exists a path $\pi = s_0, s_1, \ldots$ from s s.t. $\forall j \geq 0$ if $\forall i < j$, $(s_i \not\models \varphi_1)$ then $s_j \models \varphi_2$.
- $M, s \models A\varphi_1 V \varphi_2 \iff$ for all paths $\pi = s_0, s_1, \ldots$ from s it holds that $\forall j \geq 0$ if $\forall i < j$, $(s_i \not\models \varphi_1)$ then $s_j \models \varphi_2$.

Note that the *release* temporal operator is the dual of the U operator. That is, $\varphi_1 V \varphi_2 \equiv \neg(\neg\varphi_1 U \neg\varphi_2) \equiv G\varphi_2 \vee \varphi_2 U(\varphi_1 \wedge \varphi_2)$. Other CTL operators can be expressed using U and V, as $G\psi \equiv (False)V(\psi)$ and $F\psi \equiv True U \psi$.

For a set of states $D \subseteq S$, $D \models \varphi \iff \forall s \in D$, $s \models \varphi$. The notation $s \models \varphi$ is sometimes used if M is clear from the context.

For a CTL formula φ and $i \in \mathbb{N}$, we write φ^i to denote a sequence of i consecutive occurances of φ. This is defined formally in the following manner: $\varphi^0 = \epsilon$ (the empty formula) and $\varphi^{i+1} = \varphi^i\varphi$.
The following definitions are also used throughout this paper.

Definition 2.0.2. (Equi-labeled states)
Let M be a Kripke structure, and let $s, t \in S$ be cstates in M. Then, s, t are *equi-labeled* if $L(s_1) = L(s_2)$.

We denote $s \equiv_0 t$ if s, t are equi-labeled. We denote the equivalence class of a state s under this relation by $[s]_0$.

Definition 2.0.3. (Relation-induced function)
Let $Q \subseteq A \times B$ be a relation. We define the function $\widehat{Q} : A \to 2^B$ so that for every $a \in A$, $\widehat{Q}(a) = \{b \in B \,|\, (a, b) \in Q\}$. We also define the function $\widehat{\widehat{Q}} : 2^A \to 2^B$ such that for every $D \in 2^A$, $\widehat{\widehat{Q}}(D) = \bigcup_{d \in D} \widehat{Q}(d)$. By abuse of notations, we use $Q, \widehat{Q}, \widehat{\widehat{Q}}$ interchangeably.

Our work uses the notion of unwinding trees, defined in the following manner.

Definition 2.0.4. (Unwinding Tree)
Let $r \in S$. Let $T = (V, E)$ be a finite directed graph in the shape of a tree with root $v_0 \in V$. Let $C : V \to S$ be a *concretization function*. Then, T is an *unwinding tree of M from r*, if $C(v_0) = r$ and for $u \in V$ which is not a leaf, it holds that $(u, v) \in E \iff (C(u), C(v)) \in R$.

The tree represents a finite unwinding of the model M from the cstate r. We refer to r as the *root* of T, denoted $\mathit{root}(T)$. We use the notations $n \in V$ and $n \in T$ interchangeably. The vertices of T are called *nodes*. Edges in the tree connect a node with each of its successors. A finite sequence of consecutive nodes in the tree, $n_1, n_2, \ldots, n_k$ is called a *trace*. A trace is *maximal* in T if n_k is a leaf. A node $n \in T$ is *labeled with s* if $C(n) = s$. The set of cstates that are labeled on the tree, $\{C(n) \mid n \in V\}$, is denoted $C(T)$.

We sometimes use the node $n \in T$ and the cstate $C(n) \in S$ interchangeably. As such, we write $n \models \varphi$ when $C(n) \models \varphi$. Note that any node $n \in T$ is the root of an unwinding tree of M from $C(n)$, which is now the root.

Note also, for each node $n \in T$ which is not $\mathit{root}(T)$ there exists a single node $n' \in T$ which is the parent of n in T (that is, $(n', n) \in E$), as T is a tree. For every node in T which is not the root, we denote by $\mathit{parent}(n)$ the parent of n in T. This notation is undefined for the root of T. The *depth* of node $n \in T$ is the number of edges in the (single) path from it to $\mathit{root}(T)$. The depth of $\mathit{root}(T)$, is defined to be zero, and the depth of a node n, denoted $\mathit{depth}(n)$ equal $\mathit{depth}(\mathit{parent}(n)) + 1$. The depth of an unwinding tree T is defined as $\max_{n \in T} \mathit{depth}(n)$.

A path $\pi = s_0, s_1, \ldots$ *extends* a trace $\tau = n_0, \ldots, n_k$ if for every $0 \leq i \leq k$, $C(n_i) = s_i$. Then, π is referred to as an *extension* of τ. Note that every trace can be extended to an infinite path, as the transitions relation is total. Additionally, each path π from $C(\mathit{root}(T))$ extends a single maximal trace.

We demonstrate our method over the running example, given in Figure 2.1. Consider M over $AP = \{p, q, r\}$, presented on the left-hand-side of Figure 2.1. Assume that s_0 is the initial state. The right-hand-side of Figure 2.1 presents an unwinding tree T of M from s_0, such that for every node $n_i \in T$, it holds that $C(n_i) = s_i$, and $C(n_2') = s_2$. Throughout this running example, we consider the model checking task of checking whether $M, s_0 \models AX(ApVq)$. Note that s_3 and s_4 are unreachable from s_0 and therefore are not represented in T.

2.1 The full abstract structure

An important aspect of OMG is its use of abstraction. OMG builds an abstraction on-the-fly. That is, along its run, the algorithm gradually builds an abstract structure and keeps changing it according to the evolvement of the concrete structure using T.

For M over AP, a set of abstract states $\hat{S}$ and an abstraction function $abs : S \to \hat{S}$, we present the notion of *full abstract structure* [SG04] (sometimes referred to as the *full abstract model*), which is built by OMG lazily. Namely, we do not necessarily build the full abstract model, but only parts of it, as needed. The full abstract model is defined over the same set of atomic propositions AP. We require abs to satisfy for every $s, t \in S$ that if $abs(s) = abs(t)$ then $L(s) = L(t)$. We use the term *astate* to abbreviate the term abstract state.

Each abstract state $\hat{s}$ represents a set of concrete states that are mapped to it by abs. That is, an abstract state $\hat{s}$ represents the set $\{s \in S \mid abs(s) = \hat{s}\}$. As such, we use both these representations interchangeably, and use the notation $s \in \hat{s}$ if $abs(s) = \hat{s}$. Note that since abs is a function, each concrete state is mapped to a single abstract state. Formally, for every $\hat{s}$, $\hat{t}$ it holds that $\hat{s} \cap \hat{t} = \emptyset$.

The *full abstract structure* is defined as follows: $\widehat{M} = (\hat{S}, \hat{R}^{may}, \hat{R}^{must}, \hat{L})$, where $\hat{L} : \hat{S} \to 2^{AP}$ is the labeling function, defined for every $\hat{s} \in \hat{S}$ such that $\hat{L}(\hat{s}) = L(s)$ for some $s \in \hat{s}$. Note that this is well-defined as of the restriction on abs.

The definition of the relation $\hat{R}^{must}$ requires the notion of hyper-transitions. Given an astate $\hat{s} \in \hat{S}$ and a set $\hat{D} \subseteq \hat{S}$, a hyper-transition from $\hat{s}$ to $\hat{D}$ is a pair $(\hat{s}, \hat{D})$. If $\hat{D} = \{t\}$ is a singleton, we refer to the hyper-transition $(\hat{s}, \hat{D})$ as a transition. We identify a transition $(\hat{s}, \hat{t})$ with the hyper-transition $(\hat{s}, \{\hat{t}\})$.

The transition relations of $\widehat{M}$ are defined as follows:

- The set of may transitions $\hat{R}^{may} = \{(\hat{s}, \hat{t}) \in \hat{S} \times \hat{S} \mid \exists s' \in \hat{s}\ \exists t' \in \hat{t}\ [(s', t') \in R]\}$.

- The set of must hyper-transitions $\hat{R}^{must} = \{(\hat{s}, \hat{D}) \in \hat{S} \times 2^{\hat{S}} \mid \forall s' \in \hat{s}\ \exists t' \in \bigcup \hat{D}\ [(s', t') \in R]\}$. Note that, if $(\hat{s}, \hat{D}) \in \hat{R}^{must}$, then for every $\hat{E} \supseteq \hat{D}$, $(\hat{s}, \hat{E}) \in \hat{R}^{must}$.

OMG conducts iterative refinements of the abstract state space by applying *splits* operations.

Definition 2.1.1. (Abstract state split)
Given a set $\hat{S}$ and an abstraction function abs, $\hat{S}'$ is a *split* of $\hat{S}$ w.r.t. a property P and an astate $\hat{s} \in \hat{S}$ if there exist $\hat{s}', \hat{s}'' \in \hat{S}'$ s.t. $\hat{s}' = \{s \in \hat{s} \mid s \models P\}$, $\hat{s}'' = \{s \in \hat{s} \mid s \not\models P\}$ and $\hat{S}' = (\hat{S} \setminus \{\hat{s}\}) \cup \{\hat{s}', \hat{s}''\}$. Note that, $\hat{s}' \cup \hat{s}'' = \hat{s}$ and $\hat{s}' \cap \hat{s}'' = \emptyset$.

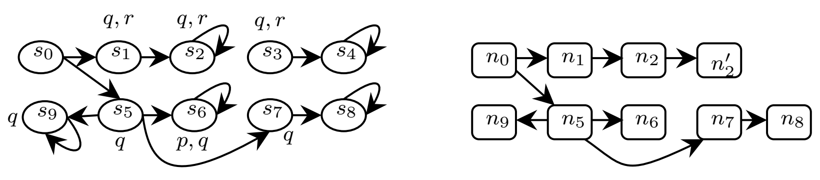

Figure 2.1: Kripke model (left) and its unwinding tree (right)

The abstraction abs' that corresponds to $\hat{S}'$ is defined as follows:

$$abs'(s) = \begin{cases} \hat{s}', & abs(s) = \hat{s} \wedge s \models P \\ \hat{s}'', & abs(s) = \hat{s} \wedge s \not\models P \\ abs(s), & otherwise \end{cases}$$

$\hat{S}'$ is a *split* of $\hat{S}$ if there exist a property P and an astate $\hat{s} \in \hat{S}$ s.t. $\hat{S}'$ is a split of $\hat{S}$ w.r.t. P and $\hat{s}$. We refer to P as the *split property*, and say that $\hat{s}$ is split and $\hat{s}', \hat{s}''$ are the *splits* of $\hat{s}$.

Definition 2.1.2. (May closure)
A pair of the form $(\hat{s}, A)$, where $A \subseteq \hat{S}$ is a *may closure* if for every abstract state $\hat{t}$ s.t. $(\hat{s}, \hat{t}) \in \hat{R}^{may}$, it follows that $\hat{t} \in A$. That it, all may transitions from $\hat{s}$ lead to an abstract state in A.

We define a notion of abstract state conformity in order to help future statements and lemmas.

Definition 2.1.3. (Abstract state conformity) Let $\hat{s}$ be an abstract state and φ be a CTL formula. Then, $\hat{s}$ *conforms* w.r.t. a formula φ if for every CTL subformula φ' of φ, it holds that: $\forall s \in \hat{s},\ s \models \varphi' \iff \hat{s} \models \varphi'$.

Chapter 3

Main Algorithm Description

In this chapter we give a high-level description of our algorithm for CTL model checking. We also present the main data structures and techniques that are used.

3.1 Abstract models corresponding to the unwinding trees

Given an unwinding tree T of a structure M, we consider the astates that "appear" on the tree. That is, the astates corresponding to concrete states that label the tree. Formally, the abstraction function *abs* is extended to the nodes of T in the following manner: $\forall n \in T$, $abs(n) = abs(C(n))$. Now, $abs(T)$ represents the set of all astates corresponding to nodes on T. That is, $abs(T) = \{abs(n) \mid n \in T\}$. Given a trace τ in T, $abs(\tau)$ represents the set of all astates corresponding to nodes on τ. That is, $abs(\tau) = \{abs(n) \mid n \in \tau\}$.

OMG assigns astates to nodes in T and updates them lazily. It builds the part of the abstract model, induced by the abstraction that is currently used, without computing the abstract model fully. In fact, OMG holds different abstract structures during its run. We use a series of abstract structures $\widehat{M}_j = (\hat{S}_j, \hat{R}_j^{may}, \hat{R}_j^{must}, \hat{L}_j)$, $j \in \mathbb{N}$, with potentially different state spaces.

The set $\hat{S}_j$ only changes due to two reasons as the algorithm progresses:

- *Abstract states are split*: Along the run of the algorithm, an astate $\hat{s}$ may be split into two parts, as described in Definition 2.1.1.

- *New cstate is discovered*: Along the run of the algorithm, it may be the case that a node n is added to T due to unwinding, but there does not exists an astate $\hat{s} \in \hat{S}_j$ such that $C(n) \in \hat{s}$. Then, OMG adds a new astate $\hat{s} = [C(n)]_0$. It is shown next that, if $C(n) \notin \bigcup \hat{S}_j$, then for every cstate $t \in \bigcup \hat{S}_j$, it holds that $C(n) \not\equiv_0 t$. As such, the new astate and each of the existing astates are disjoint.

According to the changes in $\hat{S}_j$, the rest of the components of the abstract model change as well. For each $j \in \mathbb{N}$, the abstraction (partial) function $abs_j : S \to \hat{S}_j$ is defined so that for every $s \in \bigcup \hat{S}_j$, $abs_j(s)$ is the (uniquely defined) $\hat{s} \in \hat{S}_j$ s.t. $s \in \hat{s}$. The first set of abstract states, $\hat{S}_0$, is the singleton $\{[C(root(T))]_0\} \subseteq \{[s]_0 : s \in S\}$.

Lemma 3.1.1. *At every point along the run of the algorithm, the abstract state space $\hat{S}$ satisfies that $\bigcup \hat{S} = \bigcup_{n \in T} [C(n)]_0$.*

Proof. We prove the lemma by induction over the changes in the abstract state space. For the base case, T contains a root node n, and it follows that $\hat{S} = \{abs(n)\} = \{[C(n)]_0\}$, and so the claim follows. For the induction step, assume that the abstract state space $\hat{S}$ satisfies that $\bigcup \hat{S} = \{[C(n)]_0 \mid n \in T\}$, and that the state space now changes. As explained above, the change may be caused by one of two reasons:

- Abstract state split: There exists an astate $\hat{s} \in \hat{S}$ that is split into $\hat{s}_1, \hat{s}_2$. Now, the new set of astates $\hat{S}'$ equals $(\hat{S} \setminus \hat{s}) \cup \{\hat{s}_1, \hat{s}_2\}$. As of the fact that $\hat{s} = \hat{s}_1 \cup \hat{s}_2$, it follows that $\bigcup \hat{S}' = \bigcup \hat{S}$, and so the claim holds as of the induction assumption.

- Concrete state discovery: In this case, a new concrete state s is discovered and added as a node in T. In this case, $abs(s) \neq \hat{s}$ for every $\hat{s} \in \hat{S}$. Thus, $s \notin \bigcup \hat{S}$, and by the induction assumption $s \notin \bigcup_{n \in T} [C(n)]_0$. In particular, we get that for every $n \in T$, $L(s) \neq L(n)$. Now, when adding $[s]_0$ to $\hat{S}$, the new abstract state space is $\hat{S}' = \hat{S} \cup \{[s]_0\}$. This new abstract state space satisfies the required property. ■

OMG maintains the following data about the abstract model, which is saved in *absStructure*:

- *Abstract states*: Different astates are discovered as OMG progresses. However, it may be the case that not all astates in the current $\hat{S}$ correspond to nodes on T. For example, a split of an astate $\hat{s}$ may result in astates $\hat{s}', \hat{s}''$ s.t. $\hat{s}'$ or $\hat{s}''$ contains no cstates from T. The abstract model contains all of the astates that were discovered along the run of OMG.

- *May transitions*: New astates may be created when new cstates are discovered by the algorithm and added to *item*, creating new may transitions between these astates. The algorithm sometimes learns that an astate $\hat{s}$ has no may transitions to some astate $\hat{s}'$. The abstract model records such pairs in a relation R^{may}_{no}, that contains all pairs $(\hat{s}, \hat{s}')$ for which it learns that there is no may-transition from $\hat{s}$ to $\hat{s}'$.

- *Must hyper-transitions*: Like in the case of may-transitions, the abstract model records pairs $(\hat{s}, \hat{D})$, for which it learns that $(\hat{s}, \hat{D}) \in \hat{R}^{must}$.

- *May closures*: OMG records pairs of the form $(\hat{s}, A)$, for which it is inferred that the pair is a may closure or that it is not a may closure.

The main purpose of *absStructure* is to maintain all information about the abstract model that has been learned so far. This is important since otherwise an on-the-fly algorithm may recompute again and again such information.

In addition to gathering data about the abstract model, information that is computed during a run of OMG is saved even if it is not currently used. For example, after computing the successors of a cstate s, they are recorded. Later, when computing the successors of another node n s.t. $C(n) = s$, this information is re-used.

3.2 Recursive handling of CTL formulas

OMG is a recursive algorithm. We first define the notion of a *goal*.

Definition 3.2.1. (Goal)
A *goal* is a pair $g = (n, \varphi)$, where $n \in T$ and φ is a CTL formula. We say that g *holds* if $n \models \varphi$.

The model checking task of whether a node n satisfies φ is thus translated into checking whether the goal (n, φ) holds. For that matter, the algorithm decomposes this goal into subgoals lazily, which are checked recursively. Unlike explicit model checking (presented in [CGK$^+$18]), where subformulas are checked in the entire model, in our work, subgoals are only checked when needed. For example, to check the goal $(n, p \wedge AXEGq)$, we check the goals (n, p) and (n', EGq) for every successor n' of n in T. When OMG finishes checking a goal (n, φ), it labels n with either φ or $\neg\varphi$.

OMG is composed of subprocedures, each handles CTL formulas of a certain form, according to the main connective of the formula. It contains subprocedures to handle atomic propositions, logical connectives, and the operators EX, AV, EV. We also need the following definition.

Definition 3.2.2. For an astate $\hat{s} \in \hat{S}$, $\hat{s}$ satisfies a formula φ, denoted $\hat{s} \models \varphi$, if for every $s \in \hat{s}$ it holds that $s \models \varphi$.

The following property is required when checking compound CTL formulas, and OMG makes sure it holds. It is be further explained in later sections.

Property 3.2.3. Let $g = (n, \varphi)$ be a subgoal checked by OMG. Let $\widehat{M_j}$ be the abstract model obtained after OMG finishes to evaluate g. Let $\hat{s} \in \hat{S_j}$ be the astate that satisfies $C(n) \in \hat{s}$. Then, $\forall t \in \hat{s},\ t \models \varphi \iff$ g holds.

As a consequence of the lemma, after checking a goal (n, φ), OMG labels $abs(n)$ as well. Thus, checking goals of the form (n', φ) where $abs(n) = abs(n')$ becomes easy.

3.3 Initialization and Recursive Activation

The initialization procedure is presented in Algorithm 3.1 and explained below.

Algorithm 3.1 OMG

```
Input: Kripke structure $M$, cstate $s \in S$, $\varphi \in CTL$
Output: $True$ if $M, s \models \varphi$, $False$ otherwise
 1: global $T$ = initUnwindingTree($M, s$)
 2: global $abs$ = initAbstraction($M$)
 3: global $absStructure$ = initAbstractStructure($M$)
 4: Let $g$ be the goal $(root(T), \varphi)$
 5: return RecurCtl($g$)                                  ▷ See Algorithm 3.2
```

The initialization method is given a structure M, a cstate $s \in S$ and a specification φ. The global data structures are initialized at the beginning of the run. The unwinding tree T is initialized to a tree that contains a single node labeled with s. The abstraction function *abs* is initialized to mapping s to $[s]_0$. The abstract structure, *absStructure*, is initialized to contain the astate $[s]_0$. RecurCtl is then called over the root of T and the specification, to check whether $s \models \varphi$.
The procedure RecurCtl is presented below.

Algorithm 3.2 RecurCtl

```
Input: Goal $g = (n, \varphi)$
Output: $True$ if $(n, \varphi)$ holds, $False$ otherwise
 1: if $\varphi \in \{\top, \bot\}$ then
 2:     Return the boolean value of $\varphi$
 3: Compute $\hat{s} = abs(n)$
 4: if $\hat{s}$ is labeled with $\varphi$ or $\neg\varphi$ then      ▷ for every $p \in AP$, $p$ is decided for $\hat{s}$
 5:     return the boolean value $\hat{s} \models \varphi$
 6: Let # be the main connective of $\varphi$
 7: Let $McResult$ = Handle#($g$)
 8: if $McResult$ is $True$ then
 9:     Label $abs(n)$ with $\varphi$
10: else
11:     Label $abs(n)$ with $\neg\varphi$
12: return $McResult$
```

After initialization, RecurCtl analyzes the goal $(root(T), \varphi)$, to check whether $s \models \varphi$. Depending on the main connective of φ, the appropriate subprocedure is called. RecurCtl is called by every such subprocedure to further analyze subformulas of φ.

Given a goal $g = (n, \varphi)$, every subprocedure updates *absStructure* by adding either the label φ or the label $\neg\varphi$ to the astate $abs(n)$. The formula φ is then said to be *decided* w.r.t. state $abs(n)$.

Note that along a run of OMG, if an astate is decided for some formula, it is also decided for all of its subformulas, due to both Property 3.2.3 and the recursive nature of OMG.

In the following chapters we first describe the subprocedures for checking $ApVq$ and $EpVq$ and then the subprocedures for the other operators.

Chapter 4

Handling ApVq

We provide intuition for handing formulas of the form AGp, and then extend it to formulas of the form $ApVq$.

4.1 Intuition to AG

We explain the algorithm for AGp, and then extend it for formulas of the form $ApVq$, where $p, q \in CTL$.

Consider the formula AGp. Given a cstate s, we explore the states reachable from it in order to check whether they all satisfy p. For this purpose, we develop the unwinding tree T rooted at a node n. The unwinding tree is developed one node at a time, according to some node-choosing heuristic. A promising heuristic is developing nodes with minimal depths first, unless some node is marked as "urgent", in which case it is the next node to be developed. An example for the latter case can be found in the procedure INDAV, described later in this chapter.

Whenever we develop a node m in the tree, we check whether $m \models p$. If it does, then the algorithm goes on and develops the tree further. Otherwise, the trace from n to m is a violation of the statement $n \models AGp$, and *False* is returned.

Up to this point, our method is most suitable for refutation. If S is finite, then after developing at most $|S|$ levels of the tree without halting, we may terminate with a positive answer. However, we wish for our algorithm to terminate with a positive answer before unwinding that many levels of the tree. For that, we use an inductive invariant suitable for AGp, defined in the following manner:

Definition 4.1.1. (Inductive invariant for AGp)
Let p be a CTL formula. A set $D \subseteq \hat{S}$ is an *inductive invariant for AGp* if the following conditions hold:

- for every $\hat{s} \in D$ and for every $\hat{t}$ such that $(\hat{s}, \hat{t}) \in \hat{R}^{may}$, it holds that $\hat{t} \in D$.
- for every $\hat{s} \in D$, $\hat{s} \models p$.

Recall that $\bigcup D$ consists of all cstates that are mapped to any $\hat{s} \in D$.

Observation 4.1.2. If D is an inductive invariant for AGp, then for every $s \in \bigcup D$, it holds that $s \models p$.

This holds as every cstate $s \in \bigcup D$ is assigned an astate $\hat{s} \in D$ s.t. $\hat{s} \models p$. As of Property 3.2.3, we get that $s \models p$.

Lemma 4.1.3. *Let p be a CTL formula. If D is an inductive invariant for AGp, then $\forall s \in \bigcup D$, $s \models AGp$. By Definition 3.2.2, we also have $\forall \hat{s} \in D$, $\hat{s} \models AGp$.*

Before proving Lemma 4.1.3, we prove the following lemma.

Lemma 4.1.4. *Let p be a CTL formula, D be an inductive invariant for AGp and $s \in \bigcup D$. Let $\pi = s_0, s_1, \ldots$ be a path from s. Thus, for every $k \geq 0$ it holds that $s_k \in \bigcup D$.*

Proof. We prove this by induction over k.

For the base case, $k = 0$. Indeed, as $s_0 = s$, it follows that $s \in \bigcup D$, and so the base case is proved.

We now prove the induction step. First, by the induction hypothesis, we get that $s_k \in \bigcup D$. We now prove that $s_{k+1} \in \bigcup D$.

As $s_k \in \bigcup D$, there exists an astate $\hat{s}_k \in D$ s.t. $s_k \in \hat{s}_k$. Let $\hat{s}_{k+1} \in \hat{S}$ be $abs(s_{k+1})$ (that is, $s_{k+1} \in \hat{s}_{k+1}$). As $(s_k, s_{k+1}) \in R$, we get that $(\hat{s}_k, \hat{s}_{k+1}) \in \hat{R}^{may}$, by the definition of may transitions. By Definition 4.1.1, we get that $\hat{s}_{k+1} \in D$, and so $s_{k+1} \in \bigcup D$, as required. ■

We now prove Lemma 4.1.3

Proof. Let $\hat{s} \in D$, and assume by contradiction that it does not hold that $\hat{s} \models AGp$ Then, there exists $s \in \hat{s} \subseteq \bigcup D$ such that $s \not\models AGp$. Thus, $s \models EF\neg p$, and so there exists a path $\pi = s_0, s_1, \ldots$ in M, s.t. $s = s_0$ and there exists $k \geq 0$ s.t. $s_k \models \neg p$. Now, consider the path π. It holds that $s_0 = s \in \bigcup D$, and so Lemma 4.1.4 applies for π. Thus, $s_k \in \bigcup D$. By Observation 4.1.2, we get that $s_k \models p$, which is a contradiction. We conclude that $s \models AGp$. ■

Our algorithm checks whether $abs(T)$ is an inductive invariant for AGp. If this is the case, then all may-transitions, originating in an astate in D lead to astates in D. Thus, D is an inductive invariant for AGp.

If $abs(T)$ is not an inductive invariant, then there is an astate $\hat{s} \in abs(T)$ that has a may transition to an astate $\hat{t} \notin abs(T)$. In this case, either more unwinding should be conducted or the abstraction is too coarse to prove the existence of an inductive invariant. We explain how to handle these cases in the next section, in which the more general case of $ApVq$ is discussed.

4.2 Extension to ApVq

We now extend the algorithm for AGp to the $ApVq$ case. We require all paths π from s to satisfy either Gq or $qU(q \wedge p)$, which is a necessary and sufficient condition for $s \models ApVq$ to hold.

For that, the unwinding is changed. A trace $n_0, \ldots, n_k$ is a counterexample to the statement $s \models ApVq$ if for every $i < k, n_i \models q \wedge \neg p$ and $n_k \models \neg q$. If such a trace is found, the algorithm returns a negative answer. If a node that satisfies $q \wedge p$ is found, this node is not further developed, as all paths that extend it satisfy $qU(p \wedge q)$.

We define the notion of an inductive invariant for $ApVq$, which allows OMG to prove that $s \models ApVq$ without exploring the entire state-space.

Definition 4.2.1. (Inductive invariant for $ApVq$)
Let p, q be CTL formulas. Two sets of astates (D, B), are an *inductive invariant for* $ApVq$ if the following holds:

- $\forall \hat{s} \in D, \forall \hat{t} \in \hat{R}^{may}(\hat{s}) : \hat{t} \in D \cup B$.
- $\forall \hat{s} \in B, \hat{s} \models q \wedge p$.
- $\forall \hat{t} \in D, \hat{t} \models q \wedge \neg p$.

The intuitive meaning of the invariant is that every state in it either satisfies $q \wedge p$, or it satisfies $q \wedge \neg p$ and all of its successors are contained in the invariant.

Observation 4.2.2. If (D, B) is an inductive invariant for $ApVq$, then for every $s \in \bigcup D$ it holds that $s \models q \wedge \neg p$, and for every $s \in \bigcup B$ it holds that $s \models q \wedge p$.

This holds as every cstate $s \in \bigcup D$ is assigned an astate $\hat{s} \in D$ s.t. $\hat{s} \models q \wedge \neg p$, and as of Property 3.2.3, we get that $s \models q \wedge \neg p$. Similarly, for $s \in \bigcup B$, it is assigned an astate $\hat{s} \in B$ s.t. $\hat{s} \models q \wedge p$, and as of Property 3.2.3, we get that $s \models q \wedge p$.

Lemma 4.2.3. *Let p, q be CTL formulas. If (D, B) is an inductive invariant for $ApVq$, then for every $s \in \bigcup(D \cup B)$ it holds that $s \models ApVq$, and so for every $\hat{s} \in (D \cup B), \hat{s} \models ApVq$.*

Before proving Lemma 4.2.3, we prove the following lemma.

Lemma 4.2.4. *Let p, q be CTL formulas, (D, B) be an inductive invariant for $ApVq$ and $s \in \bigcup(D \cup B)$. For every $k \geq 0$ and for every path $\pi = s_0, s_1, \ldots,$ from s one of the following holds:*

- *$\exists 0 \leq j \leq k$ s.t. $s_j \in \bigcup B$ and $s_i \in \bigcup D$ for every $0 \leq i < j$.*
- *$\forall i \geq 0$, $s_i \in \bigcup D$.*

Proof. We prove this by induction on k. For the base case, $k = 0$. Let π be a path from s. Indeed, $s_0 = s \in \bigcup D \cup B$. Thus, either $s_0 \in \bigcup D$ or $s_0 \in \bigcup B$, and so the requirement holds.

For the induction step, let π be a path from s. By the induction hypothesis, we get that one of the following holds:

- $\exists 0 \le j \le k$ s.t. $s_j \in \bigcup B$ and $s_i \in \bigcup D$ for every $i < j$. In this case, it also holds that there exists $j' = j$ such that $0 \le j' \le k+1$, $s_{j'} \in \bigcup B$ and $s_i \in \bigcup D$ for every $i < j'$, and so the claim holds.

- $\forall i \le k$, $s_i \in \bigcup D$. In this case, it holds in particular that $s_k \in \bigcup D$. Thus, there exists an astate $\hat{s}_k \in D$ s.t. $s_k \in \hat{s}_k$. Let $\hat{s}_{k+1} \in \hat{S}$ be $abs(s_{k+1})$ (that is, $s_{k+1} \in \hat{s}_{k+1}$). As $(s_k, s_{k+1}) \in R$, we get that $(\hat{s}_k, \hat{s}_{k+1}) \in \hat{R}^{may}$, by the definition of may transitions. By Definition 4.2.1, we get that $\hat{s}_{k+1} \in D \cup B$. If $\hat{s}_{k+1} \in D$, the second requirement of this lemma holds. Otherwise, if $\hat{s}_{k+1} \in B$, then the first one holds. ■

We now prove Lemma 4.2.3

Proof. Let $s \in \bigcup(D \cup B)$, and assume by way of contradiction that $s \not\models ApVq$. Thus, $s \models E(\neg p)U(\neg q)$, and so there exist a path $\pi = s_0, s_1, \ldots$ from s and $k \ge 0$ s.t. $s_k \models \neg q$, and for every $0 \le i < k$ it holds that $s_i \models \neg p$.

Consider the path π. It holds that $s_0 = s \in \bigcup D \cup B$, and so Lemma 4.2.4 applies for k. However, both possible consequences of the lemma do not hold, as shown below:

- It does not hold that $\exists 0 \le j \le k$ s.t. $s_j \in \bigcup B$ and $s_i \in \bigcup D$ for every $i < j$:

  – For every $0 \le j < k$, $s_j \not\models p$, and as of Observation 4.2.2 it holds that $s_j \notin \bigcup B$.
  – For $j = k$, $s_j \not\models q$ and so $s_j \notin \bigcup B$.

  Either way, this possible consequence does not hold.

- It does not hold that for every $0 \le i \le k$, $s_i \in \bigcup D$, as $s_k \not\models q$ and so $s_k \notin \bigcup D$ by Observation 4.2.2.

Neither of the possible results of Lemma 4.2.4 holds, which is a contradiction. Thus, $s \models ApVq$. ■

OMG returns a positive answer when $abs(T)$ can be divided into two sets D, B s.t. (D, B) is an inductive invariant for $ApVq$.

As $AGq \equiv A(false)Vq$, the algorithm for $ApVq$ is a proper generalization of the algorithm for AGq. This is due to the fact that if $p = False$ then $B = \emptyset$, in which case Definition 4.2.1 is identical to Definition 4.1.1.

Our algorithm does not change in case p, q, are general CTL formulas (and not necessarily atomic propositions). This is due to the fact that the algorithm handles subgoals by need, such that when querying whether a subgoal of the form (m, p) holds, it recurs over that subgoal before going on.

4.3 The full algorithm

4.3.1 HandleAV description

Algorithm 4.1 HANDLEAV

```
Input: Goal $g = (initNode, \varphi = ApVq)$
Output: $True$ if $initNode \models \varphi$, $False$ otherwise
 1: $ToVisit \leftarrow \{initNode\}$                         ▷ Nodes to be visited
 2: while $ToVisit \neq \emptyset$ do
 3:     Choose nextNode from $ToVisit$
 4:     Let $\tau$ be the trace from initNode to nextNode
 5:     UpdateAbstract($nextNode$)
 6:     RECURCTL($nextNode$, $q$)
 7:     if $nextNode \not\models q$ then
 8:         STRENGTHENTRACE($\tau$)
 9:         return $False$
10:     RECURCTL($nextNode$, $p$)                           ▷ $nextNode \models q$
11:     if $nextNode \not\models p$ then
12:         Add successors of $nextNode$ to $ToVisit$
13:     if INDAV($initNode$, $p$) then
14:         return $True$
15: STRENGTHENSUBTREE(initNode)
16: return $True$
```

HANDLEAV is presented in Algorithm 4.1. In line 2 we iterate over all nodes in $ToVisit$. In each iteration, HANDLEAV picks a node $nextNode$ from $ToVisit$. We say that in this iteration, $nextNode$ is *examined*. If $ToVisit$ becomes empty, then there are no more nodes to examine, and so we return *True*. An example of that is the case where $initNode \models p \wedge q$. The strengthening conducted in line 15 (explained in section 4.4) ensures that all astates in $abs(T)$ satisfy $ApVq$ as well. This information is recorded in *absStructure*.

In line 5, the data structures of *absStructure* are updated. If *absStructure* contains an astate $\hat{s}$ s.t. $abs(nextNode) = \hat{s}$, nothing is changed. Otherwise, the astate $[nextNode]_0$ is created and added to *absStructure*. Additionally, HANDLEAV sets $abs(nextNode) = [nextNode]_0$.

In line 6, RECURCTL is called to check whether $nextNode \models q$. If it does not, then τ proves that $initNode \not\models ApVq$. In that case, τ is strengthened (in order to guarantee that $abs(initNode) \not\models ApVq$ as well) and *False* is returned. All astates in $abs(\tau)$ after strengthening do not satisfy $ApVq$, and we record this in *absStructure* by labeling them with $\neg ApVq$.

If line 10 is reached, then it is known that $nextNode \models q$. Then, we check whether $nextNode \models p$. If it does, then we stop unwinding nextNode, as all paths from $C(initNode)$ that pass through nextNode (that is, all paths that extend τ) satisfy $qU(q \wedge p)$, and in particular pVq. We then call INDAV. If, however, $nextNode \not\models p$ (line 11), the successors of $nextNode$ are computed and added to T. The new nodes are added to $ToVisit$ to be examined later in the run.

In line 13, INDAV is called to check whether $abs(T)$ can be partitioned to (D, B) which form an inductive invariant for $ApVq$. If so, the algorithm returns *True*. In that case, all astates in $abs(T)$ are labeled with $ApVq$ due to Lemma 4.2.3. Otherwise, it goes on to the next iteration of the loop in line 2.

4.3.2 indAV description

The procedure INDAV is presented in Algorithm 4.2.

Algorithm 4.2 INDAV

```
Input: Unwinding tree $T$ rooted at $initNode$, $p \in CTL$
Output: True if $abs(T)$ can be partitioned to an inductive invariant, False otherwise
 1: $toCheck \leftarrow \{\hat{s} \in abs(T) \mid \hat{s} \not\models p\}$      ▷ We assume that every $\hat{s} \in abs(T)$ satisfies $q$
 2: while $toCheck \neq \emptyset$ do
 3:     Choose abstract state $\hat{s}$ from $toCheck$, and let $n \in T$ s.t. $abs(n) = \hat{s}$
 4:     isClosure ← IsMayClosure($\hat{s}$, $abs(T)$)
 5:     if isClosure is True then
 6:         Update absStructure with the may closure $(\hat{s}, abs(T))$.
 7:         Remove $\hat{s}$ from toCheck
 8:     else
 9:         Let $\hat{t} \notin abs(T)$ such that $(\hat{s}, \hat{t}) \in \hat{R}^{may}$
10:         if $C(n)$ has a successor $t \in \hat{t}$ then
11:             Continue to the next iteration of HandleAV with $nextNode = n$
12:         else
13:             SplitEX($\hat{s}, \hat{t}$)
14:         return False
15: return True
```

The algorithm INDAV iterates (in line 2) over the set of astates *toCheck*. This set is initialized to contain all astate in $abs(T)$ which do not satisfy p. We attempt to prove that each astate in that set has a may closure to $abs(T)$, in order to find an inductive invariant.

Note that for every $\hat{s} \in toCheck$ there exists $n \in T$ such that $abs(n) = \hat{s}$, by the definition of $abs(T)$. If several such nodes n exist, each of them may be chosen. Heuristically, it is preferable to choose a node with a maximal depth.

INDAV checks (in line 4) whether $(\hat{s}, abs(T))$ is a may closure (see Definition 2.1.2) [1]. If it is a may closure (line 5), INDAV records it in *absStructure* and analyzes the

[1]Before conducting an explicit check, INDAV uses the fact that if there exists a may closure in

next astate in *toCheck* after removing $\hat{s}$ from *toCheck* (so it is not chosen again). If $(\hat{s}, toCheck)$ is a may closure for every $\hat{s} \in toCheck$, we return *True* in line 15.

Otherwise, there exists an astate $\hat{s} \in toCheck$ such that $(\hat{s}, abs(T))$ is not a may closure. Before returning *False* (in line 14), we use the non-existence of the may closure to proceed.

As a may closure does not exist, there exists an astate $\hat{t} \notin abs(T)$, s.t. $(\hat{s}, \hat{t}) \in \hat{R}^{may}$. There are two possible reasons for that:

- There are reachable cstates which were not discovered, and for which there is no matching astate in $abs(T)$.
- The abstraction is too coarse, and the may transition $(\hat{s}, \hat{t})$ stems from unreachable cstates that are mapped to $\hat{s}$.

In line 10, the algorithm attempts to identify which of the two reasons holds. It does so by checking whether $C(n)$ has a successor $t \in \hat{t}$. If it does, we further unwinds T from n (line 11). Otherwise, refinement is required in order to separate the cstates from $\hat{s}$ that have a concrete successor in $\hat{t}$ from those which do not. For that, SPLITEX is applied in line 13.

SPLITEX follows Definition 2.1.1, where the astate $\hat{s}$ is split into two new astates, $\hat{s}^Y, \hat{s}^N$, according to the property $P = \{s \in S \mid \exists t \in \hat{t} \text{ s.t.} (s,t) \in R\}$. It holds that the cstates in $\hat{s}^Y$ satisfy P and the cstates in $\hat{s}^N$ do not. We now show that both new astates are not empty. As the *if* in line 10 is not taken, it holds that $C(n) \in \hat{s}^N$. Moreover, as $(\hat{s}, \hat{t}) \in \hat{R}^{may}$, there exist $s^* \in \hat{s}$ and $t^* \in \hat{t}$ such that $(s^*, t^*) \in R$. We conclude that $s^* \in \hat{s}^Y$, and so $\hat{s}^Y \neq \emptyset$.

The abstract model *absStructure* is updated according to the split. All must hyper-transitions of the form $(\hat{s}, \hat{G})$ are replaced by the must hyper-transitions $(\hat{s}^Y, \hat{G})$ and $(\hat{s}^N, \hat{G})$. Then, must hyper-transitions of the form $(\hat{t}, \hat{D})$ s.t. $\hat{t}$ is some astate and $\hat{s} \in \hat{D}$ are replaced by must hyper-transitions $(\hat{t}, ((\hat{D} \setminus \hat{s}) \cup \{\hat{s}^Y, \hat{s}^N\}))$. Moreover, we add the following data to *absStructure*: $(\hat{s}^Y, \hat{t}) \in \hat{R}^{must}$ and $(\hat{s}^N, \hat{t}) \in \hat{R}^{may}_{no}$.

Recall that, in line 15, we return *True*, as this line is reached if and only if we proved for every $\hat{s} \in toCheck$ that $(\hat{s}, abs(T))$ is a may closure. In this case, $abs(T)$ can be partitioned to (D, B) which form an inductive invariant for $ApVq$.

Example 4.3.1. We demonstrate HANDLEAV over the example in Figure 2.1. Consider the subgoal $(n_1, ApVq)$. Initially, $ToVisit = \{n_1\}$, and $nextNode = n_1$ in the first iteration. As $n_1 \models q$ and $n_1 \not\models p$, the successors of s_1 are computed and added to T (i.e., n_2 is added to T). Let $\hat{s}_1 = [s_1]_0 = \{s \in S \mid L(s) = \{q, r\}\}$. Next, HANDLEAV checks whether $(\{\hat{s}_1\}, \emptyset)$ is an inductive invariant for $ApVq$. The check fails due to the transition $(s_3, s_4) \in R$ ($s_3 \in \hat{s}_1$ and $s_4 \notin \hat{s}_1$). Since s_1 does not have a successor in $\hat{s}_1$, SPLITEX is applied over $\hat{s}_1$, resulting in $\hat{s}_1' = \{s_1, s_2\}$ and $\hat{s}_1'' = \{s_3\}$. HANDLEAV

absStructure of the form $(\hat{s}, \hat{D})$ s.t. $\hat{D} \subseteq abs(T)$, then $(\hat{s}, abs(T))$ is also a may closure.

proceeds to the next iteration, where $nextNode = n_2$. As $s_2 \models q \wedge \neg p$ it unwinds n_2, adding n_2' to T. As $abs(n_1) = abs(n_2) = \hat{s}_1'$, HANDLEAV checks if $(\{\hat{s}_1'\}, \emptyset)$ is an inductive invariant for $ApVq$, and terminates with a positive answer.

We now prove a lemma that is a part of the proof of soundness of OMG.

Lemma 4.3.2. *(Soundness of Algorithm 4.2)*
Let T be an unwinding tree for a model M and let $p, q \in CTL$. Assume that for every $\hat{s} \in abs(T)$, $\hat{s}$ conforms w.r.t. p and q (Definition 2.1.3), and also that $\hat{s} \models q$. Then, INDAV *returns True if $abs(T)$ can be partitioned to (D, B), s.t. (D, B) forms an inductive invariant for $ApVq$, and False otherwise.*

Proof. INDAV contains a single loop in line 2. The set *toCheck* is initialized before the loop, and in every iteration a single astate $\hat{s}$ is examined. In every iteration of the loop, if the *if* statement in line 5 is taken, then $\hat{s}$ is removed from *toCheck*, and otherwise the loop terminates and *False* is returned in line 14.

Thus, *True* is returned if and only if the *if* statement in line 5 is taken for every $\hat{s} \in toCheck$, which means that *toCheck* becomes empty after all of the iterations of the loop. On the other hand, *False* is returned if and only if there exists an astate $\hat{s} \in toCheck$ for which that *if* statement is not taken.

Assume first that *True* is returned. Thus, in every iteration of the loop, it is proved for some astate $\hat{s} \in toCheck$ (in line 4) that $(\hat{s}, abs(T))$ is a may closure. If *True* is returned, then this is proved for all astates $\hat{s} \in toCheck$. We prove that this implies that (D, B) is an inductive invariant for $ApVq$, where $D = \{\hat{s} \in abs(T) \mid \hat{s} \not\models p\}$ and $B = \{\hat{s} \in abs(T) \mid \hat{s} \models p\}$. Note that $abs(T) = D \cup B$.

First, for every $\hat{s} \in abs(T)$, by the assumption, $\hat{s} \models q$. By the definitions of D, B, we get that for every $\hat{s} \in B$, $\hat{s} \models q \wedge p$ and that for every $\hat{t} \in D$, $\hat{t} \models q \wedge \neg p$. Now, let $\hat{s} \in D$, and let $\hat{t} \in \hat{R}^{may}(\hat{s})$. Recall that, $\hat{s}$ is in *toCheck* at the beginning of Algorithm 4.2 (by the definition of *toCheck*). As the *if* is taken when $\hat{s}$ is examined, we know that $(\hat{s}, abs(T))$ is a may closure. By the definition of may closure, we get that $\hat{t} \in abs(T) = D \cup B$.

Assume now that *False* is returned. Thus, there exists an astate $\hat{s} \in toCheck$ s.t. the *if* statement is not taken, meaning that $(\hat{s}, abs(T))$ is not a may closure. As $\hat{s} \in toCheck$, we get that $\hat{s} \in D$. However, by Definition 2.1.2, there exists $\hat{t} \notin abs(T)$ s.t. $(\hat{s}, \hat{t}) \in \hat{R}^{may}$. In particular $\hat{t} \notin D \cup B$. This is a contradiction to the definition of an inductive invariant. Thus, $abs(T)$ cannot be split to an inductive invariant. ■

Lemma 4.3.3. *(Termination of Algorithm 4.2) If M is a finite structure, then Algorithm 4.2 terminates after a finite number of steps.*

Proof. Note that, in every step in the run of the algorithm, the abstract state space in *absStructure* is bounded by $|S|$, as no astate may be empty. As such, the number of iterations of the loop in line 2 (in Algorithm 4.2) is at most the number of astates recorded in *absStructure*, which is finite. In each iteration, INDAV checks whether some astate $\hat{s}$ satisfies that $(toClose, abs(T))$ is a may closure (line 4 in Algorithm 4.2). This check is done by retrieving data from *absStructure*, and possibly checking the satisfiabiliy of some boolean formula without quantifiers. This check is equivalent to solving SAT, which is decidable (that is, can be done in a finite number of steps). As there is a finite amount of iterations, and every iteration is conducted in a finite number of steps, the algorithm terminates in a finite number of steps as well. ■

4.4 Strengthenings

Two types of strengthenings are performed by OMG. We first give motivation for the strengthening, and then explain how it is done.

Recall that we wish for Property 3.2.3 to hold. It states that after checking a goal $g = (n, \varphi)$, it holds that $n \models \varphi \iff abs(n) \models \varphi$. This lemma is essential for our algorithm to handle general CTL formulas.

For instance, in order for D to be an inductive invariant for $AG\varphi$, where φ is a complex formula, each *astate* $\hat{s} \in D$ is required to satisfy φ, which is what the lemma ensures.

4.4.1 Trace Strengthening

Consider line 8 in Algorithm 4.1, where τ, which proves that $initNode \not\models ApVq$, is strengthened. The trace τ alone does not ensure that every cstate $t \in abs(initNode)$ has a corresponding ”violating” trace, and so it does not guarantee that Property 3.2.3 holds. The function STRENGTHENTRACE, presented in Algorithm 4.3, solves this problem.

Algorithm 4.3 STRENGTHENTRACE

```
Input: Trace $\tau = (n_0, n_1, \ldots, n_k)$
1: for each $i = k, \ldots, 1$ do
2:     SPLITEX($abs(n_{i-1})$, $abs(n_i)$)
3:     Let $\hat{s}^Y, \hat{s}^N$ be the splits of $abs(n_{i-1})$.
4:     Update $abs(n_{i-1})$ to $\hat{s}^Y$                    ▷ Explained below
```

Consider a trace of length one, denoted (n, n'), such that $n' \not\models q$. Thus, $abs(n)$ has to be refined to only contain cstates that, like $C(n)$, have a successor in $abs(n')$ [2]. For traces of the form $(n_0, n_1, \ldots, n_k)$, the refinement is done iteratively up the trace. That is, the trace (n_{i-1}, n_i) is strengthened for every $i = k, \ldots, 1$. After every split, the astates that appear on the trace change dynamically, based on the result of the split applied in the previous iteration. Let $\hat{s}$ be $abs(n_{i-1})$ before the split in line 2, which is preformed w.r.t. the updated $abs(n_i)$. Then, $abs(n_{i-1})$ is set to $\hat{s}^Y$ after the split.

[2]This is equivalent to intersecting $abs(n)$ with the pre-image of $abs(n')$ w.r.t. R.

We prove the following lemma to assert the correctness of this algorithm.

Lemma 4.4.1. *Let $\tau = (n_0, n_1, \ldots, n_k)$ be a trace in M over which* StrengthenTrace *is applied. For every $0 \leq i \leq k$, let $\hat{s}_{k-i}$ be $abs(n_{k-i})$ after i iterations of the loop in* StrengthenTrace. *Then, for every $0 \leq i \leq k$, it holds for every $s \in \hat{s}_{k-i}$ there exists a series of cstates $s_{k-i}, s_{k-i+1}, \ldots, s_k$ such that:*

- $s = s_{k-i}$.
- $(s_j, s_{j+1}) \in R$ *for every* $k - i \leq j < k$.
- $s_j \in \hat{s}_j$ *for every* $k - i \leq j \leq k$.

Moreover, the algorithm terminates in a finite number of steps.

Proof. We prove the Lemma by induction over i. For the base case, $i = 0$ and indeed the Lemma holds for the series s of length one.

Assume that the Lemma holds after i iterations of the algorithm, and consider the $(i+1)^{th}$ iteration. In this iteration, $abs(n_{k-i-1})$ is split w.r.t. the property $P = \{s \in S : \exists t \in S[(s,t) \in R \wedge t \in \hat{s}_{k-i}]\}$. Let $\hat{s}$ be $abs(n_{k-i-1})$ before the split. Note that $\hat{s}_{k-i-1}$ is the astate $abs(n_{k-i-1})$ after the split. It holds that $\hat{s}_{k-i-1} = \hat{s}^Y$, which is the split of $\hat{s}$ that satisfies P, as $C(n_{k-i-1}) \in \hat{s}$, $(C(n_{k-i-1}), C(n_{k-i})) \in R$ and $n_{k-i} \in \hat{s}_{k-i}$. Due to the split, there exists a must hyper-transition from $\hat{s}_{k-i-1}$ to $\hat{s}_{k-i}$.

Let $s \in \hat{s}_{k-i-1}$ (recall that after this split, $\hat{s}_{k-i-1} = abs(n_{k-i-1})$). Then, there exists $t \in \hat{s}_{k-i}$ s.t. $(s,t) \in R$. By the induction hypothesis, there exists a series of cstates $s_{k-i}, s_{k-i+1}, \ldots, s_k$ such that $t = s_{k-i}$ and $s_j \in \hat{s}_j$ for every $k - i \leq j \leq k$. Additionally, (s_j, s_{j+1}) for every $k - i \leq j < k$. Now, we show that the series $s, s_{k-i}, s_{k-i+1}, \ldots, s_k$ is the required series:

- s is the first cstate in the series.
- We showed that $(s,t) = (s, s_{k-i}) \in R$, and for $k - i \leq j < k$, it holds that $(s_j, s_{j+1}) \in R$ due to the induction hypothesis.
- $s \in \hat{s}_{k-i-1}$ by the definition of s, and for $k - i \leq j \leq k$ it holds by the induction hypothesis that $s_j \in \hat{s}_j$.

Regarding termination, consider an iteration of the loop. It terminates as splitting an astates and updating *abs* is done in a finite number of steps. As there are k iterations, the algorithm terminates. ■

Example 4.4.2. We demonstrate strengthening using the example in Figure 2.1. Consider the goal $(n_5, ApVq)$. Assume OMG finds that the trace $\tau = (n_5, n_7, n_8)$ is a counterexample to the statement $n_5 \models ApVq$. Before returning *False*, trace strengthening is applied to τ, in order for Property 3.2.3 to hold. In this case, it is vital as $abs(s_9) = abs(s_5)$ before strengthening, however $s_9 \models ApVq$ and $s_5 \not\models ApVq$.

Thus, we need to split $abs(s_5)$ in a manner that distinguishes between s_5, s_9. The split is done in two steps. First, we conduct SPLITEX to $abs(n_7)$ w.r.t. $abs(n_8)$. Secondly, we split $abs(n_5)$ w.r.t, *the updated* $abs(n_7)$, which is updated and changed in the previous step of the strengthening.

4.4.2 Subtree Strengthening

Subtree strengthening is required in line 15 of Algorithm 4.1, when unwinding of all traces from $initNode$ is terminated. In this case, it is guaranteed that $initNode \models ApVq$, as $initNode \models AqU(p \wedge q)$. However, it is not guaranteed that this holds for all states s s.t. $s \in abs(initNode)$. Recall that having all cstates in $abs(initNode)$ satisfy $\varphi = A(qU(p \wedge q))$ is important when φ is a subformula of the checked formula. In this case, subtree strengthening is conducted.

Consider an unwinding tree T that is composed of a root node n with successors $t_1, \ldots, t_j$, which are leaves in T. Assume that $n \models q \wedge \neg p$, and that $\forall 1 \leq i \leq j,\ t_i \models p \wedge q$. In such a case, $n \models ApVq$ since all transitions from n reach "good" states (that is, states that satisfy $p \wedge q$). Thus, we split $abs(n)$ w.r.t. the property of having *all* successors in the set of astates $\{abs(t_1), abs(t_2), \ldots, abs(t_j)\}$. Formally, we define $P = \{s \in S \mid \forall t \in R(s), t \in \bigcup_{k=1}^{j} abs(t_k)\}$. Note that n satisfies this property.

This split is referred to as SPLITAX, and it is also used in chapter 6. In case of a subtree of an arbitrary depth that should be strengthened, the generalization is done bottom-up, in a post-order scan of the subtree. This guarantees that Property 3.2.3 holds.

The procedure is given in Algorithm 4.4.

Algorithm 4.4 STRENGTHENSUBTREE

```
Input: Subtree rooted at $n$
1: if $n$ is a leaf then
2:     return
3: for each successor $n'$ of $n$ do
4:     STRENGTHENSUBTREE(n')
5: SPLITAX(n)
```

We prove the following lemma to assert the correctness of this type of strengthening.

Lemma 4.4.3. *Let T be an unwinding tree of M of depth k, over which* STRENGTHENSUBTREE *is applied. Then,* STRENGTHENSUBTREE *terminates. Moreover, after its termination, for every path $\pi = s_0, s_1, \ldots$ such that $s_0 \in abs(n_0)$ there exists a maximal trace $\tau = (n_0, n_1, \ldots, n_k)$ in T s.t. $s_i \in abs(n_i)$ for $0 \leq i \leq k$.*

Proof. We prove the Lemma by induction over k. Let $n = root(T)$. For the base case, $k = 0$, and so $T = \{n\}$. Indeed, n is a leaf in T and so the algorithm terminates.

Moreover, for each path $\pi = s_0, s_1, \ldots$ such that $s_0 \in abs(n_0)$, there exists the maximal trace (n) (of length 1), for which it holds that $C(n) \in abs(n)$ by definition.

Assume that the Lemma holds for unwinding trees of depth at most k, and let T be an unwinding tree of depth $k+1$. Let $\pi = s_0, s_1, s_2, \ldots$ s.t. $s_0 \in abs(n_0)$, and let $t_1, \ldots, t_l$ be the successors of n. We now prove termination of StrengthenSubtree for the induction step. First, as n is not of depth 0, it is not a leaf. Thus, the *if* statement in line 1 is not taken. Next, as n has a finite amount of successor and as each successor is the root of an unwinding tree of size at most k, we get by the induction assumption that each of the recursive activations terminates. Later, SplitAX is applied for n and it terminates in a finite number of steps as well. Overall, StrengthenSubtree terminates.

We now prove the rest of the claim in the Lemma. As SplitAX is conducted for n_0 in line 5, it is guaranteed that for each $s \in abs(n_0)$ and for each $t \in R(s)$ it holds that $t \in \bigcup_{i=1}^{l} abs(t_i)$. In particular, for $s_0 \in abs(n_0)$ and for $s_1 \in R(s_0)$, it holds that $s_1 \in \bigcup_{i=1}^{l} abs(t_i)$. Let $1 \leq j \leq l$ such that $s_1 \in abs(t_j)$. Consider the subtree of T rooted at t_j. It is of length at most k, and so by the induction hypothesis for the path $s_1, s_2, \ldots$ there exists a maximal path $n_1, \ldots, n_k, n_{k+1}$ such that $n_1 = t_j$ and for each $1 \leq i \leq k+1$, $s_i \in abs(n_i)$. Now, as $(s_0, s_1) \in R$, the trace $n_0, n_1, \ldots, n_k, n_{k+1}$ is a maximal trace in T. Moreover, due to the induction hypothesis and the fact that $s_0 \in abs(n_0)$ by its definition, we get that the Lemma holds. ■

We now prove soundness of Algorithm 4.1.

Lemma 4.4.4. *(Soundness of* HandleAV*)*
Let $g = (n, \varphi)$ be a goal, where $\varphi = ApVq$ and $p, q \in CTL$. Assume that after HandleAV *checks a subgoal (m, f), it holds that $abs(m)$ conforms w.r.t. f. Then, if* HandleAV *terminates, it returns True if and only if g holds. Moreover, if it terminates, $abs(n)$ conforms w.r.t. φ after termination.*

Proof. The loop in line 2 is conducted as long as $ToVisit \neq \emptyset$. It is initialized to $\{initNode\}$. In every iteration, a node nextNode is chosen from it and examined. By the structure of HandleAV, once a node is examined by the algorithm, it is never examined again (in particular, it never enters $ToVisit$ again).

In every iteration, HandleAV recurs over the subgoal $g' = (nextNode, q)$ in line 6. *False* is only returned from HandleAV if it examines a node $nextNode$ s.t. after this recursive activation, it holds that $nextNode \not\models q$.

We observe HandleAV returns *False* only if it examines a node that does not satisfy q. Thus, if a node is not a leaf in T, or it is not the last to be examined, then it satisfies q.

Moreover, let n be a node in T which is not a leaf. Then, when it is examined, the *if* statement in line 7 is not taken, as otherwise n would have been a leaf. Additionally, the *if* statement in line 11 is taken, as otherwise n would have been a leaf. Thus, the subgoal (n, q) holds and the subgoal (n, p) does not hold. In particular, we conclude that if a node is not a leaf in T, it does not satisfy p.

Assume first that *False* is returned by HANDLEAV. Then, there exists a node *nextNode* for which the *if* statement in line 7 is taken. Denote $\tau = n_0, \ldots, n_k$ (where $n_k = \mathit{nextNode}$), and let π be an infinite path that extends τ. We prove that $\pi \models (q \wedge \neg p)U(\neg q)$, and so $\mathit{initNode} \not\models ApVq$. By the observations above, we get that $\forall 0 \leq i < k$, $n_i \models q \wedge \neg p$. As we know that $\mathit{nextNode} \not\models q$, we get that indeed $\pi \models (q \wedge \neg p)U(\neg q)$, and so $\mathit{initNode} \not\models ApVq$. Thus, g does not hold.

We now prove that in this case, $abs(\mathit{initNode})$ conforms w.r.t. φ after *False* is returned. Before *False* is returned (in line 9), STRENGTHENTRACE is applied over τ. According to Lemma 4.4.1, for each $s \in abs(\mathit{initNode})$, there exists a series of cstates $s_0, s_1, \ldots, s_k$ such that $s = s_0$, $(s_i, s_{i+1}) \in R$ for $0 \leq i < k$ and $s_i \in \hat{s}_i$, where $\hat{s}_i$ equals $abs(n_i)$ after i iterations of the STRENGTHENTRACE.

Let $0 \leq i \leq k$, and let $\hat{t}_i$ be the astate $abs(n_{k-i})$ before calling STRENGTHENTRACE. It holds that $s_i \not\models p$ as $\hat{t}_i$ does not satisfy p, and $\hat{s}_i \subseteq \hat{t}_i$ (it is a split of $\hat{t}_i$). Due to similar arguments, it holds that for $0 \leq i < k$, $s_i \models q$ and $s_k \not\models q$. Let π be a path that extends τ. Thus, $\pi \models (q \wedge \neg p)U\neg q$, and so $s \not\models \varphi$. As this is true for every $s \in abs(\mathit{initNode})$, we get that $abs(\mathit{initNode})$ conforms w.r.t. φ.

Assume now that *True* is returned. As of the structure of the algorithm, *True* may be returned either in line 14 or in line 16. We show that in both cases g holds and that $abs(n)$ conforms w.r.t. φ after HANDLEAV terminates.

- Assume *True* is returned in line 14. In this case, the function INDAV is called and returns true. Let $n \in T$. We show that $n \models q$. As INDAV is called, n has been examined, and the subgoal (n, q) has been checked. If $n \not\models q$, the algorithm would return *False* in line 9. Thus, $n \models q$.

  As of the fact that for every astate $\hat{s} \in abs(T)$ there exists $n \in T$ s.t. $n \in \hat{s}$, it holds that $\hat{s} \models q$ due to the conformity assumption of this Lemma. Thus, all astates in $abs(T)$ satisfy q, and in particular they conform w.r.t. q. Moreover, by the structure of HANDLEAV we get that for every node $n \in T$, the subgoal (n, p) is checked when n is examined. Due to the conformity assumption of this Lemma, we get that all astates in T conform w.r.t. p.

  The prerequisites of Lemma 4.3.2 hold. Thus, we get that $abs(T)$ can be partitioned to (D, B) s.t. (D, B) is an inductive invariant for $ApVq$. By Lemma 4.2.3, we get that all astates $\hat{s} \in abs(T)$ satisfy $ApVq$, and in particular they conform w.r.t. φ. Moreover, as $n \in abs(n)$, $abs(n) \in abs(T)$ (and so $abs(n) \models \varphi$), we get that g holds. Thus, the Lemma holds in this case.

- Assume *True* is returned in line 16. In this case, $ToVisit$ becomes empty. As shown above, all nodes in T which are not leaves satisfy $q \wedge \neg p$. Additionally, let $n \in T$ be a leaf. We show that $n \models p$. As the successors of n are not added to $ToVisit$, the subgoal (n, p) is checked by HANDLEAV, which concludes that it holds. Due to the conformity assumption of this Lemma, we get that for each $n \in T$, $abs(n) \models q$, and $abs(n) \models p \iff$ p is a leaf.

We now show that g holds. Let $\pi = s_0, s_1, \ldots$ be a path from $C(n)$. We show that $\pi \models qU(p \wedge q)$. Let $\tau = (n_0, n_1, \ldots, n_k)$ be the single maximal trace in T (from initNode) s.t. for every $0 \leq i \leq k$, $C(n_i) = s_i$. As mentioned above, $n_i \models q$ for every $0 \leq i \leq k$. Moreover, $n_k \models p$ as n_k is a leaf (as τ is maximal) and $n_i \not\models p$ for every $0 \leq i < k$. We thus get that $\pi \models qU(p \wedge q)$. Thus, $C(n) \models AqU(p \wedge q)$, and in particular $n \models ApVq$, meaning g holds.

We now show that $abs(n)$ conforms w.r.t. φ. Let $s \in abs(n)$. It is sufficient to show that $s \models ApVq$. Let $\pi = s_0, s_1, \ldots$ be a path from s. We show that $\pi \models qU(p \wedge q)$.

According to Lemma 4.4.3, there exists a maximal trace $(n_0, n_1, \ldots, n_k)$ in T such that $s_i \in abs(n_i)$ for every $0 \leq i \leq k$. As shown above, $n_i \models q$ for every $0 \leq i \leq k$. Moreover, $n_k \models p$ as n_k is a leaf, and $n_i \not\models p$ for every $0 \leq i < k$. We thus get that $\pi \models qU(p \wedge q)$. Recall that, we assume that after checking all subgoals of the form (n, p) and (n, q), the astate $abs(n)$ conforms w.r.t. the formula checked. Thus, we get that $abs(n_i) \models q \wedge \neg p$ for $i < k$ and that $abs(n_k) \models q \wedge p$. As $s_i \in abs(n_i)$ for every $0 \leq i \leq k$, we get that $s_i \models q \wedge \neg p$ for $0 \leq i < k$ and that $s_k \models q \wedge p$. Thus, $\pi \models qU(q \wedge p)$ by definition. ■

We now prove termination of Algorithm 4.1 for finite structures.

Lemma 4.4.5. *(Termination of* HANDLEAV *for finite models)*
Let $g = (n, \varphi)$ be a goal, where $\varphi = ApVq$ and $p, q \in CTL$. Assume that for every goal of the form $g' = (m, \varphi')$, where m is a node and φ' is a proper subformula of φ, RECURCTL *checks g' in a finite number of steps. Then,* HANDLEAV *terminates in a finite number of steps as well.*

Proof. We show that HANDLEAV terminates for the goal (n, φ). We first show that INDAV is not called for more than $2 \cdot |S| + 1$ times, where $|S|$ is the number of states in M.

Consider a run of INDAV in which *False* is returned. By the structure of the algorithm, INDAV reaches line 10. Then, one of the following two options occurs, depending on whether the *if* statement in line 10 in INDAV is taken.

- If the *if* is taken, then there exist a cstate t s.t. $abs(t) \notin abs(T)$. This means that there does not exist an astate $\hat{s} \in abs(T)$ s.t. $t \in \hat{s}$. Then, a new astate is added to $abs(T)$ which contains t. Thus, every astate that is added in this case is not empty. As such, at most $|S|$ astates may be added to $abs(T)$, and so this *if* may be taken at most $|S|$ times.

- If the *if* is not taken, we first show that both splits of $\hat{s}$ are not empty. Consider the node n chosen in this iteration of INDAV. Then, as the *if* is not taken, it holds that $R(C(n)) \cap \hat{t} = \emptyset$. However, as the *if* statement in line 5 is not taken, then $(\hat{s}, \hat{t}) \in \hat{R}^{may}$, and so there exist $s \in \hat{s}, t \in \hat{t}$ such that $t \in R(s)$.

  Then, after the split in line 13, the cstates s and $C(n)$ belong to different splits, as s has a successor in $\hat{t}$ and $C(n)$ does not, and the split is done according to the property of having a successor in $\hat{t}$.

  Thus, the execution can reach that split at most $|S|$ times, as otherwise there would have been created more than $|S|$ non-empty astates, which is impossible.

We showed that there can be at most $2 \cdot |S|$ calls to INDAV in which *False* is returned. Note that if INDAV returns *True*, then HANDLEAV returns *True* as well. Recall that, INDAV and STRENGTHENSUBTREE are guaranteed to terminate due to Lemma 4.3.3 and to Lemma 4.4.3. Thus, all actions done in every iteration terminate, as of these Lemmas and as of the induction assumption over the calls to RECURCTL over proper subformulas of φ.

Now, consider an iteration of the loop in HANDLEAV in which neither *True* nor *False* are returned. Thus, INDAV is called in this iteration. As shown before, this may occur at most $2 \cdot |S| + 1$ times. Then, there may be at most $2 \cdot |S| + 1$ iterations of the loop in HANDLEAV.

Consider a run of HANDLEAV. The initialization of $ToVisit$ in line 1 obviously terminates. Then, if *True* or *False* are returned during the first $2 \cdot |S|$ iteration of the loop in line 2, then the algorithm terminates. Otherwise, more than $2 \cdot |S|$ iterations took place, which means that INDAV returned *False* $2 \cdot |S|$ times.

Consider the next iteration of the loop. If *True* or *False* are returned before INDAV is called, then the algorithm terminates. Otherwise, it necessarily returns *True* as shown above. Then, HANDLEAV returns *True* as well, and so it terminates.

Note that, if *True* is not returned from inside the loop, then the execution eventually proceeds to line 15, as shown above. There, STRENGTHENSUBTREE executed in a finite number of steps (as shown in Lemma 4.4.3), and then HANDLEAV returns *True*. Thus, in any case, the run of HANDLEAV terminates in a finite number of steps. ■

Chapter 5

Handling EpVq

We provide intuition by handing formulas of the form EGp, and then extend it to formulas of the form $EpVq$.

5.1 Intuition to EG

Let $\varphi = EGp$. Assume $p \in AP$ (as explained in chapter 4, this assumption does not change the generality of the explanation).

Similarly to the case of AGp in chapter 4, we explore the states reachable from a given state s in order to determine whether there exists an infinite path $\pi = s_0, s_1, \ldots$ from s s.t. for every $i \geq 0$ it holds that $s_i \models p$. We use the unwinding tree T differently. If a node n s.t. $n \not\models p$ is found, further unwinding it is not needed, as no infinite path that passes through n may satisfy Gp. Thus, we add the successors of n to T if and only if it satisfies p. If the successors of n are not added to T, we say that it is *pruned*.

Consider a run where there exists a trace τ s.t. every $s \in C(\tau)$ satisfies p, and there exists a cstate that appears twice in τ. This implies means that there is a lasso-shaped path from $C(root(T))$, which extends τ and satisfies Gp. Therefore, τ proves that $s \models EGp$. If there are no more nodes to examine, it means that all leaves in T were pruned. That means that $root(T) \models AF(\neg p)$. In other words, it means that $root(T) \not\models EGp$.

As in the case of AGp, abstraction is used to infer that $s \models EGp$ without exploring the entire model. We define a new kind of invariant, requiring that for each cstate s in it, there *exists* a successor within the invariant. This guarantees that there exists an infinite path π from s, which fully resides within the invariant. If all cstates in the invariant satisfy q, then $\pi \models Gq$. We formalize this notion with the following definition.

Definition 5.1.1. (Inductive invariant for EGq)
Let $q \in CTL$. A set of astates D is an *inductive invariant for EGq* if the following conditions hold:

- for every $\hat{s} \in D$ it holds that $(\hat{s}, D) \in \hat{R}^{must}$.

- for every $\hat{s} \in D$, $\hat{s} \models q$.

Observation 5.1.2. If D is an inductive invariant for EGq, then for every $s \in \bigcup D$, it holds that $s \models q$.

This holds for the same reason Observation 4.1.2 holds. Since every cstate $s \in \bigcup D$ is assigned an astate $\hat{s} \in D$ s.t. $\hat{s} \models q$, we get by Property 3.2.3 that $s \models q$.

Lemma 5.1.3. *Let q be a CTL formula. If D is an inductive invariant for EGq, then for every $\hat{s} \in D$, $\hat{s} \models EGq$. In addition, for every $s \in \bigcup D$, it holds that $s \models EGq$.*

Before proving Lemma 5.1.3, we prove the following lemma.

Lemma 5.1.4. *Let q be a CTL formula, D an inductive invariant for EGq and $s \in \bigcup D$. Let $k \geq 0$ and let $s_0, s_1, \dots, s_k$ be a consecutive sequence from s s.t. $s_j \in \bigcup D$ for every $0 \leq j \leq k$.*

Then, there exists a cstate s_{k+1} such that $(s_k, s_{k+1}) \in R$ and $s_{k+1} \in \bigcup D$. Additionally, the sequence $s_0, s_1, \dots, s_k, s_{k+1}$ is a consecutive sequence that satisfies $s_j \in \bigcup D$ for every $0 \leq j \leq k+1$.

Proof. Let $k \geq 0$ and let $s_0, s_1, \dots, s_k$ be a consecutive sequence such that $s_i \in \bigcup D$ for every $0 \leq i \leq k$.

Now, as $s_k \in \bigcup D$, there exists an astate $\hat{s}_k \in D$ s.t. $s_k \in \hat{s}_k$. By Definition 5.1.1, we get that $(\hat{s}_k, D) \in \hat{R}^{must}$. Now, due to the definition of must hyper-transitions, there exists a cstate $s_{k+1} \in \bigcup D$ s.t. $(s_k, s_{k+1}) \in R$.

We conclude that the sequence of cstates $s_0, s_1, \dots, s_k, s_{k+1}$ satisfies the required properties, as of the proof above. ■

We now prove Lemma 5.1.3

Proof. Let $s \in \bigcup D$. We define an *infinite* path $s_0, s_1, \dots$ from s. The path is defined recursively. We first define $s_0 = s$. Now, let $k \geq 0$, and assume that the s_i is defined for $0 \leq i \leq k$. Consider the consecutive sequence $s_0, s_1, \dots, s_{k-1}, s_k$. Let t be a cstate which is guaranteed to exist by Lemma 5.1.4. Then, we choose s_{k+1} to be t.

We prove that for every $i \geq 0$, it holds that $s_0, s_1, \dots, s_{i-1}$ is a consecutive sequence from s, and that s_i is well defined. For the base case $s_0 = s$ is well defined and satisfies the requirements trivially. For the induction hypothesis, assume that $s_0, s_1, \dots, s_{i-1}$ are well-defined cstates that form a consecutive sequence from s, and that $s_j \in \bigcup D$ for every $0 \leq j \leq i-1$. Now, as of Lemma 5.1.4, there exists a cstate s_i, such that $s_0, s_1, \dots, s_i$ is a consecutive sequence from s such that $s_j \in \bigcup D$ for every $0 \leq j \leq i$. Moreover, s_i is well defined.

As of the proof above, we get that π is a legal path from s, and that for every $j \in \mathbb{N}$, $s_j \in \bigcup D$. Then, by Observation 5.1.2, $s_j \models q$.

Thus, π is a legal path from s that satisfies Gq, which means that $s \models EGq$. By Definition 3.2.2, conclude that for every $\hat{s} \in D$, $\hat{s} \models EGq$. ■

5.2 Extension to EpVq

The generalization from EGp to $EpVq$ follows the same lines as the generalization from AGp to $ApVq$. That is, instead of looking for a path π from s along which p is satisfied, we wish π to satisfy Gq or $qU(p \wedge q)$. For that, HandleEV only unwinds nodes that satisfy q. HandleEV uses abstraction only to prove the existence of an infinite path that satisfies Gq, whereas finding a path that satisfies $qU(p \wedge q)$ is done using T. In the latter case, HandleEV returns *True* if it finds a trace $\tau = n_0, \ldots, n_k$ s.t. for every $0 \leq i \leq k, n_i \models q$ and $n_k \models p$. In this case, τ is strengthened to guarantee that the property also holds for the corresponding astates.

5.3 The full algorithm

5.3.1 HandleEV description

HandleEV is presented in Algorithm 5.1.

Algorithm 5.1 HandleEV

```
Input: Goal $g = (initNode, \varphi = EpVq)$
Output: True if $initNode \models \varphi$, False otherwise
 1: $ToVisit \leftarrow \{initNode\}$
 2: while $ToVisit \neq \emptyset$ do
 3:     choose nextNode from $ToVisit$
 4:     UpdateAbstract($nextNode$)
 5:     RecurCtl($nextNode$, $q$)
 6:     if $nextNode \not\models q$ then                    ▷ nextNode is not developed
 7:         continue
 8:     RecurCtl($nextNode$, $p$)                          ▷ $nextNode \models q$
 9:     if $nextNode \models p$ then
10:         StrengthenTrace(initNode, nextNode)
11:         return True
12:     else                                               ▷ $nextNode \models q \wedge \neg p$
13:         add successors of $nextNode$ to $ToVisit$
14:     if indEV(initNode, $nextNode$) then
15:         return True
16: StrengthenSubtree(initNode)
17: return False
```

The initialization, loop and choosing of nextNode of HandleEV are identical to those of Algorithm 4.1. Let τ be the trace from initNode to nextNode. If $ToVisit$ becomes empty, then all leaves in T are pruned. Then, no path that satisfies pVq can be found. Thus, we return *False* after applying subtree strengthening, to ensure that all astates in $abs(T)$ do not satisfy $EpVq$.

In line 4, we update the global data structures in the same manner described in Algorithm 4.1.

In line 6, if $nextNode \not\models q$, then HANDLEEV does not apply unwinding to that node, and continues to other nodes in $ToVisit$. The reason is that no path that satisfies pVq may pass through τ, as it satisfies $\neg pU\neg q$.

In line 8, if $nextNode \models p$ (recall that, as of the condition in line 5, $nextNode \models q$), any path π that extends τ satisfies $qU(p \wedge q)$, and in particular pVq. We label $initNode$ accordingly, strengthen τ and return *True*. The strengthening is done in order to guarantee that all astates in $abs(\tau)$ satisfy $EpVq$. Otherwise, $nextNode \models q \wedge \neg p$, and we further develop it.

In line 14, INDEV checks if τ is an abstract lasso and $abs(\tau)$ forms an inductive invariant for EGq. If so, HANDLEEV returns *True*, and otherwise it goes on to the next iteration of the loop.

5.3.2 indEV detailed description

The procedure INDEV is presented in Algorithm 5.2.

Algorithm 5.2 INDEV

```
Input: Trace $\tau = n_0, \ldots, n_k$
Output: True if $\tau$ is an abstract lasso, and $abs(\tau)$ forms an inductive invariant for
    $EGq$, False otherwise
 1: while $\exists i < k$ s.t $abs(n_i) = abs(n_k)$ do                    ▷ Abstract lasso
 2:     Let $\hat{D} = \{abs(n_i), abs(n_{i+1}), \ldots, abs(n_{k-1})\}$
 3:     $absToCheck \leftarrow \hat{D}$
 4:     while $absToCheck \neq \emptyset$ do
 5:         Choose $\hat{s}$ from $absToCheck$, and let $n \in T$ s.t. $abs(n) = \hat{s}$
 6:         isMust ← ISMUSTHYPERTRANSITION($\hat{s}$, $\hat{D}$)
 7:         if isMust is True then
 8:             Remove $\hat{s}$ from $absToCheck$
 9:         else
10:             SPLITEX($\hat{s}$, $\hat{D}$)
11:             break
12:     if $absToCheck = \emptyset$ then
13:         STRENGTHENTRACE($(n_0, n_1, \ldots, n_i)$)
14:         return True
15: return False
```

INDEV checks if $abs(\tau)$ forms an inductive invariant for EGq (which implies that $initNode \models EGq$, and so in particular $initNode \models EpVq$).

In order to avoid expensive testing, INDEV first checks whether a prerequisite for the existence of an inductive invariant holds. For that, the algorithm uses the concept of an *abstract lasso*. An abstract lasso is a trace $\tau = t_1, \ldots, t_n$ such that there exist $i < n$ s.t. $abs(t_i) = abs(t_n)$. Note that, if τ contains an inductive invariant for EGq, then extending the trace eventually yields an abstract lasso, as the invariant contains a finite number of astates.

In line 1, INDEV checks whether τ is an abstract lasso. Namely, it checks whether there exist $i < k$ s.t. $abs(t_i) = abs(t_k)$.

Thus, if τ is not an abstract lasso, INDEV returns *False*. Otherwise, let n_i be denoted the *base* of the lasso. Now, INDEV checks whether the set $absToCheck$ (initialized to $\hat{D}$, which is computed in line 2) forms an inductive invariant for EGq. It does so by iterating over $absToCheck$ (in line 4). In every iteration, it examines an astate $\hat{s} \in absToCheck$ and checks (in line 6) whether $(\hat{s}, \hat{D}) \in \hat{R}^{must}$ (like in INDAV, data in $absStructure$ is used and updated).

If $(\hat{s}, \hat{D}) \notin \hat{R}^{must}$ (line 9), then INDEV refines the abstraction by a split (described below) and then breaks the inner while loop. The split induces the need to recompute $abs(\tau)$, as an astate in $abs(\tau)$ is split. Now, INDEV returns to another iteration of the loop in line 1.

In the refinement process, $\hat{s}$ is split w.r.t. the property $P = \{s \in S \mid \exists t \in \bigcup \hat{D} : (s,t) \in R\}$. Thus, $\hat{s}$ is split into two new astates: $\hat{s}^Y$ and $\hat{s}^N$.

The changes of T and to $absStructure$ are similar to those described in Algorithm 4.1. The information gained by the split is also used to update $absStructure$: $(\hat{s}^Y, \hat{D}) \in \hat{R}^{must}$ and $(\hat{s}^N, \hat{s}) \in \hat{R}^{may}_{no}$ for every $\hat{s} \in \hat{D}$.

If $(\hat{s}, \hat{D}) \in \hat{R}^{must}$ for every $\hat{s} \in \hat{D}$, an inductive invariant for EGq is found. INDEV then strengthens the trace τ' from the base of the abstract lasso to $initNode$ (line 13), labels all astates in $abs(\tau)$ with $EpVq$ and returns *True*. As before, τ' is strengthened in order to guarantee that not only $initNode \models EGq$ but also $abs(initNode) \models EGq$.

We now prove a few Lemmas that are a part of the proof of soundness of our algorithm.

Lemma 5.3.1. *If* INDEV *returns False, then τ does not form an abstract lasso.*

Proof. Assume now that *False* is returned from INDEV. Then, it is returned in line 15. In this case, τ does not form an abstract lasso as the condition of the outer while loop is not met. ■

Lemma 5.3.2. *(Soundness of Algorithm 5.2)*
Let T be an unwinding tree for a model M and let $\tau = (n_0, n_1, \ldots, n_k)$ be a trace in T. Assume that for every $\hat{s} \in abs(\tau)$, it holds that $\hat{s} \models q$. Assume also that for every different $i, j < k$, $abs(n_i) \neq abs(n_j)$ before INDEV *is applied. Then, if Algorithm 5.2 returns True then τ is an abstract lasso, and $abs(\tau)$ forms an inductive invariant for EGq. Otherwise, if Algorithm 5.2 returns False then τ does not form an abstract lasso.*

Proof. The while loop in line 1 goes on as long as τ forms an abstract lasso. Recall that, for every different $i, j < k$ it holds that $abs(n_i) \neq abs(n_j)$ at the beginning of the run of this procedure. Then, as astates may only change due to split operations in this procedure, this inequality holds throughout the run. Thus, if an abstract lasso exists, then there exists a single index $0 \leq i < k$ such that $abs(n_i) = abs(n_k)$. Whenever an abstract lasso exists, we denote n_i the *base* of the lasso.

Consider the loop in line 4. In every iteration of that loop, if the *if* statement in line 7 is taken, $\hat{s}$ is removed from *absToCheck*, and otherwise the inner loop (in line 4) terminates. Thus, *True* is returned if and only if τ is an abstract lasso, and the *if* statement in line 7 is taken for every $\hat{s} \in absToCheck$, which means that *absToCheck* becomes empty after all of the iterations of the inner loop. On the other hand, *False* is returned if and only if τ does not form an abstract lasso.

Assume first that *True* is returned. Thus, τ is an abstract lasso at the point of termination. We prove now that $\hat{D}$ is an inductive invariant for *EGq*. Let $\hat{s} \in \hat{D}$. By the assumption, $\hat{s} \models q$. Moreover, $\hat{s}$ belongs to *absToCheck* at the beginning of Algorithm 5.2. In line 6, $\hat{s}$ is examined and it is proved that $(\hat{s}, \hat{D})$ is a must hyper-transition. This holds for every $\hat{s} \in \hat{D}$, and so $\hat{D}$ is an inductive invariant for *EGq*.

Note that $\hat{D} \subseteq abs(\tau)$. We now show that the activation of StrengthenTrace in line 13 guarantees that $abs(\tau)$ also forms an inductive invariant for *EGq*. First, note that for every j such that $0 \leq j < i$, it holds that $abs(n_j) \notin \hat{D}$, as mentioned above. Now, according to Lemma 4.4.1, for each $0 \leq j < i$, it holds that $(abs(n_j), \{abs(n_{j+1})\}) \in \hat{R}^{must}$ after j iterations of StrengthenTrace. As each astate only changes once in this case, this also holds after StrengthenTrace terminates. Now, for each $\hat{s} \in \hat{D}$, it still holds that $(\hat{s}, \hat{D})$ is a must hyper-transition, and thus so is $(\hat{s}, abs(\tau))$. For $\hat{s} \in abs(\tau) \setminus \hat{D}$, there exists $0 \leq i < k$ such that $abs(n_i) = \hat{s}$, and so $(\hat{s}, \{abs(n_{i+1})\})$ is a must hyper-transition. Then, $(\hat{s}, abs(\tau))$ is also a must hyper-transition. We get the $abs(\tau)$ is an abstract lasso that forms an inductive invariant for *EGq*.

Assume now that *False* is returned. Then, by Lemma 5.3.1, we get that τ does not form an abstract lasso. ∎

Lemma 5.3.3. *(Termination of Algorithm 5.2)*
If M is a finite structure and for every different $i, j < k$ it holds that $abs(n_i) \neq abs(n_j)$, then Algorithm 5.2 terminates after a finite number of steps.

Proof. If there does not exist $i < k$ such that $abs(n_i) = abs(n_k)$, then the algorithm terminates immediately. Otherwise, let i be such an index. Denote by $\hat{D}$ the set $\{abs(n_i), abs(n_{i+1}), \ldots, abs(n_k)\}$.

We first show that whenever the algorithm enter the outer while loop in line 1, it executes the code in that loop in a finite number of step.

indEV conducts at most $k - i + 1$ iterations of the inner while loop. The only action in each such iteration which does not obviously terminate is checking whether a pair $(\hat{s}, \hat{D})$ is a must hyper-transition. This check is done by retrieving data from *absStructure* and possibly checking the satisfiability of a quantified boolean formula, which is decidable. After the inner while loop, the rest of the code in the outer while loop is also conducted in a finite number of steps, as shown in Lemma 4.4.1.

We now show that the number of iterations in the outer while loop is finite. As shown before, for every different $i, j < k$, it holds that $abs(n_i) \neq abs(n_j)$ throughout the run. Then, n_i is the base of the abstract lasso in every iteration.

In each iteration of loop in line 1, if *True* is not returned then a single split operation is applied in line 10. As of the structure of the algorithm, the *if* in line 7 is not taken, and so it holds that $(\hat{s}, \hat{D})$ is not a must hyper-transition. Thus, there exists a cstate $s \in \hat{s}$ such that $R(s) \cap \bigcup D = \emptyset$. However, there exists an index j such that $i \leq j < k$ and $abs(n_j) = \hat{s}$. However, $(C(n_j), C(n_{j+1})) \in R$ and so $R(C(n_j)) \cap \bigcup \hat{D} \neq \emptyset$.

Thus, after the split, $\hat{s}$ is split to two non-empty astate, where each one is smaller than $\hat{s}$. Let $\hat{s}'$ be the split of $\hat{s}$ such that all cstates in $\hat{s}$ have a successor in D. However, note that there does not exist index j s.t. $i \leq j < k$ and $abs(n_j) = \hat{s} \setminus \hat{s}'$, as $(C(n_j), C(n_{j+1}) \in R$ and $abs(n_{j+1}) \in \hat{D}$.

Now, let $\hat{D}_{before}, \hat{D}_{after}$ be the astates in $\{abs(n_i), \ldots, abs(n_k)\}$ before and after the loop, respectively. Then, as a split operation is applied, it holds that $\bigcup \hat{D}_{after} \subseteq \bigcup \hat{D}_{before}$. As $s \in \bigcup D$ and $s \notin \hat{s}'$, we get that $s \in \bigcup \hat{D}_{before}$ and $s \notin \bigcup \hat{D}_{after}$. We conclude that $\bigcup \hat{D}_{after} \subsetneq \bigcup \hat{D}_{before}$.

We showed that in every iteration of the loop in line 1, the size of $\bigcup \hat{D}$ decreases. As each astate never becomes empty in these splits (as shown above), there cannot be more than $|\bigcup \hat{D}| - |\hat{D}|$ iterations of INDEV before it terminates.

Thus, the while loop in line 1 runs for a finite number of iteration. As each iteration terminates, we deduce that INDEV terminates as well. ■

Lemma 5.3.4. *(Soundness of* HANDLEEV*)*
Let $g = (n, \varphi)$ be a goal, where $\varphi = EpVq$. Assume that after HANDLEEV *checks a sub-goal (m, f), it holds that $abs(m)$ conforms w.r.t. f. Then, if* HANDLEEV *terminates, it returns True if and only if g holds. Moreover, if it terminates, $abs(n)$ conforms w.r.t. φ.*

Proof. The loop in line 2 is conducted for as long as $ToVisit \neq \emptyset$. The set $ToVisit$ is initialized to $\{initNode\}$. In every iteration, a node nextNode is chosen from it and examined. By the structure of HANDLEEV, once a node is examined by the algorithm, it is never examined by it again (in particular, it never enters $ToVisit$ again).

We observe that in every iteration of HANDLEEV, if $nextNode \not\models q$, then its successors are not added to $ToVisit$. This means that if a node is not a leaf in T, it satisfies q. Moreover, if a node is a leaf in T, and it is not examined in the last iteration before halting, it satisfies $\neg q$.

As the algorithm returns *True* if it examines a node that satisfies $p \wedge q$, we conclude that if a node is not a leaf, it does not satisfy p (as it satisfies q).

Assume first that *False* is returned by HANDLEEV. Let n be the last node that is examined in the loop in line 2 (at least one node is examined, as $ToVisit$ is initialized with $initNode$, which is examined first). It follows that $ToVisit$ becomes empty during the run, after which *False* is returned in line 17. As mentioned above, every node $n \in T$ which is not a leaf satisfies $n \models q \wedge \neg p$. Moreover, every leaf $n' \in T$ such that $n' \neq n$ satisfies $\neg q$.

We now show that $n \models \neg q$. Assume towards contradiction that $n \models q$. Thus, when HandleEV examines n, the *if* in line 6 is not taken. If $n \models p$, then *True* is returned, which is a contradiction to the assumption that *False* is returned. Then, $n \not\models p$, and so the successors of n are added to $ToVisit$. Note that, at least one node is added to $ToVisit$, as the transition relation is total. Next, one of the following holds:

- *True* it returned in line 15, which is a contradiction to the assumption that *False* is returned.
- The algorithm proceeds to line 2, and $ToVisit \neq \emptyset$ as the successors of n were added to it. This is a contradiction to the assumption that n is the last node to be examined.

Either way, we get a contradiction, and so we deduce that $n \not\models q$. Therefore, all leaves in T do not satisfy q.

StrengthenSubtree is applied in line 16. Let $s \in abs(root(T))$, and let π be a path from s. According to Lemma 4.4.3, there exists a maximal trace $\tau = (n_0, n_1, \ldots, n_k)$ in T s.t. $s_i \in abs(n_i)$ for every $0 \leq i \leq k$. For each such i, the subgoals (n_i, p) and (n_i, q) have been checked, and so by the assumption of this Lemma (Lemma 5.3.4), it holds that $abs(n_i) \models q \wedge \neg p$ for $i < k$ and $abs(n_k) \models \neg q$. Thus, $s_i \models q \wedge \neg p$ for $i < k$ and $s_k \models \neg q$, according to Definition 3.2.2. We deduce that $\pi \models \neg pU\neg q$, and so in particular $s \models A\neg pU\neg q$, which implies that $s \not\models EpVq$.

We get that $abs(root(T)) \not\models EpVq$, and in particular $n \not\models EpVq$. Thus, the Lemma holds in this case.

Assume now that *True* is returned, and let $nextNode$ be the node examined in the last iteration. Let $\tau = (n_0, n_1, \ldots, n_k)$ be the trace from $initNode$ (that is, $n_0 = initNode$) to $nextNode$ (that is, $n_k = nextNode$). Due to the structure of HandleEV, *True* may be returned either in line 11 or in line 15. We show that in both cases g holds and $abs(n)$ conforms w.r.t. φ after HandleEV terminates.

- Assume *True* is returned in line 15. In this case, the function indEV is called and returns *True*.

  We show that the prerequisites of Lemma 5.3.2 hold.

  - First, we show that every node $n \in \tau$ satisfies q. As proven before, every node $n \in \tau$ s.t. $n \neq nextNode$ satisfies q as it is not a leaf. Moreover, $nextNode \models q$ as otherwise, indEV is not called when $nextNode$ is examined. Due to the conformity assumption of this Lemma, we get that all astates in τ conform w.r.t. q, and in particular they satisfy q as the nodes in τ satisfy q.
  - We now show that for every $i \neq j$ s.t. $i, j < k$, it holds that $abs(n_i) \neq abs(n_j)$. Let i be an index such that $0 \leq i < k$. Then, n_i is examined by HandleEV before $nextNode$. As n_i is not a leaf in T and is not the last node to be examined, we get that the *if* statements in lines 6 and 9 are not

taken. Then, INDEV is applied over the arguments $initNode, n_i$ and *False* is returned (otherwise, HANDLEEV would have returned *True* then, which contradict our assumption). This is true for every $0 \leq i < k$

We prove by induction over m that for every different $i, j < m$, it holds that $abs(n_i) \neq abs(n_j)$. For $m = 1, 2$, the claim holds vacuously. Assume the claim holds for $m - 1$, and we prove it for k. According to the induction hypothesis, for every different $i, j < m - 1$, it holds that $abs(n_i) \neq abs(n_j)$. Now, consider the activation of INDEV over the two arguments $initNode, n_m$. As *False* is returned by that function call, we get by Lemma 5.3.1 that the trace $(n_0, n_1, \ldots, n_m)$ does not form an abstract lasso. Thus, $abs(n_k) \neq abs(n_j)$ for every $0 \leq j < k$. Together with the induction hypothesis, this proves the inductive claim.

We showed that the prerequisites of Lemma 5.3.2 hold. Thus, we get that $abs(\tau)$ forms an inductive invariant for EGq. By Lemma 5.1.3, we get that all astates $\hat{s} \in abs(\tau)$ satisfy EGq, and in particular they conform w.r.t. φ. Moreover, as $n \in abs(n)$, $abs(n) \in abs(\tau)$, we get that g holds. Thus, the Lemma holds in this case.

- Assume *True* is returned in line 11. Thus, there exist an iteration in which $nextNode$ is examined, in which *True* is returned. It holds that $nextNode \models q \wedge p$ and that for every $i < k$, $n_i \models q \wedge \neg p$. Due to the conformity assumption of this Lemma, it holds that $abs(n_k) \models p \wedge q$ and that for every $i < k$, $abs(n_i) \models q \wedge \neg p$.

  Recall that before *True* is returned, STRENGTHENTRACE is applied over τ in line 10. Let $\hat{s}_j$ be $abs(n_j)$ before the activation of STRENGTHENTRACE, for $0 \leq j \leq k$. Let $s_0 \in abs(n_0)$. According to Lemma 4.4.1, there exists a series of states $s_1, s_2, \ldots, s_k$ such that $(s_i, s_{i+1}) \in R$ for every $0 \leq i < k$, and $s_j \in abs(n_j)$ for every $0 \leq j \leq k$. As for every j it holds that $abs(n_j) \subseteq \hat{s}_j$, we get that $s_j \models q \wedge \neg p$ for $j < k$ and that $s_k \models q \wedge p$. Thus, $s \models EqU(q \wedge p)$, and in particular $s \models EpVq$.

  This is true for every $s \in abs(n_0)$, and so $abs(root(T)) \models EpVq$. In particular, we get that g holds. Thus, the Lemma holds in this case. ∎

Lemma 5.3.5. *(Termination of* HANDLEEV *for finite models)*
Assume M is a finite Kripke structure. Let $g = (n, \varphi)$ be a goal, where $\varphi = EpVq$ and $p, q \in CTL$. Assume that for every goal of the form $g' = (m, \varphi')$, where m is a node and φ' is a proper subformula of φ, RECURCTL *checks g' in a finite number of steps. Then,* HANDLEEV *terminates in a finite number of steps as well.*

Proof. We first show that the number of activations of INDEV over traces τ which form an abstract lasso, is at most $|S| + 1$ times. Consider a run of INDEV in which *False* is returned and in which τ is an abstract lasso at the beginning of the run. Then, indEV reaches line 10 at least once (otherwise, *True* is returned). In that line, SplitEX is applied over $\hat{s}$ w.r.t. $\{abs(n_i), abs(n_{i+1}), \ldots, abs(n_k)\}$.

We show that both splits of $\hat{s}$ are not empty. Consider the node n chosen in this iteration of indEV. Then, as the *if* in line 7 is not taken, it holds that there exists $s \in \hat{s}$ such that $R(s) \cap \bigcup_{j=i}^{k} abs(n_j) = \emptyset$. However, there exists $i \leq j \leq k$ such that $n = n_j$. Now, n has a successor in $\bigcup_{j=i}^{k} abs(n_j)$, as if $j < k$, then $C(n)$ has a successors in $abs(n_{j+1})$ and if $j = k$ then $n_i \in abs(n_i) = abs(n_k)$, and $C(n)$ has a successor in $abs(n_{i+1})$. Either way, $R(C(n)) \cap \bigcup_{j=i}^{k} abs(n_j) \neq \emptyset$.

Then, after the split in line 10, the cstates s and $C(n)$ belong to different splits, as $C(n)$ has a successor in $\bigcup_{j=i}^{k} abs(n_j)$ and s does not, and the split is done according to the property of having a successor in $\bigcup_{j=i}^{k} abs(n_j)$.

Thus, the execution can reach that split at most $|S|$ times, as otherwise there would have been created more than $|S|$ non-empty astates, which is impossible. Then, there can be at most $|S| + 1$ calls to indEV in which τ is an abstract lasso at the beginning.

Let D be the maximal branching degree of M (which is finite by assumption). Recall that, any node is examined by HandleEV at most once. Thus, every trace $\tau = (n_0, n_1, \ldots, n_k)$ in T is also examined at most once, as it is examined only when n_k is examined by HandleEV. Now, for any depth k, there is a finite number of traces of length at most k, which is $d_k = \sum_{i=0}^{k} D^i$. Thus, for every $k \in \mathbb{N}$, after at most $d_k + 1$ iterations of the loop in HandleEV, all node that are examined are of depth at least k.

For every trace $\tau = (n_0, n_1, \ldots, n_k)$ whose length is more than $|S|$, there exists $i \neq j$ such that $i, j \leq |S|$ and $C(n_i) = C(n_j)$, and in particular $abs(n_i) = abs(n_j)$.

We show that the loop in HandleEV terminates after a finite number of iterations. Consider the first $d_{|S|+1}$ iterations of the loop. If the algorithm terminates after a smaller number of iteration, then we are done. Otherwise, assume that more than $d_{|S|+1}$ take place. Note that after that many iterations, every trace τ forms an abstract lasso as shown above.

Now, consider the i^{th} iteration for $i > d_{|S|+1}$. Assume that there are n nodes in $ToVisit$ at that point. Then, if *True* or *False* are returned from the loop, the algorithm terminates. Otherwise, either indEV is called or the *if* in line 6 is taken. In the latter case, in the next iteration, the size of $ToVisit$ becomes $n - 1$. Then, after at most n iteration either the loop terminates or indEV is called. Thus, after a finite number of iterations, either the loop terminates or indEV is called at a time where τ forms an abstract lasso.

Then, as indEV can be applied that way at most $|S| + 1$ times, we get that the loop in HandleEV always halts after a finite number of iterations.

Note that all actions done in every iteration terminate, as of the induction assumption over the calls to RecurCtl over proper subformulas of φ and as of Lemma 4.4.1 and Lemma 5.3.3 (whose conditions are met, as shown in the proof of Lemma 5.3.4).

Thus, every iteration is conducted in a finite number of steps.

Consider a run of HandleEV. The initialization of $ToVisit$ in line 1 obviously terminates. Then, if *True* or *False* are returned during some iteration of the loop in line 2, then the algorithm terminates. Otherwise, as the loop terminates after a finite number of iterations, and each iteration terminates in a finite number of steps, the execution continues in line 16. Then, as StrengthenSubtree is executed in a finite number of steps (as shown in Lemma 4.4.3, HandleEV returns *True*. Thus, in any case, the run of HandleEV terminates in a finite number of steps. ■

Chapter 6

Handling EX and logical operators

6.1 Handling logical operators

The handling of formulas of the form $\varphi_1 \circ \varphi_2$ where $\circ \in \{\wedge, \vee\}$ is as expected. For example, given the goal $(n, \varphi_1 \wedge \varphi_2)$, we recur over the subgoals (n, φ_1) and (n, φ_2). We return *True* if both subgoals hold, and *False* otherwise. Note that, if the first subgoal checked does not hold, we do not check the second subgoal, and simply return *False*. Handling negation is done simply as well. Let g be a goal of the form $(n, \neg\varphi)$. Then, (n, φ) is checked. If g holds we return *False* and vice versa. In all cases, it holds that $abs(n)$ conforms w.r.t. φ after termination.

6.2 Handling EX

The recursive handling of formulas of the form EXp is the following.

Algorithm 6.1 HandleEX

```
Input: Goal $g = (initNode, \varphi = EXp)$
Output: True if $initNode \models \varphi$, False otherwise
 1: Let $ToVisit$ be the set of successors of $initNode$
 2: while $ToVisit \neq \emptyset$ do
 3:     choose nextNode from $ToVisit$
 4:     RecurCtl($nextNode$, $p$)
 5:     if $nextNode \models p$ then
 6:         Let $\tau$ be the trace $(initNode, nextNode)$
 7:         StrengthenTrace($\tau$)
 8:         return True
 9:     else
10:         Remove $nextNode$ from $ToVisit$
11: StrengthenSubtree(initNode)                    ▷ $ToVisit$ is empty
12: return False
```

To handle formulas of the form EXp, HandleEX computes (in line 1) the set of successors of $initNode$ and iterates over it. For each successor n' of $initNode$, HandleEX recurs on the subgoal (n', p) (in line 4). If it finds a successor n' for which this subgoal holds, it strengthens the trace $(initNode, n')$ (line 7), labels the astate $abs(initNode)$ with φ and returns *True*. Otherwise, $initNode \not\models EXp$. HandleEX then splits $abs(initNode)$ to $\hat{s}', \hat{s}''$ in such a way that $\hat{s}' \models \neg EXp$, and defines $abs(initNode) = \hat{s}'$. For that purpose, we use StrengthenSubtree (in line 11). The subtree we strengthen is rooted at $initNode$, and contains this node and its successors only. As of Lemma 4.4.3, Property 3.2.3 holds after HandleEX terminates.

We now prove lemmas that are a part of the correctness proof of OMG.

Lemma 6.2.1. *(Soundness of Algorithm 6.1)*
Let $g = (n, \varphi)$ be a goal, where $\varphi = EXq$ and $q \in CTL$. Assume that after HandleEX *checks a subgoal (m, f), it holds that $abs(m)$ conforms w.r.t. f after checking that subgoal. Then, if* HandleEX *terminates, it returns True if and only if g holds. Moreover, if it terminates then $abs(n)$ conforms w.r.t. φ after termination.*

Proof. In every iteration of the loop in line 2, a successor node n' of n is chosen and examined. HandleEX recurs over the subgoal (n', p), and returns *True* if and only if there exists a node $nextNode \in ToVisit$ s.t. $nextNode \models p$.

Assume first HandleEX returns *True*. Thus, there exists a node $nextNode$ s.t. $(C(initNode), C(nextNode)) \in R$ and $nextNode \models p$. Thus, $initNode \models EXp$. By the conditions of the Lemma, it follows that after the recurring over the subgoal $(nextNode, p)$, it holds that $abs(nextNode) \models p$. As $nextNode \models p$, HandleEX enters the *if* condition in line 5. The algorithm strengthens the trace τ from $initNode$ to nextNode. We now shows that $abs(initNode) \models EXp$. Let $s \in abs(initNode)$. As StrengthenTrace is applied over the trace $\tau = (initNode, nextNode)$ of length 2, Lemma 4.4.1 guarantees that there exists a cstate t such that $(s, t) \in R$ and $t \in abs(nextNode)$. Recall that $abs(nextNode) \models p$. As $t \in abs(nextNode)$, it follows that $t \models p$. It then follows that $s \models EXp$. We get that $abs(initNode) \models EXp$.

Assume now that HandleEX returns *False*. Let $n_1, \ldots, n_k$ be the successors of initNode. Thus, for every $1 \leq i \leq k$, $n_i \not\models p$. Thus, $initNode \models AX\neg p \equiv \neg EXp$.

By the conditions of the Lemma, it follows that after recurring over the subgoals $(n_1, p), (n_2, p), \ldots, (n_k, p)$, it holds that $abs(n_i) \not\models p$ for every $1 \leq i \leq k$. HandleEX reaches line 11 and strengthens the subtree of initNode. In this case, T contains the node initNode and its successors, $n_1, \ldots, n_k$. We now prove that $abs(initNode) \models AX\neg p$. Let $s \in abs(initNode)$. Let $\pi = s_0, s_1, \ldots$ be a path from s. According to Lemma 4.4.3, there exists a maximal trace $\tau = (n_0, n_1, \ldots, n_k)$ s.t. $s_i \in abs(n_i)$ for every $0 \leq i \leq k$. Due to the structure of T, it follows that $k = 1$, and that there exists $1 \leq j \leq k$ s.t. $s_1 \in abs(n_j)$. As shown above, $abs(n_j) \not\models p$ and so $s_1 \models \neg p$. Thus, $\pi \models X\neg p$ and $s \models AX\neg p$. ∎

Lemma 6.2.2. *(Termination of* HandleEX *for finite models)*
Let $g = (n, \varphi)$ be a goal, where $\varphi = EXp$ and $q \in CTL$. Assume that for every goal of the form $g' = (m, \varphi')$, where m is a node and φ' is a proper subformula of φ, RecurCtl *checks g' in a finite number of steps. Then,* HandleEV *terminates in a finite number of steps as well.*

Proof. The initialization of $ToVisit$ in line 1 terminates. Let k be the number of successors of $C(initNode)$. Now, the loop in line 2 has at most k iterations, as in every iteration a successor of $C(initNode)$ is examined.

Note that all actions done in every iteration terminate, as of the inductive assumption over the calls to RecurCtl over proper subformulas of φ and as of Lemma 4.4.1.

Then, either *True* is returned from inside the loop, or the execution continues to line 11. This line terminates due to Lemma 4.4.3 and then HandleEX returns *False* and terminates.

Thus, in any case, the run of HandleEX terminates in a finite number of steps. ■

Chapter 7

Symbolic formalization and optimizations

State-of-the-art model checking algorithms use a symbolic representation of the model and the state-space traversal. We note that the description in previous sections may lead to state enumeration. In this section we suggest the required adaptations for implementing OMG symbolically. In particular, we explain how the development of the unwinding tree and the handling of abstract states should be changed. We refer to the adapted algorithm as *Symbolic OMG* (SOMG).

Let M be a model, defined over a set of variables $\overline{v}$ and input variables $\overline{i}$. A *literal* is a variable or a negation of a variable. A *cube* is a conjunction of variables. For $p \in AP$, let $p(\overline{v})$ represent the set of states satisfying p. Let the formulas $Init(\overline{v})$ and $R(\overline{v}, \overline{i}, \overline{v}')$ be formulas representing the initial states and transitions of M, respectively. A cstate $s \in S$ is a valuation of $\overline{v}$. A cstate s can be represented by a cube $s(\overline{v})$ in a standard manner. Given a cstate s and input values $\overline{i}_0$, there is exactly one valuation $\overline{t}$ s.t. the formula $R(s, \overline{i}_0, \overline{t})$ evaluates to *True*. In other words, each valuation to $\overline{i}$ defines a single transition from s. This property is important to conduct splits efficiently.

We treat a set of cstates D and the formula $D(\overline{v})$ representing it, interchangeably. Given a set of cstates D, we define its *image* to be $\textsc{Img}[D](\overline{v}) = (\exists \overline{v}, \overline{i}\ [D(\overline{v}) \wedge R(\overline{v}, \overline{i}, \overline{v}')])[\overline{v}' \leftarrow \overline{v}]$ and its *pre-image* to be $\textsc{PreImg}[D](\overline{v}) = \exists \overline{i}, \overline{v}'\ [R(\overline{v}, \overline{i}, \overline{v}') \wedge D(\overline{v}')]$. The former represents the set of successors of the cstates in D and the latter, their set of predecessors.

7.1 Symbolic representation of abstract states

We describe how to compute a symbolic representation for an astate $\hat{s}$. If $\hat{s}$ is of the form $[s]_0$ for $s \in S$, then it is represented by the formula $\hat{s}(\overline{v}) = \bigwedge_{p \in L(s)} p(\overline{v}) \wedge \bigwedge_{q \notin L(s)} \neg q(\overline{v})$. Otherwise, it is created due to a split of an astate $\hat{t}$ w.r.t. a property P, as explained below.

7.1.1 Abstract state representation using quantifiers

We first show how to represent splits of an astate using boolean formulas with quantifiers. For each of the two kinds of splits (*EX-split*, *AX-split*), we show how to represent the set of states that satisfy P using a formula $P(\overline{v})$ with quantifiers. Then, the splits of the astate $\hat{s}$ are $\hat{s}(\overline{v}) \wedge P(\overline{v})$ and $\hat{s} \wedge \neg P(\overline{v})$. We show the formulas for each kind of split.

- *EX-split*, where $P = \{s \in S \,|\, \exists t \in S[(s,t) \in R \wedge t \in \hat{t}]\}$ for $\hat{t} \in \hat{S}$. In that case, $P(\overline{v}) = \exists \overline{v}'[R(\overline{v}, \overline{v}') \wedge \hat{t}(\overline{v}')]$.
- *AX-split*, where $P = \{s \in S \,|\, \forall t \in S[(s,t) \in R \implies t \in \bigcup_{i=1}^{k} \hat{t}_i]\}$ for abstract states $\{\hat{t}_i\}_{i=1}^{k}$. In that case, $P(\overline{v}) = \forall \overline{v}'[R(\overline{v}, \overline{v}') \implies \bigvee_{i=1}^{k} \hat{t}_i(\overline{v}')]$.

Checking whether $s \in \hat{s}$ amounts to checking if $\hat{s}(\overline{v})[\overline{v} \leftarrow s]$ evaluates to true. Checking whether (D, B) is an inductive invariant for $ApVq$ in HandleAV amounts to checking whether $(\hat{s}, \hat{D})$ is a may-closure (see Definition 2.1.2), where $\hat{s}$ is an astate and $\hat{D}$ is a set of astates. It holds that $(\hat{s}, \hat{D})$ is a may-closure if and only if the formula $\hat{s}(\overline{v}) \wedge R(\overline{v}, \overline{i}, \overline{v}') \wedge \neg \bigvee_{\hat{t} \in D} \hat{t}(\overline{v}')$ is unsatisfiable.

Checking for an inductive invariant for EGq is done similarly, as it amounts to checking whether $(\hat{s}, \hat{D}) \in \hat{R}^{must}$. It holds that $(\hat{s}, \hat{D}) \in \hat{R}^{must}$ if and only if the formula $\forall \overline{i}, \overline{v}'[\hat{s}(\overline{v}) \wedge [R(\overline{v}, \overline{i}, \overline{v}') \to \neg \bigvee_{\hat{t} \in D} \hat{t}(\overline{v}')]]$ is unsatisfiable.

7.1.2 Quantifier-free representation of abstract states

To ease the computational difficulty that stems from the usage of quantifiers, we suggest new operations that split an astate $\hat{s}$. It is split to $\hat{s}^Y$ that contains only cstates satisfying P, and to $\hat{s} \setminus \hat{s}^Y$ that may still contain some cstates satisfying P, while containing all cstates that do not satisfy P. Note that this usage of split does not fully conform with Definition 2.1.1. We employ a method similar to [HBS12] to generalize a cstate or a set of cstates (represented by a cube) into an astate representation without quantifiers.

We first show how to conduct a variant of SplitEX, denoted *EX-NEG*, in which we generalize a cstate $s \in \hat{s}$. In this variants, we compute a split $\hat{s}^Y$ of $\hat{s}$, such that every cstate $s \in \hat{s}^Y$ does not have a transition to any state in some astate $\hat{t}$. Let s be a cstate that satisfies this property. That is, there does not exists a cstate $t \in \hat{s}$ s.t. $(s,t) \in R$. Then, the formula $\varphi(\overline{v}, \overline{i}, \overline{v}') = s(\overline{v}) \wedge R(\overline{v}, \overline{i}, \overline{v}') \wedge \hat{t}(\overline{v}')$ is unsatisfiable. Note that, s is a conjunction of literals $l_1, \ldots, l_k$. Using an unsat core of φ, we find a subset of the literals, $l_{i_1}, \ldots, l_{i_m}$, s.t. $(\bigwedge_{j=1}^{m} l_{i_j}) \wedge R(\overline{v}, \overline{i}, \overline{v}') \wedge \hat{t}(\overline{v}')$ is unsatisfiable. Then, the representation of $\hat{s}^Y$ in the split is $\bigwedge_{j=1}^{m} l_{i_j} \wedge \hat{s}(\overline{v})$.

The second variant of SplitEX, denoted *EX-POS*, is one in which we generalize a cstate $s \in \hat{s}$ such s *has* a transition to a cstate $t \in \hat{t}$. As s has a successor $t \in \hat{t}$, there exists an input i_0 s.t. $(s, i_0, t) \in R$. We compute i_0 by computing a satisfying assignment to the formula $s(\overline{v}) \wedge R(\overline{v}, \overline{i}, \overline{v}') \wedge \hat{t}(\overline{v}')$. Then, similarly to *EX-NEG*, we extract a subset $l_{i_1}, \ldots, l_{i_m}$ of the literals of s using an unsat core of $s(\overline{v}) \wedge i_0(\overline{i}) \wedge R(\overline{v}, \overline{i}, \overline{v}') \wedge \neg\hat{t}(\overline{v}')$, which is unsatisfiable. The representation of the split $\hat{s}^Y$ (which contains cstates that have a transition to a cstate in $\hat{t}$) is $\bigwedge_{j=1}^{m} l_{i_j} \wedge \hat{s}(\overline{v})$.

The corresponding variations of SplitAX are dual to the variations of SplitEX presented above. For example, let s be a cstate and $\hat{t}$ be an astate. Generalizing s into an astate, s.t. all cstates in it have a transition to $\hat{t}$ (*AX-POS*), is equivalent to generalizing s into an astate in which all cstates have no successor in $\neg\hat{t}(\overline{v})$ (*EX-NEG*).

Given these representations, checking whether there exists an inductive invariant for $ApVq$ (Definition 4.2.1) or for EGq is done in the same manner describe in the previous section.

7.2 Symbolic unwinding

Unlike OMG, where a node in the unwinding tree T corresponds to a single concrete state, in SOMG every node in T corresponds to a *set of concrete states*. We denote such a set as cSet. As a result, the concrete labeling C maps each node to a cSet $C(n)$. The function *abs* is modified s.t. it can be applied to a cSet. Given a cSet D, $abs(D) = \{abs(s) \mid s \in D\}$. Equivalently, $\hat{s} \in abs(D) \iff D(\overline{v}) \wedge \hat{s}(\overline{v}) \not\equiv \bot$. For a node $n \in T$, $abs(n)$ is naturally defined as $abs(C(n))$.

Unwinding a node n in T is performed using the image operator on $C(n)$. Yet, there are cases where SOMG conducts a more fine-grained traversal of the state-space, and only applies the image operator to a subset of $C(n)$. This is achieved by splitting a cSet, which resembles a split of astates. As an example, consider a run of HandleAV where a node $n \in T$ is examined, such that for every $s \in C(n)$, it holds that $s \models q$. Assume $C(n)$ consists both of cstates that satisfy p and cstates that do not. In this case, splitting $C(n)$ to $D_1 = \{s \in C(n) \mid s \models p\}$ and $D_2 = C(n) \setminus D_1$ allows SOMG to consider only cstates that are successors of D_2. Note that, cstates in D_1 do not need further processing. We formally define the notion of a cSet split:

Definition 7.2.1. (cSet split)
Given a cSet D, the cSets D_1 and D_2 are a *split* of D if they are a non-trivial partition of D. Namely, $D_1 \neq \emptyset$ and $D_2 \neq \emptyset$.

Let $m, n \in T$ s.t. n is a successor of m. When the cSet $C(n)$ is split, n is replaced by two fresh nodes, n_1 and n_2, s.t. the following holds:

- n_1 and n_2 are successors of m in T.
- $C(n_1)$ and $C(n_2)$ are a *split* of $C(n)$.

Note that splitting a cSet allows SOMG to partition the unwinding process. Namely, instead of computing all successors of $C(n)$, it computes the successors of $C(n_1)$ and of $C(n_2)$ separately.

We emphasize the different purposes of splitting a cSet and splitting an astate. Splitting an astate is done in order to create may or must transitions between different astates. For example, SPLITEX in HANDLEAV is used to refine the abstraction, thus allowing the algorithm to find an inductive invariant. In contrast, cSet splits only allow SOMG to unwind different parts of T separately, and is not related directly to the abstract model.

It is to be pointed out that all cstates in the reachable cSets in T are reachable. The symbolic unwinding only changes the manner in which the unwinding is conducted, and not the set of cstates that are represented in T.

7.3 Adapting the algorithm

The symbolic representation of the unwinding tree changes the way the state-space is traversed. As a result, the algorithm needs to be adapted and take this change into account. Let $g = (n, \varphi)$ be a goal. Since $C(n)$ now represents a cSet, the return value of the subprocedures in SOMG cannot be Boolean, as it may be the case that some cstates in $C(n)$ satisfy φ and some do not. To solve this, each subprocedure returns a *Model Checking Result.*

Definition 7.3.1. Let $g = (n, \varphi)$ be a goal. A *model checking result* (*MC result*) for g is a pair of cSets (D_y, D_n), s.t. $D_y = \{s \in C(n) \mid s \models \varphi\}$ and $D_n = C(n) \setminus D_y$. Note that D_y, D_n form a possibly-trivial partition of $C(n)$.

In what follows, we describe the main changes in the different subprocedures of OMG, w.r.t. a goal $g = (n, \varphi)$.

Initialization, RecurCtl and logical connectives The only change in the initialization is the value of the cSet $C(\mathit{root}(T))$, which is $\{s\}$ when checking $s \models \varphi$ and is S_0, when checking $M \models \varphi$.

Consider RECURCTL. SOMG only analyzes cstates that belong to an astate for which φ is not decided. Let $A_! = \{\hat{s} \in \mathit{abs}(n) \mid \hat{s} \text{ is decided for } \varphi\}$ and $A_? = \mathit{abs}(n) \setminus A_!$. Then, n is replaced by two fresh nodes $n_!, n_?$, s.t. $C(n_!) = \{s \in C(n) \mid \mathit{abs}(s) \in A_!\}$ and $C(n_?) = C(n) \setminus C(n_!)$. Unless $C(n_?) = \emptyset$, RECURCTL continues by checking the subgoal $g' = (n_?, \varphi)$. After g' is analyzed, the cstates in $C(n_!)$ are added to the MC result of g'.

The subprocedures that handle logical connectives unify the MC results of their subgoals. For example, let $\varphi = \varphi_1 \wedge \varphi_2$ and (D_1^y, D_1^n) and (D_2^y, D_2^n) be the MC results of (n, φ_1) and (n, φ_2), respectively. SOMG returns the MC result $(D_1^y \cap D_2^y, D_1^n \cup D_2^n)$.

HandleEX Assume $\varphi = EXf$. HANDLEEX extends T with m, the successor of n s.t. $C(m) = \text{IMG}[C(n)]$. Then, RECURCTL is called over the subgoal $g' = (m, f)$, returning the MC result (D'_y, D'_n). Next, HANDLEEX defines its MC result (D_y, D_n), where

$D_y = C(n) \cap \textsc{PreImg}[D'_y]$ and $D_n = C(n) \setminus D_y$. Before returning (D_y, D_n), HandleEX refines each astate in $abs(n)$ to ensure that Property 3.2.3 holds. Refinement is done in the following manner: let $\hat{E}_y = \{\hat{s} \in abs(m) \mid \hat{s} \models f\}$. For each $\hat{s} \in abs(D_y)$, $\hat{s}$ is split using *EX-POS* w.r.t. $\hat{E}_y$. For each $\hat{s} \in abs(D_n)$, $\hat{s}$ is split using *EX-NEG* w.r.t. $\hat{E}_y$. In both cases the cSet we generalize is $C(n) \wedge \hat{s}$.

HandleAV and HandleEV Consider HandleAV, where $\varphi = ApVq$. The main changes involve the recursive handling of p and q. Note that, as indAV conducts checks that only regard astates (and so, it is not related to the manner in which unwinding is applied), it does not change.

Let $g' = (nextNode, q)$ be the subgoal checked in line 6 of HandleAV, and let (D'_y, D'_n) be the MC result for g'. If $D'_n \neq \emptyset$, there exists a trace τ from $initNode$ to $nextNode$ and cstate $s \in C(nextNode)$ s.t. $s \not\models q$. As s is reachable from $initNode$, there exists $s_0 \in C(initNode)$ s.t. $s_0 \not\models ApVq$. Thus, there exists a cSet $D_n \subseteq C(initNode)$, where $D_n \not\models ApVq$. No further analysis of D_n is required, and $C(initNode)$ is split s.t. $initNode$ is replaced by two fresh nodes, whose cSets are $D_n, D_?$, where $D_? = C(initNode) \setminus D_n$. After the cSet is split, all its successors must be updated to maintain a valid unwinding tree. This is performed by applying a cSet split for every node in τ, starting from $nextNode$ backwards. After T is updated, HandleAV strengthens τ in order to guarantee that Property 3.2.3 holds.

Then, it moves on to analyze the subgoal $g'' = (nextNode, p)$ (line 10 in Algorithm 4.1). Analyzing g'' is similar to the process described for g'. HandleAV splits $C(nextNode)$ into two cSets D''_y, D''_n, w.r.t. p. Then, D''_n undergoes further unwinding. We do not describe this process fully.

The changes applied to HandleEV are similar to the changes to HandleAV. The full details are omitted.

7.4 General optimizations to OMG

When implementing both OMG and SOMG, several optimizations should applied to increase efficiency.

- *Avoiding trivial splits*: A split of an astate is said to be *trivial* if one of its splits $\hat{s}'$ represents no concrete state. Namely, $\forall s \in S : abs(s) \neq \hat{s}'$. If this holds, no split should be conducted, so that *absStructure* does not change. This optimization may prevent many unnecessary split in the abstract state space.

- *Brother unification*: Consider a run of indAV where $abs(T) = \{\hat{s}_1, \ldots, \hat{s}_k\}$. If $\hat{s}_1, \hat{s}_2$ are the splits of $\hat{t}$, then we can replace $abs(T)$ with the set $\{\hat{s}_3, \ldots, \hat{s}_k, \hat{t}\}$ when checking for an inductive invariant. The new set of astates is smaller and simpler, and so the checking for an inductive invariant is more efficient. This action of replacement is called *reduce*. A set of astates is *reduced* if no reduce action can be performed over it. IND AV thus computes a reduced set from $abs(T)$, with which it continues. This optimization can also be applied in IND EV.

- *Lazy splits*: If a node n is assigned an astate $\hat{s}$ which undergoes a split, the value $abs(n)$ can be recomputed by need.
- *Checking multiple properties over a model*: The same data structures may be used to check multiple properties over the same model. The information retained from the check of one specification speeds up the model checking process for the next one.
- *Concrete state trimming*: In Algorithm 4.1, it is redundant to explore (that is, keep unwinding) a node n if a node n' such that $C(n) = C(n')$ has already been explored. Thus, the algorithm maintains a set of the cstates that were explored to avoid exploring such nodes n. A similar optimization is applicable in HANDLEEV.

Chapter 8

Correctness of OMG

We now prove the correctness of OMG.

Theorem 8.1. *(Partial correctness of OMG)*
Let $M = (S, I, R, L)$ be a Kripke structure and φ be a CTL formula. Let $s \in S$ be a cstate. Then, OMG is sound. *That is, if OMG terminates, it returns True if and only if $M, s \models \varphi$.*

Proof. We prove that if OMG terminates, then its answer to the model checking task is correct. That is, if OMG returns *True*, then $M, s \models \varphi$, and if OMG returns *False*, then $M, s \not\models \varphi$.

Assume that OMG terminates. Let $g_1, g_2, \dots, g_k$ be the goals checked by RecurCtl during the run of OMG, ordered such that $i < j$ if and only if the activation of RecurCtl over g_i terminates before the activation of RecurCtl over g_j.

We first prove by induction over k that if RecurCtl returns *True* over $g_k = (n, \varphi)$ then g_k holds, and if *False* is returned then g_k does not hold. Additionally, $abs(n)$ conforms w.r.t. φ after RecurCtl terminates over g_k.

For the base of the induction we prove the claim for $k = 1$. Assume towards contradiction that $\varphi \notin AP \cup \{\bot, \top\}$. Then, φ is a complex formula, and so there exists a proper subformula φ' of φ. Further, RecurCtl is applied over a subgoal g' of the form (m, φ') for some node m. Then, RecurCtl finishes checking g' before finishing to check g, as of the structure of RecurCtl and the fact that OMG eventually terminates. However, this is impossible as $g \neq g'$ and g is the only goal checked during the run of OMG.

Thus, $\varphi \in AP \cup \{\bot, \top\}$. If φ is not an atomic proposition, the *if* statement in line 1 of RecurCtl is taken and the boolean value of φ is returned in line 2. Moreover, every astate conforms w.r.t. φ by definition. Thus, the base case of the Lemma holds in this case.

Otherwise, $\varphi \in AP$. In line 3 in RecurCtl, $abs(n)$ is computed. In particular, $abs(n)$ is labeled with the same atomic propositions as $C(n)$. Therefore, the *if* statement in line 4 (of RecurCtl) is taken, and *True* is returned $\iff \varphi \in L(C(n))$, which proves soundness. As mentioned above, astates conform w.r.t. atomic propositions by the definition of the abstract structure. Thus, the base case of the Lemma holds in this case as well.

Assume now that the claim holds for k, and let $g_1, g_2, \ldots, g_k, g_{k+1}$ be a series of subgoals as described above. Now, if $\varphi \in AP \cup \{\top, \bot\}$, then the claim holds as proved above. Otherwise, φ is a complex formula. We split to cases according to the main connective of φ:

- Assume φ is of the form $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$. Due to the structure of RecurCtl, we conclude that it finishes checking the subgoals (n, φ_1) and (n, φ_2) before finishing to check g. By the induction hypothesis, $abs(n)$ conforms w.r.t. φ_1, φ_2. Then, OMG returns the appropriate result according to the logical connective. Thus, g holds. Afterwards, $abs(n)$ is labeled (in RecurCtl) with φ or $\neg\varphi$ according to the result of the model checking, and in particular $abs(n)$ conforms w.r.t. φ. A similar argument holds if φ is of the form $\neg\varphi'$.

- Assume φ is of the form $A\varphi_1 V \varphi_2$. Let $g_{i_1}, g_{i_2}, \ldots, g_{i_l}$ be the subgoals over which HandleAV calls RecurCtl. It thus holds that RecurCtl finishes checking these subgoal before checking g_{k+1}. Thus, for every $1 \leq j \leq l$ it holds that $i_j < k+1$. Denote $g_{i_j} = (n_{i_j}, \varphi_{i_j})$ for $1 \leq j \leq l$. By the induction hypothesis, for each $1 \leq j \leq l$ it holds that $abs(n_{i_j})$ conforms w.r.t. φ_{i_j} after g_{i_j} is checked. Then, the conditions of Lemma 4.4.4 hold, and so we get that *True* is returned if g_{k+1} holds and *False* otherwise. Moreover, Lemma 4.4.4 guarantees that after termination, $abs(n)$ conforms w.r.t. φ.

- For the cases where φ is of the form $E\varphi_1 V \varphi_2$ or $EX\varphi_1$, the proof follows the lines of the previous case, apart for the usage of Lemma 5.3.4 and Lemma 6.2.1, respectively.

Now, let $M, s \models \varphi$ be the model checking task checked by OMG. As OMG activates RecurCtl over the goal $g = (root(T), \varphi)$, it follows from the inductive claim that OMG returns true if and only if g holds. Thus, OMG is sound. ■

Theorem 8.2. *(Termination of OMG for finite models)*
Let $M = (S, I, R, L)$ be a finite *Kripke structure and φ be a CTL formula. Let $s \in S$ be a cstate. Then, OMG terminates in a finite number of steps.*

Proof. We prove that OMG terminates in a finite number of steps for every finite model M, cstate s and a CTL formula φ.

We show that the following property holds: for every $\varphi \in CTL$, and for every node n, when RecurCtl is applied over $g = (n, \varphi)$, it terminates in a finite number of steps. We prove this by induction over the structure of φ.

For the base case, $\varphi \in AP \cup \{\top, \bot\}$. If φ is boolean, then the *if* statement in line 1 of RecurCtl is taken. Then, RecurCtl returns in line 2. Otherwise, $\varphi \in AP$, in which case the *if* statement in line 1 is not taken. As explained before, $abs(n)$ is labeled with the same atomic propositions as $C(n)$. Therefore, the *if* statement in line 4 is taken, and the algorithm halts in the next line.

Now, assume that for every proper subformula φ' of φ and that for every node m, if RecurCtl is applied over the subgoal (m, φ'), then it terminates in a final number of steps. We prove that RecurCtl terminates when applied over g.

If $\varphi \in AP \cup \{\top, \bot\}$, then the claim holds as proved in the base case. Otherwise, φ is a complex formula. We split to cases according to the main connective of φ:

- Assume φ is of the form $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$. By the induction hypothesis, RecurCtl checks the subgoals (n, φ_1) and (n, φ_2) in a finite number of steps. Then, OMG returns the appropriate result, and in particular, it halts. Thus, it halts for φ in a finite number of steps. A similar argument holds if φ is of the form $\neg\varphi'$.

- Assume φ is of the form $A\varphi_1 V \varphi_2$. According to the induction hypothesis, for every node m and for every proper subformula φ' of φ, it holds that RecurCtl checks the goal (m, φ') in a finite number of steps. Then, when it is applied over φ, the *if* statement in line 1 is not taken. Next, RecurCtl either terminates in line 5, or it calls HandleAV which terminates according to Lemma 4.4.5. We conclude that RecurCtl terminates.

- For the cases where φ is of the form $E\varphi_1 V \varphi_2$ or $EX\varphi_1$, the proof follows the lines of the previous case, apart for the usage of Lemma 5.3.5 and Lemma 6.2.2, respectively.

Now, let $M, s \models \varphi$ be the model checking task checked by OMG. As OMG activates RecurCtl over the goal $g = (root(T), \varphi)$, it follows from the inductive claim that OMG terminates. Thus, OMG terminates in a finite number of steps. ■

Chapter 9

Related Work

The algorithm suggested in [HBS12] (*IICTL*) is the closest to our work. It conducts *CTL* model checking in an incremental and inductive manner. For every subformula of the specification it computes an over- and an under-approximation of the set of states in which it holds, and progresses by refining them. This process goes on until the algorithm halts with an answer.

The refinements that are applied in [HBS12] are used to make the approximation more precise. The refinements use generalization techniques that are similar to the ones presented in [Bra11, BSHZ11]. The refinements in OMG use a similar generalization technique to that of [HBS12]. However, it is used to achieve a different goal. OMG applies refinements to conduct splits, that create either must hyper-transitions or may closures.

Inductive proofs in [HBS12] are composed of lemmas, which prove relations between different approximations. No abstraction is used. In contrast, in OMG inductiveness is defined for abstract states, and is composed of different types of transitions (and hyper-transitions) between abstract states.

For formulas of the form *EGp*, *IICTL* generalizes a “skeleton” of concrete states in order to prove the existence of a lasso. In contrast, OMG uses must hyper-transitions to prove the existence of an infinite path, without computing the concrete states that form it. Additionally, OMG gathers and saves information on-the-fly, which is not done in [HBS12].

On-the-fly and local model checking is applied in many contexts, for instance in [BCG95, BS92, SW89b, SW89a, CVWY90, BLW02]. All algorithms use the notions of goals and subgoals. OMG uses abstraction to speed up proofs, unlike the other works. For example, the algorithm presented in [SW89b] constructs a tableau on-the-fly and unwinds the structure explicitly, until it finds an answer to the goal that it checks.

Lazy abstraction is used in [McM06, VGS12, HJMS02], as well as in OMG, although it is defined and used differently. The work in [McM06] uses interpolants for model checking software. It creates an abstraction for the program and refines it by need. Both this algorithm and OMG define notions of closure, though it is conceptually different. Our work is more general, as it handles the entire CTL.

SAT-based algorithms for model checking branching-time properties are suggested in [PWZ02, JC18, OG07, Wan05]. They resemble OMG in its unwinding of the model and search for witnesses. However, these methods do not use abstraction to speed up convergence. Moreover, OMG uses inductive invariants to speed up termination. In contrast, these methods only terminate after sufficient unwinding of the model is conducted.

May and *must transitions* are used, e.g., in [SG07, GHJ01]. In [CGJ$^+$03], a series of abstract models with *may transitions* are used for $ACTL^*$ model checking, allowing to prove but not refute properties. In [SG07], these transitions are combined with *3-valued* semantics for model checking. Given an abstract model that contains *may* and *must* transitions, a game board is built to conduct model checking. The method in [GHJ01] utilizes such transitions to define approximations for a model, in order to model check the entire *μ-calculus*. OMG uses such transitions as parts of proofs, but computes them lazily. Moreover, it uses *must hyper-transitions* instead of *must transitions*, as also done in [SG04].

Abstraction is used for CTL and μ-calculus model checking in [PH97, LA99]. These methods compute a series of approximations of the model. The work in [LA99] employs a compositional approach. Moreover, it computes an under- and an over-approximation for subformulas of the original specifications. The work in [PH97] formalizes a notion of *conservative approximations*, which is suitable for checking both universal and existential properties. The method proposed is based on abstraction-refinement. In OMG, however, it is not guaranteed that the same abstraction is used in different parts of the model. We lazily apply refinement only when and where needed.

The work in [CGM21] combines abstraction and compositional verification in order to prove the existence of fair infinite paths in infinite-state systems. Both this work and OMG use abstraction to prove the existence of infinite paths. However, the work in [CGM21] entails a compositional approach and handles fairness, whereas our algorithm is on-the-fly.

Different approaches for CTL model checking are presented in [BPR16, INH96]. In [BPR16], the model checking problem is translated to a set of Horn clauses using proof rules. The method presented in [INH96] uses *BDDs* to traverse the model. Traditional CTL model checking algorithms that use *BDDs* are presented in [CGK$^+$18]. These approaches are not on-the-fly, do not use abstraction and cannot handle infinite models, as opposed to our work.

The method presented in [Pel94] combines partial order reductions with on-the-fly model checking. Both algorithms implement on-the-fly verification, however minimizing the search space is done differently. [Pel94] uses partial order reduction, which recognizes equivalent executions w.r.t. independent actions in the model, thus reducing the state space. OMG uses abstraction to compute some of the transitions in the abstract state space, which is substantially smaller than the original state space.

Chapter 10

Conclusion and future work

This work presents an on-the-fly algorithm for CTL model checking with abstraction. Our algorithm minimizes the explored parts of the model in order to achieve efficiency. OMG can be used for model checking finite models, as well as for model checking infinite models with a finite branching degree. As a future work, we intend to incorporate concepts from OMG into the *IC3* algorithm. Another promising direction is generalizing our algorithm to handle the alternation-free fragment of the powerful μ-calculus logic.

Bibliography

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for ctl*. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*, pages 388–397, 1995.

[BLW02] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free μ-calculus. In *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, pages 128–147, 2002.

[BPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BPR16] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Efficient CTL verification via horn constraints solving. In John P. Gallagher and Philipp Rümmer, editors, *Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016*, volume 219 of *EPTCS*, pages 1–14, 2016.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2011, Austin, TX, USA*, pages 70–87, January 2011.

[BS92] Julian C. Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theor. Comput. Sci.*, 96(1):157–174, 1992.

[BSHZ11] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 144–153, 2011.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGK$^+$18] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking – Second Edition.* MIT Press, 2018.

[CGM21] Alessandro Cimatti, Alberto Griggio, and Enrico Magnago. Proving the existence of fair paths in infinite-state systems. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 104–126. Springer, 2021.

[CVWY90] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, pages 233–242, 1990.

[GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2001.

[HBS12] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. Incremental, inductive CTL model checking. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 532–547, 2012.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70. ACM, 2002.

[INH96] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. CTL model checking based on forward state traversal. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 82–87. IEEE Computer Society / ACM, 1996.

[JC18] Chuan Jiang and Gianfranco Ciardo. Improving sat-based bounded model checking for existential ctl through path reuse. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 471–487. EasyChair, 2018.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[LA99] Jørn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In *18th International Conference on Computer Aided Verification (CAV)*, pages 123–136, Seattle, WA, USA, August 2006.

[OG07] Rotem Oshman and Orna Grumberg. A new approach to bounded model checking for branching time logics. In *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, pages 410–424, 2007.

[Pel94] Doron A. Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[PH97] Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 12–23. Springer, 1997.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[PWZ02] Wojciech Penczek, Bozena Wozna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of CTL. *Fundam. Inform.*, 51(1-2):135–156, 2002.

[SG04] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 546–560, 2004.

[SG07] Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Log.*, 9(1):1, 2007.

[SW89a] Colin Stirling and David Walker. CCS, liveness, and local model checking in the linear time mu-calculus. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, pages 166–178, 1989.

[SW89b] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89)*, pages 369–383, 1989.

[VGS12] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and sat-based reachability in hardware model checking. In Gianpiero Cabodi and Satnam Singh, editors, *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 173–181. IEEE, 2012.

[Wan05] Bow-Yaw Wang. Proving forall-μ-calculus properties with sat-based model checking. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, pages 113–127, 2005.

כעת, במקום לחפש שמורה אינדוקטיבית שמורכבת ממצבים קונקרטיים, נחפש שמורה שכזו שמורכבת מהמצבים האבסטקטיים שמשויכים למצבים הקונקרטיים בהם ביקרנו. שינוי זה מאיץ משמעותית את ההתכנסות לעצירה עם תשובה חיובית.

אם קבוצת המצבים האבסטקטיים לא מהווה שמורה אינדוקטיבית, נפרוש את המבנה הלאה, על-מנת לגלות מצבים אבסטרקטיים נוספים, או שלחילופין נעדן את האבסטקציה.

אנו מרחיבים את האלגוריתם המתואר לנוסחאות מהצורה $ApVq$, כאשר V הוא אופרטור בשם **release**, המקיים $pVq \equiv Gq \vee (qU(p \wedge q))$. האלגוריתם המורחב דומה לאלגוריתם המטפל בנוסחאות מהצורה AGp.

על-מנת להבין את OMG בצורה טובה יותר, נסביר את הטיפול בנוסחאות מהצורה EGq. ביצוע הבדיקה $M, s \models EGq$ מסתכם בחיפוש מסלול אינסופי מהמצב s , כאשר לכל מצב לאורך המסלול מתקיים q. בעת ניתוח המצבים הישיגים מהמצב s , אם OMG נתקל במצב שלא מקיים את התכונה q , הוא מפסיק לנתח את אותו המצב, היות שהוא לא יכול להיות חלק ממסלול שמספק Gq . אם מסלול אינסופי לא נמצא, ואזלו המצבים שיש לנתח, האלגוריתם מחזיר תשובה שלילית. על-מנת להחזיר תשובה חיובית, OMG מחפש שמורה אינדוקטיבית. השמורה האינדוקטיבית מבטיחה כי כלל המצבים בה, פרט לכך שיספקו q, יהיו בעלי בן בשמורה עצמה.

הטיפול בנוסחאות CTL כלליות הוא רקורסיבי. לדוגמה, על-מנת לבצע את הבדיקה $s_0 \models p \wedge AXAGq$, האלגוריתם יבדוק האם $s_0 \models p$ והאם כל אחד מהבנים של s_0 מספק את הנוסחא AGq. באופן הזה, בדיקת המודל של נוסחאות CTL כלליות נעשה על-ידי בדיקת תתי-נוסחאות מעל מצבים לפי הצורך בלבד.

אופן הטיפול הרקורסיבי מעלה את הצורך בתכונה הבאה. נניח שאנו בודקים האם מתקיים $s \models \varphi$, ויהי $\hat{s}$ המצב האבסטרקטי שמשויך למצב הקונקרטי s בתום אותה בדיקה. אזי, כל המצבים הקונקרטיים שמשויך להם אותו מצב אבסטרקטי $\hat{s}$, עליהם להסכים עם s באשר לסיפוק φ. הווי אומר, כל מצב t אשר המצב האבסטרקטי שמשויך לו הוא $\hat{s}$, עליו לקיים כי $s \models \varphi \iff t \models \varphi$. נקודה זו מעלה את הצורך ב"חיזוקים", אשר מבוצעים במקומות שונים לאורך ריצת האלגוריתם.

האלגוריתם אשר אנו מציעים מתאים לבדיקת מודלים בעלי דרגת יציאה סופית לכל מצב, כאשר מספר המצבים הוא סופי או אינסופי. עם זאת, יתכן שהאלגוריתם לא יעצור עבור מבנים שאינם סופיים.

בעבודה זו, מימשנו את האלגוריתם שלנו, ושילבנו בין ייצוג מפורש לבין ייצוג סימבולי של מבנים שונים בהם משתמש האלגוריתם. עבור הייצוג הסימבולי, עשינו שימוש בפותרי SAT ובפותרי SMT. הניסויים שלנו מראים שישנן דוגמאות עבורן האלגוריתם שלנו משיג תוצאות טובות יותר מאשר $IICTL$, שהוא אלגוריתם חדיש מבוסס-SAT לביצוע בדיקת מודל לנוסחאות CTL.

יתרה מכך, הגדרנו פורמליזציה של האלגוריתם שלנו, והצגנו הן גרסא מפורשת והן גרסא סימבולית של OMG.

האלגוריתם שאנו מציעים משלב בין שתי גישות מרכזיות לבדיקת מודל: אימות עצל ואבסטרקציה. מטרת שתי גישות אלו היא התמודדות עם בעיית התפוצצות המצבים, אשר מתארת את הקושי בבדיקת מודל במערכות מודרניות. לב העניין הוא הבא - היות שמספר המצבים במודל גדל מעריכית עם גידול מספר המשתנים שמגדירים את המודל, ייצוג המודל כולו בזכרון בצורה מפורשת הופך לא ישים. עקב כך, נדרשים ייצוגים אלטרנטיביים למודל, שגוררים אחריהם קושי חישובי גדול יותר בביצוע בדיקת המודל. נתאר בקצרה שתי גישות אלו.

אימות עצל הוא גישה בבדיקת מודל שגורסת כי יש לפתח אך ורק את החלקים הנדרשים במודל על-מנת לבצע את הבדיקה עליו. כפועל יוצא, אלגוריתמים רבים שנוקטים בגישה זו מפתחים את המודל באופן שמונחה על ידי המפרט הנבדק. יתרה מכך, אלגוריתמי אימות הנוקטים בגישה זו מבצעים במקרים רבים פירוק של הנוסחה לתתי-נוסחאות שמגדיר את אופן בדיקת המפרט כולו. כך, נבדקות אך ורק תתי-הנוסחאות הנחוצות על גבי המודל. לפיכך, אלגוריתמים שמבצעים אימות עצל עשויים לעתים להצליח לבצע בדיקת מודל על מודל אינסופי, תוך פיתוח חלק סופי של המודל.

אבסטרקציה הינה גישה כללית שפועלת בצורה הבאה. בהנתן מודל M, אלגוריתם שמבצע אבסטרקציה יבנה מודל חלופי M' מתוך המודל המקורי וקטן ממנו בגודלו. מבנה זה מכונה "מבנה אבסטרקטי". לאחר מכן, תתבצע בדיקת מודל על המודל האבסטרקטי. ישנן אבסטרקציות רבות שמאפשרות הכללה של תוצאת בדיקת המודל המצומצם אל המודל המקורי, ובכך מתבצעת בדיקת המודל שלו, תוך שהושקע כוח חישוב בבדיקת המודל האבסטרקטי, הקטן יותר, בלבד. ניתן לעתים לבצע אימות פורמלי למערכות אינסופיות באמצעות טכניקת האבסטרקציה. האימות נעשה על-ידי מידול המערכת האינסופית באמצעות מודל אבסטרקטי עם מספר סופי של מצבים, וביצוע בדיקת מודל על המודל האבסטרקטי.

כעת, נתאר בצורה אינטואיטיבית חלק מהאלגוריתם שלנו, וספציפית - את ההתמודדות עם ניתוח ישיגות. בלוגיקה CTL, נוסחאות הבודקות ישיגות הן מהצורה AGp. בהינתן מצב התחלתי s_0 במודל M, הבדיקה $M, s_0 \models AGp$ שקולה לבדיקה כי כל המצבים הישגים מ- s_0 ב- M מקיימים את התכונה p.

על-מנת לבדוק תכונה זו, האלגוריתם שלנו מבצע פרישה של עץ החישוב של המבנה מתוך המצב s_0 בצורה עצלה. לאורך הבדיקה, האלגוריתם מפתח את המצבים הישיגים מ- s_0, ובודק האם כולם מספקים את התכונה p. אם נמצא מצב שאינו מקיים תכונה זו, הרי שמצאנו דוגמה נגדית, והאלגוריתם מסיים עם תשובה שלילית. עם זאת, על-מנת לסיים עם תשובה חיובית, יש לבקר בכל המצבים הישיגים.

אף על פי כן, נבחין כי אם כבר ביקרנו בכל הבנים של מצב במבנה, המשך הפיתוח שלו אינו דרוש. אם תכונה זו מתקיימת לכל המצבים במבנה בהם ביקרנו, נוכל לעצור עם תשובה חיובית. במקרה זה, נאמר כי קבוצת המצבים בהם ביקרנו מהווה **שמורה אינדוקטיבית**.

עם זאת, קבוצת המצבים הישגים עלולה להיות גדולה מאד ואף אינסופית. לפיכך, נשלב אבסטרקציה באלגוריתם שלנו, על-מנת להאיץ את ההתכנסות של האלגוריתם שלנו (ובמקרה של מבנה אינסופי, להפוך את ההתכנסות הזו לאפשרית). לכן, נשייך לכל מצב s במבנה בו נבקר מצב אבסטרקטי $\hat{s}$, שמסומן באותן נוסחאות אטומיות כמו המצב s.

תקציר

בעבודה זו נציג אלגוריתם חדיש לביצוע בדיקת מודל לשפת המפרט CTL, בשם "בדיקת מודל עצלה באמצעות אבסטרקציה מונחית" (נכנה אותה OMG). OMG משלבת אלגוריתם עצל עם שימוש באבסטרקציה.

אחת השיטות הבולטות ביותר לביצוע אימות הפורמלי מכונה בדיקת מודל. ניתן לתאר את הבעיה בצורה הבאה: בהינתן מודל של מערכת M ומפרט φ, אלגוריתם לבדיקת מודל מכריע האם המודל מקיים את המפרט, ובמידה שהוא לא מקיים אותו, האלגוריתם מספק דוגמה נגדית לכך. המפרט עשוי להיות מוגדר הן על-ידי מתכנן המערכת, הן על-ידי המפתח שלה, והן על-ידי הלקוח שעתיד להשתמש בה.

חשיבות אלגוריתמים שמבצעים בדיקת מודל הולכת וגדלה בימינו. סיבה מרכזית לכך היא שמערכות תוכנה וחומרה מודרניות הולכות והופכות לחלק מרכזי יותר בחיי הפרט. יתרה מכך, מערכות רבות נדרשות להיות עמידות לתקלות ולשיבושים, ויש צורך לוודא שהן אכן כאלו. מכל אלו עולה הצורך באלגוריתמים חדישים שמסוגלים לבצע בדיקת מודל של מערכות גדולות במהירות ובצורה נכונה.

היות שהבעיה הכללית של אימות פורמלי איננה כריעה, אלגוריתמי בדיקת מודל מטפלים אך ורק במערכות ריאקטיביות בעלות מספר סופי של מצבים, ואך ורק במפרטים שניתן לבטא בלוגיקות מסוימות. המודל לרוב מתואר כמבנה קריפקה, שמהווה צורה כללית לייצוג מכונות מצבים. המפרט לבדיקה מבוטא כנוסחה בלוגיקת זמן. ישנן לוגיקות זמן רבות, דוגמת CTL, LTL, CTL^*, ו-$\mu - calculus$. הלוגיקות השונות נבדלות בכוח הביטוי שלהן ובסיבוכיות האלגוריתמים שמשמשים לבדיקת מודל עבורן.

הלוגיקה CTL היא לוגיקת זמן מתפצל. לוגיקה זו משמשת לכתיבת מפרטים שמתארים התנהגויות של מערכות תוכנה וחומרה כאחד. באמצעות לוגיקה זו, ניתן לתאר תכונות שאינן ניתנות לתיאור בלוגיקות זמן לינארי, דוגמת הלוגיקה LTL. דוגמה לתכונה שכזו היא תכונת האתחוליות, קרי, יכולתה של מערכת לחזור למצב תחילי, עקב תהליך אתחול או תהליך שיקום. בלוגיקה CTL, ניתן לבטא תכונה זו באמצעות הנוסחה $AGEFstart$, כאשר בלוגיקה LTL לא קיימת נוסחה ששקולה לנוסחה זו.

יתרון נוסף של לוגיקה זו בא לידי ביטוי בדימיון היחסי שלה לפרגמנט חסר החילופים של הלוגיקה העוצמתית $\mu - calculus$. כתוצאה מכך, אלגוריתמים רבים שמבצעים בדיקת מודל ל- CTL ניתנים להרחבה בקלות יחסית לפרגמנט זה.

נתאר כעת את העקרונות המרכזיים המוצגים בעבודה שלנו. תהליך האימות שמבוצע על ידי

המחקר בוצע בהנחייתה של פרופסור ארנה גרימברג, בפקולטה למדעי המחשב.

תודות

בראש ובראשונה, ברצוני להודות למנחה שלי, פרופסור ארנה גרימברג. הפגישות השבועיות היוו מקור בלתי-נדלה של השראה עבורי. במהלכן, למדתי מחקר מהו, ועד כמה הוא יכול להיות מהנה. העבודה המשותפת נתנה לי ההזדמנות לרכוש כלים וניסיון מחוקרת בעלת שיעור קומה ושם עולמי. ארנה - תודה לך על האמונה בי, על התמיכה הרבה ועל חברותך.

ברצוני להודות לד"ר יקיר ויזל על שיתוף פעולה פורה ומוצלח. הידע שרכשתי בזכותך ילווה אותי בהמשך דרכי, ונהנתי מאד בעבודתנו המשותפת. ברצוני להודות גם ליתר עמיתיי ועמיתותיי במחקר, אשר היוו בעבורי השראה ומקור אדיר לייעוץ ולהכוונה.

ברצוני להודות למשפחתי האהובה, על אמונתם ועל תמיכתם בי לכל אורך לימודיי. אני מוקיר את המוטיבציה ואת הרצון ללמוד שהחדירו בי עוד בבית-הספר היסודי, ומעריך את תמיכתם הן ברגעים השמחים והן בזמנים הקשים.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

בדיקת מודל עצלה
באמצעות אבסטרציה מונחית

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

גל שדה

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
אדר התשפ"א חיפה פברואר 2021

בדיקת מודל עצלה
באמצעות אבסטרציה מונחית

גל שדה