Friedrich-Alexander-Universität
Erlangen-Nürnberg

Chair for Computer Science 8
Theoretical Computer Science

An Implementation of Global Caching for the Alternation-Free Coalgebraic μ-Calculus

Master Thesis in Computer Science

Christoph Egger

Advisors:

Daniel Hausmann   Lutz Schröder

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Erlangen, June 20, 2016

Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 20. Juni 2016

Kurzzusammenfassung

Diese Abschlussarbeit beschreibt die Implementierung eines “global caching” Algorithmus für das alternierungs-freien Fragment des coalgebraischen μ-Kalkül, COOL. COOL kann die erfüllbarkeit von Formeln im alternierungsfreien Fragment des coalgebraischen μ-Kalkül, der den relationalen μ-Kalkül und **CTL** sowie **ATL** und mehrere andere Logiken enthält, feststellen. Der “global caching” Algorithmus ist der erste Tableau-Algorithmus, der sowohl optimal ist als auch in einem Durchlauf arbeitet. Er ist eine Erweiterung des Algorithmus für das flache Fragment des coalgebraischen μ-Kalküls.

Speziell im Fall von **CTL** ist “model checking” eine in der Praxis häufig gebrauchte Problemstellung und es gibt eine Vielzahl von Anwendungen für die Erfüllbarkeitsprüfung. Folglich gibt es hierfür bereits mehrere andere Implementierungen. COOL schneidet im Allgemeinen vergleichbar mit Konkurenzimplementierungen ab und arbeitet für bestimmte Formeltypen signifikant schneller.

Diese Arbeit fasst den “global caching” Algorithmus zusammen und präsentiert eine Beschreibung der Implementierung als Teil von COOL. Weiterhin wird die Geschwindigkeit von COOL mit alternativen Werkzeugen verglichen. Schließlich werden diverse Optimierungen beschrieben, die an der aktuellen Implementierung in COOL noch vorgenommen werden können.

Abstract

This thesis presents a first implementation of a global caching algorithm for the alternation-free fragment of the coalgebraic μ-calculus, COOL. COOL can decide satisfiability of formulae in the alternation-free coalgebraic μ-calculus which includes the relational, alternation-free μ-calculus, **CTL** as well as **ATL** and several other logics. The global caching algorithm is the first optimal single-pass tableaux algorithm for all the mentioned logics and builds upon the global caching algorithm for the flat coalgebraic μ-calculus.

Especially for **CTL**, model checking is regularly used in practice and there are several applications for reasoning. Consequently there already exist a number of different implementations. COOL in general performs comparable to other implementations and is significantly faster for certain kinds of formulae.

The thesis provides a summary of the global caching algorithm and a description of the implementation in COOL as well as a performance comparison between COOL and other state of the art reasoners. Finally it provides several suggestions for further optimization.

Contents

1 Introduction

Modal logic is a corner stone of description and verification of sequential and concurrent systems. Implicit and explicit fixpoint constructs play an essential role to reach the expressiveness needed to describe real world systems.

A modal logic commonly describes graph- or tree-like structures, called Kripke frames, where certain conditions are true at nodes in this graph and certain restrictions on the child nodes hold. One of the most fundamental of these modal logics is system **K**. In **K** one has the ability to express that certain conditions must hold in all child nodes or that there must exist a child where some condition is true.

It is a common requirement when specifying, for example, sequential circuits [BCLMD94] or computer programs to add restrictions of the form "This condition needs to hold from here for all descendants" or "From here a finite path needs to exist which finally reaches a node that satisfy this condition". **CTL** is an extension to system **K** that provides exactly these kinds of restrictions.

In this thesis we are considering an implementation of the alternation-free fragment of the coalgebraic μ-calculus. The μ-calculus contains **CTL** as a fragment but allows more general fixpoint constructs. The coalgebraic generalization allows to create one reasoner, like the one described, which can be easily extended to support not only relational modal logic bot also most of the common non-standard modal logics. COOL, for example, already supports **ATL** and the alternation-free fragment of Parikh's game logic.

CTL is widely used in practice for verification. Where, for example, requirements expressed in **CTL** are verified to hold on some sequential circuit represented by Kripke structures. This process is called model checking. Reasoning solves a more general problem. Given a set of requirements, expressed for example in **CTL**, a reasoner can decide whether it is possible to construct a circuit fully satisfying the requirements. Other applications also include verification of software systems. While **CTL** reasoning certainly has potential applications it is computationally expensive and rarely used in practice.

Recently, there has been increased academic interest in **CTL** reasoners resulting in several competing implementations. All existing tableau style reasoners however either implemented multi-pass algorithms, first transforming the problem into an exponentially sized intermediate representation, or have suboptimal runtime (TreeTab [AGW07] runs in 2ExpTime while **CTL** is known to be an ExpTime complete problem).

In this thesis we describe COOL, the first optimal, single-pass tableau reasoner that is able to handle **CTL**. The algorithm implemented in COOL [HSE16] actually supports the alternation-free fragment of the modal μ-calculus which includes **CTL** (over **K**), but also includes other logics like **ATL** (using coalition logic as basis) which is used to describe multi-agent systems and the *-free fragment of Parikh's game logic [Par83; Par85] where COOL is the first-known reasoner. This is possible by using the coalgebraic generalization of modal logic used by COOL which was extended by support of explicit fixpoints as part of this thesis.

Related work

There is work on reasoners for **CTL** [ZHD09; Mar05] as well as **ATL** [Dav13] and the full μ-calculus [FL10]. These reasoners will be discussed when performing comparisons between our solver, COOL, and the alternative implementations in chapter 6.

In [GTW11] a comparison of **CTL** provers was conducted. The paper also presented a wide variety of formulae designed to show the respective strength and weaknesses of the existing provers which were reused for our comparison where we added COOL to the picture.

Coalgebraic modal logic was first introduced by Dirk Pattinson [Pat03]. [SP09] introduced the axiomatization in terms of one-step rules as used by COOL. The work we present in this thesis builds upon an already existing reasoner for coalgebraic modal logic described in [GPSWW14].

Outline

This thesis is structured as follows: We first introduce some prerequisites of our work ranging from modal logic through coalgebraic axiomatization and the fixpoint calculus this thesis adds support for. We then discuss the previous state of our tool, COOL. In chapters 4 we discuss the global caching algorithm and proceed in chapter 5 to the implementation of this algorithm, the core part of the thesis. The implementation chapter is followed by an evaluation of the implementation in comparison to state of the art solvers for several covered logics. Finally we put some notes on possible and implemented optimizations for COOL before finishing with a conclusion.

2 Prerequisites

This chapter we will briefly revisit coalgebraic modal logic as well **K**, **KD**, coalition logic and serial monotone neighborhood logic as examples. In the second part of the chapter we will introduce the coalgebraic μ-calculus.

2.1 Coalgebraic modal logic

Coalgebraic modal logic [SP09] generalizes several modal logics. An instantiation of coalgebraic modal logic is formed by a coalgebra structure (W, ξ) where W is a set of states, T is a functor and $\xi : W \to TW$ the transition function.

$$\psi, \phi ::= \bot \mid \top \mid p \mid \neg\psi \mid \psi \wedge \phi \mid \psi \vee \phi \mid \heartsuit\, \psi.$$

The operator $\heartsuit$ stands for any modal operator found in the respective logic. Every modal operator $\heartsuit$ comes with its dual operator $\overline{\heartsuit}$ such that $\overline{\heartsuit}\psi = \neg\heartsuit\neg\psi$. The semantic of the modal operators comes from the signature functor and an associated predicate lifting $[\![\heartsuit]\!]$.

Definition 2.1 (Predicate lifting)**.** A *predicate lifting* for a functor T is a natural transformation $\mathcal{Q} \to \mathcal{Q} \circ T^{OP}$ where $\mathcal{Q}$ denotes the contravariant powerset functor $Set^{OP} \to Set$, i. e. the functor that maps objects in the same way as the powerset functor and the map f to $\mathcal{Q}(f)$ which takes the preimages under f.

We will see an example for such a predicate lifting in Section 2.4. The semantic of coalgebraic modal logic then is formed as follows:

$$\begin{aligned}
[\![\psi \wedge \phi]\!] &= [\![\psi]\!] \cap [\![\phi]\!] \\
[\![\psi \vee \phi]\!] &= [\![\psi]\!] \cup [\![\phi]\!] \\
[\![\neg\psi]\!] &= W \setminus [\![\psi]\!] \\
[\![\heartsuit\psi]\!] &= \xi^{-1} \circ [\![\heartsuit]\!][\![\psi]\!]
\end{aligned}$$

2.2 Coalgebraic μ-calculus

The coalgebraic μ-calculus [CKP09] enriches the logic by adding fixpoint literals. The literals can be used to introduce a concept of iteration to the logic. This allows to add restrictions e. g. on every state that follows (however deep) from the current one. For the μ-calculus, the syntax is adapted as follows:

$$\psi, \phi ::= \bot \mid \top \mid p \mid X \mid \neg\psi \mid \psi \wedge \phi \mid \psi \vee \phi \mid \heartsuit\, \psi \mid \mu X.\, \psi_X \mid \nu X.\, \psi_X,$$

where ψ_X is a formula with distinguished variable X all of whose free occurences are positive, i. e. all occurences are under an even number of negations. To define the semantics of the μ-calculus we use the Knaster-Tarski fixpoint definition

Definition 2.2. Let $f : \mathscr{P}(W) \to \mathscr{P}(W)$. Pre-fixpoints of f are defined as $PRE_f = \{X \subseteq W \mid f(X) \subseteq X\}$ while post-fixpoints are defined as $POST_f = \{X \subseteq W \mid X \subseteq f(X)\}$. If f is monotone w. r. t. set inclusion, the least fixpoint μX is defined as the intersection of all pre-fixpoints which is also a pre-fixpoint and a fixpoint while the largest fixpoint νX is defined as the union of all post-fixpoints, also a post-fixpoint and a fixpoint.

With this definition we can now define the semantics of the coalgebraic μ-calculus

$$\begin{aligned}
[\![X]\!]_i &= i(X) & [\![\psi \wedge \phi]\!]_i &= [\![\psi]\!]_i \cap [\![\phi]\!]_i \\
[\![\mu X.\, \psi]\!]_i &= \mu [\![\psi]\!]_i^X & [\![\psi \vee \phi]\!]_i &= [\![\psi]\!]_i \cup [\![\phi]\!]_i \\
[\![\nu X.\, \psi]\!]_i &= \nu [\![\psi]\!]_i^X & [\![\neg\psi]\!]_i &= W \setminus [\![\psi]\!]_i \\
& & [\![\heartsuit\psi]\!]_i &= \xi^{-1} \circ [\![\heartsuit]\!] [\![\psi]\!]_i,
\end{aligned}$$

where $[\![\psi]\!]_i^X(G) = [\![\psi]\!]_{i;[X \mapsto G]}$ and i is a substitution mapping fixpoint variables to subsets of W.

2.3 Tableaux rules

Satisfiability in coalgebraic modal logic can be axiomatized in terms of one-step tableau rules [SP09].

Definition 2.3 (One-step tableau rules). [HSE16] We fix a set V of *(propositional) variables*. We denote by $\Lambda(V)$ the set $\{\heartsuit p \mid \heartsuit \in \Lambda, p \in V\}$ of formulas consisting of an application of a modal operator $\heartsuit$ to an element of V. Given a set U, a *U-(tableau-)sequent* is a subset of U, written $u_1, \ldots, u_n$ for $u_i \in U$ and read *conjunctively*. A *one-step tableau rule* $R = (\Gamma_0/\Gamma_1 \ldots \Gamma_n)$ consists of a $\Lambda(V)$-sequent Γ_0, the *premise*, and V-sequents $\Gamma_1, \ldots, \Gamma_n$ ($n \geq 0$), the *conclusions*, with the additional provisos that Γ_0 mentions every variable at most once, and $\Gamma_1, \ldots, \Gamma_n$ mention only variables occurring in Γ_0.

The axiomatization comes with the notion of one-step soundness and one-step completeness.

Lemma 2.4 (One-step completeness). *A set of one-step tableaux rules is one-step complete for a modal logic if for every satisfiable formula in the logic a tableau can be constructed using only the one-step rules.*

Lemma 2.5 (One-step soundness). *A set of one-step tableau rules is one-step sound for a modal logic if every formula for which a tableau can be constructed using the one-step tableaux rules is satisfiable.*

All coalgebraic logics have a one-step sound and complete axiomatization [SP09].

2.4 Examples

2.4.1 Relational modal logic

Relational modal logic is the most common modal logic. It contains two modal operators, $\Box$ and $\Diamond$ and is interpreted over Kripke structures. **CTL** is a fragment of the relational μ-calculus. In **K**, the functor for the coalgebra structure is the powerset functor $\mathscr{P}$ and the predicate lifting gives $[\![\Box]\!](A) = \{B \in \mathscr{P}(X) \mid B \subseteq A\}$ and $[\![\Diamond]\!](A) = \{B \in \mathscr{P}(X) \mid B \cap A \neq \emptyset\}$.

Definition 2.6 (Kripke structure)**.** A Kripke structure consists of a set of states S and a transition relation $R \subseteq S \times S$ as well as a map $L : S \to 2^{AP}$ which labels each state with atomic propositions from AP.

Intuitively $\Box\psi$ requires that all child nodes satisfy the formula ψ while $\Diamond\psi$ is understood to require that there exists a child node satisfying ψ.

We will consider two variants of relational modal logic: **K** is interpreted over standard Kripke frames while **KD** (and **CTL**) additionally requires that R is a total relation, i. e. for every $w \in W$ there exists at least one $w' \in W$ such that $(w, w') \in R$.

In addition to the standard tableau rules for propositional reasoning, the axiomatization of **K** contains one additional rule shown in Figure 2.1a while **KD** has both Figure 2.1a and Figure 2.1b as tableaux rules.

$$\frac{\Box a_1, \dots, \Box a_n, \Diamond b}{a_1, \dots, a_n, b}$$

(a) **K** and **KD**

$$\frac{\Box a_1, \dots, \Box a_n}{a_1, \dots, a_n}$$

(b) **KD**

Figure 2.1: tableau rules for **K** and **KD**

2.4.2 Serial monotone neighborhood logic

Serial monotone neighborhood logic [GS14] can be mapped to **KD** by replacing $\Box$ with $\Diamond\Box$ and $\Diamond$ by $\Box\Diamond$ [Par83]. The models are so called neighborhood frames.

Model of neighborhood frames. Each state is in relation with a set of neighborhoods which each are a set of states. Forming the coalgebraic μ-calculus over serial monotone neighborhood logic results in Parikh's game logic [Par83; Par85] which can be used to formalize game-theoretic problems.

$$\frac{\Box a}{a}$$

(a) $\mathbf{D_0}$

$$\frac{\Diamond b}{b}$$

(b) $\mathbf{K_0}$

$$\frac{\Box a, \Diamond b}{a, b}$$

(c) $\mathbf{K_1}$

Figure 2.2: Tableau rules for serial monotone neighborhood logic

2.4.3 Coalition logic

In coalition logic, we consider a fixed set of agents $N = \{1, \dots n\}$. Subsets of N are called coalitions. The logic has modal operators $[\{C\}]$ for each $C \subseteq N$ which can be intuitively read as "coalition C has a strategy to ensure that … ". Formally, coalition logic is interpreted over *game* *frames*. Each state x has function f_x associated with the domain $S_1 \times \cdots \times S_n$ where each S_i is a finite set of actions available to agent i in state x.

ATL [AHK02; Sch08] is a fragment of the coalgebraic μ-calculus applied on coalition logic. The coalitions A_i need to be pairwise disjoint and, for the first rule, contained in D.

$$\frac{[A_1]a_1, \ldots, [A_n]a_n, \langle D\rangle b, \langle N\rangle c_1, \ldots, \langle N\rangle c_m}{a_1, \ldots, a_n, b, c_1, \ldots, c_m} \qquad \frac{[A_1]a_1, \ldots, [A_n]a_n}{a_1, \ldots, a_n}$$

Figure 2.3: Tableau rules for coalition logic

3 COOL

COOL [GPSWW14] is a generic reasoner for modal logics. As such, COOL is given an input formula in one of the supported logics and decides whether that formula is satisfiable. Cool is designed to work generically with any modal logic which is representable in the coalgebraic framework. Several of these are already implemented in COOL including K, KD and Coalition Logics as well as combinations of logics expressed as choice or fusion (functor product or coproduct). Propositional reasoning in COOL is delegated to a dedicated satsolver. COOL had no previous support for fixpoint logics. In the following, we will describe the basic structure of the existing COOL which serves as a basis for the implementation of fixpoint support discussed in Section 5. We will cover some details of data representation as well as the basic working of the reasoner.

3.1 Formula representations

During the different preparation steps, the input formula is represented in three different forms, each serving a particular purpose. The formula is first parsed into an AST structure for easy structural manipulation. It is then converted to the hash-consed version where physical and structural identity coincide. Finally the formula is converted into an array representation suitable for fast access by the reasoner.

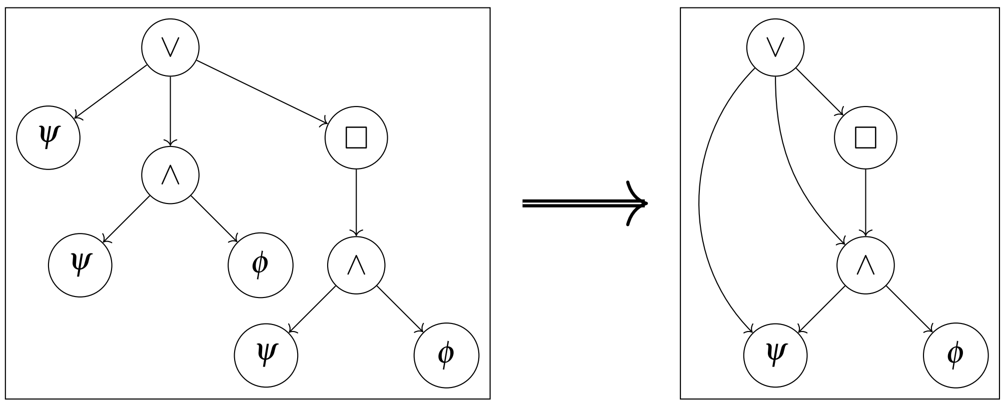
3.1.1 AST representation

Initially, COOL transforms the textual representation of a formula into a syntax tree. This is done via a standard recursive-descending parser. Standard operator precedence rules for propositional logic are respected, i. e. $\wedge$ binds stronger than $\vee$. All supported modal operators have equal precedence and bind stronger than any propositional connective. COOL already transforms some propositional constructs during this phase. For example implications are converted to equivalent disjunctions.

The AST representation is well suited to do transformations. Therefore COOL executes several transformation steps there already. First, the parsed formula is converted into negation-normal-form (NNF). Negations may only occur directly attached to propositional atoms. Afterwards, COOL performs a simplification run on the AST. In this step, conjuncts and disjuncts involving $\top$ or $\bot$ are transformed as well as, for example, $\Box\top$ which becomes $\top$.

3.1.2 Hash-Consed representation

Subsequently, the formula is converted to hash-consed form. This step primarily converts the syntax tree to a directed acyclic graph by merging all nodes of the AST that are structurally equivalent. Additionally, for each node a second node is created representing the negated formula in NNF. When finished converting to hash-consed form, COOL has constructed a node for every subformula appearing in the input formula.



AST representation Hash-Consed representation

3.1.3 Array representation

The final representation, which is then used by the reasoner, is an array structure. During hash-consing, all possible subformulae have been calculated. COOL now numbers all formulae sequentially and allocates several arrays each holding an entry for every subformula. For example the formula $\psi = \phi_1 \vee \phi_2$ will result in $type[\mathscr{I}_\psi] = OrF$, $dest1[\mathscr{I}_\psi] = \mathscr{I}_{\phi_1}$ and $dest2[\mathscr{I}_\psi] = \mathscr{I}_{\phi_2}$. $\mathscr{I}_\psi$ here is the consecutively assigned integer id of the formula ψ. Additional arrays exist to hold, for example, the index of boxes in multimodal K. A set of formulae can now be represented as a bit string where some formula ϕ is present at a node iff the $\mathscr{I}_\phi$th bit is set.

3.2 Reasoner structure

The reasoner then creates a bipartite graph with two sorts of nodes. States are saturated nodes, meaning they only contain non-clashing propositional atoms and modal operators. Cores, often referred to as pre-states, are sets of formulae that represent the result of a modal step.

Each reasoner node may be in one of four possible states: *satisfiable* and *unsatisfiable* nodes are fully processed by the reasoner and are known to be (un-)successful. *Open* nodes have been fully expanded, all their child nodes are known, but it is not yet known whether they are satisfiable. *Expandable* nodes have been created but not all of their child nodes have been created yet.

3.2.1 State nodes

State nodes, or states, are saturated, meaning that all the formulae present at a state are either just propositional atoms or modal operators. States belong to an instantiation of the coalgebraic logic which provides a set of tableaux rules. As COOL allows for composition of different logics there may be several different sets of tableaux rules active when COOL processes a formula.

The expansion of a state creates a two-step structure. For each application of a modal tableau rule (encoded within the functor), several child cores can be created. Satisfiability of a state has to be considered as a $\forall\exists$ structure. A state is successful if for every application of a modal rule there exists at least one child core that is successful.

3.2.2 Core nodes

Core nodes, or cores, are the results of expanding a state. The formulae in a core are therefore the formulae directly in a modal literal from the previous state. If a state contains $\heartsuit\psi$ as one of its formulae, a child core now may exist which contains the formula ψ. Cores are expanded by minisat [ES03] and said to be successful if one of their children is satisfiable.

Cores are transformed into conjunctive normal form as consumed by minisat. For every formula each connective is represented recursively by a fresh literal and a set of disjuncts. For example the formula $\phi = \psi_1 \vee \psi_2$ is encoded into $\neg lit(\phi) \vee lit(\psi_1) \vee lit(\psi_2)$ while $\phi = \psi_1 \wedge \psi_2$ is transformed to $\neg lit(\phi) \vee lit(\psi_1), \neg lit(\phi) \vee lit(\psi_2)$. As all the literals representing formulae present in the core node are added conjunctively, those implications are enough to encode the formula for the satisfiability solver. Modal subformulae ($\heartsuit\psi$) are translated directly into a literal and therefore treated by minisat just like propositional atoms.

Minisat will then produce a satisfying assignment. This assignment details exactly which constructs in each formula were chosen allowing COOL to reconstruct the propositional tableau steps needed to reach the state. The extracted satisfying assignment is then added to minisat when generating the next state. This ensures a different state is extracted each time. This process is repeated until all possible solutions for the core have been constructed.

3.3 Reasoner execution

COOL executes reasoner steps until the query formula is known to be either successful or unsuccessful. The reasoner starts with a single core representing the query formula and can then either expand one of the nodes or try to assign success to nodes. This process is repeated until the reasoner graph is fully expanded or a result has been found earlier. Expansion of nodes has been discussed in section 3.2.

A node can become satisfiable for several reasons. It can be *internally* satisfiable. This happens if a core contains only the formula $\top$ or a state is created which cannot be expanded further, which happens for example in **K** if the state contains no diamond literal. A core which cannot be expanded contains a propositional contradiction and is therefore unsatisfiable. Nodes can also become *locally* (un-)satisfiable if their child nodes meet the conditions laid out in section 3.2.

COOL now runs in waves. First, for every expandable core it creates one new state and for every new state it then creates all rule applications and all their child cores. This results in a modified breadth-first search which expands both one propositional and one modal step at a time and not expanding cores fully. After these expansions, COOL will try to assign success to the nodes before doing another wave. Also, whenever COOL encounters internal unsatisfiability it will try to propagate this information to the parents and mark them also as unsatisfiable if possible. Details will follow in Section 7.1.

4 Global caching algorithm

The global caching algorithm [HSE16] is an optimal single-pass tableaux algorithm for deciding satisfiability in the alternation-free μ-calculus. It operates by constructing, determinizing and solving a Büchi game on-the-fly. It is an extension of the global caching algorithm for the flat fragment of the coalgebraic μ-calculus [HS15].

Remark 4.1 (Global caching). Global caching [GN07a; GN07b] describes a family of algorithms constructing graph-shaped tableaux in a way that avoids generating any tableau node twice.

Definition 4.2 (Büchi game). A *Büchi game* is a directed graph $G = (V, E)$ where V are the nodes in the graph and $E \subseteq V \times V$ is a relation and a subset of nodes $F \subseteq V$. The graph is partitioned into nodes belonging to player one and nodes belonging to player two. In each node belonging to player one, the first player chooses an outgoing edge and in each node belonging to player two, the second player may choose an outgoing edge. Player one *wins* the game iff he can enforce to visit a node in F infinitely often. That is, he can choose nodes in his turns in such a way, that for every choice player two might make, the chosen path will pass through F infinitely often.

Before describing the global caching algorithm for the alternation-free μ-calculus, we need some definitions.

4.1 Fischer-Ladner-Closure

Definition 4.3 (Fischer-Ladner Closure). The Fischer-Ladner-Closure (FLC) [FL79; Koz88] of a formula ψ is the smallest set of formulae such that:

$$\begin{aligned}
& \psi \in FLC(\psi) \\
\phi \in FLC(\psi) &\Rightarrow \neg\phi \in FLC(\psi) \\
\phi \wedge \varphi \in FLC(\psi) &\Rightarrow \phi \in FLC(\psi), \varphi \in FLC(\psi) \\
\phi \vee \varphi \in FLC(\psi) &\Rightarrow \phi \in FLC(\psi), \varphi \in FLC(\psi) \\
\heartsuit\phi \in FLC(\psi) &\Rightarrow \phi \in FLC(\psi) \\
\eta X.\phi \in FLC(\psi) &\Rightarrow \phi[X \mapsto \eta X.\phi] \in FLC(\psi)
\end{aligned}$$

The FLC of some formula ψ is finite and no larger than $2 \cdot \psi$ [Koz83].

Intuitively, the FLC contains all formulae that may be encountered as result of tableaux rules. As such it can be used for encoding of formulae and for complexity bounds.

4.2 Deferrals

Definition 4.4 (Deferrals). [HSE16] Given fixpoint literals $\chi_1, \dots, \chi_n$, where $\chi_i = \eta X_i.\, \psi_i$, we say that the substitution $[X_1 \mapsto \chi_1]; \dots; [X_n \mapsto \chi_n]$ is *built over* χ_n if $\chi_i <_f \chi_{i+1}$ for all $1 \leq i < n$, where we write $\psi <_f \mu X.\phi$ if $\psi \leq \phi$ and ψ is open and occurs free in ϕ. We say that e is *irreducible* if there is no sequence $[X_1 \mapsto e_1];\ldots;[X_n \mapsto e_n]$ built over some e_n such that $e = e_1([X_2 \mapsto e_2];\ldots;[X_n \mapsto e_n])$. A closed least fixpoint that is irreducible is an *eventuality*. A formula ψ *belongs* to an eventuality e_n, if $\psi = \alpha\sigma$ for some α and σ such that $\sigma = [X_1 \mapsto e_1];\ldots;[X_n \mapsto e_n]$ is built over e_n and $\alpha <_f e_1$. We refer to formulas that belong to some eventuality e as *e-deferrals* and denote the set of e-deferrals by $dfr(e)$.

Intuitively, deferrals encode the fact that least fixpoints must not require infinite unfolding but rather, after a finite number of steps, reach a state where the argument to the fixpoint literal holds independent of the unfolding of the fixpoint. As such we need a form of *deferral tracking*: We need a notion of if a deferral still is present in the result of a tableau rule application or whether it has been finished (and the eventuality therefore has reached its finite conclusion).

To see how the algorithm tracks these eventualities, we use the notion of *inheriting* a formula. A node inherits some formula ψ from a modal literal $\heartsuit\psi$ if ψ was produced by applying a tableau rule on $\heartsuit\psi$. Similarly, for propositional tableaux rules a formula ψ is inherited from ϕ if ψ is the result of a tableau rule applied on ϕ.

The algorithm now works by annotating each reasoner node with a *focus*. The *focus* is a subset of the deferrals at the nodes. When creating a successor node $(\Gamma, d_{\Delta\rightsquigarrow\Gamma})$, all deferrals in the successor node are added to the *focus* iff they are inherited from focused nodes in the original node (Δ, d) where $d_{\Delta\rightsquigarrow\Gamma}$ denotes the step of *tracking* the deferrals in the focus. If the focus of a node becomes the empty set, the algorithm *refocuses*: All deferrals in the successor node are added to the focus.

In terms of Büchi games the set of reasoner nodes can be identified with V, the relation E with the successor relation on the reasoner nodes and F contains all nodes in V which have the empty set as focus.

4.3 Propagation

Finally, success is determined by the algorithm based on a propagation step. In terms of Büchi games the propagation step solves the (partial) game. We start by giving some definitions.

Definition 4.5. Let $C \subseteq \mathbf{C}$ be a set of focussed nodes. We define the functions $f : \mathscr{P}(C) \to \mathscr{P}(C)$ and $g : \mathscr{P}(C) \to \mathscr{P}(C)$ by

$$f(Y) = \{(\Delta, d) \in C \mid \forall \Sigma \in Cn(\Delta).\, \exists \Gamma \in \Sigma.\, (\Gamma, d_{\Delta\rightsquigarrow\Gamma}) \in Y\}$$
$$g(Y) = \{(\Delta, d) \in C \mid \exists \Sigma \in Cn(\Delta).\, \forall \Gamma \in \Sigma.\, (\Gamma, d_{\Delta\rightsquigarrow\Gamma}) \in Y\}$$

for $Y \subseteq C$. We refer to C as the *base set* of f and g.

Definition 4.6 (Proof transitionals). [HSE16] For $H \subseteq C \subseteq \mathbf{C}$, we define the *proof transitionals* $\hat{f}_H : \mathscr{P}(C) \to \mathscr{P}(C)$, $\hat{g}_H : \mathscr{P}(C) \to \mathscr{P}(C)$ by

$$\hat{f}_H(G) := (f(H \cap G) \cap \overline{F}) \cup (f(H) \cap F) = f(H \cap G) \cup (f(H) \cap F)$$
$$\hat{g}_H(G) := (g(H \cup G) \cup F) \cap (g(H) \cup \overline{F}) = g(H) \cup (g(H \cup G) \cap \overline{F}),$$

for $G \subseteq C$, where $F = \{(s,d) \in C \mid d = \emptyset\}$ and $\overline{F} = \{(s,d) \in C \mid d \neq \emptyset\}$ are the sets of focused nodes with empty and non-empty focus, respectively.

Lemma 4.7. *[HSE16] The proof transitionals are monotone w.r.t. set inclusion, i.e. if $H' \subseteq H$, $G' \subseteq G$, then $\hat{f}_{H'}(G') \subseteq \hat{f}_H(G)$ and $\hat{g}_{H'}(G') \subseteq \hat{g}_H(G)$.*

Definition 4.8 (Propagation). [HSE16] For $G \subseteq \mathbf{S}$, we define $E_G, A_G \subseteq G \times G$ as

$$E_G = \nu H.\mu G.\ \hat{f}_H(G) \quad \text{and} \quad A_G = \mu H.\nu G.\ \hat{g}_H(G),$$

where E_G is the set of *successful* nodes and A_G the set of *unsuccessful* nodes. Note that during intermediate propagation (when the reasoner graph has not yet been fully expanded), E_G and A_G do not necessarily form a partition of the set of nodes. However both sets will only monotonely increase.

4.4 The algorithm

Decide satisfiability of a closed formula ϕ_0.

1. (Initialization) Let $G := \emptyset$, $\Gamma_0 := \{\phi_0\}$ be the initial node and $U := \{\Gamma_0\}$ the set of nodes which can still be expanded.
2. (Expansion) Pick $t \in U$ and let $G := G \cup \{t\}$, $U := (U - \{t\}) \cup (\bigcup Cn(t) - G)$.
3. (Intermediate propagation) Optional: Compute E_G and/or A_G. If $(\Gamma_0, d(\Gamma_0)) \in E_G$, return 'Yes'. If $(\Gamma_0, d(\Gamma_0)) \in A_G$, return 'No'.
4. If $U \neq \emptyset$, continue with Step 2.
5. (Final propagation) Compute E_G. If $(\Gamma_0, d(\Gamma_0)) \in E_G$, return 'Yes', else 'No'.

Soundness and completeness of the algorithm has been proven in [HSE16]:

Theorem 4.9 (Soundness). *The algorithm returns 'Yes' on input ϕ_0 if ϕ_0 is satisfiable.*

Corollary 4.10 (Completeness). *If a run of the algorithm with input ϕ_0 returns 'Yes', then ϕ_0 is satisfiable.*

4.4.1 Complexity

Our algorithm has optimal complexity (given that the problem is known to be ExpTime-hard for all logics discussed in Section 2.4):

Theorem 4.11. *The global caching algorithm decides the satisfiability problem of the alternation-free μ-calculus in* ExpTime, *more precisely in time $2^{\mathcal{O}(n)}$.*

Additionally, by means of step 3 in the algorithm may terminate before the graph has been fully expanded, that is, the global caching algorithm may, for some formulae, terminate before creating an exponentially large data structure.

5 Implementation

In this chapter, we will discuss the integration of the global caching algorithm as described in the previous chapter into COOL. We start with a discussion of parser extensions and additional preprocessing. We then discuss the additional initialization needed for the reasoner. Finally we discuss the changed propagation algorithm for success checking.

5.1 Preparations

Preparation is structured into three phases. First, the input formula is parsed. In a second step all fixpoint logic operators are normalized to explicit fixpoint literals. Finally the prepared formula is verified to have several properties needed for correct operation of the algorithm.

5.1.1 Parsing

For the μ-calculus, COOL needs additional syntax for fixpoint literals, variables and – to handle **CTL** – the operators from **CTL**. Especially the text-like operators of **CTL** can cause conflicts with previously-allowed atoms. As a result, the **CTL** operators – as well as the strings "MU" and "NU" – are no longer useable as atoms with fixpoints. It is advised to use only lower-case characters for atoms if possible.

Variables bound by fixpoint literals in the input may be shadowed. Consider for example the variable X in $\mu X.(\mu X.\Box X) \vee \Diamond X$. Here, the X below the $\Box$ is bound by the inner fixpoint literal while the X below the $\Diamond$ is bound by the outer fixpoint literal. COOL replaces variable literals by fresh symbols in such a way that each symbol uniquely belongs to exactly one fixpoint literal. Note that this limits the identification of subformulae in the algorithm to only those formulae not containing any fixpoint literals. It is desirable to properly identify equivalent subformulae even in the presence of fixpoints – this would allow for example to directly identify $AG\ p \wedge \neg AG\ p$ as a contradiction and avoid exploring equivalent subformulae appearing in different places of the input formula twice. This is, however, left aside as a future optimization.

Least fixpoints literals can be expressed equivalently by the two forms $\mu X.\psi$ and $MU\ X.\psi$ while largest fixpoints literals can be written as $\nu X.\psi$ and $NU\ X.\psi$. **CTL** formulae are accepted in the same syntax as, for example, by TreeTab [AGW07] (e. g. $AG\ \psi, A(\phi\ U\ \psi)$) as there does not seem to be a strict syntax implemented by all solvers. For a full list of **CTL** construct, see Figure 5.1.

5.1.2 CTL transformation

COOL does not handle **CTL** directly but considers it to be an alternative syntax for the relational μ-calculus. **CTL** formulae are therefore converted to their explicit form generating fresh variable names for the implicit **CTL** fixpoints. This also allows combining **CTL** operators with the more expressive syntax of the μ-calculus. Also, while **CTL** is normally understood as an extension to **KD**, COOL allows **CTL** operators with any logic featuring the same set of modal operators but with different tableaux rules. This currently includes **K** and serial monotone neighborhood logic [GS14]. **CTL** formulae are translated to the μ-calculus as follows:

Figure 5.1: Transformation of **CTL** constructs

$$\begin{array}{rcl}
 & & A(\phi\, U\, \psi) \Rightarrow \mu X.(\psi \vee (\phi \wedge \Box X)) \\
AF\,\phi \Rightarrow \mu X.(\phi \vee \Box X) & & E(\phi\, U\, \psi) \Rightarrow \mu X.(\psi \vee (\phi \wedge \Diamond X)) \\
EF\,\phi \Rightarrow \mu X.(\phi \vee \Diamond X) & & A(\phi\, R\, \psi) \Rightarrow \nu X.(\psi \wedge (\phi \vee \Box X)) \\
AG\phi \Rightarrow \nu X.(\phi \wedge \Box X) & & E(\phi\, R\, \psi) \Rightarrow \nu X.(\psi \wedge (\phi \vee \Diamond X)) \\
EG\phi \Rightarrow \nu X.(\phi \wedge \Diamond X) & & A(\phi\, B\, \psi) \Rightarrow \nu X.(\neg\psi \wedge (\phi \vee \Box X)) \\
 & & E(\phi\, B\, \psi) \Rightarrow \nu X.(\neg\psi \wedge (\phi \vee \Diamond X))
\end{array}$$

5.1.3 Verification of properties

For correct results, the input formula ψ needs to satisfy several properties. Each property is verified in a recursive check executed after **CTL** normalization and before further processing the formula. Note that any pure **CTL** formula trivially satisfies all criteria due to the fact that it only includes a small number of valid fixpoint schemes in its implicit syntax.

Definition 5.1 (Positivity)**.** A fixpoint literal $\eta\ X.\psi$ is said to appear positive iff every occurence of X in ψ appears under an even number of negations. In NNF, this is equivalent to the condition that X does not occur negated.

Positivity implies monotonicity of the logic (w. r. t. set inclusion). This is required to ensure that fixpoints actually exist. The requirement therefore is also in place for solvers of the full μ-calculus and not a restriction on the supported fragment. The core of the algorithm testing for positivity can be found in Listing A.2. It keeps for every bound variable a counter of negations between the binding and the current position and, when encountering a variable, checks whether the counter is even.

Definition 5.2 (Guardedness)**.** A fixpoint literal $\eta\ X.\psi$ is called guarded iff in ψ the fixpoint variable X only occurs in modal literals.

Guardedness is a common restriction on the μ-calculus. There exist methods to convert unguarded formulae to guarded ones [FL11] which come with an exponential increase in size. The core algorithm to verify guardedness is reproduced in Listing A.3. While recursively examining the formula, it keeps a list of variables which have been newly bound and clears this list every time it passes a modal operator.

Definition 5.3 (Alternation-Freeness)**.** A closed formula ψ is said to be alternation-free iff every fixpoint literal $\eta\ X.\phi$ within ψ is either enclosed in another fixpoint literal of the same kind or contains no free variables.

Example 5.4. The formulae $\mu\ X.\mu\ Y.\Box(X \wedge Y)$ and $\mu\ X.(\nu\ Y.\Box\ Y) \wedge \Box\ X$ are alternation-free while $\mu\ X.\nu\ Y.\Box(X \wedge Y)$ is not because the inner fixpoint is enclosed in the dual fixpoint type and contains a free occurence of X.

Finally, alternation-freeness is the central restriction of the algorithm presented. This still allows however to fully include e. g. **CTL** and **ATL** which are just fragments of the flat μ-calculus with different base logics. The algorithm can be found in Listing A.1. In contrast to the other tests which operate in pre-order, alternation-freeness is checked by a post-order traversal of the AST. When unwinding the recursion and traversing a fixpoint literal, the algorithm notes the kind of fixpoint (least or largest fixpoint) iff the fixpoint literal contains free variables or a special mark otherwise. At every fixpoint literal it is verified that the current fixpoint literal is of the same kind as the note or it was noted that the argument of the fixpoint operator contains no open fixpoint literal. For all binary connectives, the note of both branches need to be merged. If only one side is marked as having an open fixpoint, that mark is inherited by the binary operator. The same is true when both branches have been marked to contain the same kind of fixpoint literal. If the branches have non-matching marks, the next enclosing fixpoint needs to break the assertion of alternation-freeness for one of the paths and a failure is signaled.

5.2 Initialization

After preprocessing the formula, the reasoner needs to be initialized. The set of relevant formulae now needs to take into account the fixpoint expansions as calculated by the FLC as described in Definition 4.3. The algorithm recursively traverses the formula in hash-consed form (see Subsection 3.1.2) in pre-order and collects all the visited formulae. This step is also responsible for creating expansions of the fixpoint literals. Whenever a fixpoint literal is encountered, the algorithm substitutes all free occurrences of the respective fixpoint variable by the fixpoint literal. This expansion can result in variable shadowing (as can be seen in Example 5.5) which needs to be taken into account when implementing the substitution. The fixpoint is then represented in the array form (See section 3.1.3) with a silent transition from itself to the unfolded version.

The algorithm will stop on any formula it has encountered before and therefore unfold every fixpoint exactly once. This guarantees termination of the algorithm after $2 \cdot n$ steps for any formula of size n.

Example 5.5. $\mu X.\Box(\mu Y.\Diamond(X \wedge Y))$ will be unfolded to $\Box(\mu Y.\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge Y))$ where Y is bound twice by nested fixpoints. The FLC for this formula will now contain the following formulae:

- $\mu X.\Box(\mu Y.\Diamond(X \wedge Y))$
- $\Box(\mu Y.\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge Y))$
- $\mu Y.\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge Y)$
- $\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge \mu Y.\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge Y))$
- $\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge \mu Y.\Diamond(\mu X.\Box(\mu Y.\Diamond(X \wedge Y)) \wedge Y)$

5.2.1 Deferral annotation

The algorithm constructing the FLC is also responsible for annotating the formulae as deferrals if they belong to some eventuality. It does so by keeping a list of fixpoint variables belonging to least fixpoints. Whenever a least fixpoint literal is encountered the respective variable is appended to the end of this list. When processing the next subformula, all variables that occur no longer free in the formula are removed. The first element in the list then is the eventuality that this subformula belongs to.

When a subformula is encountered that has already been processed but was not marked as deferral while the list of variables is not empty and the subformula should therefore be marked as deferral, the information is updated. This is necessary because the variable X is not encountered when first processing $\mu X . \psi$ while this literal is a deferral for itself.

5.3 Graph construction

Once the FLC has been computed and annotated with the corresponding eventuality, COOL continues constructing the reasoner graph. Each node in the reasoner contains a distinct tuple $(\mathit{formulae}, \mathit{focus})$, where *formulae* is taken from the COOL version without fixpoints and represents the set containing all formulae at this reasoner node. *focus* is the focus of the node containing the subset of *formulae* of deferrals that have not yet been finished. The reasoner then starts with a core node that contains the input formula as its only element and has the empty set as focus.

5.3.1 Deferral tracking

For modal steps, a formula in the child core has to be marked as still being a deferral if the formula and one of its sources in the state are marked as deferral belonging to the same fixpoint and one of these source formulae was contained in the focus.

Example 5.6. Consider a state in **K** with two formulae $\Box\psi$ and $\Diamond\psi$ with both formulae as well as ψ marked as deferral to some fixpoint identified by its variable X_i. Now according to the tableau rules for **K**, one child core is created containing just ψ for both the box and diamond rule. The formula ψ in the core now needs to be marked as unfinished eventuality if at least one of $\Box\psi$ and $\Diamond\psi$ in the parent state was marked as deferral.

The implementation of deferral tracking for **K** is reproduced in a simplified way (the removed code is irrelevant for the fixpoint implementation in COOL) in Listing A.4. In line 12 (22-23) it is checked if the original formula was a deferral and whether both the created formula and the original formula belong to the same eventuality. Lines 11-14 handle the diamond while lines 21-25 take care of the boxes. For other logics, the code is essentially the same. Every time a formula is added to the generated core, deferral tracking is performed and the created formula added to the focus if it matches the requirements.

When expanding a core, the formulae of the newly created state are extracted from the results of the sat solver. The extracted literal can either be a (negated) propositional atom, in which case it is never a deferral, or a modal formula $\heartsuit\psi$. This modal formula is a deferral if the formula where it is derived from has been marked as deferral to the same fixpoint as $\heartsuit\psi$ and was contained in the focus.

5.3.2 Refocusing

Deferral tracking can yield reasoner nodes with empty focus. If such a node is expanded, the children need to be *refocused*. When refocusing, all formulae which are marked as deferral to some fixpoint, if any, are added to the focus and the reasoner continues with tracking these deferrals. Refocusing as part of a modal step can be seen in the code in Listing A.4: Line 4 tests whether refocusing is needed and in lines 11 and 21 all deferrals are added to the focus bypassing the deferral tracking condition.

5.4 Deciding success

The final step the reasoner has to perform is deciding whether a formula is *successful*. We distinguish three different conditions for *success*.

Definition 5.7 (Success)**.** A reasoner node can become (un-)successful in three different ways, *internally*, *locally* or *globally*. A state can become *internally* successful if no more modal rules can be applied. A core can be *internally* successful if it only contains the single formula $\top$ and unsuccessful if the formulae are propositionally contradictory. It is *locally* (un-)successful as a consequence of the success of its children and *globally* (un-)successful, if it is part of an (un-)successful cycle.

Definition 5.8. A core node can *step out of* (*step into*) a set of nodes N iff there exist a child node c with $c \notin N$ ($c \in N$). Similarly a state node can *step out of* (*step into*) a set of nodes N iff every rule application creates at least one child node c with $c \notin N$ ($c \in N$).

We can now proceed to the algorithms deciding the different sorts of success.

5.4.1 Local successfulness

States are successful if all rule applications yield at least one successful child node. In **K**, **KD** or the serial monotone neighborhood logic each rule yields exactly one child node and therefore this condition could be simplified to requiring, that all children are successful. Some other coalgebraic logics however have rules that yield more than one child and the children created by the same rule application have to be considered disjunctively.

Cores in contrast are successful if they have at least one successful child.

5.4.2 Global successfulness

The algorithm (which is reproduced in Listing A.5) considers two sets of nodes. The set X^0 contains all nodes that are admissible as part of a path to a finishing node and initially contains all open or successful nodes as well as all expandable cores. As cores only need one successful child node, cores can, in contrast to states, be marked as successful before being fully expanded. The set X^0 is formed by *setStates* and *setCores* in the implementation. The set A_X^0 (*setFinishingStates* and *setFinishingCores*) contains all nodes, which are considered admissible for finishing eventualities and initially contains all successful nodes as well as all nodes with empty focus. These sets are constructed using the *stateCollector* (lines 18-41) and *coreCollector* (lines 46-60) functions.

The algorithm then computes the fixpoints from Definition 4.8: It calculates the set of nodes X^{n+1} which are admissible as part of the path in the next step: Starting with the nodes in A_X^n, it adds all parent nodes which can already step into X^{n+1} to that set. This is repeated for all nodes added to X^{n+1} until no more node can be added. It then sets $A_X^{n+1} = A_X^n \cap X^{n+1}$ and continues with the next iteration.

Once $X^n = X^{n+1}$ the fixpoint iteration can be stopped. All nodes still contained in X^n are successful and can be marked as such. In terms of Büchi games, the set X^n contains now only nodes where player one can enforce that unfocused nodes are visited infinitely often.

5.4.3 Global unsuccessfulness

The algorithm for unsuccessful cycles works similar to the one for successful ones. It initializes the sets A_X^0 and X^0 the same way with the only difference that expandable states are considered instead of expandable cores as states only need one contradictory child to be marked unsuccessful.

The algorithm initializes the set E_X^{n+1} with X^n. Every node from A_X^n is then removed from E_X^{n+1} if it steps out of E_X^{n+1}. This process is repeated for all parents of removed nodes until no more node can be removed from E^{n+1}.

At this point, all nodes in E^{n+1} are unsuccessful (i. e. nodes where player two can enforce that only focused nodes are passed from some time on) and marked as such. The algorithm then sets $A_X^{n+1} = A_X^n \setminus E_X^{n+1}$ and $X^{n+1} = X^n \setminus E_X^{n+1}$. Once $E_X^{n+1} = \emptyset$, no more nodes can be marked unsuccessful and the algorithm terminates.

6 Comparison

In this chapter we will compare COOL to reasoners supporting several subsets of logics supported by COOL. Unfortunately only **CTL** has a considerable number of other reasoners as well as established benchmark formulae. Measurements were done on an Intel i7-4790 CPU running at 3.60 GHz. The machine had 16 GiB of Memory and the stack limit was raised to 1 GiB to accommodate some of the other solvers.

6.1 CTL reasoning

COOL supports **CTL** as the flat fragment of the μ-calculus over KD. An extensive study of algorithms and existing reasoners for **CTL** has been conducted in [GTW11]. I used a selection of the formulae used in this evaluation as well as our own *early* formulae and a randomly generated set of **CTL** formulae in this comparison.

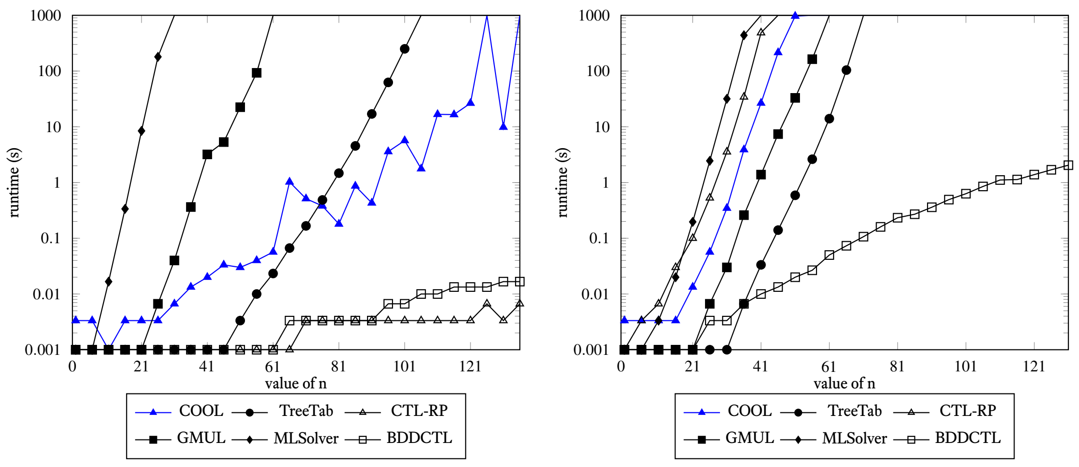
6.1.1 Reasoner

There exist two general types of **CTL** reasoners. Tableaux based reasoners, COOL being one of them, are using a top-down approach. Other examples of tableau based reasoners are TreeTab, GMUL and MLSolver. The resolution based approach has completely different runtime characteristics and can decide formulae quickly that are expensive for all tableaux based reasoners while being considerably slower on other kinds of formulae. We include CTL-RP and BDDCTL as bottom-up reasoners in the following experiments.

Remark 6.1 (Compacting)**.** Some of the reasoners used profit heavily from an preprocessing step called compacting. This process is described in [GTW11]. A formula containing $AG\ \phi_1 \wedge \cdots \wedge AG\ \phi_n$ is transformed to the equivalent form $AG\ (\phi_1 \wedge \cdots \wedge \phi_n)$. The reference comparison also includes a step called "3-compact" that works around a limitation in BDDCTL on the size of the temporal formula while maintaining most of the benefit of compacting by only collecting groups of three AG for the transformation. This step is left out in this comparison which may cause BDDCTL to fail processing formulae it otherwise could have processed but should not negatively impact BDDCTL runtime where it could process a formula.

TreeTab

TreeTab is considered to be the best tableaux reasoner for **CTL**. It implements a single-pass algorithm described in [AGW07] which is, like COOL, able to conclude before fully expanding the graph. TreeTab inherently uses a depth-first strategy however and may therefore expand exponentially sized branches before considering short branches that already determine the formula. TreeTab is also, in contrast to COOL, not optimal and has a worst case runtime of 2EXPTIME.



(a) montali_sat1 (b) montali_unsat1

Figure 6.1: Montali 1

GMUL

GMUL is an implementation of a tableaux algorithm first described in [BPM83]. It is, in general, the fastest optimal tableaux based **CTL** reasoner. However, GMUL needs to fully expand the reasoner graph before it can decide whether the eventualities are successful.

MLSolver

MLSolver [FL10] is an optimal reasoner for the full μ-calculus which includes the alternation-free μ-calculus supported by COOL as a fragment. MLSolver works by transforming the input formula into a parity game and using a game solver on the output. We compare COOL against MLSolver on **CTL**, and later in this chapter, on the alternation-free fragment. However MLSolver does not support the same coalgebraic generalization as COOL and can therefore not be used for e. g. ATL which is discussed below.

MLSolver was used with the same optimizations as described in [GTW11], `-opt comp -opt litpro -pgs recursive`, which improves significantly on the baseline performance of MLSolver.

CTL-RP

CTL-RP is a resolution-based reasoner described in [ZHD09]. CTL-RP works in a clausal-normal-form SNF^{g}_{CTL}. Eventualities are handled in a second reasoner phase after all resolvents are found. It implements an optimal algorithm finishing in ExpTime. As a bottom-up reasoner, CTL-RP performs significantly differently from the tableaux solvers.

BDDCTL

BDDCTL [Mar05] utilizes ordered binary decision diagrams to represent the formulae. Like CTL-RP it implements a bottom-up approach and delays checking for eventualities to a second phase.

6.1.2 *montali* Formulae

The *montali* formulae are derived from a paper by Montali et. al.[Mon+08] and originally designed to show advantage of abductive logic programming instead of LTL for verification of business processes. They later found their way into **LTL** and **CTL** benchmarks. They encode a payment process consisting of n sequential steps where k of them may fail. The *montali* formulae used in the comparison set k to one.

They can be decided efficiently by resolution-based solvers but are difficult for most tableau-based solvers. The results from COOL for the satisfiable case however are promising and COOL seems to scale considerably better on these compared to the other tableaux solvers while it exhibits comparable behavior on the unsat formulae.

The montali formulas are created using the following fragments:

$$\begin{aligned}
\phi_1^i &:= AF\ p_1 \\
\phi_m^i &:= AF\ (p_i \wedge AX\ \phi_{m-1}^i) \\
\psi_n &:= AG\ \bigwedge_{i=0}^{n-1} (p_i \Rightarrow AX\ A(\neg p_i U p_{i+1})).
\end{aligned}$$

The satisfiable *montali* formulae are formed by $\phi_k^0 \wedge \psi_n$ while the unsatisfiable *montali* formula are formed by $\phi_k^0 \wedge \psi_n \wedge \neg\phi_k^n$ with k set to one.

COOL works remarkably well on the satisfiable examples. Considering the ψ_n part of the montali formulae, the reasoner needs to construct a sequence of states which contain p_i for strictly increasing i. COOL seems to be able to quickly dismiss nodes which contain several p_i and therefore postpone the eventuality $A(\neg p_i U p_{i+1})$ indefinitely.

6.1.3 *exp* Formulae

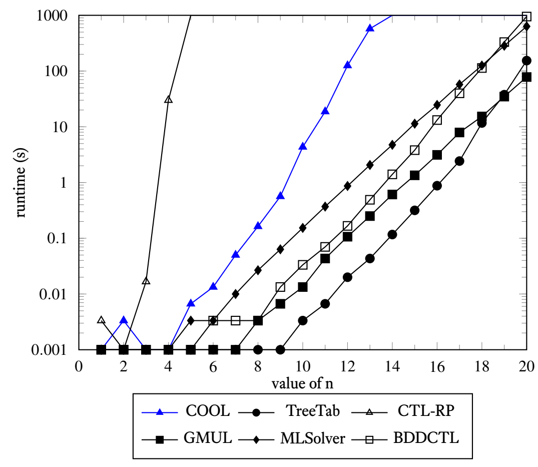
The *exp* formulae are designed to create an exponentially sized graph with exactly one cycle spanning the whole graph. this is achieved by implementing a binary counter and, for the unsatisfiable formulae, requiring that the counter never has all bits set to ones.

This cycle can only be decided after fully exploring the search-space, therefore enforcing exponential runtime. These formulae can therefore be considered the worst case for COOL as the fastest solutions here would be fully expanding the reasoner graph and only then trying to finish the eventualities. The preference to simpler algorithms can also be observed in GMUL overtaking TreeTab for the largest samples of these formulae.

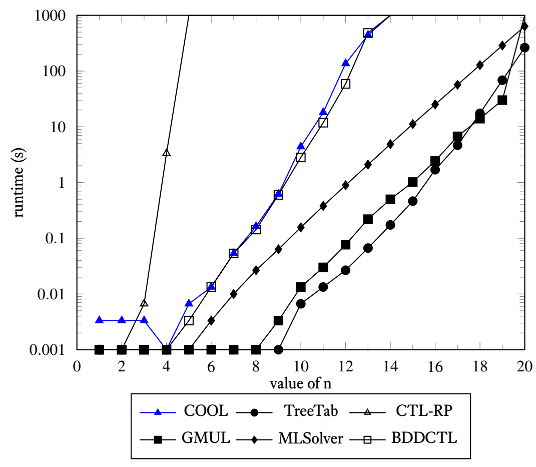
In the next chapter we will discuss some effort to get COOL closer to this optimal behavior on the *exp* style functions while still performing well on less extreme examples.

6.1.4 *early* and *early_gc* Formulae

The *early* formulae express a n-bit binary counter which, after a few steps starts another counter postponing an eventuality indefinitely. The formulae have, like the *exp* formulae, an exponentially large search space created by the counter. However the postponed eventuality, which branches from the cycle created by the main counter, allows to already conclude without expanding an exponentially large part of the search space.

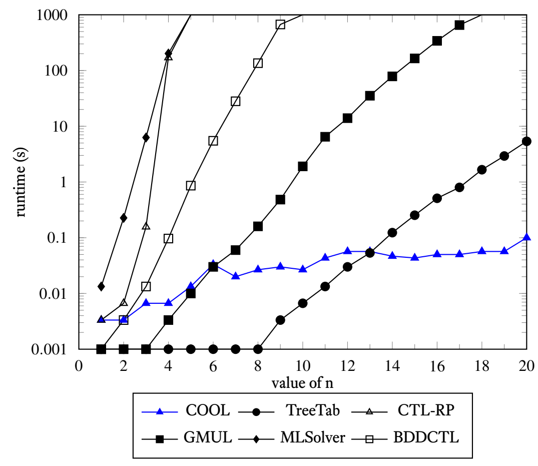


(a) exp_sat

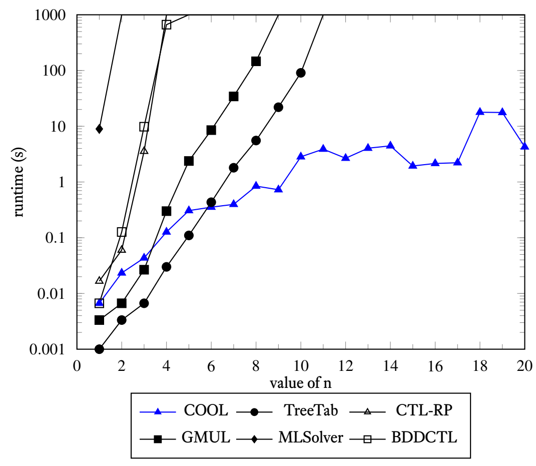


(b) exp_unsat

Figure 6.2: exp



(a) early

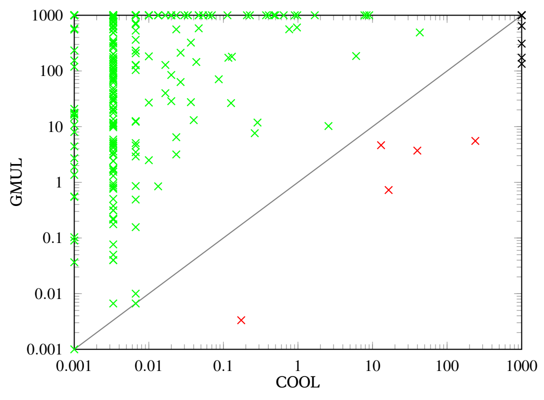


(b) early_gc

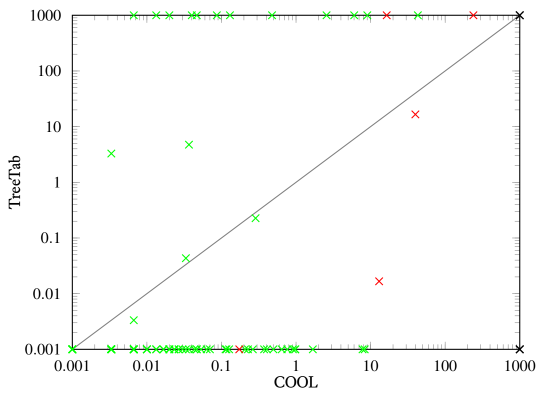
Figure 6.3: early

$$
\begin{aligned}
c_n(x,i) &= (\neg x_{n-i} \wedge AX\ x_{n-i} \wedge \psi_n(x,i-1)) \vee (x_{n-i} \wedge AX\ \neg x_{n-i} \wedge c_n(x,i-1)) \\
init(x,m) &= AG\ ((start_x \to (x \wedge \textstyle\bigwedge_{0 \le i < m} \neg x_i)) \wedge (x \to EX\ x)) \\
early(n,j,k) &= start_p \wedge init(p,n) \wedge init(r,k) \wedge AG\ ((r \to c(r,k)) \wedge (p \to c(p,n))) \wedge \\
&\quad AG\ ((\textstyle\bigwedge_{0 \le i \le j} p_i \to EX(start_r \wedge EF\ p)) \wedge \neg(p \wedge r) \wedge (r \to AX\ r)) \\
early_{gc}(n,j,k) &= early(n,j,k) \wedge b \wedge init(q,n) \wedge AG\ (\neg(p \wedge q) \wedge \neg(q \wedge r) \wedge (q \to c(q,n))) \\
&\quad \wedge\ AG\ (AF\ b \wedge (b \to (EX\ p \wedge EX\ start_q \wedge AX\ \neg b)))
\end{aligned}
$$

As anticipated, COOL quickly detects the unsuccessful cycle and is therefore able to decide the formulas quickly. While TreeTab should, in general, be able to finish early here as well, the results indicate that it always follows the exponentially large cycle. This is likely due to the depth-first traversal which forces TreeTab to finish the large cycle if it starts to explore it.



(a) random with GMUL



(b) random with TreeTab

Figure 6.4: *random* **CTL** formulae of size 100

6.1.5 *random* Formulae

Additionally, we evaluated the performance of **CTL** reasoners on a randomly generated set of formulae constructed of the propositional operators $\vee$ and $\wedge$ and the **CTL** operators AX, EX, $AG, AF, EG, EF, A(\phi U \psi), A(\phi B \psi), E(\phi U \psi), E(\phi B \psi)$ where each of the propositional operators was twice as likely to appear as each of the **CTL** operators. The formulae were constructed by choosing a random operator and placing it in one of the free slots of the to-be constructed formula. All slots still empty at the end were filled with one of the propositional atoms in positive or negated form.

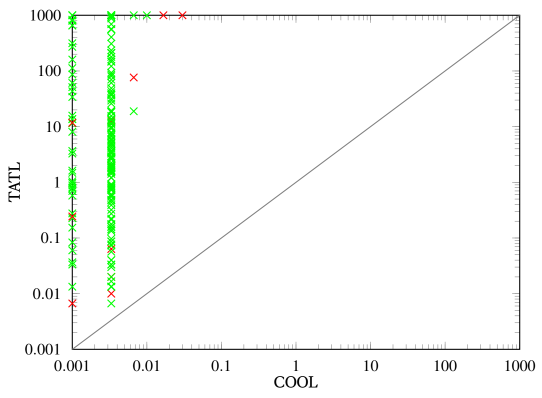
We used 100 operators in each formula and five propositional atoms. This resulted in a collection of formulae where the reasoners were able to solve most queries within the bounds of the experiment while still being hard enough to create meaningful results. The red marks in the scatter plot signify unsatisfiable formulae while the green ones mark the results of satisfiable formulae.

Of the 300 generated formula, 287 can be decided by at least one reasoner while for 13 all reasoners failed to terminate in time. Of these 287 formulae, COOL finished 216 in less than 0.01 seconds and 32 additional ones in less than 0.1 seconds. Remarkably, GMUL consistently performed better than COOL on the unsatisfiable formulae while COOL was faster for all satisfiable formulae solved in time. The considerable advantage of TreeTab compared to COOL is likely due to the depth-first expansion of TreeTab which provides good results if most of the paths are enough to decide the formulae. COOL can be adapted to work in a depth-first manner as well. However the advantage of COOL on the *early* and *montali_sat* formulae is due to the breadth-first expansion strategy and both modes have their respective advantages..

6.1.6 Results

COOL works remarkably well compared to the other tableau based reasoners and performs similarly to the best of them for each class of formulae. The *early* formulae show the strength of COOL as optimal single-pass reasoner.

COOL has no known set of formulae where it performs asymptotically worse than other tableaux solvers while it is the only solver fast on the early formulae. Bottom-Up solvers still perform better for certain problems. One would therefore probably still recommend hybrid solver as in[GTW11] but replace TreeTab by COOL.



(a) three propositional variables

Figure 6.5: *random* ATL formulae of size 50

Even in cases, where COOL performs exceedingly well, a comparatively large initial constant time cost can be observed. This is likely due to the fact that COOL actually operates on the alternation-free μ-calculus in coalgebraic generality while the other solvers only concern themselves with **CTL**.

6.2 ATL reasoning

Cool supports ATL as the flat fragment of the μ-calculus over coalition logic. The other solver working on this logic is called TATL[Dav13].

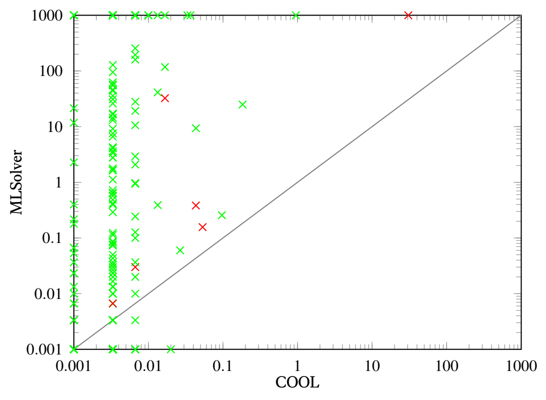
6.2.1 *random* Formulae

Unfortunately, for **ATL** there does not exist a comparable set of benchmark formulae we could consider. To get an impression of the performance of COOL and TATL we used a set of randomly generated formulae. The generation was similar to the construction already used for **CTL** but after replacing the **CTL** operators by ATL ones. In contrast to the **CTL** case, COOL does not directly support the usual syntax so the formulae are written in two different representations: a classical ATL representation as consumed by TATL and the explicit fixpoint syntax as consumed by COOL. We used formulas built with 50 operators which covers the full range of runtime allowed within the parameters of the experiment. During our experiments we discovered some discrepancies between the results from COOL and the ones generated by TATL. As a result TATL answered differently on three out of the 200 formulae used for this experiment. The author of TATL has confirmed that this is a bug and the result reported by COOL is correct.

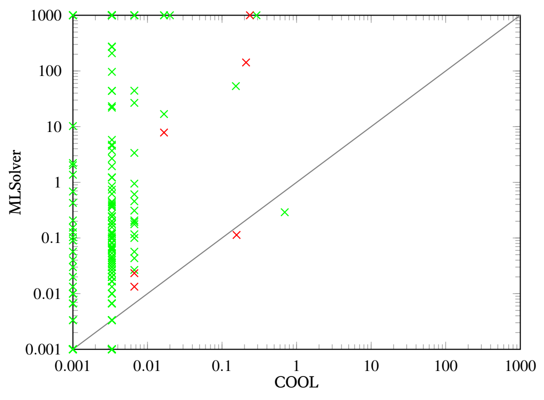
As can be seen in figure 6.5a COOL finishes for all the formulae in less than 0.01 seconds. TATL only solved 28 of them in less than 0.1 seconds and 120 in less than one second. 22 formulae took longer than 10 seconds for TATL.

6.3 Alternation-free μ-calculus

No other reasoner is known to support the alternation-free fragment of the μ-calculus. MLSolver actually supports the full μ-calculus and therefore is the only competition to COOL when



(a) three propositional atoms



(b) ten propositional atoms

Figure 6.6: *random* formulae from the relational μ-calculus of size 250

considering the alternation-free Fragment. For measuring the performance of COOL and ML-Solver, random formulae from the alternation-free fragment have been created.

6.3.1 *random* Formulae

For evaluation of the alternation-free μ-calculus we primarily used a set of randomly generated formulae. The construction of formulae was similar to the one used for the random CTL formulae. All formulae contain exactly 250 instances of logic constructors (from $\vee$, $\wedge$, $\Box$, $\Diamond$, μ and ν where $\vee$ and $\wedge$ were twice as likely as the other constructs). The formulae were constructed by choosing a random constructor and placing it in one of the free slots of the to-be constructed formula. All slots still empty at the end were filled with either one of the propositional atoms in positive or negated form or a fixpoint variable valid at this point according to the rules in 5.1.3. The experiment was conducted with three and ten propositional atoms.

The number of constructors was chosen to have MLSolver complete reasoning on most of them within the memory boundaries imposed. Of the 250 formulae with ten atoms, MLSolver was unable to solve 40 within the bounds of the experiment, all due to memory exhaustion while COOL could answer all queries. On the set with three atoms, MLSolver failed to answer 55 queries within the bounds, here two were timeouts while 53 were due to memory limits again. COOL again could answer all queries within the bounds.

In the set of formulae with three atoms, MLSolver finished 94 in less than 0.01 seconds and 130 in less than 0.1 second. Only 36 took longer than 1 second. MLSolver performed comparably on the formulae with ten atoms finishing in less than 0.01 seconds for 98 formulae and in less than 0.1 seconds for 145 formulae while 22 formulae took longer than one seconds. COOL needed more than 0.01 seconds for 12 of the formulae with ten atoms and 17 of the formulae with three atoms.

In conclusion it is fair to say that COOL, while only supporting a fragment of the μ-calculus implemented in MLSolver, is consistently faster than MLSolver on that fragment as far as this can be derived from comparing random formulae.

7 Optimization

While COOL without further optimizations already provides decent performance, a number of optimizations can be made to improve the speed further. Most of the optimizations described here are further work and cannot be evaluated. However some optimizations are already implemented in COOL and in these cases an evaluation is included. We also show the performance of GMUL and TreeTab, the other main tableaux solvers, in comparison to give an impression of state of the art performance.

7.1 When to propagate

As we have seen earlier, during the main reasoner loop, COOL has two possible actions it can execute. It can either expand additional nodes or execute the propagation algorithm. COOL needs to execute the propagation algorithm regularly as this is the step where success of the formula is decided and propagation may cause COOL to finish early. However, propagation is an expensive operation with runtime complexity cubic in the number of reasoner nodes. Therefore a good balance has to be found between running propagation often enough to profit from finishing early and not spending too much of the time in the propagation step.

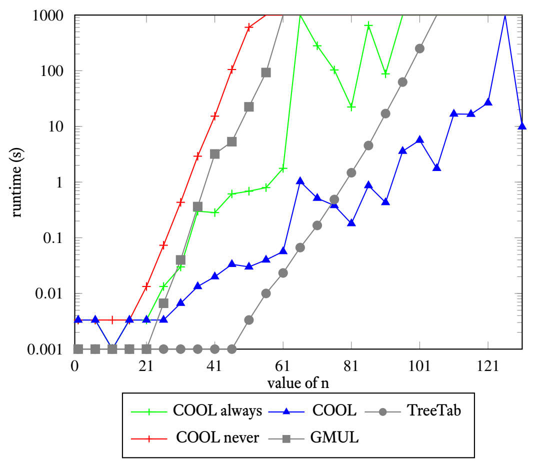
The current implementation runs the propagation step after a number of waves proportional to the number of currently open state nodes in the reasoner graph. In Figure 7.1, results for this strategy are displayed in blue. The red values show the runtime of COOL when it is instructed to postpone the propagation step until the full reasoner graph has been constructed while the green values show the behavior if propagation is done after every wave of expansions.

One extreme example in terms of propagation are the *exp_sat* formulae from Subsection 6.1.3. Here the full reasoner graph needs to be constructed before propagation can come to any conclusion. This setting favors simpler algorithms which expand the reasoner graph quickly. As can be seen in Figure 7.1b, COOL performs comparably to other state of the art solvers if it is instructed to never run the propagation step until the graph is fully constructed – the optimal strategy for these formulae – while the versions which run the propagation step regularly are performing significantly worse.

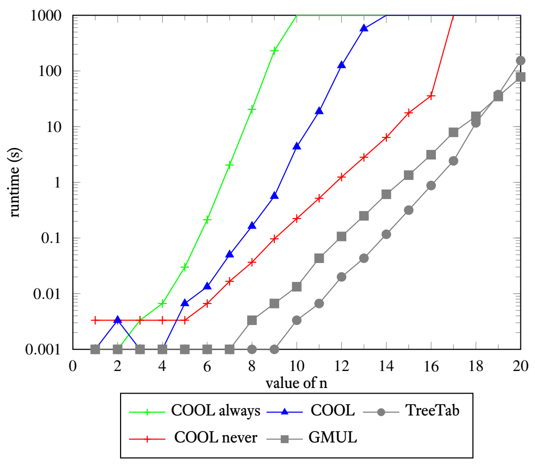
For the remaining examples, the currently implemented strategy performs consistently better than the simple version which propagates after every wave, and significantly better than the version which postpones propagation to the end. Both strategies which include intermediate propagation outperform the other tableaux reasoners on the *early* formulae and asymptotically conjecturedly also on the *montali* example where the current propagation strategy clearly performs better than any existing tableaux solver.

Note that in both, Figure 7.1c and Figure 7.1b, the version of COOL which performs full expansion before attempting propagation fails due to the memory limit and cannot fully use the allocated time. The current implementation of COOL is relatively demanding on memory. A discussion of this problem can be found in Section 7.6 where I consider some improvements in this area as well.

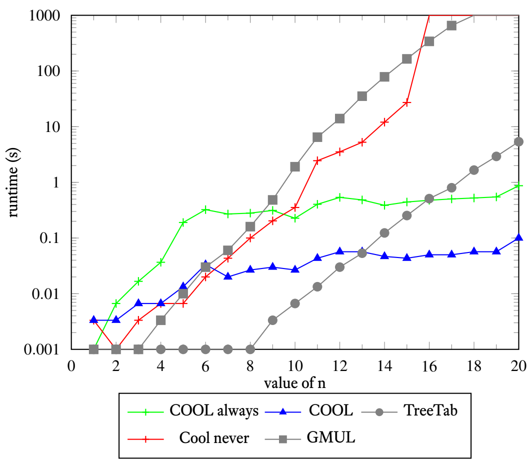
While the current propagation strategy is already relatively satisfying, there is at least one further improvement that can be implemented. Propagation can only find new locally successful



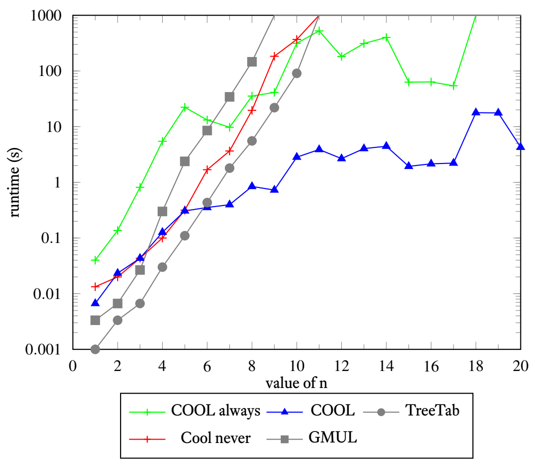
(a) montali_sat1



(b) exp_sat



(c) early



(d) early_gc

Figure 7.1: Different propagation heuristics

nodes if an internally successful node has been created and identified or a node expansion got references to nodes already known as (un-)successful. Globally successful nodes can only be found if there is at least one cycle in the graph.

Consequently there are two heuristics one can add. As long as there are no cycles in the reasoner graph, propagation should only be attempted if an internally successful node has been created or node expansion resulted in a node already present in the graph (and marked as (un-)successful there). Secondly, every time a cycle (or just any edge back into the reasoner graph) is created, this serves as a strong indication that propagation may find more successful nodes. One does, however, want to wait for some time after such an edge has been created to allow all nodes which are part of the cycle to be sufficiently expanded for the propagation algorithm to succeed.

7.2 Local propagation

COOL as implemented only verifies the (un-)satisfiability of nodes using the fixpoint computations for global satisfiability described in Subsection 5.4.2 and 5.4.3. As we have already noted in the previous section, this propagation is an expensive operation. The full propagation algorithm, however, is only necessary to find globally successful nodes while locally successful nodes can be identified in a simpler algorithm.

Both, local successfulness and local unsuccessfulness can be propagated using a depth-first search on the reasoner graph visiting all open nodes (and potentially some of the expandable nodes as detailed in Subsection 5.4.2). At every visited node the local success condition is checked and if no conclusion can be drawn yet, the children are processed. When returning from the children the local condition is then reevaluated. As this algorithm is only quadratic in the number of nodes, local propagation can be performed more often and used as a first step in the global propagation algorithm.

Additionally this local propagation algorithm can collect only these successful nodes that have open or expandable parent nodes. Global propagation can then use only these successful nodes instead of all successful nodes in the graph thereby reducing the complexity of this algorithm as well.

7.3 Removing edges

Improving on the last optimization one observes, that some edges to (un-)successful nodes become superfluous for further propagation steps. As cores are $\exists$-quantified, an unsuccessful child state does not contribute anything to deciding the success of a core. As a result, these edges can be removed without changing the semantics of that core. Similarly, states are $\forall\exists$-quantified. Therefore a rule application with at least one successful child core does not contribute to the result of that state and the edge to this child can be safely removed.

Any subgraphs that become disconnected from the main graph due to the removal of edges are at that point not needed any more for the reasoner. Consequently any expandable nodes in such a subgraph no longer needs to be expanded and the nodes in there do not need to be considered when running one of the propagation algorithms. However, the full subgraph should be kept as – due to the global caching structure of the algorithm – new edges may be created connecting such a subgraph again and all inferred information on its nodes need to be still present in this case.

Implementing edge removal fits well into the local propagation algorithm described previously. That algorithm already visits all nodes reachable from the initial core and can reconstruct the set of reachable, expandable nodes for the next waves as well as remove all the edges that are no longer needed as part of its recursion.

7.4 Variable renaming

As discussed in Subsection 5.1.1, the way COOL currently handles variable names is not optimal. Preferably any subformula ψ – even if it contains fixpoint variables – would get the same representation in hash-consed form independent from its location in the input formula. Additionally negations should be marked properly, i. e. $EF\ \psi$ should be marked as the negation of $\neg EF\ \psi = AG\ \neg\psi$. As the hash-consing module already assigns an unique integer to each subformula, as part of the implementation, the transformation into hash-consed form can perform a deterministic variable renaming based on this integer and – as it always has the representation of $\neg\psi$ when constructing the fixpoint literal $\eta X.\psi$ – annotate the negations as part of this process.

7.5 Semantic branching

Semantic branching is a common optimization for reasoners which can already be found in propositional reasoners and has been successfully implemented in GMUL. Given a formula $\phi \vee \psi$, when exploring ψ, we do not need to consider paths subsumed in ϕ as these would already satisfy the left-hand side of the disjunction. This will, in several cases, limit the search-space of the right-hand side to options not already considered when expanding the left-hand side. This can be implemented by changing the tableaux rule for disjunction:

$$\frac{\phi \vee \psi}{\phi \qquad \psi} \Rightarrow \frac{\phi \vee \psi}{\phi \qquad \neg\phi \wedge \psi}$$

Remark 7.1. Implementing semantic branching may increase the search space. Consider the disjunction $\phi \vee \psi$ where ϕ is a complex subformula and ψ is much simpler. While the reasoner in both cases has to consider ϕ when processing the left-hand side of the disjunct, the second tableau rule now generates $\neg\phi \wedge \psi$ where the reasoner has to keep track of ϕ again. Semantic branching is however still assumed to be beneficial for, at least, CTL formulae and worthwhile to evaluate with the alternation-free μ-calculus in COOL.

7.6 Memory usage

As we have seen previously, COOL needs a large amount of memory for its operation. In several cases this can amount to about 50 times the memory usage expected when examining the abstract algorithm. The dominant cause for this memory consumption is due to the way the external satisfiability solver, minisat, is used in COOL. COOL needs a fully minisat instance at every expandable core to generate additional child states in every wave. This minisat however is also kept when the core enters the open state and can no longer be expanded. This is a result of an optimization in old COOL called backjumping. Deallocating minisat when a core reaches the open state will reduce the memory footprint of COOL by about 60 % for certain sets of formulae (i. e. the *exp* formulae we discussed earlier). Further work in this regard will evaluate whether to reintegrate the old optimization (i. e. backjumping) into local propagation or properly remove the late usage of minisat from COOL when doing fixpoint reasoning.

7.7 Additional normalization

The preprocessing of formulae in COOL performs some normalization steps on the inputs when creating negation normal form. However some basic normalization steps are still missing. COOL does not consider commutativity or associativity when comparing formulae.

For commutativity a formula $\phi \vee \psi$ should be reordered such that the subformula with the smaller integer assigned by hash-consing appears first and both $\phi \vee \psi$ and $\psi \vee \phi$ result in the same hash-consed node. $\wedge$ can be handled in the same way.

Associativity can either be implemented by replacing the binary $\vee$ and $\wedge$ nodes by nodes taking a set of child nodes and transforming a formula of the fom $(\phi_1 \vee \phi_2) \vee \phi_3$ to the form $\vee\{\psi_1, \psi_2, \psi_3\}$ or by normalizing to some form $\psi_1 \vee (\psi_2 \vee (\psi_3 \vee \ldots))$ with increasing hash-conse numbers. The latter option, while more complex in general, fits considerably better into the reasoner design with the array representation.

7.8 Subset Nodes

Consider a reasoner node n with a set of formulae s and a focus f which is marked as successful. A second reasoner node n' with formulae s' and focus f' can already be marked as successful iff $s' \subseteq s$ and $f' \subseteq f$ as n places a superset of restrictions on the model compared to n'. The dual result holds for unsuccessful nodes.

It is generally assumed, that finding arbitrary subset nodes is too expensive to make this interesting as a general purpose optimization. However within the global caching algorithm described here, there are two special cases which might be interesting to explore. If we have an fully focused node n (that is, $f = s$) which is already marked as successful and a node n' which has the same set of formulae s as n but a smaller focus, n' can be marked as successful. This works similarly for fully focused, unsuccessful nodes.

Finding these unfocused (or fully focused) nodes is relatively cheap as one needs to only find one specific node in a hashtable, an implementation of this optimization will need to ensure that such nodes are created regularly. If so, one can then try to add some heuristics to further improve the results such as delaying further consideration of a node n' if their unfocused (or fully focused) relative exists but has not been marked as (un-)successful.

8 Conclusion

In this thesis we have described an extension to COOL, a reasoner for coalgebraic modal logics, to allow certain fixpoint constructs: The alternation-free coalgebraic μ-calculus. This extension covers several common modal logics like **CTL**, **ATL**, **PDL** and fragments of Parikh's game logic.

COOL is the first and only reasoner to support several of these logics. For several logics where other reasoners exist, like **ATL** and the relational μ-calculus, COOL provides significant performance improvements compared to the existing tools. In fact, COOL performs comparable to the best known tableaux reasoners for **CTL**, a logic where several optimized reasoners exist and is able to provide this level of performance to all supported logics.

The implementation in COOL is fully generic to the coalgebraic nature of COOL and thus easily extensible to any other modal logic which fits this framework making COOL the obvious choice when implementing an optimized reasoner for a new (coalgebraic) fixpoint logic.

Bibliography

[AGW07] Pietro Abate, Rajeev Goré, and Florian Widmann. "One-Pass Tableaux for Computation Tree Logic". In: *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 07*. LNCS. Springer, 2007, pp. 32–46.

[AHK02] R. Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-time temporal logic". In: *J. ACM* 49 (2002), pp. 672–713.

[BCLMD94] Jerry R Burch, Edmund M Clarke, David E Long, Kenneth L McMillan, and David L Dill. "Symbolic model checking for sequential circuit verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (1994), pp. 401–424.

[BPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. "The temporal logic of branching time". In: *Acta informatica* 20.3 (1983), pp. 207–226.

[CKP09] Corina Cîrstea, Clemens Kupke, and Dirk Pattinson. "EXPTIME Tableaux for the Coalgebraic μ-Calculus". In: *Computer Science Logic, CSL 2009*. Vol. 5771. LNCS. Springer, 2009, pp. 179–193.

[Dav13] Amélie David. "TATL: Implementation of ATL tableau-based decision procedure". In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2013, pp. 97–103.

[ES03] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *Theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[FL10] Oliver Friedmann and Martin Lange. "A solver for modal fixpoint logics". In: *Electronic Notes in Theoretical Computer Science* 262 (2010), pp. 99–111.

[FL11] Oliver Friedmann and Martin Lange. "The Modal μ-Calculus Caught Off Guard". In: *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2011*. Vol. 6793. LNCS. Springer, 2011, pp. 149–163.

[FL79] M. Fischer and R. Ladner. "Propositional dynamic logic of regular programs". In: *J. Comput. Sys. Sci.* 18 (1979), pp. 194–211.

[GN07a] R. Goré and L. Nguyen. "EXPTIME Tableaux for $\mathcal{ALC}$ Using Sound Global Caching". In: *Description Logics, DL 2007*. Vol. 250. CEUR Workshop Proc. 2007.

[GN07b] R. Goré and L. Nguyen. "EXPTIME Tableaux with Global Caching for Description Logics with Transitive Roles, Inverse Roles and Role Hierarchies". In: *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2007*. Vol. 4548. LNCS. Springer, 2007, pp. 133–148.

[GPSWW14] Daniel Gorín, Dirk Pattinson, Lutz Schröder, Florian Widmann, and Thorsten Wißmann. "COOL–a generic reasoner for coalgebraic hybrid logics (system description)". In: *Automated Reasoning*. Springer, 2014, pp. 396–402.

[GS14] Daniel Gorín and Lutz Schröder. "Subsumption Checking in Conjunctive Coalgebraic Fixpoint Logics". In: *Proc. Advances in Modal Logic, AiML 2014*. Ed. by Rajeev Goré, Barteld Kooi, and Agi Kurucz. College Publications, 2014, pp. 254–273.

[GTW11] Rajeev Goré, Jimmy Thomson, and Florian Widmann. "An experimental comparison of theorem provers for CTL". In: *2011 Eighteenth International Symposium on Temporal Representation and Reasoning*. IEEE. 2011, pp. 49–56.

[HS15] Daniel Hausmann and Lutz Schroder. "Global Caching for the Flat Coalgebraic μ-Calculus". In: *Temporal Representation and Reasoning (TIME), 2015 22nd International Symposium on*. IEEE. 2015, pp. 121–130.

[HSE16] Daniel Hausmann, Lutz Schröder, and Christoph Egger. "Global Caching for the Alternation-free Coalgebraic μ-Calculus". In: *27th International Conference on Concurrency Theory (CONCUR 2016)*. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, to appear.

[Koz83] D. Kozen. "Results on the propositional μ-calculus". In: *Theoret. Comput. Sci.* 27 (1983), pp. 333–354.

[Koz88] Dexter Kozen. "A finite model theorem for the propositional μ-calculus". In: *Stud. Log.* 47 (1988), pp. 233–241.

[Mar05] Will Marrero. "Using BDDs to decide CTL". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 222–236.

[Mon+08] Marco Montali, Paolo Torroni, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello. "Verification from declarative specifications using logic programming". In: *Logic Programming*. Springer, 2008, pp. 440–454.

[Par83] Rohit Parikh. "Propositional Game Logic". In: *Foundations of Computer Science, FOCS 1983*. IEEE Computer Society, 1983.

[Par85] R. Parikh. "The logic of games and its applications". In: *Ann. Discr. Math.* 24 (1985), pp. 111–140.

[Pat03] Dirk Pattinson. "Coalgebraic modal logic: Soundness, completeness and decidability of local consequence". In: *Theoretical Computer Science* 309.1 (2003), pp. 177–193.

[Sch08] Sven Schewe. "Synthesis of distributed systems". PhD thesis. Universität des Saarlands, 2008.

[SP09] Lutz Schröder and Dirk Pattinson. "PSPACE Bounds for Rank-1 Modal Logics". In: *ACM Trans. Comput. Log.* 10 (2009), 13:1–13:33.

[ZHD09] Lan Zhang, Ullrich Hustadt, and Clare Dixon. "A refined resolution calculus for CTL". In: *Automated Deduction–CADE-22*. Springer, 2009, pp. 245–260.

List of Definitions

A Selected code samples

A.1 Verification of Properties

Listing A.1: verifyMuAltFree

```
let rec verifyMuAltFree formula =
  let proc = verifyMuAltFree in
  match formula with
  | EX (_,a) | AX (_,a) -> proc a
  | OR (a,b) | AND (a,b) ->
     let (atype, afree) = proc a
     and (btype, bfree) = proc b in
     if (compare atype "μ" == 0 && compare btype "ν" == 0) ||
          (compare atype "ν" == 0 && compare btype "μ" == 0) then
       raise (CoAlgException ("formula␣not␣alternation-free"));
     if compare atype "none" == 0 then
       (btype, List.flatten [afree; bfree])
     else
       (atype, List.flatten [afree; bfree])
  | MU (s, f) ->
     let (fptype, free) = proc f in
     (if (compare fptype "ν" == 0) then
        raise (CoAlgException ("formula␣not␣alternation-free")));
     let predicate x = compare x s != 0 in
     let newfree = List.filter predicate free in
     if newfree = [] then
       ("none", [])
     else
       ("μ", newfree)
  | NU (s, f) ->
     let (fptype, free) = proc f in
     (if (compare fptype "μ" == 0) then
        raise (CoAlgException ("formula␣not␣alternation-free")));
     let predicate x = compare x s != 0 in
     let newfree = List.filter predicate free in
     if newfree = [] then
       ("none", [])
     else
       ("ν", newfree)
  | VAR s -> ("none", [s])
  | _ -> (* base case: handle recursion *)
```

Listing A.2: verifyMuPositive

```
let rec verifyMuMonotone negations formula =
  let proc = verifyMuMonotone negations in
  match formula with
  | MU (s, f)
  | NU (s, f) ->
     let newNeg = (s, 0) :: negations in
     verifyMuMonotone newNeg f
  | VAR s ->
     let finder (x, _) = compare x s == 0 in
     let (_, neg) = List.find finder negations in
     if ((neg land 1) != 0) then raise (CoAlgException ("formula␣not␣monotone"))
  | NOT a ->
     let increment (s, n) = (s, n+1) in
     let newNeg = List.map increment negations in
     verifyMuMonotone newNeg a
  | _ -> (* base case: handle recursion *)
```

Listing A.3: verifyMuGuarded

```
let rec verifyMuGuarded unguarded formula =
  let proc = verifyMuGuarded unguarded in
  match formula with
  | MU (s, f)
  | NU (s, f) ->
     let newUnguard = s :: unguarded in
     verifyMuGuarded newUnguard f
  | VAR s ->
     let finder x = compare x s == 0 in
     if List.exists finder unguarded then
       raise (CoAlgException ("formula␣not␣guarded"))
  | _ -> (* base case: handle recursion *)
```

A.2 Deferral tracking

Listing A.4: mkRuleK

```
(** directly return a list of rules **)
let mkRuleList_MultiModalK sort bs defer sl : rule list =
  assert (List.length sl = 1);
  let refocusing = bsetCompare (bsetMakeRealEmpty ()) defer = 0 in
  let getRules f acc =
    if lfGetType sort f = ExF then (* f = ∃R.C,i.e. a diamond *)
      let bs1 = bsetMake () in
      let defer1 = bsetMakeRealEmpty () in
      let nextf = (lfGetDest1 sort f) in
      bsetAdd bs1 nextf; (* bs1 := { C }          *)
      if (refocusing && (lfGetDeferral sort nextf) != (Hashtbl.hash "ε")) ||
           ((bsetMem defer f) && (lfGetDeferral sort f) = (lfGetDeferral sort nextf))
      then
        bsetAdd defer1 nextf;
      let (role : int) = lfGetDest3 sort f in (* role := R *)
      let filterFkt f1 =
        if lfGetType sort f1 = AxF && lfGetDest3 sort f1 = role then
          (* if f1 = ∀R.D then bs1 = bs1  { D } *)
          let nextf1 = (lfGetDest1 sort f1) in
          bsetAdd bs1 nextf1;
          if (refocusing && (lfGetDeferral sort nextf1) != (Hashtbl.hash "ε")) ||
               ((bsetMem defer f1) &&
                  (lfGetDeferral sort f1) = (lfGetDeferral sort nextf1)) then
            bsetAdd defer1 nextf1
          else ()
        else ()
      in
      bsetIter filterFkt bs; (* bs1 := bs1  { D | some ∀"R.D" ∈ bs } *)
      let s1 = List.hd sl in (* [s1] := sl *)
      let rle = (dep f, lazylistFromList [(s1, bs1, defer1)]) in
      rle::acc
    else acc
  in
  bsetFold getRules bs []
```

A.3 Successfulness checking

Listing A.5: propagateSatMu

```
let propagateSatMu () =
  let setFinishingStates = setEmptyState () in
  let setFinishingCores = setEmptyCore () in
  let setStates = setEmptyState () in
  let setCores = setEmptyCore () in
  let emptySet = bsetMakeRealEmpty () in
  let openstates = ref 0 in

  (* Collect two sets of nodes. All nodes that may be satisfiable
   * after this iteration are collected into setStates/setCores.
   *
   * As every cycle containing a node with empty focus or an
   * satisfiable node should be considered satisfiable, collect these
   * decisive nodes into setFinishingStates/setFinishingCores
   *
   * This also marks in trivial cases nodes as satisfiable.
   *)
  let stateCollector state =
    match stateGetStatus state with
    | Unsat -> ()
    | Sat ->
       setAddState setStates state;
      setAddState setFinishingStates state
    | Expandable -> ()
    | Open ->
       openstates := !openstates + 1;
       (* States with no rules are satisfiable *)
       if List.length (stateGetRules state) == 0 ||
            (* KD generates nodes with just True as formula *)
         bsetCompare (bsetMake ()) (stateGetBs state) == 0
       then begin
         setAddState setFinishingStates state;
         stateSetStatus state Sat
       end else begin
         setAddState setStates state;
         if bsetCompare (stateGetDeferral state) emptySet == 0
         then begin
           setAddState setFinishingStates state
         end
         else ()
       end

  (* As it is enough for a core to have one successfull child, we can
   * also handle (some) expandable cores.
   *)
  and coreCollector core =
    match coreGetStatus core with
    | Unsat -> ()
    | Sat ->
       setAddCore setCores core;
       setAddCore setFinishingCores core
    | Expandable
    | Open ->
       setAddCore setCores core;
      if bsetCompare (coreGetDeferral core) emptySet == 0
      then begin
        setAddCore setFinishingCores core
      end
      else ()
  in
  graphIterStates stateCollector;
  graphIterCores coreCollector;

  setPropagationCounter !openstates;

  (* In a fixpoint the set called setStates / setCores is narrowed
   * down.
   *
   * In each step only those states and cores are retained in setStates
   * / setCores which reach one of setFinishing{States,Cores} in
   * finitely many steps. This new set of States / Cores is collected
   * as allowed{States,Cores} during each fixpoint iteration.
   *
   * Only those finishing nodes are retained that have allowed or
   * Successfull Children.
   *)
  let rec fixpointstep setStates setCores =
    let allowedStates = setEmptyState () in
    let allowedCores = setEmptyCore () in

    let rec visitParentStates (core : core) : unit =
      if not (setMemCore allowedCores core)
      then begin
        setAddCore allowedCores core;
        let verifyParent (state,_) =
          let rules = stateGetRules state in
          let ruleiter (dependencies, corelist) =
            List.exists (fun (core : core) -> setMemCore allowedCores core ||
                                              coreGetStatus core == Sat)
              corelist
          in
          if List.for_all ruleiter rules
          then visitParentCores state
        in
        List.iter verifyParent (coreGetParents core)
      end

    and visitParentCores (state : state) : unit =
      if not (setMemState allowedStates state)
      then begin
        setAddState allowedStates state;
        let verifyParent core =
          let acceptable =
            List.exists (fun (state : state) -> setMemState allowedStates state ||
                                                stateGetStatus state == Sat)
              (coreGetChildren core)
          in
          if acceptable
          then visitParentStates core
        in
        List.iter verifyParent (stateGetParents state)
      end
    in

    (* All rule applications need to still be potentially Sat for a
     * finishing State to be a valid startingpoint for this fixpoint.
     *)
    let checkFinishingState (state : state) =
      let ruleiter (dependencies, corelist) : bool =
        List.for_all (fun (core : core) ->
                        coreGetStatus core == Unsat ||
                          coreGetStatus core == Expandable ||
                            not (setMemCore setCores core)) corelist
      in
      if not (List.exists ruleiter (stateGetRules state)) then begin
        visitParentCores state
      end

    (* There needs to be a State still potentially Sat for this core
     * to be considered for the fixpoint
     *)
    and checkFinishingCore (core : core) =
      if not (List.for_all (fun (state : state) ->
                              stateGetStatus state == Unsat ||
                                stateGetStatus state == Expandable ||
                                  not (setMemState setStates state))
                            (coreGetChildren core))
      then begin
        visitParentStates core
      end
    in
    setIterState checkFinishingState setFinishingStates;
    setIterCore checkFinishingCore setFinishingCores;


    if (setLengthState setStates) = (setLengthState allowedStates) &&
         (setLengthCore setCores) = (setLengthCore allowedCores)
    then begin
        setIterState (fun state -> stateSetStatus state Sat) setStates;
        setIterCore (fun core -> coreSetStatus core Sat;
                                 if core == graphGetRoot ()
                                 then raise (CoAlg_finished true)
                                 else ()) setCores;
      end else
      fixpointstep allowedStates allowedCores
  in
  fixpointstep setStates setCores
```