Towards Combining Model Checking and Proof Checking

YING JIANG[1], JIAN LIU[2], GILLES DOWEK[3] AND KAILIANG JI[4]

[1]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, 100190 Beijing, China*
[2]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences, 100190 Beijing, China*
[3]*Inria and ENS de Cachan, 61, avenue du Président Wilson 94235 Cachan cedex, Paris, France*
[4]*University Paris Diderot, 8 Place Aurélie Nemours, 75013 Paris, France*
Email: jy@ios.ac.cn, liujian@ios.ac.cn, gilles.dowek@ens-cachan.fr, jkl@irif.fr

Model checking and automated theorem proving are two pillars of formal verification methods. This paper investigates model checking from an automated theorem proving perspective, aiming at combining the expressiveness of automated theorem proving and the complete automaticity of model checking. It places the focus on the verification of temporal logic properties of Kripke models. The main contributions are: (1) introducing an extended computation tree logic that allows polyadic predicate symbols; (2) designing a proof system for this logic, taking Kripke models as parameters; (3) developing a proof search algorithm for this system and a new automated theorem prover to implement it. The verification process of the new prover is completely automatic, and produces either a counterexample when the property does not hold, or a certificate when it does. The experimental results compare well to existing state-of-the-art tools on some benchmarks, and the efficiency is illustrated by application to an air traffic control problem.

Keywords: CTL Model checking; Automated theorem proving; Continuation-passing style; Doubly on-the-fly style

1. INTRODUCTION

Model checking [1, 2, 3] and automated theorem proving [4, 5, 6] are two pillars of formal verification methods. They differ by the fact that model checking often uses decidable logics, such as propositional modal logics, while automated theorem proving mostly uses undecidable ones, such as first-order logic. Nevertheless, model checking and automated theorem proving have a lot in common, in particular, both of them are often based on a recursive decomposition of problems, through the application of rules.

This paper investigates model checking from an automated theorem proving perspective, aiming at combining the expressiveness of automated theorem proving and the complete automaticity of model checking.

The first contribution of this paper is to propose a slight extension of Computation Tree Logic (CTL) [7, 8], called CTL_P. In this extension, we may refer explicitly to states of the model under consideration. For instance, the proposition $P(s)$ expresses what is usually expressed with the judgment $s \models P$. Thus P here is not a proposition symbol, but a unary predicate symbol. This transformation can be compared to the introduction of adverbial phrases in natural languages, where we can say not only "The sky will be blue in the future" but also "The sky will be blue on Monday". A proposition such as $EX(P)(s)$ must then be written as $EX_x(P(x))(s)$. Indeed, as the symbol P is now a unary predicate symbol, it must be applied to a variable, which is bound by the modality EX. This allows to introduce polyadic predicates that do not only express properties of states, but also relations between states. For instance, we can express the existence of a sequence of states $s_0, s_1, \ldots$ such that each s_{i+1} occurs after s_i and that one can buy a left shoe at some state s_m and then the right shoe of the same pair at a later state s_n. This property is expressed by the formula $EF_x(EF_y(P(x,y))(x))(s_0)$ in CTL_P.

The second contribution of this paper is to propose a sequent-calculus-like proof system for CTL_P, called simply SCTL. The proof search in SCTL coincides with checking the validity of properties described by CTL_P formulae in a Kripke model. We prove that SCTL is sound and complete and the proof search in this system always terminates.

When designing such a proof system, one of the main issues is to handle co-inductive modalities, for instance, asserting the existence of an infinite sequence of which all elements satisfy some property. It is tempting to reflect this infinite sequence as an infinite proof and then use the finiteness of the model to prune the search-tree in a proof search method. Instead, we use the finiteness of the model to keep our proofs finite, like in the usual sequent calculus. This is the purpose of the *merge* rules of SCTL (Figure 4). However, the way we use sequents in SCTL is a bit unusual: as we prove a sequent of the form $\vdash A \Rightarrow B$ by proving $\vdash B$ or by proving $\vdash \neg A$, and not by proving $A \vdash B$, we do not need hypotheses to prove implications. So we use the left hand

side of the sequent to store visited states for co-inductive modalities.

SCTL enjoys several advantages. For example, when a given property holds in a Kripke model, it permits to give a certificate (a proof tree) for the property. Such a certificate can be further verified by an independent proof checker, increasing the confidence in the proved property, and can also be combined with proofs built by other means. Conversely, when a given property does not hold, it permits to generate a counterexample as a proof of the negation of the property, instead of a sequence of states as in traditional model checkers. In particular, when providing a counterexample for a formula containing nested modalities, such as $EG_x(EG_y(P(x,y))(x))(s)$, we need to provide a finite tree labeled with states, in such a way that for each state a labeling a leaf of this tree, the formula $EG_y(P(a,y))(a))$ does not hold. Thus, for each of these states, we need to provide another tree. As we shall see, such a hierarchical tree can be represented as a proof of the formula $AF_x(AF_y(\neg P(x,y))(x))(s)$ in SCTL.

The third contribution of this paper is an implementation of SCTL. Instead of translating temporal formulae to Quantified Boolean Formulae (QBFs) [9] or to the format of an existing theorem prover [10], we develop a new automated theorem prover tailored for SCTL (SCTLProV[5]), in the programming language OCaml.[6] Designing our own system permits us to combine the advantages of model checking and automated theorem proving, and gives us a lot of freedom to optimize it. For example, SCTLProV adopts several techniques, such as global variable storage or storage based on Binary Decision Diagram (BDD), which are commonly used in traditional model checkers, but cannot be realized in the usual theorem provers like iProver Modulo. On the other hand, theorem provers usually output proof trees as a diagnosis of the system under verification, while in traditional model checkers, only sequences of states representing the counterexamples can be produced. One inheritance from traditional theorem provers is that SCTLProV produces proof trees in any case. Another inheritance is that, in SCTLProV, the Kripke model is not always defined extensionally with a list of states and a list of transitions, but the type of states can be any datatype and the transitions can be defined by an arbitrary computable function mapping a state to the list of its successors. Therefore SCTLProV provides a more expressive input language than most traditional model checkers. This choice allows us to tackle, for instance, the *Small Aircraft Transportation System* (SATS) [11, 12], that is often difficult to solve for model checkers using a more extensional description of their input.

To illustrate the efficiency of SCTLProV, we compared the experimental data of SCTLProV, on several benchmarks, with the following tools: an automated theorem prover iProver Modulo[7], a QBF-based bounded model checker Verds[8], two BDD-based symbolic model checkers NuSMV[9] and NuXMV[10], and the verification toolbox CADP[11] for action based models. The experimental results show that SCTLProV compares well with these tools.

The reason for selecting the tools above-mentioned is: firstly, the focus of this work is on solving CTL model checking problems while, as far as we know, NuSMV and NuXMV are the most successful model checkers up-to-date in this area; secondly, benchmark #1 in this paper is originally proposed by Zhang [9] in order to compare the bounded model checker Verds with NuSMV, and reused by Ji [10] to compare the theorem prover iProver Modulo with Verds; finally, as one of the state-of-the-art explicit model checkers, CADP is widely used both in industrial and academic fields.

The efficiency of SCTLProV depends on the following design choices: the first is that, unlike traditional symbolic model checkers or bounded model checkers, SCTLProV searches states in a doubly on-the-fly style (both the transition relation and the formula are unfolded on-the-fly) [13, 14]. Thus, the state space is usually not needed to be fully generated. This avoids enumerating unneeded states during the verification procedure. The second is that, unlike traditional on-the-fly model checking algorithms for CTL [13, 14], our proof search algorithm is in continuation-passing style [15] which permits to reduce stack operations.

Links between model checking and automated theorem proving have been investigated for long, for instance, Bounded Model Checking (BMC) [16, 9, 17] is based on a reduction of model checking to satisfiability of Boolean or quantified Boolean formulae. In the literature, different proof systems for temporal logic have been proposed [8, 18, 19, 20, 21, 22]. Proof systems for temporal logics usually deal with the validity problem [8, 18, 22], where a temporal formula is provable if and only if it holds in all models. However, the proof system SCTL is parameterized by Kripke models, in the sense that, for any given Kripke model $\mathcal{M}$, a CTL_P formula is provable in SCTL if and only if it holds in $\mathcal{M}$.

The rest of the paper is organized as follows. In Section 2, we introduce the logic CTL_P. In Section 3, we introduce the proof system SCTL. In Section 4, first we present the proof search algorithm for SCTL and the complete automatic prover SCTLProV which is an implementation of SCTL; then the design choices of SCTLProV will be discussed in more detail. In Section 5, we deal with the verification of properties under fairness constraints in SCTL. In Section 6, first we compare, on several benchmarks, SCTLProV with iProver Modulo, Verds, NuSMV, NuXMV, and CADP, respectively; then we illustrate the application of SCTLProV by modeling and analyzing SATS and verifying its safety property. Appendix A describes in detail the pseudo code of the proof search algorithm. Appendix B shows the details of the experimental data for benchmark #1,

[5] https://github.com/sctlprov/sctlprov_code
[6] http://ocaml.org/
[7] http://www.ensiie.fr/~guillaume.burel/blackandwhite_iProverModulo.html.en
[8] http://lcs.ios.ac.cn/~zwh/verds/index.html
[9] http://nusmv.fbk.eu/
[10] https://nuxmv.fbk.eu/
[11] http://cadp.inria.fr/

#2, #3, and #4. Appendix C shows the detailed proofs for the soundness and completeness of the proof system SCTL. Appendix D shows the detailed proof for the correctness of the proof search method.

2. CTL_P

In this section, we present the logic $\mathsf{CTL}_P(\mathcal{M})$ taking a Kripke model $\mathcal{M}$ as the parameter.

DEFINITION 2.1 (Kripke model). *A Kripke model $\mathcal{M}$ is given by*

- *a finite non-empty set S, whose elements are called states,*
- *a binary relation $\longrightarrow$ defined on S, such that for each s in S, there exists a finite number of elements and at least one element s' in S, such that $s \longrightarrow s'$,*
- *and a family of relations, each being a subset of S^n for some natural number n.*

We write $\mathsf{Next}(s)$ for the set $\{s' \in S \mid s \longrightarrow s'\}$ which is always finite and not empty. A *path* is a finite or infinite sequence of states $s_0,...,s_n$ or $s_0,s_1,...$ such that for each i, if s_i is not the last element of the sequence, then $s_{i+1} \in \mathsf{Next}(s_i)$. A *path-tree* is a finite or infinite tree labeled by states such that for each internal node labeled by a state s, the children of this node are labeled by exactly the elements of $\mathsf{Next}(s)$.

Properties of such a model are expressed in a language tailored for this model that contains a constant for each state s, also written as s; and a predicate symbol for each relation P, also written as P.

The grammar of $\mathsf{CTL}_P(\mathcal{M})$ formulae is displayed below:

$$\phi := \left\{ \begin{array}{l} \top \mid \bot \mid P(t_1,...,t_n) \mid \neg P(t_1,...,t_n) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \\ AX_x(\phi)(t) \mid EX_x(\phi)(t) \mid AF_x(\phi)(t) \mid EG_x(\phi)(t) \mid \\ AR_{x,y}(\phi_1,\phi_2)(t) \mid EU_{x,y}(\phi_1,\phi_2)(t) \end{array} \right.$$

where x, y are variables, and each of t and $t_1 \ldots t_n$ is either a constant or a variable.

Note that in this language, modalities are applied to formulae and states, binding variables in these formulae. More explicitly, modalities AX, EX, AF, and EG bind the variable x in ϕ, and modalities AR and EU bind respectively the variable x in ϕ_1 and y in ϕ_2. Also, the negation is applied to atomic formulae only, so, as usual, negations must be pushed inside the formulae. We use the notation $(t/x)\phi$ for the substitution of t for all free occurrences of x in ϕ. As usual, in presence of binders, substitution avoids variable captures.

The following abbreviations are used.

- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$;
- $EF_x(\phi)(t) \equiv EU_{z,x}(\top,\phi)(t)$;
- $ER_{x,y}(\phi_1,\phi_2)(t) \equiv EU_{y,z}(\phi_2,((z/x)\phi_1 \wedge (z/y)\phi_2))(t) \vee EG_y(\phi_2)(t)$, where z is a variable that occurs neither in ϕ_1 nor in ϕ_2;
- $AG_x(\phi)(t) \equiv \neg(EF_x(\neg\phi)(t))$;
- $AU_{x,y}(\phi_1,\phi_2)(t) \equiv \neg(ER_{x,y}(\neg\phi_1,\neg\phi_2)(t))$.

Hereafter, each of AX, EX, AF, EF, AU, and EU is called an *inductive modality*; and each of AR, ER, AG, and EG is called a *co-inductive modality*. A formula starting with an inductive modality is called an *inductive formula*; and a formula starting with a co-inductive modality is called a *co-inductive formula*.

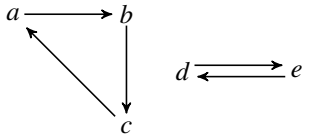
DEFINITION 2.2 (Validity). *Let $\mathcal{M}$ be a Kripke model and ϕ be a closed CTL_P formula. The validity of the formula ϕ in the model $\mathcal{M}$ is defined by induction on ϕ in Figure 1.*

From this definition, we obtain $\mathcal{M} \models EF_x(\phi)(s)$, if there exists an infinite path $s_0,s_1,...$ starting from s and a natural number j such that $\mathcal{M} \models (s_j/x)\phi$, etc.

EXAMPLE 1. The formula

$$EF_x(EF_y(P(x,y))(x))(a)$$

expresses the existence of a path starting from state a, that contains two states related by P. This formula is valid in a Kripke model formed with the relation depicted below



and the set $P = \{\langle a,c\rangle\}$, but not in that formed with the same relation and the set $P = \{\langle a,d\rangle\}$ instead.

EXAMPLE 2. This example is inspired from the example presented in [23], where the specification of the motion planning of a multi-robot [24] system is characterized by CTL formulae. The specification states that, on a partitioned map, each robot moving from an initial section will eventually reach its destination section; meanwhile, along the movement, each robot should avoid entering some specific section. Here we focus on "spatial" properties (i.e., properties that characterize relations between states). Consider a special robot, e.g. an unmanned vehicle, that is designed to move on the surface of a planet. The planet is partitioned into a finite number of small areas. The unmanned vehicle moves from one area to another at a time, each position of the unmanned vehicle is considered as a state, and the moves from one position to another form the transition relation. A basic property of which a unmanned vehicle designer has to think is that: a unmanned vehicle should not stay infinitely long in a small set of areas. To be more precise, for a given distance σ, at any state s, the unmanned vehicle will eventually move to some state s' such that the distance (not the number of moves) between s and s' is longer than σ. This property can be easily characterized by the CTL_P formula $AG_x(AF_y(D_\sigma(x,y))(x))(s_0)$, where s_0 is the landing position of the unmanned vehicle, i.e., the initial state; and atomic formula $D_\sigma(x,y)$ characterizes the spatial property that the distance between state x and state y is longer than σ. This scenario is depicted in Figure 2. This kind of properties can be easily expressed in CTL_P, but cannot be elegantly, even impossibly, expressed in CTL, as there are no mechanisms to speak about specific states in

$\mathcal{M} \models P(s_1,...,s_n)$, if $\langle s_1,...,s_n\rangle \in P$ with P an n-ary relation on $\mathcal{M}$;
$\mathcal{M} \models \neg P(s_1,...,s_n)$, if $\langle s_1,...,s_n\rangle \notin P$ with P an n-ary relation on $\mathcal{M}$;
$\mathcal{M} \models \top$ is always the case;
$\mathcal{M} \models \bot$ is never the case;
$\mathcal{M} \models \phi_1 \wedge \phi_2$, if $\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$;
$\mathcal{M} \models \phi_1 \vee \phi_2$, if $\mathcal{M} \models \phi_1$ or $\mathcal{M} \models \phi_2$;
$\mathcal{M} \models AX_x(\phi_1)(s)$, if for each state s' in $\mathsf{Next}(s)$, $\mathcal{M} \models (s'/x)\phi_1$;
$\mathcal{M} \models EX_x(\phi_1)(s)$, if there exists a state s' in $\mathsf{Next}(s)$ such that $\mathcal{M} \models (s'/x)\phi_1$;
$\mathcal{M} \models AF_x(\phi_1)(s)$, if there exists a finite path-tree T such that T's root is labeled by s, and for each leaf s', $\mathcal{M} \models (s'/x)\phi_1$;
$\mathcal{M} \models EG_x(\phi_1)(s)$, if there exists an infinite path $s_0, s_1, ...$ starting from s, such that for all natural numbers i, $\mathcal{M} \models (s_i/x)\phi_1$;
$\mathcal{M} \models AR_{x,y}(\phi_1,\phi_2)(s)$, if there exists a path-tree T such that T's root is labeled by s, and for each node $s' \in T$, $\mathcal{M} \models (s'/y)\phi_2$ and for each leaf $s'' \in T$, $\mathcal{M} \models (s''/x)\phi_1$;
$\mathcal{M} \models EU_{x,y}(\phi_1,\phi_2)(s)$, if there exists a finite path $s_0,...,s_n$ starting from s such that $\mathcal{M} \models (s_n/y)\phi_2$ and for all $i < n$, $\mathcal{M} \models (s_i/x)\phi_1$.

FIGURE 1. Validity of a formula in CTL_P

the syntax of CTL, even in the semantics of CTL, only one state is under consideration at a time, so it is hard to express relationships between states.

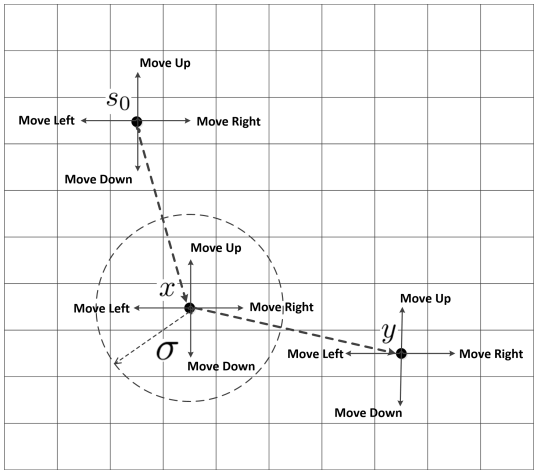


FIGURE 2. Possible positions of an unmanned vehicle

3. SCTL

In this section, we present $\mathsf{SCTL}(\mathcal{M})$, a proof system for $\mathsf{CTL}_P(\mathcal{M})$ taking a Kripke model as the parameter.[12] Unlike the usual proof systems, where a formula is provable if and only if it is valid in all models, a formula is provable in $\mathsf{SCTL}(\mathcal{M})$ if and only if it is valid in the Kripke model $\mathcal{M}$.

3.1. Proofs

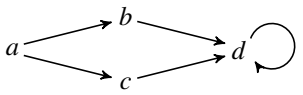
Consider the formula $AF_x(P(x))(s)$. This formula is valid if there exists a finite path-tree whose root is labeled by s, and all the leaves are in P. Such a tree can be called a *proof* of the formula $AF_x(P(x))(s)$.

Consider now the formula $AF_x(AF_y(P(x,y))(x))(s)$ that contains nested modalities. To justify the validity of this formula, one needs to provide a path-tree whose root is labeled by s, and at each leaf a, the formula $AF_y(P(a,y))(a)$ is valid. To justify the validity of the formula $AF_y(P(a,y))(a)$, one needs to provide other path-trees. These hierarchical trees can be formalized with the following proof rules.

$$\frac{\vdash (s/x)\phi}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-}\mathsf{R}_1$$

$$\frac{\vdash AF_x(\phi)(s_1) \quad \dots \quad \vdash AF_x(\phi)(s_n)}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-}\mathsf{R}_2\ \{s_1,...s_n\}=\mathsf{Next}(s)$$

EXAMPLE 3. Consider the Kripke model formed with the following relation and the set $P = \{b, c\}$.



A proof of the formula $AF_x(P(x))(a)$ is

$$\dfrac{\dfrac{\dfrac{}{\vdash P(b)}\ \text{atom-}\mathsf{R}}{\vdash AF_x(P(x))(b)}\ \mathbf{AF}\text{-}\mathsf{R}_1 \qquad \dfrac{\dfrac{}{\vdash P(c)}\ \text{atom-}\mathsf{R}}{\vdash AF_x(P(x))(c)}\ \mathbf{AF}\text{-}\mathsf{R}_1}{\vdash AF_x(P(x))(a)}\ \mathbf{AF}\text{-}\mathsf{R}_2$$

where besides the rules $\mathbf{AF}$-R_1 and $\mathbf{AF}$-R_2, we use the following rule.

$$\frac{}{\vdash P(s_1,...,s_n)}\ \text{atom-}\mathsf{R}\ \langle s_1,...,s_n\rangle \in P$$

EXAMPLE 4. Consider the model formed with the same relation as in Example 3 and the set $Q = \{\langle b,d\rangle, \langle c,d\rangle\}$. A proof of the formula $AF_x(AF_y(Q(x,y))(x))(a)$ is given in Figure 3.

In the usual sequent calculus, the right rule of disjunction can either be formulated in a multiplicative way

$$\frac{\Gamma \vdash \phi_1, \phi_2, \Delta}{\Gamma \vdash \phi_1 \vee \phi_2, \Delta}$$

or in an additive way

$$\frac{\Gamma \vdash \phi_1, \Delta}{\Gamma \vdash \phi_1 \vee \phi_2, \Delta}$$

[12] This system was initially introduced in [25].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash Q(b,d)}\ \text{atom-R}}{\vdash AF_y(Q(b,y))(d)}\ \mathbf{AF}\text{-R}_1}{\vdash AF_y(Q(b,y))(b)}\ \mathbf{AF}\text{-R}_2}{\vdash AF_x(AF_y(Q(x,y))(x))(b)}\ \mathbf{AF}\text{-R}_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash Q(c,d)}\ \text{atom-R}}{\vdash AF_y(Q(c,y))(d)}\ \mathbf{AF}\text{-R}_1}{\vdash AF_y(Q(c,y))(c)}\ \mathbf{AF}\text{-R}_2}{\vdash AF_x(AF_y(Q(x,y))(x))(c)}\ \mathbf{AF}\text{-R}_1}{\vdash AF_x(AF_y(Q(x,y))(x))(a)}\ \mathbf{AF}\text{-R}_2$$

FIGURE 3. A proof of $AF_x(AF_y(Q(x,y))(x))(a)$

$$\frac{\Gamma \vdash \phi_2, \Delta}{\Gamma \vdash \phi_1 \vee \phi_2, \Delta}$$

These two formulations are equivalent in presence of structural rules. For instance, the proof of the sequent $\vdash P \Rightarrow P$, that is $\vdash \neg P \vee P$, in the multiplicative system

$$\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \text{axiom}}{\vdash \neg P, P}\ \neg\text{-right}}{\vdash \neg P \vee P}\ \vee\text{-right}$$

can be rewritten in the additive one

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}\ \text{axiom}}{\vdash \neg P, P}\ \neg\text{-right}}{\vdash \neg P, \neg P \vee P}\ \vee\text{-right}}{\vdash \neg P \vee P, \neg P \vee P}\ \vee\text{-right}}{\vdash \neg P \vee P}\ \text{contraction-right}$$

The contraction rule, or the multiplicative rule, is needed to build the sequent $\vdash \neg P, P$ that is provable even if none of its weakenings $\vdash \neg P$ and $\vdash P$ is, because we do not know whether the formula P is true or false.

Our proof system SCTL needs neither contraction rules nor multiplicative $\vee$-R rules, because for each atomic formula P, either P is provable or $\neg P$ is. Therefore, the sequent $\vdash \neg P \vee P$ is proved by proving either the sequent $\vdash \neg P$ or the sequent $\vdash P$.

As we have neither multiplicative $\vee$-R rules nor structural rules, if we start with a sequent $\vdash \phi$, where the formula ϕ does not contain co-inductive sub-formulae, then each sequent in the proof has one formula on the right of $\vdash$ and none on the left. Thus, as all sequents have the form $\vdash \phi$, the left rules and the axiom rule can be dropped as well. In other words, unlike the usual sequent calculus, our proof system, like Hilbert systems, is tailored for deduction, not for hypothetical deduction.

The case of co-inductive formulae, for instance $EG_x(P(x))(s)$, is more complex than that of the inductive formulae. To justify its validity, one needs to provide an infinite sequence that is an infinite path-tree with only one branch, such that the root of the tree is labeled by s, and each state in the infinite sequence verifies P. As the model is finite, we can always restrict to regular trees and use a finite representation of such trees. This leads us to introduce a rule, called **EG**-merge, that permits to prove a sequent of the form $\vdash EG_x(P(x))(s)$, provided such a sequent already occurs lower in the proof. To make this rule local, we re-introduce hypotheses Γ to record part of the history of the proof.[13] The sequents have therefore the form $\Gamma \vdash \phi$, with a non-empty Γ in this particular case only, and the **EG**-merge rule is then just an instance of the axiom rule.

The above discussion leads to the rules of SCTL depicted in Figure 4.

3.2. Soundness and completeness

Proposition 3.1 and 3.2 below permit to transform finite structures into infinite ones and will be used in the Soundness proof, while Proposition 3.3 and 3.4 permit to transform infinite structures into finite ones and will be used in the Completeness proof.

PROPOSITION 3.1 (Finite to infinite sequences). *Let $s_0, ..., s_n$ be a finite sequence of states such that for all i between 0 and $n-1$, $s_i \longrightarrow s_{i+1}$, and $s_n = s_p$ for some p between 0 and $n-1$. Then there exists an infinite sequence of states $s'_0, s'_1, ...$ such that $s_0 = s'_0$ and for all i, $s'_i \longrightarrow s'_{i+1}$, and all the s'_j are among $s_0, ..., s_n$.*

Proof. Take the sequence $s_0, ..., s_{p-1}, s_p, ..., s_{n-1}, s_p, ...$, where $s_0 = s'_0$. □

PROPOSITION 3.2 (Finite to possibly infinite path-trees). *Let S be a set of states and T be a finite path-tree such that each leaf is labeled with a state which is either in S or also a label of a node on the branch from the root of T to this leaf. Then, there exists a possibly infinite path-tree T' such that all the leaves are labeled by elements of S, and all the labels of T' are the labels of T.*

Proof. Consider for T' the path-tree whose root is labeled by the root of T and such that for each node s, if s is in S, then s is a leaf of T', otherwise the successors of s are the elements of $\mathsf{Next}(s)$. It is easy to check that all the nodes of T' are labeled by labels of T. □

PROPOSITION 3.3 (Infinite to finite sequences). *Let $s_0, s_1, ...$ be an infinite sequence of states such that for all i, $s_i \longrightarrow s_{i+1}$. Then there exists a finite sequence of states $s'_0, ..., s'_n$ such that for all i between 0 and $n-1$, $s'_i \longrightarrow s'_{i+1}$, $s'_n = s'_p$ for some p between 0 and $n-1$, and all the s'_j are among $s_0, s_1, ...$*

Proof. As the number of states is finite, there exists p and n such that $p < n$ and $s_p = s_n$. Take the sequence $s_0, ..., s_n$. □

[13] Intuitively, we use the left hand side of the sequent to store visited states for co-inductive modalities.

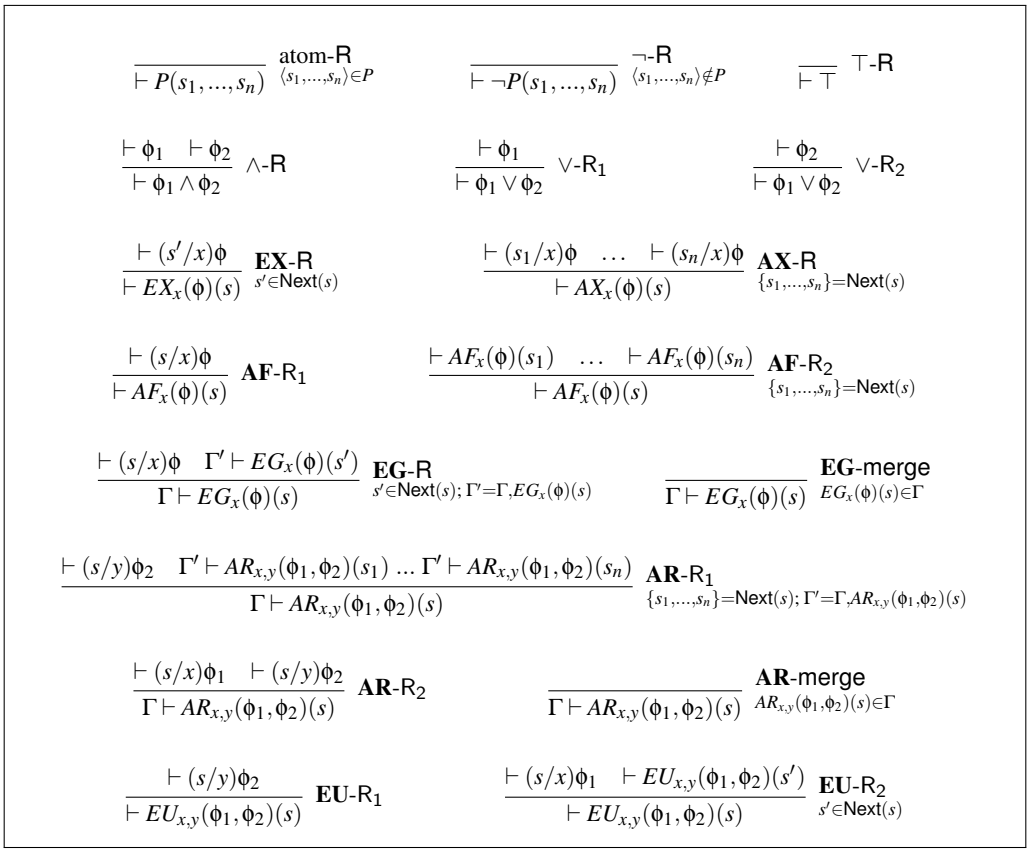


FIGURE 4. SCTL($\mathcal{M}$)

PROPOSITION 3.4 (Possibly infinite to finite path-trees). *Let S be a set of states and T be a possibly infinite path-tree such that each leaf is labeled by a state in S. Then, there exists a finite path-tree such that each leaf is labeled with a state which is either in S or also a label of a node on the branch from the root of T to this leaf.*

Proof. As the number of states is finite, on each infinite branch, there exists p and n such that $p < n$ and $s_p = s_n$. Prune the path-tree at node s_n. This path-tree is finitely branching and each branch is finite, hence, by König's lemma, it is finite. □

THEOREM 3.1 (Soundness and completeness). *If ϕ is closed, then the sequent $\vdash \phi$ has a proof in SCTL($\mathcal{M}$) if and only if $\mathcal{M} \models \phi$ for the given Kripke model $\mathcal{M}$.*

Proof. The soundness and completeness are guaranteed by the finiteness of the Kripke model and Proposition 3.1, 3.2, 3.3 and 3.4 above-mentioned. The details are presented in Appendix C. □

4. IMPLEMENTATION

In this section, a proof search method for SCTL is presented. Based on this method, a proof search algorithm is designed. Based on this algorithm, a complete automatic prover SCTLProV[14] is developed, verifying CTL$_P$ properties over any Kripke models. Finally, we discuss the relations of the techniques adopted in SCTLProV with those in some other CTL model checking approaches.

4.1. A proof search method for SCTL

The basic idea of the proof search method for SCTL is as follows: firstly, an order is given over the inference rules of SCTL with the same conclusion; secondly, for each inference rule, an order is given over its premises. This way, proving the sequent transforms into proving all its premises with some specific order. The purpose of continuation-passing tree (Definition 4.1) is to encode these two orders into a tree-like structure when proving a sequent. This proof search method leads to the CPT rewriting system displayed in Figure 5.

4.1.1. Continuations and CPT rewriting system

The major technique for the proof search method is based on the concept of continuation, which is usually used in compiling and programming [15, 26]. Basically, a continuation is an explicit representation of "the rest of the computation" to happen.

DEFINITION 4.1 (Continuation-passing tree). A continuation-passing tree *(CPT for short) is a binary tree such that*

- *every leaf is labeled by either* $\mathfrak{t}$ *or* $\mathfrak{f}$*, where* $\mathfrak{t}$ *and* $\mathfrak{f}$ *are*

[14] https://github.com/sctlprov/sctlprov_code

$\mathsf{cpt}(\vdash \top, c_1, c_2) \rightsquigarrow c_1 \quad \mathsf{cpt}(\vdash \bot, c_1, c_2) \rightsquigarrow c_2$
$\mathsf{cpt}(\vdash P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1 \quad [\langle s_1,...,s_n\rangle \in P]$
$\mathsf{cpt}(\vdash P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_2 \quad [\langle s_1,...,s_n\rangle \notin P]$
$\mathsf{cpt}(\vdash \neg P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_2 \quad [\langle s_1,...,s_n\rangle \in P]$
$\mathsf{cpt}(\vdash \neg P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1 \quad [\langle s_1,...,s_n\rangle \notin P]$
$\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2)$
$\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$
$\mathsf{cpt}(\vdash AX_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s_1/x)\phi, \mathsf{cpt}(\vdash (s_2/x)\phi, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi, c_1, c_2), ..., c_2), c_2), c_2)$ $[\{s_1,...,s_n\} = \mathsf{Next}(s)]$
$\mathsf{cpt}(\vdash EX_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s_1/x)\phi, c_1, \mathsf{cpt}(\vdash (s_2/x)\phi, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi, c_1, c_2)...)))$ $[\{s_1,...,s_n\} = \mathsf{Next}(s)]$
$\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow c_2 \quad [AF_x(\phi)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/x)\phi, c_1, \mathsf{cpt}(\Gamma' \vdash AF_x(\phi)(s_1), \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash AF_x(\phi)(s_n), c_1, c_2)..., c_2), c_2))$ $[\{s_1,...,s_n\} = \mathsf{Next}(s), AF_x(\phi)(s) \notin \Gamma, \text{and } \Gamma' = \Gamma, AF_x(\phi)(s)]$
$\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow c_1 \quad [EG_x(\phi)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/x)\phi, \mathsf{cpt}(\Gamma' \vdash EG_x(\phi)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi)(s_n), c_1, c_2)...)), c_2)$ $[\{s_1,...,s_n\} = \mathsf{Next}(s), EG_x(\phi)(s) \notin \Gamma, \text{and } \Gamma' = \Gamma, EG_x(\phi)(s)]$
$\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s), c_1, c_2) \rightsquigarrow c_1 \quad \left[(AR_{x,y}(\phi_1,\phi_2)(s) \in \Gamma\right]$
$\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/y)\phi_2, \mathsf{cpt}(\vdash (s/x)\phi_1, c_1, \mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_1), \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n),$ $c_1, c_2)..., c_2), c_2)), c_2)\left[\{s_1,...,s_n\} = \mathsf{Next}(s), AR_{x,y}(\phi_1,\phi_2)(s) \notin \Gamma, \text{ and } \Gamma' = \Gamma, AR_{x,y}(\phi_1,\phi_2)(s)\right]$
$\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s), c_1, c_2) \rightsquigarrow c_2 \quad \left[EU_{x,y}(\phi_1,\phi_2)(s) \in \Gamma\right]$
$\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/y)\phi_2, c_1, \mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1,\phi_2)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1,\phi_2)(s_n),$ $c_1, c_2)...)), c_2)) \quad \left[\{s_1,...,s_n\} = \mathsf{Next}(s), EU_{x,y}(\phi_1,\phi_2)(s) \notin \Gamma, \text{ and } \Gamma' = \Gamma, EU_{x,y}(\phi_1,\phi_2)(s)\right]$

FIGURE 5. Rewritings over CPTs

two different symbols;

- *every internal node is labeled by an SCTL sequent.*

For each internal node in a CPT, the left sub-tree is called its t-continuation*, and the right one its* f-continuation*. A CPT c with a SCTL sequent $\Gamma \vdash \phi$ labeled at its root is called a CPT corresponding to the sequent $\Gamma \vdash \phi$, and often denoted by $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$, or visually by*

$$\begin{array}{c} \Gamma \vdash \phi \\ \diagup \quad \diagdown \\ c_1 \quad c_2 \end{array}$$

where c_1 is the t*-continuation of c, and c_2 the* f*-continuation.*

The proof system SCTL is implemented by the CPT rewriting system given in Figure 5, where conditions are put in brackets. Since there is no congruence rule to allow the application of a rewrite rule to a sub-expression of a CPT, reductions always occur at the root of a CPT.

The aim of these conditional rewrite rules is to decide, for a given sequent $\Gamma \vdash \phi$, whether the corresponding CPT, that is $\mathsf{cpt}(\Gamma \vdash \phi, \mathfrak{t}, \mathfrak{f})$, reduces to $\mathfrak{t}$ or to $\mathfrak{f}$. To do so, we analyze the form of the formula ϕ. If, for instance, it is $\phi_1 \wedge \phi_2$, we transform, using one of the rewrite rules, the tree $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, \mathfrak{t}, \mathfrak{f})$ into $\mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, \mathfrak{t}, \mathfrak{f}), \mathfrak{f})$ which is depicted as

$$\begin{array}{ccc} & \vdash \phi_1 & \\ & \diagup \quad \diagdown & \\ \vdash \phi_2 & & \mathfrak{f} \\ \diagup \quad \diagdown & & \\ \mathfrak{t} \quad \mathfrak{f} & & \end{array}$$

expressing that if the attempt to prove $\vdash \phi_1$ succeeds, then we attempt to prove $\vdash \phi_2$, otherwise it just returns a negative result denoted by $\mathfrak{f}$. Then the reduction procedure will apply to $\mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, \mathfrak{t}, \mathfrak{f}), \mathfrak{f})$ by considering the form of ϕ_1, and so on.

EXAMPLE 5. The proof search method can be illustrated by searching the proof in Example 3. The rewriting steps are depicted in Figure 6.

Step 1. Consider the CPT on the left side of $\overset{1}{\rightsquigarrow}$, the root of it is labeled by $\vdash AF_x(P(x))(a)$ which is the sequent to prove. We need to decide whether $\vdash AF_x(P(x))(a)$ is provable, which is not known at that moment yet, so the leaves of the CPT are its t-continuation and f-continuation, representing the rest of the proof search. Then we have to decide successively

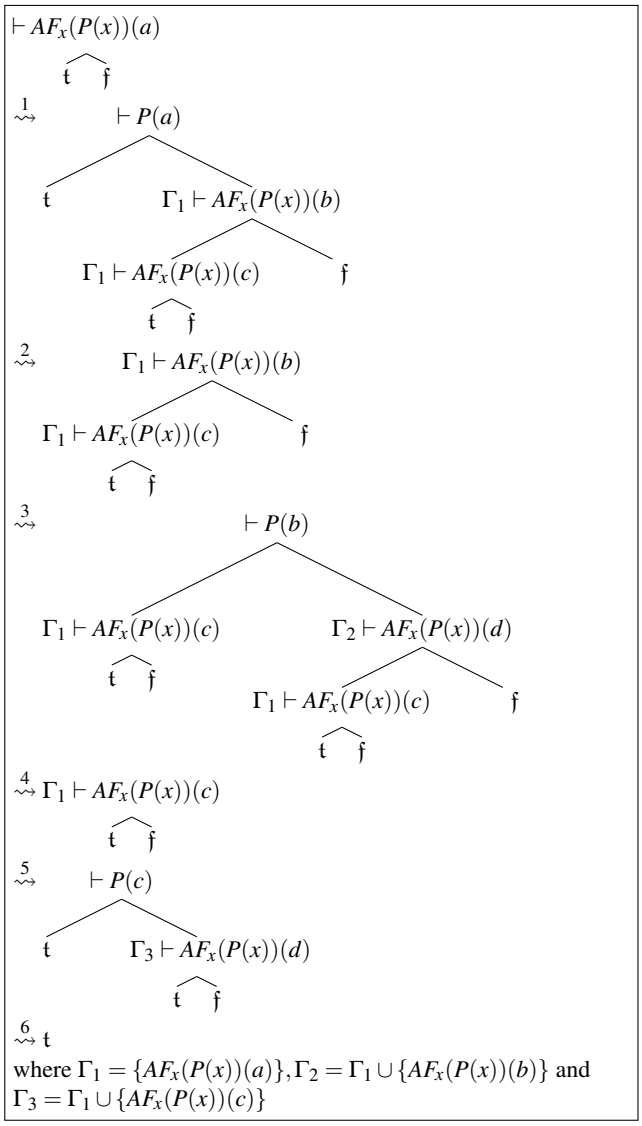


FIGURE 6. An illustration of CPT rewritings

- If $\vdash AF_x(P(x))(a)$ is provable by applying the **AF**-R_1 rule, that is, if $\vdash P(a)$ is provable.
- If $\vdash AF_x(P(x))(a)$ is provable by applying the **AF**-R_2 rule, that is, if $\Gamma_1 \vdash AF_x(P(x))(b)$ and $\Gamma_1 \vdash AF_x(P(x))(c)$ are provable, successively.

The above information is encoded into a single CPT that is on the right side of $\overset{1}{\rightsquigarrow}$.

Step 2. Since the atomic formula $P(a)$ is not provable, we can only use the **AF**-R_2 rule to prove $\vdash AF_x(P(x))(a)$, so the CPT on the left side of $\overset{2}{\rightsquigarrow}$ reduces to its right sub-tree (f-continuation) which is the CPT on the right side of $\overset{2}{\rightsquigarrow}$.

Step 3. Like at step 1, we need to decide if $\Gamma_1 \vdash AF_x(P(x))(b)$ is provable, which is not known at that moment. According to the CPT which is on the left side of $\overset{3}{\rightsquigarrow}$, we know that if $\Gamma_1 \vdash AF_x(P(x))(b)$ is provable then we need to decide if $\Gamma_1 \vdash AF_x(P(x))(c)$ is provable. Then we have to decide successively

- If $\Gamma_1 \vdash AF_x(P(x))(b)$ is provable by applying the **AF**-R_1 rule, that is, if $\vdash P(b)$ is provable.
- If $\Gamma_1 \vdash AF_x(P(x))(b)$ is provable by applying the **AF**-R_2 rule, that is, if $\Gamma_2 \vdash AF_x(P(x))(d)$ is provable.

The above information is encoded into a single CPT that is on the right side of $\overset{3}{\rightsquigarrow}$.

Step 4. Like at step 2, we can judge that the atomic formula $P(b)$ is provable immediately. So the CPT on the left side of $\overset{4}{\rightsquigarrow}$ reduces to its left sub-tree (t-continuation) which is on the right side of $\overset{4}{\rightsquigarrow}$.

Step 5. Like at step 1 and 3, we can not judge whether the sequent $\Gamma_1 \vdash AF_x(P(x))(c)$ is provable immediately, so we encode the two steps to find successively the proofs of $\vdash P(c)$ and $\Gamma_3 \vdash AF_x(P(x))(d)$ into the CPT which is on the right side of $\overset{5}{\rightsquigarrow}$;

Step 6. Like at step 2 and 4, as the atomic formula $P(c)$ is provable, the CPT on the left side of $\overset{6}{\rightsquigarrow}$ reduces to its left sub-tree (t-continuation) which is t. Now, the proof search of $\vdash AF_x(P(x))(a)$ terminates, and we can judge that this sequent is provable.

It is worth to note that the proof system SCTL is sound and complete, but the proof search in SCTL does not always terminate : this is the case when the formula under consideration is an inductive formula and it is not provable. However, if we introduce merge rules for each inductive modalities as well, then proof search always terminates. More precisely, although we do not need merge rules for inductive formulae (formulae starting with modalities AF or EU etc.) in the proof system SCTL, we do need merge rules for this kind of formulae in the CPT rewriting system which provides a proof search method for SCTL. This is because, for instance, to search a proof of the sequent $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$, we need to find a finite path of states, say $s_0(=s),\ldots,s_n$, in the Kripke model $\mathcal{M}$ under consideration, such that ϕ_2 holds at state s_n, and ϕ_1 holds at all other state s_i (if any). However, $\mathcal{M}$ may also contain infinite paths of states, say $s_0(=s),s_1,\ldots$ such that ϕ_1 holds at each state s_i ($i \geq 0$), and ϕ_2 does not hold at any of them. On such an infinite path, the proof search procedure for the sequent $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$ cannot terminate. To deal with this kind of scenarios, in the CPT rewriting system, the merge rules for inductive formulae are introduced.

Note also that merges for inductive and co-inductive formulae are different in the CPT rewriting system. Intuitively, in the case of co-inductive formulae: consider, for example, the sequent $\Gamma \vdash EG_x(\phi)(s)$.

- If $EG_x(\phi)(s) \in \Gamma$, then we have $\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow c_1$ according to the CPT rewriting system, and we proceed with the reduction of c_1, where $\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma \vdash EG_x(\phi)(s)$.
- If $EG_x(\phi)(s) \notin \Gamma$, then one of the following cases holds:
  - if $\vdash (s/x)\phi$ is not provable in SCTL, then we have $\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow^* c_2$, and proceed with the reduction of c_2, where $\mathsf{cpt}(\Gamma \vdash$

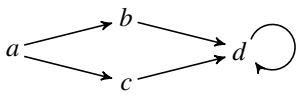
$EG_x(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma \vdash EG_x(\phi)(s)$;
- otherwise, $EG_x(\phi)(s)$ is added into the context, and we proceed with the reduction of a corresponding CPT of a sequent $\Gamma' \vdash EG_x(\phi)(s')$, where $\Gamma' = \Gamma \cup \{EG_x(\phi)(s)\}$ and $s' \in \mathsf{Next}(s)$.

In the case of inductive formulae: consider, for example, the sequent $\Gamma \vdash AF_x(\phi)(s)$.

- If $AF_x(\phi)(s) \in \Gamma$, then there exists a rewriting step $\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow c_2$ according to the CPT rewriting system, and we proceed with the reduction of c_2, where $\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma \vdash AF_x(\phi)(s)$.
- If $AF_x(\phi)(s) \notin \Gamma$, then one of the following cases holds:
  - if $\vdash (s/x)\phi$ is provable in SCTL, then we have $\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow^* c_1$, and proceed with the reduction of c_1, where $\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma \vdash AF_x(\phi)(s)$;
  - otherwise, $AF_x(\phi)(s)$ is added into the context, and for all $s' \in \mathsf{Next}(s)$, we proceed with the reduction of a corresponding CPT of the sequent $\Gamma' \vdash AF_x(\phi)(s')$, where $\Gamma' = \Gamma \cup \{AF_x(\phi)(s)\}$[15].

The example below illustrates the use and the above-mentioned difference of merges in the CPT rewriting system.

EXAMPLE 6. Consider the Kripke model formed with the following relation and two sets $P = \{a, b, d\}$ and $Q = \{b, c\}$.



We apply the proof search method to the co-inductive formulae $EG_x(P(x))(a)$ and $EG_x(Q(x))(a)$ and to the inductive formulae $AF_x(\neg P(x))(a)$ and $AF_x(Q(x))(a)$, respectively.

- The sequent $\vdash EG_x(P(x))(a)$ is provable in SCTL, and the CPT $\mathsf{cpt}(\vdash EG_x(P(x))(a), \mathfrak{t}, \mathfrak{f})$ reduces to $\mathfrak{t}$ via the following rewriting steps:

$$\begin{array}{l}\mathsf{cpt}(\vdash EG_x(P(x))(a), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma \vdash EG_x(P(x))(b), \mathfrak{t}, \mathsf{cpt}(\Gamma \vdash EG_x(P(x))(c), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma_1 \vdash EG_x(P(x))(d), \mathfrak{t}, \mathsf{cpt}(\Gamma \vdash EG_x(P(x))(c), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma_2 \vdash EG_x(P(x))(d), \mathfrak{t}, \mathsf{cpt}(\Gamma \vdash EG_x(P(x))(c), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow \\ \mathfrak{t}\end{array}$$

  where $\Gamma = \{EG_x(P(x)(a))\}$, $\Gamma_1 = \Gamma \cup \{EG_x(P(x))(b)\}$, and $\Gamma_2 = \Gamma_1 \cup \{EG_x(P(x))(d)\}$.
- The sequent $\vdash EG_x(Q(x))(a)$ is not provable in SCTL, and the CPT $\mathsf{cpt}(\vdash EG_x(Q(x))(a), \mathfrak{t}, \mathfrak{f})$ reduces to $\mathfrak{f}$ via the following rewriting steps:

$$\begin{array}{l}\mathsf{cpt}(\vdash EG_x(Q(x))(a), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow \\ \mathsf{cpt}(\vdash Q(a), \mathsf{cpt}(\Gamma \vdash EG_x(Q(x))(b), \mathfrak{t}, \mathsf{cpt}(\Gamma \vdash EG_x(Q(x))(c), \mathfrak{t}, \mathfrak{f})), \mathfrak{f}) \rightsquigarrow \\ \mathfrak{f}\end{array}$$

  where $\Gamma = \{EG_x(Q(x)(a))\}$.
- The sequent $\vdash AF_x(\neg P(x))(a)$ is not provable in SCTL, and the CPT $\mathsf{cpt}(\vdash AF_x(\neg P(x))(a), \mathfrak{t}, \mathfrak{f})$ reduces to $\mathfrak{f}$ via the following rewriting steps:

$$\begin{array}{l}\mathsf{cpt}(\vdash AF_x(\neg P(x))(a), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma \vdash AF_x(\neg P(x))(b), \mathsf{cpt}(\Gamma \vdash AF_x(\neg P(x))(c), \mathfrak{t}, \mathfrak{f}), \mathfrak{f}) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma_1 \vdash AF_x(\neg P(x))(d), \mathsf{cpt}(\Gamma \vdash AF_x(\neg P(x))(c), \mathfrak{t}, \mathfrak{f}), \mathfrak{f}) \rightsquigarrow^* \\ \mathsf{cpt}(\Gamma_2 \vdash AF_x(\neg P(x))(d), \mathsf{cpt}(\Gamma \vdash AF_x(\neg P(x))(c), \mathfrak{t}, \mathfrak{f}), \mathfrak{f}) \rightsquigarrow \\ f\end{array}$$

  where $\Gamma = \{AF_x(\neg P(x)(a))\}$, $\Gamma_1 = \Gamma \cup \{AF_x(\neg P(x))(b)\}$, and $\Gamma_2 = \Gamma_1 \cup \{AF_x(\neg P(x))(d)\}$.
- The sequent $\vdash AF_x(\neg Q(x)(a))$ is provable in SCTL, and the CPT $\mathsf{cpt}(\vdash AF_x(\neg Q(x))(a), \mathfrak{t}, \mathfrak{f})$ reduces to $\mathfrak{t}$ via the following rewriting steps:

$$\begin{array}{l}\mathsf{cpt}(\vdash AF_x(\neg Q(x))(a), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow \\ \mathsf{cpt}(\vdash \neg Q(a), \mathfrak{t}, \mathsf{cpt}(\Gamma \vdash AF_x(\neg Q(x))(b), \mathsf{cpt}(\Gamma \vdash AF_x(\neg Q(x))(c), \mathfrak{t}, \mathfrak{f}), \mathfrak{f})) \rightsquigarrow \\ t\end{array}$$

  where $\Gamma = \{AF_x(\neg Q(x)(a))\}$.

4.1.2. Termination and correctness

To prove the termination of the CPT rewriting system, the following definitions are needed.

DEFINITION 4.2 (Lexicographic path ordering [27, 28]). *Let F be a set of function symbols, where each element has a fixed arity and let $\succeq$ be a quasi-ordering (i.e., a binary relation that is reflexive and transitive) on F. The* lexicographic path ordering $\succeq_{\mathsf{lpo}}$ *on the set $T(F)$ of terms over F is defined recursively as follows.*

$$s = f(s_1, ..., s_m) \succeq_{\mathsf{lpo}} g(t_1, ..., t_n) = t$$

if and only if at least one of the following assertions holds:

1. $s_i \succeq_{\mathsf{lpo}} t$ *for some* $i \in \{1, ..., m\}$.
2. $f \succ g$ *and* $s \succ_{\mathsf{lpo}} t_j$ *for all* $j \in \{1, ..., n\}$.
3. $f = g$ *and* $(s_1, ..., s_m) \succeq'_{\mathsf{lpo}} (t_1, ..., t_n)$ *and* $s \succ_{\mathsf{lpo}} t_j$ *for all* $j \in \{2, ..., n\}$, *where* $\succeq'_{\mathsf{lpo}}$ *is the lexicographic ordering induced by* $\succeq_{\mathsf{lpo}}$.

As usual, $f \succ g$ (resp. $s \succ_{\mathsf{lpo}} t$) is shorthand for "$f \succeq g$ and $f \neq g$" (resp. "$s \succeq_{\mathsf{lpo}} t$ and $s \neq t$").

PROPOSITION 4.1 (Well-foundedness of lexicographic path orderings). *Let $\succeq$ be a quasi-ordering on a set F of function symbols where each element has a fixed arity, and $\succeq_{\mathsf{lpo}}$ be the lexicographic path ordering on the set $T(F)$ of terms over F. Then $\succeq_{\mathsf{lpo}}$ is well-founded iff $\succeq$ is.*

Proof. See the proof proposed by Dershowitz [27]. □

DEFINITION 4.3 (Weight of a sequent). *Given a Kripke model $\mathcal{M}$ with cardinal n and a sequent $\Gamma \vdash \phi$ in SCTL($\mathcal{M}$), let $|\phi|$ denote the size of ϕ defined in the usual way, and $|\Gamma|$ denote the cardinal of a context Γ. We define the* weight of the sequent $\Gamma \vdash \phi$ *as*

$$w(\Gamma \vdash \phi) = \langle |\phi|, (n - |\Gamma|) \rangle$$

It easy to check that, given a Kripke model, the weight of a sequent is well defined, and that the lexicographic order on the set of sequent weights induces a quasi-ordering on the set of sequents, which is well-founded.

[15] With a little abuse of notation, we add $AF_x(\phi)(s)$ into the context, instead of $\neg AF_x(\phi)(s)$.

PROPOSITION 4.2 (Termination). *$\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$ always rewrites to $\mathfrak{t}$ or $\mathfrak{f}$ in finitely many steps.*

Proof. Note first that, for any CPT c, if the rewriting of c is terminating, then it must end with either $\mathfrak{t}$ or $\mathfrak{f}$. Now we prove that the rewriting of a CPT always terminates. Consider a CPT $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$, we define first the set of functions symbols as follows:

$$F = \{\mathfrak{t}, \mathfrak{f}, \mathsf{cpt}\} \cup \mathsf{Seq}$$

where Seq is the set of sequents that appear in the rewriting procedure of $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$, cpt has arity 3, and other elements in F have arity 0. The quasi-ordering $\succeq$ over F is defined as follows :

- $\mathsf{cpt} \succ \mathfrak{t}$;
- $\mathsf{cpt} \succ \mathfrak{f}$;
- $\Gamma \vdash \phi \succ \mathsf{cpt}$ for each sequent $\Gamma \vdash \phi$;
- $\Gamma \vdash \phi \succ \Gamma' \vdash \phi'$ iff $w(\Gamma \vdash \phi) > w(\Gamma' \vdash \phi')$, where $>$ is the lexicographic order on pairs of natural numbers.

It is easy to check that $\succeq$ is well-founded. By Definition 4.2, $\succeq$ induces a lexicographic path ordering $\succeq_{\mathsf{lpo}}$ on CPTs. By Proposition 4.1, $\succeq_{\mathsf{lpo}}$ is also well-founded. Then it is sufficient to prove that, for any CPTs c and c',

$$c \rightsquigarrow c' \text{ implies } c \succ_{\mathsf{lpo}} c'.$$

We analyze each rule $c \rightsquigarrow c'$ displayed in the CPT rewriting system. Suppose without loss of generality that c is in the form of $\mathsf{cpt}(\vdash \phi, c_1, c_2)$.

- If $\phi = \top, \bot, P(s_1,...,s_m)$ or $\neg P(s_1,...,s_m)$, then according to Definition 4.2 (1), we have $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2) \succ_{\mathsf{lpo}} c_1$ and c_2 because c_1, c_2 are sub-terms of $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$.
- If $\phi = \phi_1 \wedge \phi_2$, then according to Definition 4.2 (3), we have $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2)$, because $\vdash \phi_1 \wedge \phi_2 \succ \vdash \phi_1$, and $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_2, c_1, c_2)$ and $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} c_2$.
- If $\phi = \phi_1 \vee \phi_2$, then according to Definition 4.2 (3), we have $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$ because $\vdash \phi_1 \vee \phi_2 \succ \vdash \phi_1$, and $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} c_1$ and $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_2, c_1, c_2)$.
- If $\phi = AX_x(\phi_1)(s)$, then by Definition 4.2 (3), we have $\mathsf{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash (s_1/x)\phi_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2), ..., c_2), c_2)$, because $\Gamma \vdash AX_x(\phi_1)(s) \succ \vdash (s_i/x)\phi_1$, and $\mathsf{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash (s_i/x)\phi_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)..., c_2), c_2)$, and $\mathsf{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} c_2$, where $\mathsf{Next}(s) = \{s_1, ..., s_n\}$, and $i \in \{1, ..., n\}$.
- The case of EX is analogous to that of AX.
- If $\phi = EG_x(\phi_1)(s)$, consider the following cases:
  - If $EG_x(\phi_1)(s) \in \Gamma$, then by Definition 4.2 (1), we have $\mathsf{cpt}(\Gamma \vdash EG_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} c_1$.
  - If $EG_x(\phi_1)(s) \notin \Gamma$, then by Definition 4.2 (3), we have $c \succ_{\mathsf{lpo}} c'$, as $\Gamma \vdash EG_x(\phi_1)(s) \succ \vdash (s/x)\phi_1$, and $\Gamma \vdash EG_x(\phi_1)(s) \succ \Gamma' \vdash EG_x(\phi_1)(s_i)$ for all $i \in \{1, ..., n\}$ where $\mathsf{Next}(s) = \{s_1, ..., s_n\}$, and $\Gamma' = \Gamma \cup \{EG_x(\phi_1)(s)\}$.
- The case of AF is analogous to that of EG.
- If $\phi = AR_{x,y}(\phi_1, \phi_2)(s)$, consider the cases below:
  - If $AR_{x,y}(\phi_1, \phi_2)(s) \in \Gamma$, then by Definition 4.2 (1), we have $\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \succ_{\mathsf{lpo}} c_1$.
  - If $AR_{x,y}(\phi_1, \phi_2)(s) \notin \Gamma$, then by Definition 4.2 (3), we have, $c \succ_{\mathsf{lpo}} c'$ because $\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \vdash (s/y)\phi_2$, $\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \vdash (s/x)\phi_1$, and $\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s_i)$ for all $i \in \{1, ..., n\}$, where $\mathsf{Next}(s) = \{s_1, ..., s_n\}$, and $\Gamma' = \Gamma \cup \{AR_{x,y}(\phi_1, \phi_2)(s)\}$.
- The case of EU is analogous to that of AR.

□

The correctness of the proof search method is ensured by the following proposition.

PROPOSITION 4.3 (Correctness). *For any given closed formula ϕ, $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \mathfrak{t}$ if and only if $\vdash \phi$ is provable in SCTL.*

Proof. The detailed proof is presented in Appendix D. □

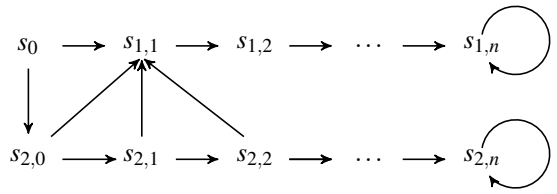
4.2. The proof search algorithm

Now we present the proof search algorithm, which is based on the CPT rewriting system. In this algorithm, much like in traditional model checking, we will store all the visited states (and the states currently been visited as well) in order to avoid visiting them several times, while keeping completeness. More precisely, in the proof search procedure of a formula ϕ, we use two global variables[16] (in the form of hash tables or BDDs) to store respectively the set of visited states on which a sub-formula of ϕ holds or not. This way, we avoid to visit repeatedly the states in these two global variables, and hence the repeated rewriting steps in the CPT rewriting system. The use of these two global variables is in fact an implementation of the merge rules. Note that this improvement does not break the termination property or the correctness property of the proof search method, as we omit repeated rewriting steps on CPTs only.

Thanks to an anonymous referee, we can use the example below to illustrate the application of the global variable storage above-mentioned.

EXAMPLE 7. Consider a Kripke model that has $2n+2$ $(n > 1)$ states: $s_0, s_{1,1}, s_{1,2}, ..., s_{1,n-1}, s_{1,n}, s_{2,0}, ..., s_{2,n}$ and a set $P = \{s_0, s_{1,1}, s_{1,2}, ..., s_{1,n-1}, s_{2,0}, ..., s_{2,n}\}$, that is the property P holds at all states of this model except at state $s_{1,n}$. The graph of transition relation of this model is depicted as follows.

[16] A more detailed explanation of the global variables in the proof search algorithm can be found in Appendix A.



When proving the formula $EG_x(P(x))(s_0)$, we need to rewrite the CPT $\mathsf{cpt}(\vdash EG_x(P(x))(s_0), \mathfrak{t}, \mathfrak{f})$. Suppose that $s_{1,1}$ is searched before $s_{2,0}$ in $\mathsf{Next}(s_0)$, and $s_{1,1}$ is searched before $s_{2,i+1}$ in $\mathsf{Next}(s_{2,i})$ for all i between 0 and $n-1$. The corresponding rewriting steps of the CPT are depicted in Figure 7,

$\mathsf{cpt}(\vdash EG_x(P(x))(s_0), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{1,1}), \mathfrak{t}, \mathsf{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{1,1} \vdash EG_x(P(x))(s_{1,2}), \mathfrak{t}, \mathsf{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{1,n-1} \vdash EG_x(P(x))(s_{1,n}), \mathfrak{t}, \mathsf{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{1,1}), \mathfrak{t}, \mathsf{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathfrak{t}, \mathfrak{f})) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{2,n-1} \vdash EG_x(P(x))(s_{2,n}), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^*$
$\mathsf{cpt}(\Gamma_{2,n} \vdash EG_x(P(x))(s_{2,n}), \mathfrak{t}, \mathfrak{f}) \rightsquigarrow \mathfrak{t}$

FIGURE 7. Rewriting steps of $\mathsf{cpt}(\vdash EG_x(P(x))(s_0), \mathfrak{t}, \mathfrak{f})$

where

$$\Gamma_0 = \{EG_x(P(x))(s_0)\},$$
$$\Gamma_{1,i} = \{EG_x(P(x))(s_0), EG_x(P(x))(s_{1,1}), ..., EG_x(P(x))(s_{1,i})\},$$
$$\Gamma_{2,i} = \{EG_x(P(x))(s_0), EG_x(P(x))(s_{2,0}), ..., EG_x(P(x))(s_{2,i})\}.$$

In Figure 7, before rewriting $\mathsf{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathfrak{t}, \mathfrak{f})$, all states in the path $\pi = s_{1,1}, s_{1,2}, ..., s_{1,n}$ are already visited, and each time we rewrite $\mathsf{cpt}(\Gamma_{2,i} \vdash EG_x(P(x))(s_{2,i+1}), \mathfrak{t}, \mathfrak{f})$ ($i \in \{0, ..., n-1\}$), all states in π need to be visited one more time. That is because our proof search method is in a depth-first search style,

When using a global variable M to store visited states, repeatedly visiting π can be avoided. Indeed, M stores all visited state s for which $EG_x(P(x))(s)$ does not hold, and before we rewrite $\mathsf{cpt}(\Gamma_{2,i} \vdash EG_x(P(x))(s_{2,i+1}), \mathfrak{t}, \mathfrak{f})$ ($i \in \{0, ..., n-1\}$), all states in π are already stored in M and hence no need to be visited again.

The pseudo code of the proof search algorithm is depicted in Figure 8, in which the notations are listed as follows.

- $\vdash \psi$ denotes the sequent to be proved in SCTL.
- c denotes the CPT to be rewritten.
- pt and ce denote the proof tree and counterexample to be constructed during the proof search, respectively.
- $\mathsf{M}^{\mathfrak{t}}$ (resp. $\mathsf{M}^{\mathfrak{f}}$) is used to store the set of visited states on which a sub-formula of ψ holds (resp. does not hold), and they are used to avoid visiting the same state repeatedly.
- $\mathsf{visited}$ is used to store temporally the visited states by sub-formula of ψ, starting with EU or AR (if any), and these states will be added into either $\mathsf{M}^{\mathfrak{t}}$ or $\mathsf{M}^{\mathfrak{f}}$ in later rewriting steps.
- We associate a set of actions A to each CPT c, written as c^{A}. When rewriting a CPT c, each action in A needs to be performed. For instance, for a CPT $\mathsf{c} = \mathsf{cpt}(\Gamma \vdash \phi, \mathsf{c}_1, \mathsf{c}_2)$, three sets of actions $\mathsf{A}_0, \mathsf{A}_1, \mathsf{A}_2$ are associated to $\mathsf{c}, \mathsf{c}_1, \mathsf{c}_2$, respectively. $\mathsf{c}^{\mathsf{A}_0}$ is written as $\mathsf{cpt}^{\mathsf{A}_0}(\Gamma \vdash \phi, \mathsf{c}_1^{\mathsf{A}_1}, \mathsf{c}_2^{\mathsf{A}_2})$. When rewriting c, the actions in A_0 need to be performed. The actions in A_1 cannot be performed until c rewrites to c_1. When rewriting c_1, the actions in A_1 need to be performed. The case of c_2 is similar. Note that A_1 contains actions for constructing the proof tree of $\Gamma \vdash \phi$, since we do not know if $\Gamma \vdash \phi$ is provable until c rewrites to c_1; while A_2 contains actions for constructing the counterexample, and each such an action cannot be performed unless c rewrites to c_2.

There is a "while" loop in the pseudo code, which is to repeatedly rewrite the CPT c, until either $\mathfrak{t}$ or $\mathfrak{f}$ is reached. For the sake of brevity, only the subroutine ProveAnd is presented (Figure 9), where $\mathsf{ac}(\mathfrak{t}, \mathsf{parent}, \mathsf{children})$ represents the operation of adding each element in children as a child to the node parent in the tree $\mathfrak{t}$. These operations are wrapped as actions in the pseudo code. The complete explanation of the pseudo code is in Appendix A.

A note on the complexity of the proof search algorithm. According to the proof search algorithm, for a given CPT $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$, each sub-formula of ϕ appears, in the worst case, $|\mathcal{M}|$ times at the roots of CPTs that appear in the rewriting steps starting with $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$, where $|\mathcal{M}|$ is the number of states in the Kripke model under consideration. Therefore, the time complexity of the proof search algorithm is $O(|\phi| \times |\mathcal{M}|)$, where $|\phi|$ is the size of the formula ϕ to be proved.

4.3. SCTLProV

In this section, we develop a new automated theorem prover called SCTLProV.[17] As is depicted in Figure 10, SCTLProV reads and interprets an input file containing a description of a Kripke model and a finite number of formulae — the properties (of this model) to be verified. It searches for a proof of each of these formulae and outputs a certificate (resp. True) when the verification succeeds, and a counterexample (resp. False), that is a proof of the negation of the formula, when it does not.

A note on the input language of SCTLProV. In the input languages of most traditional model checkers, a state in a Kripke model is usually expressed as a fixed number of values, and the types of these values are relatively simple, for instance, the Boolean data type, the integral data type, enumerated data types, etc. Such input languages usually cannot express complicated data structures such as linked lists with arbitrary finite length. In the input language of SCTLProV, the Kripke model is not always defined extensionally with a list of states and a list of transitions.

[17] https://github.com/sctlprov/sctlprov_code

Input: An SCTL formula ψ.
Output: A pair (r,t), where r is a Boolean, and t is either pt or ce.
function PROOFSEARCH
  let $c = \mathsf{cpt}^{\emptyset}(\vdash \psi, t^{\emptyset}, f^{\emptyset})$
  let $pt = ce = \langle$tree with a single node: $\vdash \psi\rangle$
  let $M^t = M^f = visited = \langle$empty hash table$\rangle$
  while c has the form $\mathsf{cpt}^{A_0}(\Gamma \vdash \phi, c_1^{A_1}, c_2^{A_2})$ **do**
    $\forall a \in A_0$, perform action a
    case ϕ **is**
      $\top$: $c := c_1^{A_1}$
      $\bot$: $c := c_2^{A_2}$
      $P(s_1,\ldots,s_n)$:
        if $\langle s_1,\ldots,s_n\rangle \in P$ **then** $c := c_1^{A_1}$ **else** $c := c_2^{A_2}$ **end if**
      $\neg P(s_1,\ldots,s_n)$:
        if $\langle s_1,\ldots,s_n\rangle \in P$ **then** $c := c_2^{A_2}$ **else** $c := c_1^{A_1}$ **end if**
      $\phi_1 \wedge \phi_2$: ProveAnd$(\vdash \phi_1 \wedge \phi_2)$
      $\phi_1 \vee \phi_2$: ProveOr$(\vdash \phi_1 \vee \phi_2)$
      $EX_x(\phi_1)(s)$: ProveEX$(\vdash EX_x(\phi_1)(s))$
      $AX_x(\phi_1)(s)$: ProveAX$(\vdash AX_x(\phi_1)(s))$
      $EG_x(\phi_1)(s)$: ProveEG$(\Gamma \vdash EG_x(\phi_1)(s))$
      $AF_x(\phi_1)(s)$: ProveAF$(\Gamma \vdash AF_x(\phi_1)(s))$
      $EU_{x,y}(\phi_1,\phi_2)(s)$: ProveEU$(\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s))$
      $AR_{x,y}(\phi_1,\phi_2)(s)$: ProveAR$(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s))$
    end case
  end while
  if $c = t^A$ **then**
    $\forall a \in A$, perform a
    return $(\mathsf{true}, \mathsf{pt})$
  end if
  if $c = f^A$ **then**
    $\forall a \in A$, perform a
    return $(\mathsf{false}, \mathsf{ce})$
  end if
end function

FIGURE 8. The proof search algorithm

let $A_{12} = A_1 \cup \{ac(pt, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1, \vdash \phi_2\})\}$
let $A_{21} = A_2 \cup \{ac(ce, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1\})\}$
let $A_{22} = A_2 \cup \{ac(ce, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_2\})\}$
let $c' = \mathsf{cpt}^{\emptyset}(\vdash \phi_1, \mathsf{cpt}^{\emptyset}(\vdash \phi_2, c_1^{A_{12}}, c_2^{A_{22}}), c_2^{A_{21}})$
$c := c'$

FIGURE 9. ProveAnd$(\vdash \phi_1 \wedge \phi_2)$

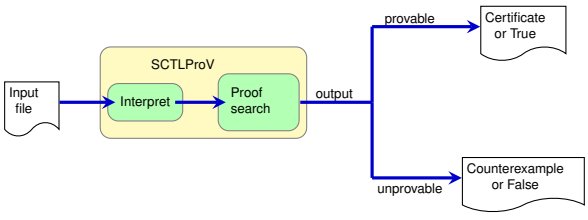


FIGURE 10. A general work flow of SCTLProV

Here the type of states can be any datatype, in particular, it is easy to express linked lists with arbitrary finite length in the language, and the transitions are defined by an arbitrary computable function mapping a state to the list of its successors. This choice allows us to tackle, for instance, the SATS system, that is often difficult to solve for model checkers using a more extensional description of their input.

4.4. Relations with some model checking techniques

Now we discuss the techniques adopted in SCTLProV with those in some other CTL model checking approaches.

4.4.1. BDD-based symbolic model checking

When a Kripke model contains mostly Boolean variables, for instance in model checking for hardware problems, using BDDs to store states is an effective way to reduce space occupation during the verification procedure. The best known BDD-based symbolic model checker is NuSMV [29, 30], as well as its extension NuXMV [31]. To illustrate the verification procedure in a BDD-based symbolic model checker, let us consider, for instance, a Kripke model $\mathcal{M}$ with the initial state s_0 and a transition relation T. To check whether $\mathcal{M}, s_0 \models EF\phi$ holds in such a model checker, say NuSMV, first one needs to calculate the least fixed point $\text{lfp} = \mu Y.(\phi \vee EXY)$, then check whether $s_0 \in \text{lfp}$ [29, 30]. Calculating lfp corresponds to unfolding the transition relation T, where states that are not reachable from s_0 may be involved.

The verification procedure in SCTLProV differs from traditional CTL symbolic model checkers. For instance, unlike in NuSMV, there is no need for SCTLProV to calculate a fixed point of the transition relation. Instead, unfolding of the transition relation stops as soon as the given property is proved or refuted. Moreover, SCTLProV can store visited states either directly (using hash tables) when the model under consideration contains many non-Boolean variables, or using BDDs when the model contains mostly Boolean variables. In the latter case, unlike NuSMV that encodes models and properties into BDDs before searching state space, SCTLProV searches states directly on the Kripke model, using BDDs to store the visited states only.

4.4.2. On-the-fly model checking

The on-the-fly style of searching state space helps to avoid exploring unneeded states. Indeed, in an on-the-fly model checker, there is usually no need to generate the full state space. Traditional on-the-fly CTL model checking algorithms [14, 13] are usually recursive, i.e., the unfolding of the formula and the transition rules are preformed recursively. These recursive based algorithms usually involve a lot of stack operations when verifying properties of large Kripke models. These stack operations may consume a lot of time and stack space during the verification processes.

In SCTLProV, the proof search of a formula mimics a double on-the-fly style model checking, that is, unfolding on demand both the transition relation of the given Kripke model and the formula to be verified. However, unlike traditional on-the-fly model checking algorithms, our algorithm is in continuation-passing style, which contains only constant stack operations [26]. In the programming language theory, a *continuation* is an explicit representation of the *the rest of the computation*. A function is said to be in continuation-passing style (CPS), if it takes an extra argument, the continuation, which decides what will happen

to the result of the function. This method, usually used in compiling and programming, can help, among others, to reduce considerably the size of stacks [32, 15, 26].

We would like to compare our algorithm in SCTLProV to those given in [14] and in [13], respectively. However, as far as we know, there are no tools based on these algorithms that can fully solve CTL model checking problems. To show that using continuation-passing style is not a trivial improvement, we designed specifically a recursion variant of SCTLProV, called SCTLProV$_R$.[18] Unlike SCTLProV that uses continuations, SCTLProV$_R$ uses recursive calls to prove sub-formulae and search state space. We will compare the experimental results of SCTLProV and SCTLProV$_R$ in Section 6.2.

4.4.3. Bounded model checking

In traditional BMC tools, temporal formulae are unfolded on a set of traces with limited length once for all. For example, in model checking $\mathcal{M}, s_0 \models_{k+1} EF\phi$, one unfolding step of the EF formula involves k unfolding steps of the transition relation T, that is, BMC tools need to deal with the bulky formula [16]:

$$[\mathcal{M}, EF\phi]_{k+1} := \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} \phi(s_j)$$

To avoid exploring unnecessary states in $\mathcal{M}$, SCTLProV unfolds on demand the transition relation T. Thus, in SCTLProV, one unfolding step of a formula involves at most one unfolding step of the transition relation. Indeed, to verify $\vdash EF_x(\phi)(s_0)$, SCTLProV unfolds the transition relation T and the formula $EF_x(\phi)(s_0)$ as

$$\begin{array}{l}\mathsf{unfold}(S, EF_x(\phi)(s_i)) := \\ \phi(s_i) \vee ((s_i \notin S) \wedge T(s_i, s_{i+1}) \wedge \mathsf{unfold}(S \cup \{s_i\}, EF_x(\phi)(s_{i+1})))\end{array}$$

where S is a set representing the visited states during the proof search, which is in fact the implementation of the merge rule of Figure 4.

5. SCTL AND FAIRNESS CONSTRAINTS

Fairness is an important aspect in verifying concurrent systems. Fairness assumptions often rule out unrealistic behaviors, and are often necessary to establish liveness properties [3]. For instance, in a mutual exclusion algorithm of two processes, we usually need to consider a fair scheduling of the execution of the processes, i.e., no process waits infinitely long. Such fairness constraints cannot be directly expressed in SCTL, since constraints over paths cannot be defined in SCTL. Here we define SCTL formulae under fairness constraints. Our definition of fairness coincides with that in [29], i.e., the path quantifiers apply to those paths along which each formula in a given set holds infinitely often.

In the rest of this section, we suppose without loss of generality that a Kripke model is given implicitly.

We define the fairness constraint C as a finite set of SCTL formulae. An infinite path is fair under fairness constraint C if and only if, for each $\phi \in C$, ϕ is valid infinitely often on this path. An SCTL formula $EG_x(\phi)(t)$ under fairness constraint C, written as $E_C G_x(\phi)(t)$, is valid if and only if there exists an infinite path, fair under C, starting from state t such that for all state s on this path, $(s/x)\phi$ is valid. Similarly, an SCTL formula $AF_x(\phi)(t)$ under fairness constraint C, written as $A_C F_x(\phi)(t)$, is valid if and only if for each infinite path, fair under C, starting from state t such that there exists a state s on this path and $(s/x)\phi$ is valid.

Similar to [29], other SCTL formulae under fairness constraints can be characterized as follows:

$$\begin{array}{c}E_C X_x(\phi)(t) = EX_x(\phi \wedge E_C G_x(\top)(x))(t) \\ A_C X_x(\phi)(t) = AX_x(\phi \vee A_C F_x(\bot)(x))(t) \\ E_C U_{x,y}(\phi_1, \phi_2)(t) = EU_{x,y}(\phi_1, \phi_2 \wedge E_C G_z(\top)(y))(t) \\ A_C R_{x,y}(\phi_1, \phi_2)(t) = AR_{x,y}(\phi_1, \phi_2 \vee A_C F_z(\bot)(y))(t)\end{array}$$

Since SCTL is sound and complete, to prove $E_C G_x(\phi)(t)$ is equivalent to prove $EG_x(\phi)(t)$ where only fair paths are considered, i.e., to prove the existence of a fair path on which ϕ is always provable. Similarly, to prove $A_C F_x(\phi)(t)$ is equivalent to prove $AF_x(\phi)(t)$ where only fair paths are considered, i.e., to prove the absence of a fair path on which ϕ can never be provable. Thus, to prove SCTL formulae under fairness constraints, we need a mechanism to decide the existence of fair paths. The merge rules in SCTL and Proposition 5.1 and 5.2 below ensure that we can decide the existence of a fair path in finitely many steps. To be more precise, when applying the merge rules, we check the fairness of the paths constructed and discard those that are not fair: i.e., we only consider merges where each formula in the given fairness constraint C is provable at some state of a cycle.

PROPOSITION 5.1. *Let C be a finite set of SCTL formulae and $\sigma = s_0, s_1, \ldots$ be an infinite sequence of states such that $s_i \longrightarrow s_{i+1}$ for all i. If every element of C is valid infinitely often on σ, then there exists a finite sequence of states $\sigma_f = s'_0, s'_1, \ldots, s'_n$ such that*

- *each s'_j is a state on σ,*
- *$s'_j \longrightarrow s'_{j+1}$ for all $0 \le j \le n-1$,*
- *$s'_n = s'_p$ for some $0 \le p \le n-1$,*
- *for every $\phi \in C$, ϕ is valid at some state s'_q, where $p \le q \le n$.*

Proof. As the number of states is finite, there exists a finite set of states S, such that each state in S appears infinitely often in σ, and each state not in S appears finitely often in σ. Then, each formula in C must be valid at some state in S, this is because otherwise, if there exists a formula $\phi \in C$ which is not valid on any state in S, then ϕ must be valid only finitely often on σ. Assume $S = \{s_{i_1}, s_{i_2}, \ldots, s_{i_k}\}$ such that $i_1 \le i_2 \le \ldots \le i_k$, then let σ_f be a prefix of σ: $\sigma_f = s'_0, s'_1, \ldots, s'_n = s_0, \ldots, s_{i_1}, \ldots, s_{i_2}, \ldots, s_{i_k}, \ldots, s_{i_1}, \ldots, s_{i_2}, \ldots, s_{i_k}$. It is easy to check that all the requested conditions are satisfied. □

PROPOSITION 5.2. *Let C be a finite set of SCTL formulae, and $\sigma_f = s_0, s_1, \ldots, s_n$ be a finite sequence of states such that*

[18] https://github.com/sctlprov/sctlprov_r_code

- $s_i \longrightarrow s_{i+1}$ *for all* $0 \leq i \leq n-1$,
- $s_p = s_n$ *for some* $0 \leq p \leq n$,
- *every formula in C is valid at some state between* s_p *and* s_n *in the sequence.*

Then there exists an infinite sequence of states $\sigma = s'_0, s'_1, \ldots$ *such that*

- *each* s'_j *is a state on* σ_f,
- $s'_i \longrightarrow s'_{i+1}$ *for all* $i \geq 0$,
- *every formula in C is valid infinitely often on* σ.

Proof. It is sufficient to take

$$\sigma = s_0, \ldots, s_{p-1}, s_p, \ldots, s_{n-1}, s_p, \ldots, s_{n-1}, \ldots$$

□

6. EXPERIMENTAL RESULTS AND APPLICATIONS

In this section, we discuss the feasibility and the efficiency of SCTLProV. First, we use a simple example to show the performance of SCTLProV, then we evaluate several benchmarks and compare the experimental results with the following verification tools: a Resolution-based theorem prover iProver Modulo [33], a QBF-based bounded model checker Verds (version 1.49), a BDD-based unbounded model checker NuSMV (version 2.6.0) and its extension NuXMV (version 1.0.0), and a formal verification toolbox CADP. Finally, we apply SCTLProV to a safety problem in air traffic control.

All benchmarks used in this paper are available online.[19] All examples and benchmarks are tested on a Linux platform with 3.0 GB memory and a 2.93GHz × 4 CPU, and the time limit is 20 minutes.

6.1. An illustrative example

EXAMPLE 8 (A mutual exclusion problem [34]). This example concerns a mutual exclusion algorithm of two concurrent processes (process *A* and process *B*). *Mutual Exclusion* means that at any time, the number of processes in the critical section is at most one. A sketch of the algorithm is depicted in Figure 11, where "`flag`" indicates that if there exists a process running at the moment, and "`mutex`" (has the initial value 0) is the number of processes that have entered the critical section. A violation of *Mutual Exclusion* means that at some state of the program, more than one process have entered the critical section, that is, the value of "`mutex`" is 2.

We use SCTLProV to check this algorithm.

The input file for this problem is depicted in Figure 12, where both the Kripke model and the property to be verified are defined. More precisely, two additional variables "`a`" and "`b`" are used to indicate the program counters of the two processes, respectively; "`ini`" stands for the initial state; "`bug(s)`" stands for the atomic formula meaning that *at state "`s`" both process A and process B are in the critical section*, that is, *a bug of the algorithm is found at state "`s`"*; and "`find_bug`" stands for the property to be checked: from the initial state, a state "`s`" such that "`bug(s)`" holds is always reachable.

[19] https://github.com/sctlprov/sctlprov_benchmarks

```
/* Process A */                 /* Process B */
1: while(flag);/*wait*/         1: while(flag);/*wait*/
2: flag = true;                 2: flag = true;
3: mutex ++;                    3: mutex ++;
/*critical section*/            /*critical section*/
4: mutex --;                    4: mutex --;
5: flag = false;                5: flag = false;
```

FIGURE 11. A sketch of process A and process B

```
Model mutual()
{
Var {
flag : Bool; mutex : (0 .. 2);
a : (1 .. 5); b : (1 .. 5);
}
Init {
flag := false; mutex := 0; a := 1; b := 1;
}
Transition {
a = 1 && flag = false : {a := 2;};
a = 2 : {a := 3; flag := true;};
/*A has entered the critical section*/
a = 3 : {a := 4; mutex := mutex + 1;};
/*A has left the critical section*/
a = 4 : {a := 5; mutex := mutex - 1;};
a = 5 : {flag := 0;};
b = 1 && flag = false : {b := 2;};
b = 2 : {b := 3; flag := true;};
/*B has entered the critical section*/
b = 3 : {b := 4; mutex := mutex + 1;};
/*B has left the critical section*/
b = 4 : {b := 5; mutex := mutex - 1;};
b = 5 : {flag := 0;};
/*If none of the conditions above are satisfied,
then the current state goes to itself.*/
(a = 1 || b = 1) && flag = true: {}
}
Atomic {bug(s) := s(mutex = 2);}
Spec{find_bug := EU(x, y, TRUE, bug(y), ini);}
}
```

FIGURE 12. The input file "mutual.model"

We check this property by using the following command.

```
sctl -output output.out mutual.model
```

The result is as follows, which indicates that the property "`find_bug`" holds for the Kripke model, that is, the mutual exclusion property is violated in the model, and hence a bug is found in the algorithm above-mentioned.

```
verifying on the model mutual...
find_bug: EU(x,y, TRUE, bug(y), ini)
find_bug is true.
```

The output file "output.out" contains a proof tree of the verified property (Figure 13). Each node of the tree is in the form of

$$id : seqt\ [id_1, \ldots, id_n]$$

where *seqt* stands for a sequent, *id* the identity number of the sequent, and $id_1, \ldots, id_n$ the identity numbers of the premises of the sequent.

```
0: |- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=1;b:=1})
[4, 1]
4: {flag:=false;mutex:=0;a:=1;b:=1}
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=1})
[7, 5]
1: |- TRUE []
7: {flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=2})
[23, 20]
5: |- TRUE []
23:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=0;a:=3;b:=2})
[27, 24]
20: |- TRUE []
27:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=2})
[31, 28]
24: |- TRUE []
31:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=3})
[35, 32]
28: |- TRUE []
35:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
{flag:=true;mutex:=1;a:=4;b:=3}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=2;a:=4;b:=4})
[37]
32: |- TRUE []
37: |- bug({flag:=true;mutex:=2;a:=4;b:=4})  []
```

FIGURE 13. The output file "output.out"

This proof tree can be visualized by a visualization tool VMDV[20] (Visualization for Modeling, Demonstration, and Verification). VMDV is a 3D visualization tool for theorem provers, which provides interfaces for the integration with different theorem provers [35]. As depicted in Figure 14, we use in fact VMDV to visualize in 3D space both the proof tree and the Kripke model provided by SCTLProV , where the root node of the proof tree, containing the information about the verified property, is highlighted in red and so does the corresponding state (i.e., the initial state) in the Kripke model.

Note that the violation of the mutual exclusion property can be avoided if the algorithm is modified (Figure 15). We use SCTLProV to check the modified algorithm: the input file is depicted in Figure 16, where variable "x" and variable "y" are used to indicate that if process *A* and process *B* are running, respectively; and variable "turn" is used to indicate that it is whose turn to enter the critical section.

The output below shows that the modified algorithm is indeed a solution for the mutual exclusion exclusion problem.

```
verifying on the model mutual...
find_bug: EU(x, y, TRUE, bug(y), ini)
find_bug is false.
```

[20] https://github.com/terminatorlxj/VMDV

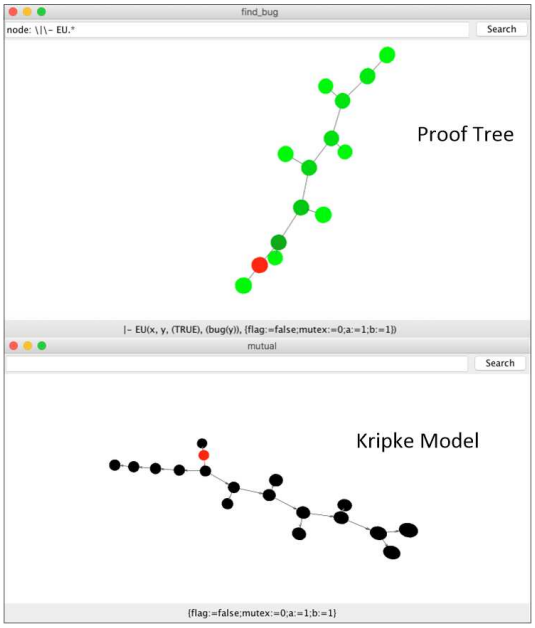


FIGURE 14. Visualization of the proof tree and the Kripke model in the illustrative example

```
/* Process A */
1: x = true;
2: turn = 1;
/*wait*/
3: while(y&&turn!=2);
4: mutex ++;
/*critical section*/
5: mutex --;
6: x = false;
```

```
/* Process B */
1: y = true;
2: turn = 2;
/*wait*/
3: while(x&&turn!=1);
4: mutex ++;
/*critical section*/
5: mutex --;
6: y = false;
```

FIGURE 15. A simple solution of the mutual exclusion problem

6.2. Randomly generated programs

We consider two benchmarks in this part. Benchmark #1 is originally proposed by Zhang [9]. Benchmark #2 is based on benchmark #1 by increasing the number of state variables. The randomness of the test cases in these benchmarks makes it rather fair for different CTL model checkers, and helps us understand the strengths and weaknesses of each of the tools.

6.2.1. Benchmark #1

Benchmark #1 was originally introduced by Zhang [9] in the evaluation of model checkers Verds and NuSMV, later, Ji [10] re-used it in the evaluation of the theorem prover iProver Modulo and the model checker Verds. For self containment of this paper, we briefly restate it here. This benchmark consists of 2880 randomly generated test cases where two types of random Boolean programs are considered—Concurrent Processes and Concurrent Sequential Processes.

Programs with Concurrent Processes. The parameters of the first set of random Boolean programs are as follows.

```
Model mutual()
{
Var {
x:Bool; y:Bool; mutex:(0 .. 2);
turn:(1 .. 2);
a:(1 .. 6); b:(1 .. 6);}
}
Init {
x := false; y := false;
mutex := 0; turn := 1;
a := 1; b := 1;
}
Transition {
a = 1 : {a := 2; x := true;};
a = 2 : {a := 3; turn := 1;};
a = 3 && (y = false || turn = 2): {a := 4;};
/*A has entered the critical section*/
a = 4 : {a := 5; mutex := mutex + 1;};
/*A has left the critical section*/
a = 5 : {a := 6; mutex := mutex - 1;};
a = 6 : {x := false;};
b = 1 : {b := 2; y := true;};
b = 2 : {b := 3; turn := 2;};
b = 3 && (x = false || turn = 1): {b := 4;};
/*B has entered the critical section*/
b = 4 : {b := 5; mutex := mutex + 1;};
/*B has left the critical section*/
b = 5 : {b := 6; mutex := mutex - 1;};
b = 6 : {y := false;};
/*If none of the conditions above are satisfied,
then the current state goes to itself.*/
(a != 3 && (y = true && turn = 1)) ||
(b != 3 && (x = true && turn = 2)) : {};
}
Atomic {bug(s) := s(mutex = 2);}
Spec {find_bug := EU(x, y, TRUE, bug(y), ini);}
}
```

FIGURE 16. The input file for the modified algorithm

a: number of processes
b: number of all variables
c: number of shared variables
d: number of local variables in a process

Each shared variable is initially set to a random value in $\{0,1\}$, and each local variable is initially set to 0. For each process, each shared or local variable is set to the negation of a variable randomly chosen from all variables. One test different sizes of the programs with 3 processes ($a = 3$), and let b vary over the set of values $\{12,24,36\}$, then set $c = b/2, d = c/a$.

Programs with Concurrent Sequential Processes. Apart from a,b,c,d specified above, another two parameters for the second set of random Boolean programs are as follows.

t: number of transitions in a process
p: number of parallel assignments in each transition

For each Concurrent Sequential Process, in addition to b Boolean variables, there is a local variable representing the program location, which has c possible values. Each shared variable is initially set to a random value in $\{0,1\}$, and each local variable is initially set to 0. For each transition of a process, p pairs of shared and local variables are randomly chosen among all variables, such that the first element of such a pair is set to the negation of the second element of the pair. Transitions are numbered from 0 to $t-1$, and are executed consecutively, and when the end of the sequence of the transitions is reached, it loops back to the execution of the transition numbered 0. For this type of programs, we test different sizes of the programs with 2 processes ($a = 2$), and let b vary in the set of values $\{12,16,20\}$, and then set $c = b/2, d = c/a, t = c$, and $p = 4$.

There are 24 properties to be checked in this benchmark: properties P_{01} to P_{12} are depicted in Table 1, and P_{13} to P_{24} are simply the variations of P_{01} to P_{12} by replacing $\wedge$ and $\bigvee$ by $\vee$ and $\bigwedge$, respectively. In both programs with Concurrent Processes and programs with Concurrent Sequential Processes, each of 24 properties is tested on 20 test cases for each value of b.

P_{01}	$AG(\bigvee_{i=1}^{c} v_i)$
P_{02}	$AF(\bigvee_{i=1}^{c} v_i)$
P_{03}	$AG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$
P_{04}	$AG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$
P_{05}	$EG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$
P_{06}	$EG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$
P_{07}	$AU(v_1, AU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{08}	$AU(v_1, EU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{09}	$AU(v_1, AR(v_2, \bigvee_{i=3}^{c} v_i))$
P_{10}	$AU(v_1, ER(v_2, \bigvee_{i=3}^{c} v_i))$
P_{11}	$AR(AXv_1, AXAU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{12}	$AR(EXv_1, EXEU(v_2, \bigvee_{i=3}^{c} v_i))$

TABLE 1. Properties $P_{01}, P_{02}, \ldots, P_{12}$ to be checked in benchmark #1 and #2

6.2.2. *Benchmark #2*

In benchmark #2, the value of b in benchmark #1 is increased to $\{48,60,72,252,504,1008\}$ for Concurrent Processes, and $\{24,28,32,252,504,1008\}$ for Concurrent Sequential Processes, respectively. There are 5760 test cases in benchmark #2. Like in benchmark #1, all test cases in benchmark #2 are also randomly generated. The properties to be checked are the same as those in benchmark #1.

6.2.3. *Experimental data*

The experimental results for benchmark #1 and #2 are presented as follows. The interested reader is referred to Appendix B for the detailed data.

Experimental data for benchmark #1. For 2880 test cases in this benchmark, iProver Modulo can solve 1816 (63.1%) cases, Verds can solve 2230 (77.4%) cases, SCTLProV can solve 2870 (99.7%) cases, and both NuSMV and NuXMV can solve all (100%) cases. The numbers of test cases where SCTLProV runs faster are 2823 (98.0%) comparing with iProver Modulo, 2858 (99.2%) comparing with Verds, 2741 (95.2%) comparing with NuSMV, and 2763 (95.9%) comparing with NuXMV.

Experimental data for benchmark #2. For 5760 test cases in this benchmark, iProver Modulo can solve 2748 (44.7%) cases, Verds can solve 2226 (38.6%) cases, NuSMV can solve 728 (12.6%) cases, NuXMV can solve 736 (12.8%) cases, and SCTLProV can solve 4441 (77.1%) cases. The numbers of test cases where SCTLProV runs faster are 4441 (77.1%) comparing with iProver Modulo, 4438 (77.0%) comparing with Verds, and 4432 (76.9%) compar-

ing both with NuSMV and with NuXMV.

Table 2 shows the test cases (in benchmark#1 and #2) that can be solved by one and only one of the five tools.[21] For instance, the fourth line of the table shows that, among 480 test cases with Concurrent Processes, each having 48 state variables, there are 15 (3.1%) cases can only be solved by NuSMV or NuXMV, whereas 46 (9.6%) cases can only be solved by SCTLProV. To sum up, Table 2 shows that among 8640 test cases in benchmark #1 and #2, there are 61 (0.7%) cases that can only be solved by NuSMV or NuXMV, and 1250 (14.5%) cases can only be solved by SCTLProV. It also shows that all test cases that are solvable by either iProver Modulo or Verds can be also solved by one of the other three tools.

Programs	iProver Modulo	Verds	NuXMV / NuXMV	SCTLProV
CP ($b=12$)	0	0	0	0
CP ($b=24$)	0	0	0	0
CP ($b=36$)	0	0	10 (2.1%)	0
CP ($b=48$)	0	0	15 (3.1%)	46 (9.6%)
CP ($b=60$)	0	0	2 (0.4%)	37 (7.7%)
CP ($b=72$)	0	0	0	54 (11.3%)
CP ($b=252$)	0	0	0	72 (15.0%)
CP ($b=504$)	0	0	0	43 (9.0%)
CP ($b=1008$)	0	0	0	7 (1.5%)
CSP ($b=12$)	0	0	0	0
CSP ($b=16$)	0	0	6 (1.3%)	0
CSP ($b=20$)	0	0	2 (0.4%)	0
CSP ($b=24$)	0	0	17 (3.5%)	55 (11.5%)
CSP ($b=28$)	0	0	9 (1.9%)	188 (39.2%)
CSP ($b=32$)	0	0	0	194 (40.4%)
CSP ($b=252$)	0	0	0	176 (36.7%)
CSP ($b=504$)	0	0	0	187 (39.0%)
CSP ($b=1008$)	0	0	0	191 (39.8%)
Sum	0	0	61 (0.7%)	1250 (14.5%)

TABLE 2. Test cases in benchmark #1 and #2 that can be solved by one and only one of the five tools

Figure 17 and Figure 18 show respectively that SCTLProV uses less time and space than the other four verification tools.

6.2.4. *Continuation vs. recursion*

To show the importance of using continuation-passing style in the proof search algorithm, we have implemented a recursive version of SCTLProV, denoted by SCTLProV$_R$, and compared the time efficiency between them. In benchmark #1 and #2, SCTLProV solves about 10% more test cases than SCTLProV$_R$, and it outperforms SCTLProV$_R$ in most solvable cases (Table 3). Indeed, SCTLProV$_R$ is more sensitive to the number of state variables than SCTLProV (Figure 19).

REMARK 1. In the comparison of average verification time between SCTLProV and SCTLProV$_R$, we extend the number of state variables in the case of Concurrent Sequential Processes by adding $b = 52$ and $b = 72$ in benchmark #2.

[21] We do not make a distinction between the experimental data for NuSMV and for NuXMV, because they are almost the same.

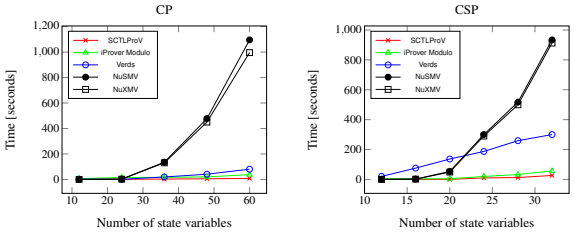


FIGURE 17. Average verification time in benchmark #1 and #2

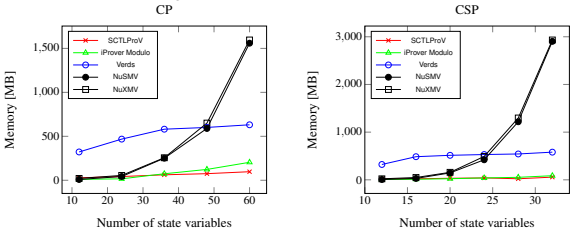


FIGURE 18. Average memory usage in benchmark #1 and #2

Bench	Solvable		t(SCTLProV) < t(SCTLProV$_R$)
	SCTLProV	SCTLProV$_R$	
#1	2862 (99.4%)	2682 (93.1%)	2598 (90.2%)
#2	4446 (77.2%)	3826 (66.4%)	3841 (71.9%)

TABLE 3. SCTLProV vs. SCTLProV$_R$

6.3. Programs with fairness constraints

In this part, first we evaluate benchmark #3, which models mutual exclusion algorithms and ring algorithms[22]. Then, we compare the experimental results of SCTLProV, Verds, NuSMV, and NuXMV. We do not consider iProver Modulo here, because iProver Modulo cannot handle CTL properties under fairness constraints [10].

Benchmark #3 consists of two sets of concurrent programs: the mutual exclusion algorithms and the ring algorithms. Each kind of algorithms consists of a set of concurrent processes running in parallel.

In the mutual exclusion algorithms, the scheduling of processes is simple: for all i between 0 and $n-2$, process $i+1$ performs a transition after process i, and process 0 performs a transition after process $n-1$. Each formula in the algorithms needs to be verified under the fairness constraint, that is, no process waits infinitely long.

Each process in the mutual exclusion algorithms has three internal states: noncritical, trying, and critical. The number of processes varies from 6 to 51. There are five properties specified by CTL formulae to be verified in mutual exclusion algorithms (Table 4). In these formulae, non_i (resp. try_i, cri_i) indicates that process p_i has internal state noncritical (resp. trying, critical). According to the scheduling algorithm, processes 0 and 1 are not symmetric, as exemplified by the difference in performance between the properties P_4 and P_5.

Each process in the ring algorithms consists of 5 Boolean internal variables indicating the internal state, and a Boolean variable indicating the output. Each process receives a

[22] http://lcs.ios.ac.cn/~zwh/verds/verds_code/bp12.rar

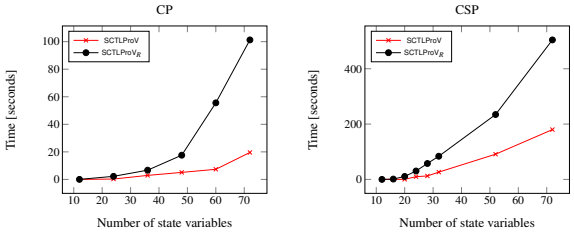


FIGURE 19. Average verification time in SCTLProV vs. SCTLProV$_R$

Prop	Mutual Exclusion Algorithms
P_1	$EF(cri_0 \wedge cri_1)$
P_2	$AG(try_0 \Rightarrow AF(cri_0))$
P_3	$AG(try_1 \Rightarrow AF(cri_1))$
P_4	$AG(cri_0 \Rightarrow Acri_0U(\neg cri_0 \wedge A\neg cri_0Ucri_1))$
P_5	$AG(cri_1 \Rightarrow Acri_1U(\neg cri_1 \wedge A\neg cri_1Ucri_0))$

TABLE 4. Properties to be verified in the mutual exclusion algorithms

Boolean value as the input during its running time. For a ring algorithm with processes $p_0, p_1, ..., p_n$, the internal state of p_i depends on the output of process p_{i-1}, and the output of p_{i-1} depends on its internal state, where $1 \leq i \leq n$. The internal state of p_0 depends on the output of process p_n, and the output of p_n depends on the internal state of its own. The number of processes varies from 3 to 10. There are four properties specified by CTL formulae to be verified in ring algorithms (Table 5). In these formulae, out_i indicates that the output of process p_i is Boolean value *true*.

Prop	Ring Algorithms
P_1	$AGAFout_0 \wedge AGAF\neg out_0$
P_2	$AGEFout_0 \wedge AGEF\neg out_0$
P_3	$EGAFout_0 \wedge EGAF\neg out_0$
P_4	$EGEFout_0 \wedge EGEF\neg out_0$

TABLE 5. Properties to be verified in the ring algorithms

Experimental data for benchmark #3. Among 262 test cases in this benchmark, Verds can solve 168 (64.1%) cases, both NuSMV and NuXMV can solve 71 (27.1%) cases, and SCTLProV can solve 211 (80.2%) cases (Table 6). There are 194 (74.0%) test cases on which SCTLProV runs faster and uses less memory comparing with Verds, and 211 (80.2%) comparing both with NuSMV and with NuXMV (Table 7).

Programs	Verds	NuSMV	NuXMV	SCTLProV
mutual exclusion	136 (59.1%)	50 (21.7%)	50 (21.7%)	191 (83.0%)
ring	32 (100%)	21 (65.6%)	21 (65.6%)	20 (62.5%)
Sum	168 (64.1%)	71 (27.1%)	71 (27.1%)	211 (80.2%)

TABLE 6. The solvable cases in Verds, NuSMV, NuXMV, and SCTLProV

Table 8 shows the test cases in benchmark #3 that can be solved by one and only one of the four tools. There are 5 (1.9%) test cases that can only be solved by Verds, and there are 66 (25.2%) test cases that can only be solved by SCTLProV. There is no test case that can only be solved by NuSMV or by NuXMV.

Programs	Verds	NuSMV	NuXMV
mutual exclusion	187 (81.3%)	191 (83.0%)	191 (83.0%)
ring	7 (21.9%)	20 (62.5%)	20 (62.5%)
Sum	194 (74.0%)	211 (80.2%)	211 (80.2%)

TABLE 7. The test cases where SCTLProV both runs faster and uses less memory than Verds, NuSMV, and NuXMV, respectively

Programs	Verds	NuSMV/NuXMV	SCTLProV
mutual exclusion	0	0	66 (28.7%)
ring	5 (15.6%)	0	0
Sum	5 (1.9%)	0	66 (25.2%)

TABLE 8. Test cases that can be solved by one and only one of the four tools

The detailed experimental data for benchmark #3 are shown in Appendix B.3.

6.4. The VLTS benchmark

In this part, we evaluate benchmark #4, which is also called the VLTS (Very Large Transition Systems) benchmark[23], which was originally proposed as a part of the CADP[24] (Construction and Analysis of Distributed Processes) toolbox [36]. As a formal verification toolbox, CADP focuses on action-based models, for instance, Labeled Transition Systems (LTSs) and Markov Chains. As pointed out by the authors of CADP: *this benchmark has been obtained from the modeling of various communication protocols and concurrent systems, many of which correspond to real life and industrial systems.*

There are 40 test cases in benchmark #4, where deadlocks and livelocks are to be detected for each test case. All test cases in benchmark #4 are encoded in BCG (Binary-Coded Graphs) format, which is a binary format designed for encoding large state spaces [36]. When verifying test cases in this benchmark in SCTLProV, each BCG file was parsed into a Kripke model, and perform the proving procedure on the Kripke model. We compare the experimental results on benchmark #4 between SCTLProV and CADP.

Experimental data for benchmark #4. For 40 test cases in this benchmark, when detecting deadlocks, SCTLProV uses less time than CADP in 33 (82.5%) cases, and uses less memory than CADP in 7 (17.5%) cases; when detecting livelocks, SCTLProV uses less time than CADP in 22 (55%) cases, and uses less memory than CADP in 6 (15%) cases (Table 9).

Cases	t(SCTLProV) < t(CADP)	m(SCTLProV) < m(CADP)
deadlock	33 (82.5%)	7 (17.5%)
livelock	22 (55.0%)	6 (15.0%)

TABLE 9. Test cases where SCTLProV uses less time and less memory than CADP, respectively

[23] http://cadp.inria.fr/resources/vlts/
[24] http://cadp.inria.fr/

More details about the experiment on benchmark #4 are explained in Appendix B.4.

6.5. Discussion of the experimental results

In the evaluation of benchmark #1 to benchmark #3, the performances of NuSMV, NuXMV, Verds, iProver Modulo, and SCTLProV are affected by two factors: the number of state variables and the type of the property to be checked. The performances of NuSMV and NuXMV are mainly affected by the number of state variables, while the performances of iProver Modulo, Verds, and SCTLProV are mainly affected by the type of the property to be checked. When the number of state variables is rather small (such as in benchmark #1), NuSMV and NuXMV solves more test cases than iProver Modulo, Verds and SCTLProV, but when the number of state variables becomes larger (such as test cases in benchmark #2 and benchmark #3), the other three tools outperform them. On the other hand, when checking properties where nearly all states must be searched (such as *AG* properties), NuSMV and NuXMV usually perform better than iProver Modulo, Verds and SCTLProV. However, when checking the other properties (including some *AG* properties), iProver Modulo, Verds and SCTLProV are more efficient in time and in space, since they usually need to search and check much less states than NuSMV and NuXMV. To be precise, iProver Modulo, Verds and SCTLProV scale up better than NuSMV and NuXMV when checking these properties, and SCTLProV scales up better than both iProver Modulo and Verds, and outperforms them in most solvable cases.

In the evaluation of benchmark #4, the performances of SCTLProV and CADP are affected both by the number of states and the property to be checked. Moreover, SCTLProV runs faster than CADP in more than half of the test cases.

6.6. An application to the analysis of air traffic control protocols

As an application to an engineering problem, we present a concept of operations for the *Small Aircraft Transportation System* (SATS) [11, 12] in SCTLProV[25].

In this concept of operations, the airspace volume surrounding an airport facility, called *the self controlled area*, is divided into 15 zones (Figure 20). For instance, the zone `holding3(right)` is a holding pattern at 3000 feet on the right side of the self controlled area. Each zone contains a list of aircraft and 24 transition rules that specify different SATS-procedures. For instance, the rule **Vertical Entry (right)** specifies the vertical entry of an aircraft in the zone `holding3(right)`.

The model is non-deterministic, that is, for a given state, several transitions are possible and all must be considered. As there are no a priori bounds on the number of aircraft in each zone, the number of states in the model is potentially infinite. However, the number of states that are reachable from the initial state is finite: an enumeration of the model shows that there are 54221 such states (and around 3000 in the simplified model where departure operations are not considered [11]).

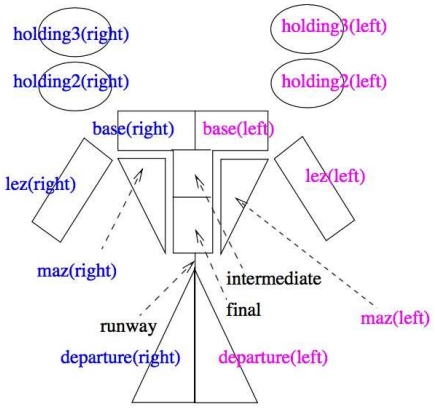


FIGURE 20. SCA zones, where right and left are relative to the pilot facing the runway, i.e., opposite from the reader point of view [11]

There are eight properties of the model that we want to verify with SCTLProV, for instance that the SATS concept does not allow more than four simultaneous landing operations and none of the 15 zones contains too many aircraft (each zone is given a maximum number of aircraft and the actual number of aircraft is never higher than this number). The safety property is the conjunction of these eight properties.

The verification problem is to check that this property holds on every reachable state from the initial state (the state where there are no aircraft on each zone of the self controlled area), so the formula to be checked is $AG_x(\phi)(e)$ where ϕ is the safety property and e is the initial state.

This is a typical model checking problem, but this problem is known to be cumbersome for traditional model checkers [11] because:

- Each state of the model is represented by a complex data structure. For instance, a number of state variables are represented by lists of aircraft with unbounded length.
- The transition rules of the model are represented by complex algorithms. For instance, some transitions rules involve recursive operations on lists of aircraft.
- The properties to be verified in the model are also represented by complex algorithms. For instance, some of the properties are inductively defined over lists of aircraft.

However, this example fits well in SCTLProV which provides a more expressive input language than most traditional model checkers. Indeed, SCTLProV provides both readable notations for the definition of data structures such as records or lists with unbounded length, and arbitrary algorithms for the definitions of transition rules and of

[25] `https://github.com/sctlprov/sctlprov_sats`

properties. So we have been able to check in SCTLProV that the safety property holds on every reachable state from the initial state in the model, and the verification was executed in less than 30 seconds on the same machine as which the benchmarks are evaluated.

7. CONCLUSION AND FUTURE WORKS

This paper provided a first step towards combining model checking and proof checking. We proposed a branching-time logic CTL_P which extends CTL with polyadic predicate symbols. CTL_P permits not only to discuss properties of one fixed state, but also relations between different states. We designed then a proof system SCTL for CTL_P, and developed a new automated theorem prover SCTLProV from scratch, tailored for SCTL. The particular aspects of SCTLProV are as follows: (1) it performs verification automatically and directly over any given Kripke model; (2) in addition to generating counterexamples when the verification of the given property fails, SCTLProV permits to give a certificate for the property when it succeeds; (3) it performs verification in a continuation-passing style and a doubly on-the-fly style, thanks to the syntax and inference rules of SCTL. In a nutshell, SCTLProV can be seen as either a theorem prover or a model checker, when solving CTL model checking problems, SCTLProV can give more instructive information than traditional model checkers, and can use more optimization strategies than traditional theorem provers.

As comparisons to other CTL model checking tools, we considered an automated theorem prover iProver Modulo, a QBF-based bounded model checker Verds, two BDD-based symbolic model checkers: NuSMV and its extension NuXMV, and a formal verification toolbox CADP. There are four benchmarks considered in the comparisons: benchmark #1 was originally introduced by Zhang [9] in the evaluation of Verds and NuSMV, and later, Ji [10] also used this benchmark in the evaluation of iProver Modulo and Verds; benchmark #2 was based on benchmark #1 by extending the number of state variables into tens, hundreds, and even thousands; benchmark #3 concerned verification with fairness constraints; benchmark #4 focused on deadlock and livelock detections. The experimental results show that SCTLProV performs well in terms of time and space consumption and can be considered complementary to model checkers such as NuSMV and NuXMV.

When applying SCTLProV on benchmark #4, LTSs have to be translated into Kripke models, which can increase considerably the number of states. Thus, the efficiency of SCTLProV can be further improved if the verification can perform directly over LTSs. One object of our future works is to design an SCTL-like proof system taking an LTS as the parameter.

So far, fairness constraints cannot be expressed in the syntax of CTL_P nor in SCTL. Another object of our future works is to extend the logic and the proof system in order to formalize fairness constraints.

SCTLProV runs in a single-threaded mode. It seems not straightforward to extend it into a multi-threaded mode, because OCaml does not support thread-level parallelism. So the third object of our future works is to design a parallel version of SCTLProV.

ACKNOWLEDGEMENTS

The authors would like to thank all anonymous referees for their valuable comments which improve significantly the present paper. The first author would like also to thank Ahmed Bouajjani and Wenhui Zhang for the beneficial discussions on model checking. This work was supported by the CAS-INRIA project VIP (GJHZ1844) and the NSFC-ANR project LOCALI (NSFC 61161130530 and ANR-11-IS02-00201).

REFERENCES

[1] Clarke, E. M., Grumberg, O., and Peled, D. (2001) *Model checking*. MIT Press, Cambridge, MA, USA.

[2] Bouajjani, A., Jonsson, B., Nilsson, M., and Touili, T. (2000) Regular model checking. *Proceedings of Computer Aided Verification, 12th International Conference, CAV 2000*, Chicago, IL, USA, 15-19 July, pp. 403–418. Springer-Verlag, Berlin.

[3] Baier, C. and Katoen, J. (2008) *Principles of Model Checking*. MIT Press, USA.

[4] Fitting, M. (1996) *First-Order Logic and Automated Theorem Proving, Second Edition* Graduate Texts in Computer Science. Springer-Verlag, New York.

[5] Loveland, D. W. (1978) *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. Elsevier, Amsterdam.

[6] Burel, G. (2009) Automating theories in intuitionistic logic. *Proceedings of Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009*, Trento, Italy, 16-18 September, pp. 181–197. Springer-Verlag, Berlin.

[7] Emerson, E. A. and Clarke, E. M. (1982) Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, **2**, 241–266.

[8] Emerson, E. A. and Halpern, J. Y. (1985) Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, **30**, 1–24.

[9] Zhang, W. (2014) QBF encoding of temporal properties and QBF-based verification. *Proceedings of IJCAR 2014*, Vienna, 19-22 July, pp. 224–239. Springer-Verlag, Berlin.

[10] Ji, K. (2015) CTL model checking in deduction modulo. *Proceedings of Automated Deduction - CADE-25*, Berlin, 1-7 August, pp. 295–310. Springer International Publishing, Switzerland.

[11] Muñoz, C. A., Dowek, G., and Carreño, V. (2004) Modeling and verification of an air traffic concept of operations. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004*, Boston, Massachusetts, USA, 11-14 July, pp. 175–182. ACM, USA.

[12] NASA/TM-2004-213006 (2004) *Abstract Model of SATS Concept of Operations: Initial Results and Recommendations*. NASA, USA.

[13] Bhat, G., Cleaveland, R., and Grumberg, O. (1995) Efficient on-the-fly model checking for *ctl**. *Proceedings of LICS'95*, San Diego, California, USA, 26-29 June, pp. 388–397. IEEE Computer Society, USA.

[14] Vergauwen, B. and Lewi, J. (1993) A linear local model checking algorithm for CTL. *Proceedings of CONCUR '93, 4th International Conference on Concurrency Theory*, Hildesheim, Germany, 23-26 August, pp. 447–461. Springer-Verlag, Berlin.

[15] Appel, A. W. (2006) *Compiling with Continuations (corr. version)*. Cambridge University Press, UK.

[16] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999) Symbolic model checking without BDDs. In Cleaveland, W. R. (ed.), *Proceedings of TACAS'99*, Amsterdam, the Netherlands, 20-28 March, LNCS, **1579**, pp. 193–207. Springer, USA.

[17] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003) Bounded model checking. *Advances in Computers*, **58**, 117–148.

[18] Fisher, M., Dixon, C., and Peim, M. (2001) Clausal temporal resolution. *ACM Trans. Comput. Log.*, **2**, 12–56.

[19] Gabbay, D. M. and Pnueli, A. (2008) A sound and complete deductive system for ctl* verification. *Logic JOURNAL of the IGPL*, **16**, 499–536.

[20] Pnueli, A. and Kesten, Y. (2002) A deductive proof system for CTL. *Proceedings of CONCUR 2002*, Brno, Czech Republic, 20-23 August, pp. 24–40. Springer-Verlag, Berlin.

[21] Reynolds, M. (2001) An axiomatization of full computation tree logic. *J. Symb. Log.*, **66**, 1011–1057.

[22] Brünnler, K. and Lange, M. (2008) Cut-free sequent systems for temporal logic. *J. Log. Algebr. Program.*, **76**, 216–225.

[23] Partovi, A. and Lin, H. (2014) Assume-guarantee cooperative satisfaction of multi-agent systems. *Proceedings of American Control Conference, ACC 2014*, USA, 4-6 June, pp. 2053–2058. IEEE, USA.

[24] Craig, J. J. (1989) *Introduction to Robotics - Mechanics and Control (2. ed.)*. Prentice Hall, USA.

[25] ISCAS-SKLCS-14-01 (2014) *A Logical Approach to CTL*. ISCAS-SKLCS, Beijing, China.

[26] Sestoft, P. (2012) *Programming Language Concepts*, Undergraduate Topics in Computer Science, **50**. Springer International Publishing, Switzerland.

[27] Dershowitz, N. (1987) Termination of rewriting. *J. Symb. Comput.*, **3**, 69–116.

[28] Sarnat, J. and Schürmann, C. (2009) Lexicographic path induction. *Proceedings of Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009.*, Brasilia, Brazil, 1-3 July, pp. 279–293. Springer, Berlin.

[29] McMillan, K. L. (1993) *Symbolic Model checking*. Springer, USA.

[30] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999) Nusmv: A new symbolic model verifier. *Proceedings of CAV'99*, Trento, Italy, 6-10 July, pp. 495–499. Springer-Verlag, Berlin.

[31] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014) The nuxmv symbolic model checker. *Proceedings of Computer Aided Verification - 26th International Conference, CAV 2014*, Vienna, Austria, 18-22 July, pp. 334–342. Springer International Publishing, Switzerland.

[32] Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbolic Computation*, **6**, 233–248.

[33] Burel, G. (2011) Experimenting with deduction modulo. In Sofronie-Stokkermans, V. and Bjørner, N. (eds.), *Proceedings of CADE 2011*, Wroclaw, Poland, July 31-August 5, pp. 162–176. Springer-Verlag, Berlin.

[34] Peterson, G. L. (1981) Myths about the mutual exclusion problem. *Inf. Process. Lett.*, **12**, 115–116.

[35] Liu, J., Jiang, Y., and Chen, Y. (2017) VMDV: A 3d visualization tool for modeling, demonstration, and verification. *11th International Symposium on Theoretical Aspects of Software Engineering*, Nice, France, 13-15 September, pp. 141–147. IEEE, USA.

[36] Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2013) CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, **15**, 89–107.

APPENDIX

APPENDIX A. PSEUDO CODE OF THE PROOF SEARCH ALGORITHM

In this section, we explain the pseudo code of our proof search algorithm in details.

First, let us define the notion of formula schema. For an $\mathsf{SCTL}(\mathcal{M})$ formula ϕ with modality AF, EG, AR, or EU, a schema ϕ^- of ϕ is defined by replacing the state constant in ϕ by $\_$. For instance, the schema of the formula $EU_{x,y}(\phi_1,\phi_2)(s)$ is $EU_{x,y}(\phi_1,\phi_2)(\_)$, and the schema of the formula $AF_x(\phi')(s')$ is $AF_x(\phi')(\_)$. A formula schema can be viewed as a context of a state constant in an $\mathsf{SCTL}(\mathcal{M})$ formula.

The pseudo code of the proof search algorithm is depicted in Figure A.1, in which the notations are listed as follows.

- $\vdash \psi$ denotes the sequent to be proved in SCTL.
- c denotes the CPT to be rewritten.
- pt and ce denote the proof tree and counterexample to be constructed during the proof search, respectively.
- M^t (resp. M^f) is used to store the set of visited states on which a sub-formula of ψ holds (resp. does not hold), and they are used to avoid visiting the same state repeatedly. Note that M^t and M^f are explained as hash tables. In fact, each of these two global variables remembers a set of states for each formula schema. Thus, a formula schema is a key, and a set of states is a value to these two hash tables. We denote, for instance, $\mathsf{M}^\mathsf{t}_{\mathsf{EG}_x(\phi)(\_)}$ to the set of states S such that $\forall s \in S$, $EG_x(\phi)(s)$ holds, and $\mathsf{M}^\mathsf{f}_{\mathsf{EG}_x(\phi)(\_)}$ to the set of states S such that $\forall s \in S$, $EG_x(\phi)(s)$ does not hold.
- visited is used to store temporally the visited states by sub-formula of ψ, starting with EU or AR (if any), and these states will be added into either M^t or M^f in later rewriting steps. Like M^t and M^f, visited can also be seen as a hash table. For instance, we denote $\mathsf{visited}_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$ to the set of visited states during the proof search of an EU formula.
- We associate a set of actions A to each CPT c, written as c^A. When rewriting a CPT c, each action in A needs to be performed. For instance, for a CPT $\mathsf{c} = \mathsf{cpt}(\Gamma \vdash \phi, \mathsf{c}_1, \mathsf{c}_2)$, three sets of actions $\mathsf{A}_0, \mathsf{A}_1, \mathsf{A}_2$ are associated to $\mathsf{c}, \mathsf{c}_1, \mathsf{c}_2$, respectively. $\mathsf{c}^{\mathsf{A}_0}$ is written as $\mathsf{cpt}^{\mathsf{A}_0}(\Gamma \vdash \phi, \mathsf{c}_1^{\mathsf{A}_1}, \mathsf{c}_2^{\mathsf{A}_2})$. When rewriting c, the actions in A_0 need to be performed. The actions in A_1 cannot be performed until c rewrites to c_1. When rewriting c_1, the

actions in A_1 need to be performed. The case of c_2 is similar. Note that A_1 contains actions for constructing the proof tree of $\Gamma \vdash \phi$, since we do not know if $\Gamma \vdash \phi$ is provable until c rewrites to c_1; while A_2 contains actions for constructing the counterexample, and each such an action cannot be performed unless c rewrites to c_2.

There is a "while" loop in the pseudo code, which is to repeatedly rewrite the CPT c, until either t or f is reached. In the proof search algorithm, the rewriting steps for the cases where ϕ is an atomic formula or the negation of a atomic formula is presented, and the explanation for other cases are presented separately in the following Subsections.

Input: An SCTL formula ψ.
Output: A pair (r,t), where r is a Boolean, and t is either pt or ce.

1: **function** PROOFSEARCH
2: let $c = \mathsf{cpt}^{\emptyset}(\vdash \psi, t^{\emptyset}, f^{\emptyset})$
3: let $pt = ce = \langle$tree with a single node: $\vdash \psi\rangle$
4: let $M^t = M^f =$ visited $= \langle$empty hash table$\rangle$
5: **while** c has the form $\mathsf{cpt}^{A_0}(\Gamma \vdash \phi, c_1^{A_1}, c_2^{A_2})$ **do**
6: $\forall a \in A_0$, perform action a
7: **case** ϕ **is**
8: $\top$: $c := c_1^{A_1}$
9: $\bot$: $c := c_2^{A_2}$
10: $P(s_1,...,s_n)$:
11: **if** $\langle s_1,...,s_n\rangle \in P$ **then** $c := c_1^{A_1}$ **else** $c := c_2^{A_2}$ **end if**
12: $\neg P(s_1,...,s_n)$:
13: **if** $\langle s_1,...,s_n\rangle \in P$ **then** $c := c_2^{A_2}$ **else** $c := c_1^{A_1}$ **end if**
14: $\phi_1 \wedge \phi_2$: ProveAnd$(\vdash \phi_1 \wedge \phi_2)$
15: $\phi_1 \vee \phi_2$: ProveOr$(\vdash \phi_1 \vee \phi_2)$
16: $EX_x(\phi_1)(s)$: ProveEX$(\vdash EX_x(\phi_1)(s))$
17: $AX_x(\phi_1)(s)$: ProveAX$(\vdash AX_x(\phi_1)(s))$
18: $EG_x(\phi_1)(s)$: ProveEG$(\Gamma \vdash EG_x(\phi_1)(s))$
19: $AF_x(\phi_1)(s)$: ProveAF$(\Gamma \vdash AF_x(\phi_1)(s))$
20: $EU_{x,y}(\phi_1,\phi_2)(s)$: ProveEU$(\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s))$
21: $AR_{x,y}(\phi_1,\phi_2)(s)$: ProveAR$(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s))$
22: **end case**
23: **end while**
24: **if** $c = t^A$ **then**
25: $\forall a \in A$, perform a
26: **return** (true, pt)
27: **end if**
28: **if** $c = f^A$ **then**
29: $\forall a \in A$, perform a
30: **return** (false, ce)
31: **end if**
32: **end function**

FIGURE A.1. The proof search algorithm

Appendix A.1. ProveAnd and ProveOr

The pseudo code for the ProveAnd case is depicted in Figure A.2, where $\mathsf{ac}(t, \mathsf{parent}, \mathsf{children})$ represents the operation of adding each element in children as a child to the node parent in the tree t. The pseudo code for this case is explained as follows.

- Line 4: When c_1 is reached from c' in the future steps, both $\vdash \phi_1$ and $\vdash \phi_2$ are provable, then both $\vdash \phi_1$ and $\vdash \phi_2$ should be added as the children of $\vdash \phi_1 \wedge \phi_2$ in the proof tree (Line 1). Otherwise, if either of the inner or outer c_2 is reached from c', then the unprovability of either $\vdash \phi_1$ (Line 2) or $\vdash \phi_2$ (Line 3) is added as the evidence of negating $\vdash \phi_1 \wedge \phi_2$ in the counterexample.
- Line 5: Rewrite c to c'.

1: let $A_{12} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1, \vdash \phi_2\})\}$
2: let $A_{21} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1\})\}$
3: let $A_{22} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_2\})\}$
4: let $c' = \mathsf{cpt}^{\emptyset}(\vdash \phi_1, \mathsf{cpt}^{\emptyset}(\vdash \phi_2, c_1^{A_{12}}, c_2^{A_{22}}), c_2^{A_{21}})$
5: $c := c'$

FIGURE A.2. ProveAnd$(\vdash \phi_1 \wedge \phi_2)$

Figure A.3 shows the details of ProveOr, which is the dual case of ProveAnd.

1: let $A_{22} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_1, \vdash \phi_2\})\}$
2: let $A_{11} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_1\})\}$
3: let $A_{12} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_2\})\}$
4: let $c' = \mathsf{cpt}^{\emptyset}(\vdash \phi_1, c_1^{A_{11}}, \mathsf{cpt}^{\emptyset}(\vdash \phi_2, c_1^{A_{12}}, c_2^{A_{22}}))$
5: $c := c'$

FIGURE A.3. ProveOr$(\vdash \phi_1 \vee \phi_2)$

Appendix A.2. ProveEX and ProveAX

In the case of ProveEX, we let $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and the pseudo code for this case is depicted in Figure A.4. The analysis is analogous to that of Figure A.2, except that in Line 5 and Line 6, when c' rewrites to the ith c_1 in the future steps, then $\vdash (s_i/x)\phi_1$ should be added as the children of $\vdash EX_x(\phi_1)(s)$ in the proof tree (Line 4), and otherwise, when c' rewrites to c_2, then $\vdash (s_1/x)\phi_1,...,\vdash (s_n/x)\phi_1$ should be added as the children of $\vdash EX_x(\phi_1)(s)$ in the counterexample (Line 3).

1: */* For notation purpose, here we refer "k" to $EX_x(\phi_1)(\_)$,* and
2: *"k(s)" to $EX_x(\phi_1)(s)$.*/*
3: $A_2 := A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \vdash \mathsf{k}(s), \{\vdash (s_1/x)\phi_1,...,\vdash (s_n/x)\phi_1\})\}$
4: $\forall i \in \{1,...,n\}$, let $A_{1i} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \vdash \mathsf{k}(s), \{\vdash (s_i/x)\phi_1\})\}$
5: let $c' = \mathsf{cpt}^{\emptyset}(\vdash (s_1/x)\phi_1, c_1^{A_{11}}, \mathsf{cpt}^{\emptyset}(...\mathsf{cpt}^{\emptyset}(\vdash (s_n/x)\phi_1, c_1^{A_{1n}},$
6: $c_2^{A_2})...))$
7: $c := c'$

FIGURE A.4. ProveEX$(\vdash EX_x(\phi_1)(s))$

Figure A.5 shows the details of ProveAX, which is the dual case of ProveEX.

1: */* For notation purpose, here we refer "k" to $AX_x(\phi_1)(\_)$,* and
2: *"k(s)" to $AX_x(\phi_1)(s)$.*/*
3: $A_1 := A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \vdash \mathsf{k}(s), \{\vdash (s_1/x)\phi_1,...,\vdash (s_n/x)\phi_1\})\}$
4: $\forall i \in \{1,...,n\}$, let $A_{2i} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \vdash \mathsf{k}(s), \{\vdash (s_i/x)\phi_1\})\}$
5: let $c' = \mathsf{cpt}^{\emptyset}(\vdash (s_1/x)\phi, \mathsf{cpt}^{\emptyset}(...\mathsf{cpt}^{\emptyset}(\vdash (s_n/x)\phi, c_1^{A_1}, c_2^{A_{2n}}),...),$
6: $c_2^{A_{21}})$
7: $c := c'$

FIGURE A.5. ProveAX$(\vdash AX_x(\phi_1)(s))$

Appendix A.3. ProveEG **and** ProveAF

The pseudo code for the ProveEG case is depicted in Figure A.6, where states(Γ) represents the set of states that appear in Γ. In this case, we let $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and $\Gamma' = \Gamma \cup \{\mathsf{EG}_x(\phi_1)(s)\}$. The pseudo code for this case is explained as follows.

- Line 3 – 4: If $\mathsf{EG}_x(\phi_1)(s)$ does not hold, then just rewrite c to c_2.
- Line 5 – 7: If $\mathsf{EG}_x(\phi_1)(s)$ holds, or $\mathsf{EG}_x(\phi_1)(s) \in \Gamma$, then rewrite c to c_1, and every formula in Γ holds and its corresponding state is thus added into $\mathsf{M}^t_{\mathsf{EG}_x(\phi_1)(\_)}$.
- Line 15 – 16: The CPT c' is constructed with the following information integrated into c'.
  1. If c' rewrites to the outer c_2 in the future steps, then $\vdash (s/x)\phi_1$ is not provable, and thus $\vdash (s/x)\phi_1$ will be added as the evidence of negating $\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$ in the counterexample (Line 9). At the same time, since $\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$ is not provable, we add s into $\mathsf{M}^f_{\mathsf{EG}_x(\phi_1)(\_)}$ (Line 13).
  2. If c' rewrites to the inner c_2, then $\Gamma' \vdash \mathsf{EG}_x(\phi_1)(s_1),...,\Gamma' \vdash \mathsf{EG}_x(\phi_1)(s_n)$ are all added as the evidence of negating $\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$ in the counterexample (Line 10). At the same time, since $\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$ is not provable, we add s into $\mathsf{M}^f_{\mathsf{EG}_x(\phi_1)(\_)}$ (Line 14).
  3. If c' rewrites to the ith c_1, then both $\vdash (s/x)\phi_1$ and $\Gamma' \vdash \mathsf{EG}_x(\phi_1)(s_i)$ are provable, and thus should be added as the evidence of proving $\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$ in the proof tree (Line 11 – 12).
- Line 17: Rewrite c to c'.

1: */* For notation purpose, here we refer "k" to $EG_x(\phi_1)(\_)$ and*
2: *"k(s)" to $EG_x(\phi_1)(s)$.*/*
3: **if** $s \in \mathsf{M}^f_k$ **then**
4: $\quad c := c_2^{A_2}$
5: **else if** $s \in \mathsf{M}^t_k$ or $k(s) \in \Gamma$ **then**
6: $\quad c := c_1^{A_1}$
7: $\quad \mathsf{M}^t_k := \mathsf{M}^t_k \cup \mathsf{states}(\Gamma)$
8: **else**
9: $\quad$ let $A_{20} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1\})\}$
10: $\quad$ let $A_{2n} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_1),...,\Gamma' \vdash k(s_n)\})\}$
11: $\quad \forall i \in \{1,...,n\}$, let
12: $\quad\quad A_{2i} = A_2 \cup \{\mathsf{ac}(\mathsf{pt}, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \Gamma' \vdash k(s_i)\})\}$
13: $\quad A_{10} := A_{10} \cup \{\mathsf{M}^t_k := \mathsf{M}^t_k \cup \{s\}\}$
14: $\quad A_{1n} := A_{1n} \cup \{\mathsf{M}^t_k := \mathsf{M}^t_k \cup \{s\}\}$;
15: $\quad$ let $c' = \mathsf{cpt}^{\emptyset}(\vdash (s/x)\phi_1, c_1^{A_{10}}, \mathsf{cpt}^{\emptyset}(\Gamma' \vdash k(s_1), \mathsf{cpt}^{\emptyset}(...\mathsf{cpt}^{\emptyset}($
16: $\quad\quad \Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...), c_2^{A_{21}}))$
17: $\quad c := c'$
18: **end if**

FIGURE A.6. ProveEG($\Gamma \vdash \mathsf{EG}_x(\phi_1)(s)$)

Figure A.7 shows the details of ProveAF, which is the dual case of ProveEG.

1: */* For notation purpose, here we refer "k" to $AF_x(\phi_1)(\_)$ and*
2: *"k(s)" to $AF_x(\phi_1)(s)$.*/*
3: **if** $s \in \mathsf{M}^t_k$ **then**
4: $\quad c := c_1^{A_1}$
5: **else if** $s \in \mathsf{M}^f_k$ or $k(s) \in \Gamma$ **then**
6: $\quad c := c_2^{A_2}$
7: $\quad \mathsf{M}^f_k := \mathsf{M}^f_k \cup \mathsf{states}(\Gamma)$
8: **else**
9: $\quad$ let $A_{10} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1\})\}$
10: $\quad$ let $A_{1n} = A_1 \cup \{\mathsf{ac}(\mathsf{pt}, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_1),...,\Gamma' \vdash k(s_n)\})\}$
11: $\quad \forall i \in \{1,...,n\}$, let
12: $\quad\quad A_{2i} = A_2 \cup \{\mathsf{ac}(\mathsf{ce}, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \Gamma' \vdash k(s_i)\})\}$
13: $\quad A_{10} := A_{10} \cup \{\mathsf{M}^t_k := \mathsf{M}^t_k \cup \{s\}\}$
14: $\quad A_{1n} := A_{1n} \cup \{\mathsf{M}^t_k := \mathsf{M}^t_k \cup \{s\}\}$;
15: $\quad$ let $c' = \mathsf{cpt}^{\emptyset}(\vdash (s/x)\phi_1, c_1^{A_{10}}, \mathsf{cpt}^{\emptyset}(\Gamma' \vdash k(s_1), \mathsf{cpt}^{\emptyset}(...\mathsf{cpt}^{\emptyset}($
16: $\quad\quad \Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...), c_2^{A_{21}}))$
17: $\quad c := c'$
18: **end if**

FIGURE A.7. ProveAF($\Gamma \vdash \mathsf{AF}_x(\phi_1)(s)$)

Appendix A.4. ProveEU **and** ProveAR

The *EU* case is more complex than the *EG* case. The reason is that the *EG* case terminates when a cycle is detected, while the *EU* case does not terminate after detecting a number of cycles, until a non-cycle that satisfies a certain condition is detected. In this case, we let $\{s_1,...,s_n\} = \mathsf{Next}(s)$, $\Gamma' = \Gamma \cup \{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)\}$, and use reachable(S) to denote the set of visited states that can reach some state in S, where S represents the set of states in a finite path. Each state in reachable(S) is either in S, or in a finite path whose last state lies in a cycle overlapping with S at some state. The set reachable(S) can be calculated on-the-fly by remembering all visited cycles and merges and then selecting all states in each merge whose last state either lies in S, or lies in a cycle overlapping with S at some state.

The pseudo code for ProveEU (Figure A.8) is explained as follows.

- Line 3 – 6: If $\mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$ holds, then just rewrite c to c_1. At the same time, all states in reachable(states(Γ)) are added into $\mathsf{M}^t_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$ and removed from $\mathsf{M}^f_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$.
- Line 7 – 9: If s has been visited, then just rewrite c to c_2. At the same time, all states in states(Γ) are added into $\mathsf{M}^f_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$ to avoid being visited again.
- Line 26 – 27: The CPT c' is constructed with the following information integrated into c'.
  1. If c' rewrites to the first c_1, then $\vdash (s/y)\phi_2$ should be added to the proof tree as the evidence of proving $\Gamma \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$ (Line 12). At the same time, if $(s/y)\phi_2$ holds, then for each state s' that can reach s, $\mathsf{EU}_{x,y}(\phi_1,\phi_2)(s')$ also holds, so we add reachable(states(Γ) $\cup \{s\}$) into $\mathsf{M}^t_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$, and remove all states in reachable(states(Γ) $\cup \{s\}$) from $\mathsf{M}^f_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$ (Line 13 – 14).
  2. If c' rewrites to the ith other c_1, then both

$\vdash (s/x)\phi_1$ and $\Gamma' \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s_i)$ are provable, and thus these two sequents are added to the proof tree as the children of $\Gamma \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$ (Line 19 – 20).

3. If c' rewrites to the outer c_2, then both $\vdash (s/x)\phi_1$ and $\vdash (s/y)\phi_2$ should be added to the counterexample as the children of $\Gamma \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$ (Line 16). At the same time, all states in $\mathsf{states}(\Gamma)\cup\{s\}$ should be added into $M^f_{\mathsf{EU}_{x,y}(\phi_1,\phi_2)(\_)}$ to avoid being visited again (Line 17).
4. If c' rewrites to the inner c_2, then $\Gamma' \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s_1),...,\Gamma' \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s_n)$ should all be added to the counterexample as the children of $\Gamma \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$ (Line 18).

- Line 28: Rewrite c to c'.

Figure A.9 shows the details of ProveAR, which is the dual case of ProveEU.

```
1: /* For notation purpose, here we refer "k" to EU_{x,y}(φ1,φ2)(_) and
2: "k(s)" to EU_{x,y}(φ1,φ2)(s).*/
3: if s ∈ M^t_k then
4:     c := c_1^{A_1}
5:     M^t_k := M^t_k ∪ reachable(states(Γ))
6:     M^f_k := M^f_k \ reachable(states(Γ))
7: else if s ∈ M^f_k or k(s) ∈ Γ then
8:     c := c_2^{A_2}
9:     M^f_k := M^f_k ∪ states(Γ)
10: else
11:     let A10 = A1 ∪ {
12:         ac(pt, Γ ⊢ k(s), {⊢ (s/y)φ2}),
13:         M^t_k := M^t_k ∪ reachable(states(Γ) ∪ {s}),
14:         M^f_k := M^f_k \ reachable(states(Γ) ∪ {s})}
15:     let A20 = A2 ∪ {
16:         ac(ce, Γ ⊢ k(s), {⊢ (s/x)φ1, ⊢ (s/y)φ2}),
17:         M^f_k := M^f_k ∪ states(Γ) ∪ {s}}
18:     let A2n = A2 ∪ {ac(ce, Γ ⊢ k(s), {Γ' ⊢ k(s1),...,Γ' ⊢ k(sn)})}
19:     ∀i ∈ {1,...,n}, let
20:         A1i = A1 ∪ {ac(pt, Γ ⊢ k(s), {⊢ (s/x)φ1, Γ' ⊢ k(si)})}
21:     if Γ = ∅ then
22:         visited_k := {s}
23:     else
24:         visited_k := visited_k ∪ {s}
25:     end if
26:     let c' = cpt^∅(⊢ (s/y)φ2, c_1^{A10}, cpt^∅(⊢ (s/x)φ1, cpt^∅(Γ' ⊢ k(s1),
27:         c_1^{A11}, cpt^∅(...cpt^∅(Γ' ⊢ k(sn), c_1^{A1n}, c_2^{A2n})...)), c_2^{A20}))
28:     c := c'
29: end if
```

FIGURE A.8. ProveEU($\Gamma \vdash \mathsf{EU}_{x,y}(\phi_1,\phi_2)(s)$)

```
1: /* For notation purpose, here we refer "k" to AR_{x,y}(φ1,φ2)(_) and
2: "k(s)" to AR_{x,y}(φ1,φ2)(s).*/
3: if s ∈ M^f_k then
4:     c := c_2^{A_2}
5:     M^f_k := M^f_k ∪ reachable(states(Γ))
6:     M^t_k := M^t_k \ reachable(states(Γ))
7: else if s ∈ M^t_k or k(s) ∈ Γ then
8:     c := c_1^{A_1}
9:     M^t_k := M^t_k ∪ states(Γ)
10: else
11:     let A10 = A1 ∪ {
12:         ac(pt, Γ ⊢ k(s), {⊢ (s/x)φ1, ⊢ (s/y)φ2}),
13:         M^t_k := M^t_k ∪ states(Γ) ∪ {s}}
14:     let A1n = A1 ∪ {ac(pt, Γ ⊢ k(s), {Γ' ⊢ k(s1),...,Γ' ⊢ k(sn)})}
15:     let A20 = A2 ∪ {
16:         ac(ce, Γ ⊢ k(s), {⊢ (s/y)φ2}),
17:         M^f_k := M^f_k ∪ reachable(states(Γ) ∪ {s}),
18:         M^t_k := M^t_k \ reachable(states(Γ) ∪ {s})
19:     ∀i ∈ {1,...,n}, let
20:         A2i = A2 ∪ {ac(ce, Γ ⊢ k(s), {⊢ (s/x)φ1, Γ' ⊢ k(si)})}
21:     if Γ = ∅ then
22:         visited_k := {s}
23:     else
24:         visited_k := visited_k ∪ {s}
25:     end if
26:     let c' = cpt^∅(⊢ (s/y)φ2, cpt^∅(⊢ (s/x)φ1, c_1^{A10}, cpt^∅(Γ' ⊢ k(s1),
27:         cpt^∅(...cpt^∅(Γ' ⊢ k(sn), c_1^{A1n}, c_2^{A2n})...), c_2^{A21})), c_2^{A20})
28:     c := c'
29: end if
```

FIGURE A.9. ProveAR($\Gamma \vdash \mathsf{AR}_{x,y}(\phi_1,\phi_2)(s)$)

APPENDIX B. DETAILED EXPERIMENTAL DATA FOR BENCHMARK #1, #2, #3, AND #4

We show the detailed experimental data for benchmark #1, #2, #3, and #4 as follows.

Appendix B.1. Benchmark #1 (Table B.1 and B.2)

Table B.1 shows that SCTLProV outperforms iProver Modulo and Verds, and is almost as good as NuSMV and NuXMV: NuSMV and NuXMV solve all the 2880 test cases, while SCTLProV solves 2870 (99.7%) test cases.

Let us now turn to the efficiency. SCTLProV is much faster than the four other tools (Table B.2). Among the test cases that can be solved by SCTLProV and iProver Modulo, SCTLProV is faster in 98.0% of these test cases, 99.2% when compared with Verds, 95.2% when compared with NuSMV and 95.9% when compared with NuXMV.

Programs	iProver Modulo	Verds	NuSMV	NuXMV	SCTLProV
CP ($b=12$)	467 (97.3%)	433 (90.2%)	480 (100%)	480 (100%)	480 (100%)
CP ($b=24$)	372 (77.5%)	428 (89.2%)	480 (100%)	480 (100%)	480 (100%)
CP ($b=36$)	383 (79.8%)	416 (86.7%)	480 (100%)	480 (100%)	470 (97.9%)
CSP ($b=12$)	177 (36.9%)	370 (77.1%)	480 (100%)	480 (100%)	480 (100%)
CSP ($b=16$)	164 (34.2%)	315 (65.6%)	480 (100%)	480 (100%)	480 (100%)
CSP ($b=20$)	253 (52.7%)	268 (55.8%)	480 (100%)	480 (100%)	480 (100%)
Sum	1816 (63.1%)	2230 (77.4%)	2880 (100%)	2880 (100%)	2870 (99.7%)

TABLE B.1. Solvable cases in five tools

Programs	iProver Modulo	Verds	NuSMV	NuXMV
CP ($b=12$)	480 (100%)	480 (100%)	430 (89.6%)	431 (89.8%)
CP ($b=24$)	480 (100%)	480 (100%)	456 (95.0%)	458 (95.4%)
CP ($b=36$)	454 (94.6%)	467 (97.3%)	441 (91.9%)	446 (92.9%)
CSP ($b=12$)	480 (100%)	480 (100%)	464 (96.7%)	465 (96.9%)
CSP ($b=16$)	474 (98.8%)	473 (98.5%)	472 (98.3%)	474 (98.8%)
CSP ($b=20$)	455 (94.8%)	478 (99.6%)	478 (99.6%)	479 (99.8%)
Sum	2823 (98.0%)	2858 (99.2%)	2741 (95.2%)	2763 (95.9%)

TABLE B.2. Cases where SCTLProV runs faster than iProver Modulo, Verds, NuSMV, and NuXMV, respectively

Appendix B.2. Benchmark #2 (Table B.3 and B.4)

Our benchmark #2 investigates the performances of iProver Modulo, Verds, NuSMV, NuXMV, and SCTLProV when

the size of the model increases.

To do so, we increase the number of variables in the random Boolean programs to {48, 60, 72, 252, 504, 1008} for Concurrent Processes, and {24, 28, 32, 252, 504, 1008} for Concurrent Sequential Processes, respectively. There are 5760 test cases in benchmark #2. Like in benchmark #1, all test cases in benchmark #2 are also randomly generated. The properties to be checked are the same as those in benchmark #1.

Counting the number of test cases that can be solved in 20 minutes, we see that SCTLProV scales up better (Table B.3, Table B.4) than the other four tools: SCTLProV solves more test cases than the other tools, and SCTLProV outperforms the other tools in most solvable test cases.

Programs	iProver Modulo	Verds	NuXMV	NuXMV	SCTLProV
CP ($b=48$)	375 (78.1%)	400 (83.3%)	171 (35.6%)	176 (36.7%)	446(92.9%)
CP ($b=60$)	360 (75.0%)	403 (84.0%)	22 (4.6%)	23 (4.8%)	440(91.7%)
CP ($b=72$)	347 (72.3%)	383 (79.8%)	0	0	437 (91.0%)
CP ($b=252$)	299 (62.3%)	216 (45.0%)	0	0	371 (77.3%)
CP ($b=504$)	292 (60.8%)	0	0	0	335 (69.8%)
CP ($b=1008$)	271 (56.5%)	0	0	0	278 (57.9%)
CSP ($b=24$)	190 (39.6%)	235 (49.0%)	421 (87.7%)	423 (88.1%)	430 (89.6%)
CSP ($b=28$)	172 (35.8%)	229 (47.7%)	106 (22.1%)	108 (22.5%)	426 (88.8%)
CSP ($b=32$)	158 (32.9%)	224 (46.7%)	8 (1.7%)	6 (1.3%)	418 (87.1%)
CSP ($b=252$)	114 (23.6%)	136 (28.3%)	0	0	312 (65.0%)
CSP ($b=504$)	108 (22.5%)	0	0	0	295 (61.5%)
CSP ($b=1008$)	62 (12.9%)	0	0	0	253 (52.7%)
Sum	2748 (47.7%)	2226 (38.6%)	728 (12.6%)	736 (12.8%)	4441 (77.1%)

TABLE B.3. Solvable cases in five tools

Programs	iProver Modulo	Verds	NuSMV	NuXMV
CP ($b=48$)	446 (92.9%)	444 (92.5%)	442 (92.1%)	442 (92.1%)
CP ($b=60$)	440 (91.7%)	440 (91.7%)	440 (91.7%)	440 (91.7%)
CP ($b=72$)	437 (91.0%)	437 (91.0%)	437 (91.0%)	437 (91.0%)
CP ($b=252$)	371 (77.3%)	371 (77.3%)	371 (77.3%)	371 (77.3%)
CP ($b=504$)	335 (69.8%)	335 (69.8%)	335 (69.8%)	335 (69.8%)
CP ($b=1008$)	278 (57.9%)	278 (57.9%)	278 (57.9%)	278 (57.9%)
CSP ($b=24$)	430 (89.6%)	429 (89.4%)	426 (88.8%)	426 (88.8%)
CSP ($b=28$)	426 (88.8%)	426 (88.8%)	425 (88.5%)	425 (88.5%)
CSP ($b=32$)	418 (87.1%)	418 (87.1%)	418 (87.1%)	418 (87.1%)
CSP ($b=252$)	312 (65.0%)	312 (65.0%)	312 (65.0%)	312 (65.0%)
CSP ($b=504$)	295 (61.5%)	295 (61.5%)	295 (61.5%)	295 (61.5%)
CSP ($b=1008$)	253 (52.7%)	253 (52.7%)	253 (52.7%)	253 (52.7%)
Sum	4441 (77.1%)	4438 (77.0%)	4432 (76.9%)	4432 (76.9%)

TABLE B.4. Cases where SCTLProV runs faster than iProver Modulo, Verds, NuSMV, and NuXMV, respectively

Appendix B.3. Benchmark #3 (Table B.5)

The detailed experimental data of verifying test cases in benchmark #3 is depicted in Table B.5.

Prop	NoP	Mutual Exclusion Algorithms							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	6	0.286	321.99	0.153	9.07	0.270	21.18	0.025	26.16
	12	1.278	322.08	19.506	76.98	21.848	89.25	0.022	29.23
	18	4.719	426.45	-	-	-	-	0.057	33.99
	24	11.989	601.55	-	-	-	-	0.130	43.31
	30	26.511	926.25	-	-	-	-	0.270	59.24
	36	52.473	1287.57	-	-	-	-	0.538	84.41
	42	100.071	1944.95	-	-	-	-	0.928	121.55
	48	-	-	-	-	-	-	1.440	173.07
	51	-	-	-	-	-	-	1.783	194.03
P_2	6	0.375	322.07	0.054	9.07	0.048	21.31	0.022	23.73
	12	2.011	322.02	22.774	76.96	21.733	89.24	0.045	30.49
	18	7.958	446.71	-	-	-	-	0.120	40.54
	24	23.448	692.30	-	-	-	-	0.272	47.89
	30	48.800	1026.48	-	-	-	-	0.620	81.72
	36	105.183	1619.01	-	-	-	-	1.145	122.62
	42	-	-	-	-	-	-	1.951	153.07
	48	-	-	-	-	-	-	3.001	246.78
	51	-	-	-	-	-	-	3.572	318.29
P_3	6	0.331	322.02	0.089	9.04	0.033	21.27	0.022	23.82
	12	2.059	322.07	22.749	76.91	21.897	89.22	0.043	31.77
	18	7.995	449.13	-	-	-	-	0.141	43.44
	24	23.578	696.74	-	-	-	-	0.356	58.59
	30	51.774	1138.27	-	-	-	-	0.774	107.76
	36	106.027	1628.84	-	-	-	-	1.423	136.78
	42	-	-	-	-	-	-	2.373	186.89
	48	-	-	-	-	-	-	4.301	318.44
	51	-	-	-	-	-	-	4.690	411.43
P_4	6	0.446	321.97	0.089	9.04	0.033	21.27	0.023	23.38
	12	8.289	552.62	22.749	76.91	21.897	89.22	-	-
	18	-	-	-	-	-	-	-	-
	24	-	-	-	-	-	-	-	-
	30	-	-	-	-	-	-	-	-
	36	-	-	-	-	-	-	-	-
	42	-	-	-	-	-	-	-	-
	48	-	-	-	-	-	-	-	-
	51	-	-	-	-	-	-	-	-
P_5	6	0.430	322.03	0.031	9.09	0.047	21.19	0.090	23.80
	12	3.398	363.78	22.747	77.01	22.029	89.17	0.074	34.01
	18	18.176	783.24	-	-	-	-	0.181	47.49
	24	87.432	2382.82	-	-	-	-	0.453	74.86
	30	-	-	-	-	-	-	0.990	122.81
	36	-	-	-	-	-	-	1.807	216.72
	42	-	-	-	-	-	-	3.112	319.21
	48	-	-	-	-	-	-	5.030	412.76
	51	-	-	-	-	-	-	5.900	535.78

Prop	NoP	Ring Algorithms							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	3	0.168	322.09	0.040	10.02	0.045	22.08	0.011	28.62
	4	0.216	322.12	0.299	22.46	0.255	34.96	0.732	48.25
	5	0.301	322.07	2.421	59.31	1.195	71.53	-	-
	6	0.449	322.13	22.127	80.49	17.967	92.82	-	-
	7	0.740	322.19	147.895	224.17	131.735	236.50	-	-
	8	1.115	322.09	1135.882	865.04	1083.48	877.36	-	-
	9	1.646	322.07	-	-	-	-	-	-
	10	2.232	321.96	-	-	-	-	-	-
P_2	3	0.201	322.59	0.058	10.74	0.068	22.73	0.060	23.95
	4	0.367	322.52	0.583	40.29	0.562	52.61	0.135	29.32
	5	0.786	336.80	5.164	62.29	5.295	74.62	0.513	31.08
	6	1.656	403.81	39.085	81.85	37.969	93.96	1.672	32.79
	7	3.352	533.55	246.123	229.07	241.375	241.15	4.217	36.36
	8	6.567	761.43	-	-	-	-	9.900	39.77
	9	9.536	929.86	-	-	-	-	21.939	42.88
	10	15.403	1334.47	-	-	-	-	43.865	51.73
P_3	3	0.189	322.61	0.045	10.03	0.071	22.32	0.012	26.19
	4	0.272	322.54	0.296	22.46	0.299	34.96	0.935	43.11
	5	0.376	322.63	2.357	59.31	2.526	71.63	160.67	1278.22
	6	0.574	322.75	22.147	80.49	21.304	92.93	-	-
	7	0.933	322.78	147.567	224.17	141.134	236.74	-	-
	8	1.512	322.79	-	-	-	-	-	-
	9	2.144	322.74	-	-	-	-	-	-
	10	2.896	336.75	-	-	-	-	-	-
P_4	3	0.158	322.09	0.066	10.00	0.171	22.32	0.044	27.84
	4	0.190	322.05	0.356	22.46	0.367	34.95	0.134	29.79
	5	0.263	322.04	2.726	59.31	2.781	71.63	0.433	30.98
	6	0.385	322.07	27.013	80.48	24.794	94.95	1.424	33.12
	7	0.528	322.07	181.007	224.16	166.725	236.61	3.993	35.20
	8	0.815	322.14	-	-	-	-	10.104	39.83
	9	1.138	322.19	-	-	-	-	23.465	45.98
	10	1.574	321.98	-	-	-	-	48.656	50.68

TABLE B.5. Time and memory usage in benchmark #3

Appendix B.4. Benchmark #4 (Table B.6 and Table B.7)

In benchmark #4, in order to detect deadlocks and livelocks of the test cases in SCTLProV, LTSs need to be transformed into Kripke models. The transformation and experimental results are explained as follows.

First, we translate LTSs into Kripke models as follows.

Given an LTS $\mathcal{L} = \langle s_0, S, Act, \rightarrow \rangle$ where $s_0 \in S$ is the initial state, S is a finite set of states, Act is a finite set of actions, and $\rightarrow \subseteq S \times Act \times S$ is the transition relation. $\mathcal{L}$ is translated into a Kripke model $\mathcal{M} = \langle s'_0, S', \longrightarrow, \mathcal{P} \rangle$ by performing the following transformation, where $S' = (S \cup \{s_d\}) \times (Act \cup \{.\})$, $s'_0 \in S'$, $\longrightarrow \subseteq S' \times S'$, and $\mathcal{P} = \{P, Q\}$.

- Let s'_0 be $(s_0, \cdot)$, where $\cdot \notin Act$ is a special action symbol;
- Add $(s_d, \cdot)$ to S' and $(s_d, \cdot) \longrightarrow (s_d, \cdot)$ to the transition relation of $\mathcal{M}$, where $s_d \notin S$ is distinguished from states in S, and $(s_d, \cdot)$ is distinguished from other states in S';
- Apply the following step repeatedly until no more states or transitions is added to $\mathcal{M}$:
  For all $a \in Act$, if $s_1 \xrightarrow{a} s_2$ is in $\mathcal{L}$, then add a transition $(s_1, b) \longrightarrow (s_2, a)$ to $\mathcal{M}$ for all $(s_1, b) \in S'$ where $b \in Act \cup \{\cdot\}$, and add state (s_2, a) into S'; for all state $(s, a) \in S'$, if s has no successor in $\mathcal{L}$, then add a

transition $(s,a) \longrightarrow (s_d,\cdot)$ to $\mathcal{M}$;
- Finally, let $P = \{(s_d,\cdot)\}$ and $Q = \{(s,\tau) \in S' \mid s \in S\}$.

Note that in the above transformation, we add a special state $(s_d,\cdot)$ to the transformed Kripke model, where $s_d \notin S$ and $\cdot \notin Act$. Also, for every state $(s,a) \in S'$ where s has no successor in the LTS to be transformed, add a special transition $(s,a) \longrightarrow (s_d,\cdot)$ to the transformed Kripke model; and to make the transition relation $\longrightarrow$ total, we add another special transition $(s_d,\cdot) \longrightarrow (s_d,\cdot)$ to the transformed Kripke model. The addition of the special state and transitions are necessary in this transformation. It is because otherwise, the transformed structure may not be called a Kripke model, since there may exist states that have no successors.

Then, we explain that the existence of deadlocks and livelocks in LTSs can be detected by performing the following verifications in the transformed Kripke models.

Deadlocks. A deadlock state in an LTS is a state which has no successor. When detecting deadlocks of an LTS $\mathcal{L}$ in SCTLProV, we first translate $\mathcal{L}$ into a Kripke model $\mathcal{M}$ using the above transformation. It can be observed that a deadlock state is reachable from s_0 in $\mathcal{L}$ if and only if $(s_d,\cdot)$ is reachable from $(s_0,\cdot)$ in $\mathcal{M}$.

Thus, we can detect deadlocks in an LTS $\mathcal{L}$ by performing the proof search of the formula

$$EF_x(P(x))((s_0,\cdot))$$

in the transformed Kripke model $\mathcal{M}$. This formula is provable if and only if $(s_0,\cdot) \longrightarrow^* (s,a) \longrightarrow (s_d,\cdot)$, where a is an action, and s has no successor in $\mathcal{L}$. Thus, the proof search of the formula can be used to detect deadlocks in $\mathcal{L}$.

We perform the deadlock detection both in SCTLProV and CADP on this benchmark. For 40 test cases in this benchmark, both SCTLProV and CADP can solve all test cases. Moreover, SCTLProV uses less time than CADP in 33 (82.5%) test cases, and uses less memory than CADP in 7 (17.5%) test cases. The detailed experimental data is depicted in Table B.6.

Livelocks. Livelocks in LTSs are represented as cycles of one or more τ transitions. The detection of livelocks is more complex than deadlocks. The reason is that not only states, but also actions are to be examined in the exploration of the LTS.

It can be observed that for an LTS $\mathcal{L}$ and the Kripke model $\mathcal{M}$ by performing the above transformation on $\mathcal{L}$, a livelock exists in $\mathcal{L}$ if and only if there exists a cycle of states where each state of the cycle satisfies Q. This is analyzed as follows.

- ($\Rightarrow$) If there exists a cycle of τ transitions in $\mathcal{L}$, then the cycle has the form $s_p \xrightarrow{\tau} s_{p+1} \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s_p$, where either $s_p = s_0$, or there exists a path $s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{p-1}} s_p$. Then according to the transformation, there exists a cycle of the form $(s_p,a) \longrightarrow (s_{p+1},\tau) \longrightarrow \cdots \longrightarrow (s_n,\tau) \longrightarrow (s_p,\tau) \longrightarrow (s_{p+1},\tau)$ in $\mathcal{M}$, where $(s_0,\cdot) \longrightarrow^* (s_p,a)$, and $a = a_{p-1}$.

Name	Deadlocks	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	26.48	0.40	10.95
cwi_1_2	No	0.13	27.71	0.39	10.80
vasy_1_4	No	0.14	27.83	0.39	10.71
cwi_3_14	Yes	0.29	25.79	0.40	10.82
vasy_5_9	Yes	0.14	25.70	0.40	10.82
vasy_8_24	No	0.17	28.90	0.43	10.79
vasy_8_38	Yes	0.14	25.76	0.39	10.74
vasy_10_56	No	0.22	29.78	0.43	10.86
vasy_18_73	No	0.24	31.68	0.47	11.81
vasy_25_25	Yes	0.97	33.52	2.18	23.26
vasy_40_60	No	0.21	29.42	0.46	15.08
vasy_52_318	No	0.59	41.09	0.65	16.69
vasy_65_2621	No	1.41	77.02	2.09	109.03
vasy_66_1302	No	0.89	34.92	1.25	14.13
vasy_69_520	Yes	0.23	27.47	0.51	11.84
vasy_83_325	Yes	0.21	27.96	0.48	11.32
vasy_116_368	No	0.67	35.27	0.77	14.40
cwi_142_925	Yes	0.28	28.33	0.57	12.72
vasy_157_297	Yes	0.18	27.14	0.45	11.48
vasy_164_1619	No	2.53	48.39	1.53	22.90
vasy_166_651	Yes	0.29	31.19	0.55	13.30
cwi_214_684	Yes	0.39	34.39	0.63	22.94
cwi_371_641	No	1.36	40.41	1.24	42.92
vasy_386_1171	No	2.14	74.11	1.66	45.12
cwi_566_3984	Yes	0.78	38.53	1.11	21.92
vasy_574_13561	No	21.56	272.11	9.72	188.21
vasy_720_390	Yes	0.23	28.49	0.48	12.89
vasy_1112_5290	No	10.2	89.81	6.54	97.47
cwi_2165_8723	No	16.51	166.74	14.55	185.58
cwi_2416_17605	Yes	3.19	87.61	3.38	71.80
vasy_2581_11442	Yes	2.40	74.11	2.68	58.43
vasy_4220_13944	Yes	2.85	89.50	3.20	73.82
vasy_4338_15666	Yes	3.41	96.21	3.83	80.59
vasy_6020_19353	No	29.41	413.40	74.24	649.41
vasy_6120_11031	Yes	2.35	82.57	2.60	67.01
cwi_7838_59101	No	92.92	1018.67	140.21	1019.55
vasy_8082_42933	No	7.85	309.74	7.82	240.69
vasy_11026_24660	Yes	4.82	149.80	5.15	134.17
vasy_12323_27667	Yes	5.40	164.73	5.67	149.09
cwi_33949_165318	No	296.86	2159.58	636.39	2972.61

TABLE B.6. Deadlock detection in SCTLProV and CADP

- ($\Leftarrow$) If there exists a cycle of the form $(s_p,\tau) \longrightarrow (s_{p+1},\tau) \longrightarrow \cdots \longrightarrow (s_n,\tau) \longrightarrow (s_p,\tau)$ which is reachable from $(s_0,\cdot)$ in $\mathcal{M}$, then $(s_0,\cdot) \longrightarrow^* (s_m,\tau)$ for some (s_m,τ) in the cycle, then we have $s_0 \xrightarrow{a_0} \cdots \xrightarrow{a'} s' \xrightarrow{\tau} s_m$ in $\mathcal{L}$, and there exists a cycle $s_p \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_p$ in $\mathcal{L}$ that is reachable from s_0.

Thus, we can detect livelocks in an LTS $\mathcal{L}$ by performing the proof search of the formula

$$EF_x(EG_y(Q(y))(x))((s_0,\cdot))$$

in the transformed Kripke model $\mathcal{M}$. This formula is provable if and only if there exists a livelock in $\mathcal{L}$.

We perform the livelock detection both in SCTLProV and CADP on this benchmark. For 40 test cases in this benchmark, both SCTLProV and CADP can solve all test cases. Moreover, SCTLProV uses less time than CADP in 22 (55%) test cases, and uses less memory than CADP in 6 (15%) test cases. The detailed experimental data is depicted in Table B.7.

APPENDIX C. PROOF OF SOUNDNESS AND COMPLETENESS OF SCTL

The soundness and completeness of SCTL are guaranteed by the finiteness of the Kripke model. In the soundness

Name	Livelocks	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	27.45	0.48	14.57
cwi_1_2	No	0.14	27.74	0.50	14.61
vasy_1_4	No	0.14	27.70	0.48	14.66
cwi_3_14	No	0.16	28.18	0.50	14.57
vasy_5_9	No	0.16	28.08	0.51	14.68
vasy_8_24	No	0.18	29.09	0.53	14.75
vasy_8_38	No	0.20	28.86	0.52	14.73
vasy_10_56	No	0.23	30.33	0.55	15.34
vasy_18_73	No	0.28	33.07	0.56	16.25
vasy_25_25	No	1.12	33.45	2.25	27.84
vasy_40_60	No	0.26	30.23	0.57	19.65
vasy_52_318	Yes	0.21	27.66	0.55	15.45
vasy_65_2621	No	5.31	280.57	1.98	113.40
vasy_66_1302	No	2.40	38.33	1.30	18.50
vasy_69_520	No	1.04	33.41	0.85	16.39
vasy_83_325	No	0.83	39.27	0.76	19.40
vasy_116_368	No	1.19	42.14	0.86	15.74
cwi_142_925	No	2.67	46.30	1.22	17.89
vasy_157_297	No	0.80	33.99	0.78	17.91
vasy_164_1619	No	3.69	53.30	1.51	23.44
vasy_166_651	No	1.60	49.98	1.02	26.02
cwi_214_684	Yes	0.26	29.93	0.63	16.81
cwi_371_641	Yes	0.26	30.63	0.62	17.41
vasy_386_1171	No	2.91	80.16	1.55	41.75
cwi_566_3984	No	13.32	106.25	3.95	54.08
vasy_574_13561	No	27.26	272.05	8.17	188.69
vasy_720_390	No	0.86	31.45	0.76	17.45
vasy_1112_5290	No	10.49	89.86	5.24	97.93
cwi_2165_8723	Yes	1.87	61.94	2.15	48.80
cwi_2416_17605	Yes	3.10	87.61	3.44	76.30
vasy_2581_11442	No	32.70	326.38	14.78	214.93
vasy_4220_13944	No	43.93	423.03	24.71	330.85
vasy_4338_15666	No	47.55	479.15	28.06	344.64
vasy_6020_19353	Yes	3.23	100.24	3.57	106.43
vasy_6120_11031	No	30.59	425.64	38.37	437.71
cwi_7838_59101	Yes	11.34	250.68	11.58	236.09
vasy_8082_42933	No	119.77	1123.85	106.49	908.29
vasy_11026_24660	No	60.86	698.85	108.97	804.34
vasy_12323_27667	No	68.50	793.83	134.44	898.61
cwi_33949_165318	Yes	33.89	732.05	34.60	738.78

TABLE B.7. Livelock detection in SCTLProV and CADP

proof, Proposition 3.1 and 3.2 permit to transform finite structures into infinite ones, and in the completeness proof, Proposition 3.3 and 3.4 permit to transform infinite structures into finite ones. The detailed soundness and completeness proofs are given as follows.

THEOREM C.1 (Soundness). *If ϕ is closed, and the sequent $\vdash \phi$ has a proof π in SCTL($\mathcal{M}$), then $\mathcal{M} \models \phi$ for the given Kripke model $\mathcal{M}$.*

Proof. By induction on the structure of the proof π.

- If the last rule of π is atom-R, then the proved sequent has the form $\vdash P(s_1,...,s_n)$, hence $\mathcal{M} \models P(s_1,...,s_n)$.
- If the last rule of π is $\neg$-R, then the proved sequent has the form $\vdash \neg P(s_1,...,s_n)$, hence $\mathcal{M} \models \neg P(s_1,...,s_n)$.
- If the last rule of π is $\top$-R, the proved sequent has the form $\vdash \top$ and hence $\mathcal{M} \models \top$.
- If the last rule of π is $\wedge$-R, then the proved sequent has the form $\vdash \phi_1 \wedge \phi_2$. By induction hypothesis $\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$, hence $\mathcal{M} \models \phi_1 \wedge \phi_2$.
- If the last rule of π is $\vee$-R_1 or $\vee$-R_2, then the proved sequent has the form $\vdash \phi_1 \vee \phi_2$. By induction hypothesis $\mathcal{M} \models \phi_1$ or $\mathcal{M} \models \phi_2$, hence $\mathcal{M} \models \phi_1 \vee \phi_2$.
- If the last rule of π is **AX**-R, then the proved sequent has the form $\vdash AX_x(\phi_1)(s)$. By induction hypothesis, for each s' in Next(s), $\mathcal{M} \models (s'/x)\phi_1$, hence $\mathcal{M} \models AX_x(\phi_1)(s)$.
- If the last rule of π is **EX**-R, then the proved sequent has the form $\vdash EX_x(\phi_1)(s)$. By induction hypothesis, for some s' in Next(s), $\mathcal{M} \models (s'/x)\phi_1$, hence $\mathcal{M} \models EX_x(\phi_1)(s)$.
- If the last rule of π is **AF**-R_1 or **AF**-R_2, then the proved sequent has the form $\vdash AF_x(\phi_1)(s)$. We associate a finite path-tree $|\pi|$ to the proof π by induction in the following way.
  - If the proof π ends with the **AF**-R_1 rule with a sub-proof ρ of the sequent $\vdash (s/x)\phi_1$, then the path-tree contains a single node s.
  - If the proof π ends with the **AF**-R_2 rule, with sub-proofs $\pi_1,...,\pi_n$ of the sequent $\vdash AF_x(\phi_1)(s_1),...,\vdash AF_x(\phi_1)(s_n)$, respectively, then $|\pi|$ is the path-tree $s(|\pi_1|,...,|\pi_n|)$.

  The path-tree $|\pi|$ has root s, and for each leaf s' of $|\pi|$, the sequent $\vdash (s'/x)\phi_1$ has a proof smaller than π. By induction hypothesis, for each leaf s' of $|\pi|$, $\mathcal{M} \models (s'/x)\phi_1$. Hence $\mathcal{M} \models AF_x(\phi_1)(s)$.
- If the last rule of π is **EG**-R, then the proved sequent has the form $\vdash EG_x(\phi_1)(s)$. We associate a finite sequence $|\pi|$ to the proof π by induction in the following way.
  - If the proof π ends with the **EG**-merge rule, then the sequence contains a single element s.
  - If the proof π ends with the **EG**-R rule, with sub-proofs ρ and π_1 of the sequents $\vdash (s/x)\phi_1$ and $\Gamma, EG_x(\phi_1)(s) \vdash EG_x(\phi_1)(s')$, respectively, then $|\pi|$ is the sequence $s|\pi_1|$.

  The sequent $|\pi| = s_0, s_1, ..., s_n$ is such that $s_0 = s$; for all i between 0 and $n-1$, $s_i \longrightarrow s_{i+1}$; for all i between 0 and n, the sequent $\vdash (s_i/x)\phi_1$ has a proof smaller than π; and s_n is equal to s_p for some p between 0 and $n-1$. By induction hypothesis, for all i, we have $\mathcal{M} \models (s_i/x)\phi_1$. Using Proposition 3.1, there exists an infinite sequence $s'_0, s'_1, ...$ such that for all i, we have $s'_i \longrightarrow s'_{i+1}$, and $\mathcal{M} \models (s'_i/x)\phi_1$. Hence, $\mathcal{M} \models EG_x(\phi_1)(s)$.
- If the last rule of π is **AR**-R_1 or **AR**-R_2, then the proved sequent has the form $\vdash AR_x(\phi_1,\phi_2)(s)$. We associate a finite path-tree $|\pi|$ to the proof π by induction in the following way.
  - If the proof π ends with the **AR**-R_1 rule with sub-proofs ρ_1 and ρ_2 of the sequents $\vdash (s/x)\phi_1$ and $\vdash (s/x)\phi_2$, respectively, or with the **AR**-merge rule, then the path-tree contains a single node s.
  - If the proof π ends with the **AR**-R_2 rule, with sub-proofs $\rho, \pi_1, ..., \pi_n$ of the sequents $\vdash (s/y)\phi_2$, $\Gamma, AR_{x,y}(\phi_1,\phi_2)(s) \vdash AR_{x,y}(\phi_1,\phi_2)(s_1), ...,$ $\Gamma, AR_{x,y}(\phi_1,\phi_2)(s) \vdash AR_{x,y}(\phi_1,\phi_2)(s_n)$, respectively, then $|\pi|$ is the path-tree $s(|\pi_1|,...,|\pi_n|)$.

  The path-tree $|\pi|$ has root s, and for each node s' of $|\pi|$, the sequent $\vdash (s'/y)\phi_2$ has a proof smaller than π; and for each leaf s', either the sequent $\vdash (s'/x)\phi_1$ has a proof smaller than π, or s' is also a label of a node on the branch from the root of $|\pi|$ to this leaf. By

induction hypothesis, for each node s' of this path-tree $\mathcal{M} \models (s'/x)\phi_2$ and for each leaf s', either $\mathcal{M} \models (s'/x)\phi_1$ or s' is also a label of a node on the branch from the root of $|\pi|$ to this leaf. Using Proposition 3.2, there exists a possibly infinite path-tree T' such that for each node s' of T', $\mathcal{M} \models (s'/y)\phi_2$, and for each leaf s' of T', $\mathcal{M} \models (s'/x)\phi_1$. Thus, $\mathcal{M} \models AR_{x,y}(\phi_1,\phi_2)(s)$.

- If the last rule of π is **EU**-R_1 or **EU**-R_2, then the proved sequent has the form $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$. We associate a finite sequence $|\pi|$ to the proof π by induction in the following way.
  - If the proof π ends with the **EU**-R_1 rule with a sub-proof ρ of the sequent $\vdash (s/y)\phi_2$, then the sequence contains a single element s.
  - If the proof π ends with the **EU**-R_2 rule, with sub-proofs ρ and π_1 of the sequents $\vdash (s/x)\phi_1$ and $\vdash EU_{x,y}(\phi_1,\phi_2)(s')$, respectively, then $|\pi|$ is the sequence $s|\pi_1|$.

  The sequence $|\pi| = s_0,...,s_n$ is such that $s_0 = s$; for each i between 0 and $n-1$, $s_i \longrightarrow s_{i+1}$; for each i between 0 and $n-1$, the sequent $\vdash (s_i/x)\phi_1$ has a proof smaller than π; and the sequent $\vdash (s_n/y)\phi_2$ has a proof smaller than π. By induction hypothesis, for each i between 0 and $n-1$, $\mathcal{M} \models (s_i/x)\phi_1$ and $\mathcal{M} \models (s_n/y)\phi_2$. Hence, $\mathcal{M} \models EU_{x,y}(\phi_1,\phi_2)(s)$.
- The last rule cannot be a merge rule.

□

THEOREM C.2 (Completeness). *If ϕ is closed, and $\mathcal{M} \models \phi$ for the given Kripke model $\mathcal{M}$, then the sequent $\vdash \phi$ is provable in SCTL($\mathcal{M}$).*

Proof. By induction over the size of ϕ.

- If $\phi = P(s_1,...,s_n)$, then as $\mathcal{M} \models P(s_1,...,s_n)$, the sequent $\vdash P(s_1,...,s_n)$ is provable with the rule atom-R.
- If $\phi = \neg P(s_1,...,s_n)$, then as $\mathcal{M} \models \neg P(s_1,...,s_n)$, the sequent $\vdash \neg P(s_1,...,s_n)$ is provable with the rule $\neg$-R.
- If $\phi = \top$, then $\vdash \top$ is provable with the rule $\top$-R.
- If $\phi = \bot$, then it is not the case that $\mathcal{M} \models \bot$.
- If $\phi = \phi_1 \wedge \phi_2$, then as $\mathcal{M} \models \phi_1 \wedge \phi_2$, $\mathcal{M} \models \phi_1$ and $\mathcal{M} \models \phi_2$. By induction hypothesis, the sequents $\vdash \phi_1$ and $\vdash \phi_2$ are provable. Thus the sequent $\vdash \phi_1 \wedge \phi_2$ is provable with the $\wedge$-R rule.
- If $\phi = \phi_1 \vee \phi_2$, as $\mathcal{M} \models \phi_1 \vee \phi_2$, $\mathcal{M} \models \phi_1$ or $\mathcal{M} \models \phi_2$. By induction hypothesis, the sequent $\vdash \phi_1$ or $\vdash \phi_2$ is provable and the sequent $\vdash \phi_1 \vee \phi_2$ is provable with the $\vee$-R_1 or $\vee$-R_2 rule, respectively.
- If $\phi = AX_x(\phi_1)(s)$, as $\mathcal{M} \models AX_x(\phi_1)(s)$, for each state s' in Next(s), we have $\mathcal{M} \models (s'/x)\phi_1$. By induction hypothesis, for each s' in Next(s), the sequent $\vdash (s'/x)\phi_1$ is provable. Using these proofs and the **AX**-R rule, we build a proof of the sequent $\vdash AX_x(\phi_1)(s)$.
- If $\phi = EX_x(\phi_1)(s)$, as $\mathcal{M} \models EX_x(\phi_1)(s)$, there exists a state s' in Next(s) such that $\mathcal{M} \models (s'/x)\phi_1$. By induction hypothesis, the sequent $\vdash (s'/x)\phi_1$ is provable. With this proof and the **EX**-R rule, we build a proof of the sequent $\vdash EX_x(\phi_1)(s)$.
- If $\phi = AF_x(\phi_1)(s)$, as $\mathcal{M} \models AF_x(\phi_1)(s)$, there exists a finite path-tree T such that T has root s, for each internal node s', the children of this node are labeled by the elements of Next(s'), and for each leaf s', $\mathcal{M} \models (s'/x)\phi_1$. By induction hypothesis, for every leaf s', the sequent $\vdash (s'/x)\phi_1$ is provable. Then, to each sub-tree T' of T, we associate a proof $|T'|$ of the sequent $\vdash AF_x(\phi_1)(s')$ where s' is the root of T', by induction, as follows.
  - If T' contains a single node s', then the proof $|T|$ is built with the **AF**-R_1 rule from the proof of $\vdash (a/x)\phi_1$ given by the induction hypothesis.
  - If $T' = s'(T_1,...,T_n)$, then the proof $|T|$ is built with the **AF**-R_2 rule from the proofs $|T_1|,...,|T_n|$ of the sequents $\vdash AF_x(\phi_1)(s_1),...,\vdash AF_x(\phi_1)(s_n)$, respectively, where $s_1,...,s_n$ are the elements of Next(s').

  This way, the proof $|T|$ is a proof of the sequent $\vdash AF_x(\phi_1)(s)$.
- If $\phi = EG_x(\phi_1)(s)$, as $\mathcal{M} \models EG_x(\phi_1)(s)$, there exists a path $s_0,s_1,...$ such that $s_0 = s$ and for all i, $\mathcal{M} \models (s_i/x)\phi_1$. By induction hypothesis, all the sequents $\vdash (s_i/x)\phi_1$ are provable. Using Proposition 3.3, there exists a finite sequence $T = s_0,...,s_n$ such that for all i, $s_i \longrightarrow s_{i+1}$, the sequent $\vdash (s_i/x)\phi_1$ is provable and s_n is some s_p for $p < n$. We associate a proof $|s_i,...,s_n|$ of the sequent $EG_x(\phi_1)(s_0),\ldots,EG_x(\phi_1)(s_{i-1}) \vdash EG_x(\phi_1)(s_i)$ to each suffix of T by induction as follows.
  - The proof $|s_n|$ is built with the **EG**-merge rule.
  - If $i \leq n-1$, then the proof $|s_i,...,s_n|$ is built with the **EG**-R rule from the proof of $\vdash (s_i/x)\phi_1$ given by the induction hypothesis and the proof $|s_{i+1},...,s_n|$ of the sequent $EG_x(\phi_1)(s_0),...,EG_x(\phi_1)(s_i) \vdash EG_x(\phi_1)(s_{i+1})$.

  This way, the proof $|s_0,...,s_n|$ is a proof of the sequent $\vdash EG_x(\phi_1)(s)$.
- If $\phi = AR_{x,y}(\phi_1,\phi_2)(s)$, as $\mathcal{M} \models AR_{x,y}(\phi_1,\phi_2)(s)$, there exists a possibly infinite path-tree such that the root of this tree is s, for each node s', $\mathcal{M} \models (s'/y)\phi_2$ and for each leaf s', $\mathcal{M} \models (s'/x)\phi_1$. By induction hypothesis, for each node s' of the path-tree, the sequent $\vdash (s'/y)\phi_2$ is provable and for each leaf s' of the path-tree, the sequent $\vdash (s'/x)\phi_1$ is provable. Using Proposition 3.4, there exists a finite path-tree T such that for each node s' of T, the sequent $\vdash (s'/y)\phi_2$ is provable, and for each leaf s', either the sequent $\vdash (s'/x)\phi_1$ is provable or s' is also a label of a node on the branch from the root of T to this leaf. Then, to each sub-tree T' of T, we associate a proof $|T'|$ of the sequent $AR_{x,y}(\phi_1,\phi_2)(s_1),...,AR_{x,y}(\phi_1,\phi_2)(s_m) \vdash AR_{x,y}(\phi_1,\phi_2)(s')$ where s' is the root of T' and $s_1,...,s_m$ is the sequence of nodes in T from the root of T to the root of T'.

– If T' contains a single node s', and the sequent $\vdash (s'/x)\phi_1$ is provable then the proof $|T'|$ is built with the **AR**-R_1 rule from the proofs of $\vdash (s'/x)\phi_1$ and $\vdash (s'/y)\phi_2$ given by the induction hypothesis.
– If T' contains a single node s', and s' is among $s_1,...,s_m$, then the proof $|T'|$ is built with the **AR**-merge rule.
– If $T' = s'(T_1,...,T_n)$, then the proof $|T'|$ is built with the **AR**-R_2 rule from the proofs $\vdash (s'/y)\phi_2$ given by the induction hypothesis and the proofs $|T_1|,...,|T_n|$ of the sequents

$$AR_{x,y}(\phi_1,\phi_2)(s_1),...,AR_{x,y}(\phi_1,\phi_2)(s_m), AR_{x,y}(\phi_1,\phi_2)(s') \vdash AR_{x,y}(\phi_1,\phi_2)(s'_1)$$
$$...$$
$$AR_{x,y}(\phi_1,\phi_2)(s_1),...,AR_{x,y}(\phi_1,\phi_2)(s_m), AR_{x,y}(\phi_1,\phi_2)(s') \vdash AR_{x,y}(\phi_1,\phi_2)(s'_n)$$

respectively, where $s'_1,...,s'_n$ are the elements of $\mathsf{Next}(s')$.

This way, the proof $|T|$ is a proof of the sequent $\vdash AR_{x,y}(\phi_1,\phi_2)(s)$.

• If $\phi = EU_{x,y}(\phi_1,\phi_2)(s)$, as $\mathcal{M} \models EU_{x,y}(\phi_1,\phi_2)(s)$, there exists a finite sequence $T = s_0,...,s_n$ such that $\mathcal{M} \models (s_n/y)\phi_2$ and for all i between 0 and $n-1$, $\mathcal{M} \models (s_i/x)\phi_1$. By induction hypothesis, the sequent $\vdash (s_n/y)\phi_2$ is provable and for all i between 0 and $n-1$, the sequent $\vdash (s_i/x)\phi_1$ is provable. We associate a proof $|s_i,...,s_n|$ of the sequent $\vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$ to each suffix of T by induction as follows.

– The proof $|s_n|$ is built with the **EG**-R_1 rule from the proof of $\vdash (s_n/y)\phi_2$ given by the induction hypothesis.
– If $i \leq n-1$, then the proof $|s_i,...,s_n|$ is built with the **EG**-R_2 rule from the proof of $\vdash (s_i/x)\phi_1$ given by the induction hypothesis and the proof $|s_{i+1},...,s_n|$ of the sequent $\vdash EU_{x,y}(\phi_1,\phi_2)(s_{i+1})$.

This way, the proof $|s_0,...,s_n|$ is a proof of the sequent $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$.

□

APPENDIX D. PROOF OF THE CORRECTNESS OF THE PROOF SEARCH METHOD

PROPOSITION D.1. *Given a closed formula* ϕ, $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \mathfrak{t}$ *iff* $\vdash \phi$ *is provable.*

Proof. The proof is split into two cases: given a closed formula ϕ,

$\Rightarrow$: $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \mathfrak{t}$ implies that $\vdash \phi$ is provable;
$\Leftarrow$: $\vdash \phi$ is provable implies that $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \mathfrak{t}$.

For the "$\Rightarrow$" case, we prove a generalized proposition: for any two different CPTs c_1, c_2 and a sequent $\Gamma \vdash \phi$ where ϕ is a closed formula, $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 implies that $\Gamma \vdash \phi$ is provable (i.e., a proof tree T of $\Gamma \vdash \phi$ can be constructed). The proof of this proposition is by induction on the weight (Definition 4.3) of $\Gamma \vdash \phi$.

Before we prove the "$\Rightarrow$" case, it should be noted that for a CPT $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ where $c_1 \neq c_2$, and ϕ is a closed formula, if $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2, then $\mathsf{cpt}(\Gamma \vdash \phi, c'_1, c'_2)$ rewrites to c'_1 before it rewrites to c'_2 for any two different CPTs c'_1 and c'_2. This is because in the rewriting steps from $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ to c_1, neither c_1 nor c_2 is examined until c_1 is to be rewritten. Thus, we can safely replace c_1 by c'_1, and c_2 by c'_2 in the rewriting steps from $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ to c_1, and obtain the rewriting steps from $\mathsf{cpt}(\Gamma \vdash \phi, c'_1, c'_2)$ to c'_1. Analogously, if $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ rewrites to c_2 before it rewrites to c_1, then $\mathsf{cpt}(\Gamma \vdash \phi, c'_1, c'_2)$ rewrites to c'_2 before it rewrites to c'_1 for any two different CPTs c'_1 and c'_2. These assertions are used in the proof of the "$\Rightarrow$" case.

Now the proof of the "$\Rightarrow$" case is presented as follows.

• If $\Gamma \vdash \phi = \vdash \top$, trivial.
• If $\Gamma \vdash \phi = \vdash P(s_1,...,s_n)$ where $P(s_1,...,s_n)$ is an atomic formula, then the only way that $\mathsf{cpt}(\vdash P(s_1,...,s_n), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 is that $\mathsf{cpt}(\vdash P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1$, in which case $\langle s_1,...,s_n\rangle \in P$. Then, it is sufficient to take the following proof tree as T.

$$\frac{}{\vdash P(s_1,...,s_n)}\ \text{atom-R}$$

• If $\Gamma \vdash \phi = \vdash \neg P(s_1,...,s_n)$ where $P(s_1,...,s_n)$ is an atomic formula, then the only way that $\mathsf{cpt}(\vdash \neg P(s_1,...,s_n), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 is that $\mathsf{cpt}(\vdash \neg P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1$, in which case $\langle s_1,...,s_n\rangle \notin P$. Then, it is sufficient to take the following proof tree as T.

$$\frac{}{\vdash \neg P(s_1,...,s_n)}\ \neg\text{-R}$$

• If $\Gamma \vdash \phi = \vdash \phi_1 \wedge \phi_2$, then the only way that $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 is that $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2) \rightsquigarrow^* \mathsf{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$, then according to the induction hypothesis, both $\vdash \phi_1$ and $\vdash \phi_2$ are provable. This is because otherwise, if either $\vdash \phi_1$ or $\vdash \phi_2$ is not provable, then according to the induction hypothesis, $\mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2)$ will rewrite to c_2 before it rewrites to c_1. Suppose T_1 and T_2 are the proof trees constructed for $\vdash \phi_1$ and $\vdash \phi_2$, respectively, then it is sufficient to take the following proof tree as T.

$$\frac{T_1 \quad T_2}{\vdash \phi_1 \wedge \phi_2}\ \wedge\text{-R}$$

• If $\Gamma \vdash \phi = \vdash \phi_1 \vee \phi_2$, then there are two ways that $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2:

1. $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$, and $\mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$ rewrites to c_1 before it rewrites to $\mathsf{cpt}(\vdash \phi_2, c_1, c_2)$. In this case, according to the induction hypothesis, $\vdash \phi_1$ is provable. Suppose T_1 is the proof tree constructed for $\vdash \phi_1$,

then it is sufficient to take the following proof tree as T.

$$\frac{T_1}{\vdash \phi_1 \vee \phi_2}\ \vee\text{-R}_1$$

2. $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$, $\mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$ rewrites to $\mathsf{cpt}(\vdash \phi_2, c_1, c_2)$ before it rewrites to c_1, and $\mathsf{cpt}(\vdash \phi_2, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2. In this case, according to the induction hypothesis, $\vdash \phi_2$ is provable. Suppose T_2 is the proof tree constructed for $\vdash \phi_2$, then it is sufficient to take the following proof tree as T.

$$\frac{T_2}{\vdash \phi_1 \vee \phi_2}\ \vee\text{-R}_2$$

- If $\Gamma \vdash \phi = \vdash EX_x(\phi_1)(s)$, and $\{s_1, ..., s_n\} = \mathsf{Next}(s)$, then $\mathsf{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 implies that there exists $s_i \in \mathsf{Next}(s)$ such that $\vdash (s_i/x)\phi_1$ is provable. This is because otherwise, if for all $s_j \in \mathsf{Next}(s)$, $\vdash (s_j/x)\phi_1$ is not provable, then according to the induction hypothesis, $\mathsf{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s_1/x)\phi_1, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$ $\rightsquigarrow^*$ $\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2) \rightsquigarrow^* c_2$, and c_1 has not been rewritten in the above rewriting steps. Suppose T' is the proof tree constructed for $\vdash (s_i/x)\phi_1$, then it is sufficient to take the following proof tree as T.

$$\frac{T'}{\vdash EX_x(\phi_1)(s)}\ \mathbf{EX}\text{-R}$$

- If $\Gamma \vdash \phi = \vdash AX_x(\phi_1)(s)$, as the dual case of EX, the analysis for the AX case is analogous to that of the EX case.
- If $\Gamma \vdash \phi = \Gamma \vdash EG_x(\phi_1)(s)$, and $\{s_1, ..., s_n\} = \mathsf{Next}(s)$, then $\mathsf{cpt}(\Gamma \vdash EG_x(\phi_1)(s), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 implies that one of the following two cases holds.

1. $EG_x(\phi_1)(s) \in \Gamma$;
2. $EG_x(\phi_1)(s) \notin \Gamma$, and $\mathsf{cpt}(\Gamma \vdash EG_x(\phi_1)(s), c_1, c_2) \rightsquigarrow^*$ $\mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}($ $...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)), c_2) \rightsquigarrow^*$ $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}(...$ $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)) \rightsquigarrow^*$ $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_i), c_1, \mathsf{cpt}(...$ $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)) \rightsquigarrow^* c_1$, where $\Gamma' = \Gamma \cup \{EG_x(\phi_1)(s)\}$ and $1 \le i \le n$.

If case 1 holds, then it is sufficient to take the following proof tree as T.

$$\frac{}{\Gamma \vdash EG_x(\phi_1)(s)}\ \mathbf{EG}\text{-merge}$$

If case 2 holds, then by induction hypothesis, both $\vdash (s/x)\phi_1$ and $\Gamma' \vdash EG_x(\phi_1)(s_i)$ are provable. Suppose T_1 and T_2 are the constructed proof trees of $\vdash (s/x)\phi_1$ and $\Gamma' \vdash EG_x(\phi_1)(s_i)$, respectively, then it is sufficient to take the following proof tree as T.

$$\frac{T_1 \quad T_2}{\Gamma \vdash EG_x(\phi_1)(s)}\ \mathbf{EG}\text{-R}$$

- If $\Gamma \vdash \phi = \Gamma \vdash AF_x(\phi_1)(s)$, as the dual case of EG, the analysis for the AF case is analogous to that of the EG case.
- If $\Gamma \vdash \phi = \Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)$, $\{s_1, ..., s_n\} = \mathsf{Next}(s)$, and $\Gamma' = \Gamma \cup \{EU_{x,y}(\phi_1, \phi_2)(s)\}$, then $\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 implies that one of the following two cases holds.

1. $\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow^*$ $\mathsf{cpt}(\vdash (s/y)\phi_2, c_1, \mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n), c_1, c_2)...)), c_2)) \rightsquigarrow^* c_1$;
2. $\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow^*$ $\mathsf{cpt}(\vdash (s/y)\phi_2, c_1, \mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n), c_1, c_2)...)), c_2)) \rightsquigarrow^*$ $\mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n), c_1, c_2)...)) \rightsquigarrow^*$ $\mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_i), c_1, \mathsf{cpt}(...\mathsf{cpt}($ $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n), c_1, c_2)...)) \rightsquigarrow^* c_1$ where $1 \le i \le n$.

If case 1 holds, then by induction hypothesis, $\vdash (s/y)\phi_2$ is provable. Suppose T' is the constructed proof tree of $\vdash (s/y)\phi_2$, then it is sufficient to take the following proof tree as T.[26]

$$\frac{T'}{\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)}\ \mathbf{EU}\text{-R}_1$$

If case 2 holds, then by induction hypothesis, both $\vdash (s/x)\phi_1$ and $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_i)$ are provable. Let T_1 and T_2 be the constructed proof trees of $\vdash (s/x)\phi_1$ and $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_i)$, respectively, and it is sufficient to take the following proof tree as T.

$$\frac{T_1 \quad T_2}{\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)}\ \mathbf{EU}\text{-R}_2$$

- If $\Gamma \vdash \phi = \Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s)$, as the dual case of EU, the analysis for the AR case is analogous to that of the EU case.

For the "$\Leftarrow$" case, we also prove a generalized proposition: for two different CPTs c_1 and c_2, and a closed formula ϕ,

- if ϕ is an inductive formula, then $\vdash \phi$ is provable implies that $\mathsf{cpt}(\vdash \phi, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2;

[26] In the rules of the proof system, the context of an EU sequent is always empty. However, to make the proof easier to follow, the contexts of EU sequents are kept. The contexts of EU sequents does not break the structure of the proof tree since this kind of contexts are not examined in the proof tree. The purpose of introducing contexts for the EU case is explained in Section 4.1.1.

- if ϕ is not an inductive formula, then $\Gamma \vdash \phi$ is provable implies that $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ rewrites to c_1 before it rewrites to c_2;

The proof of this proposition is by induction on the proof tree of $\Gamma \vdash \phi$ ($\Gamma = \emptyset$ when ϕ is an inductive formula).

- If the last rule in the proof tree is atom-R, then suppose $\phi = P(s_1,...,s_n)$ where $P(s_1,...,s_n)$ is an atomic formula, and $\vdash P(s_1,...,s_n)$ is provable, then $\langle s_1,...,s_n\rangle \in P$, then $\mathsf{cpt}(\vdash P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1$.
- If the last rule in the proof tree is $\neg$-R, then suppose $\phi = \neg P(s_1,...,s_n)$ where $P(s_1,...,s_n)$ is an atomic formula, and $\vdash \neg P(s_1,...,s_n)$ is provable, then $\langle s_1,...,s_n\rangle \notin P$, then $\mathsf{cpt}(\vdash \neg P(s_1,...,s_n), c_1, c_2) \rightsquigarrow c_1$.
- If the last rule in the proof tree is $\wedge$-R, then suppose $\phi = \phi_1 \wedge \phi_2$ and $\vdash \phi$ is provable, then according to the rules of the proof system, both $\vdash \phi_1$ and $\vdash \phi_2$ are provable, then according to the induction hypothesis, $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2) \rightsquigarrow^* \mathsf{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.
- If the last rule in the proof tree is $\vee$-R, then $\phi = \phi_1 \vee \phi_2$ and $\vdash \phi$ is provable, then according to the rules of the proof system, either $\vdash \phi_1$ or $\vdash \phi_2$ is provable:
  1. if $\vdash \phi_1$ is provable, then according to the induction hypothesis, $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2)) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps;
  2. if $\vdash \phi_1$ is not provable but $\vdash \phi_2$ is, then according to the induction hypothesis, either $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2)) \rightsquigarrow^* c_1$, or $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2)) \rightsquigarrow^* \mathsf{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.
- If the last rule in the proof tree is **EX**-R, then suppose $\phi = EX_x(\phi_1)(s)$, $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and $\vdash \phi$ is provable, then according to the rules of the proof system, there exists $s_i \in \mathsf{Next}(s)$ such that $\vdash (s_i/x)\phi_1$ is provable, and for all j such that $1 \le j < i$, $\vdash (s_j/x)\phi_1$ is not provable. Then, according to the induction hypothesis, either
  $\mathsf{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$
  $\mathsf{cpt}(\vdash (s_1/x)\phi_1, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$
  $\rightsquigarrow^*$
  $\mathsf{cpt}(\vdash (s_j/x)\phi_1, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$
  $\rightsquigarrow^* c_1$ where $1 \le j < i$, or
  $\mathsf{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$
  $\mathsf{cpt}(\vdash (s_1/x)\phi_1, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$
  $\rightsquigarrow^*$
  $\mathsf{cpt}(\vdash (s_i/x)\phi_1, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$
  $\rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.
- If the last rule in the proof tree is **AX**-R, then as the dual case of EX, the analysis for the AX case is analogous to that of the EX case.
- If the last rule in the proof tree is **EG**-R or **EG**-merge, then suppose $\phi = EG_x(\phi_1)(s)$, $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and $\Gamma \vdash EG_x(\phi_1)(s)$ is provable, then either $EG_x(\phi_1)(s) \in \Gamma$, or both $\vdash (s/x)\phi_1$ and $\Gamma' \vdash EG_x(\phi_1)(s_i)$ are provable, and $\Gamma' \vdash EG_x(\phi_1)(s_j)$ is not provable for each j such that $1 \le j < i$, where $1 \le i \le n$ and $\Gamma' = \Gamma \cup \{EG_x(\phi_1)(s)\}$. In the former case, c rewrites to c_1 before it rewrites to c_2. In the latter case, according to the induction hypothesis, either
  $\mathsf{cpt}(\Gamma \vdash EG_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$
  $\mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}(...$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)), c_2) \rightsquigarrow^*$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}(...$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)) \rightsquigarrow^*$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_j), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n),$
  $c_1, c_2)...)) \rightsquigarrow^* c_1$ where $1 \le j < i$, or
  $\mathsf{cpt}(\Gamma \vdash EG_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$
  $\mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}(...$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n), c_1, c_2)...)), c_2) \rightsquigarrow^*$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n),$
  $c_1, c_2)...)) \rightsquigarrow^*$
  $\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_i), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi_1)(s_n),$
  $c_1, c_2)...)) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.
- If the last rule in the proof tree is **AF**-R$_1$ or **AF**-R$_2$, then suppose $\phi = AF_x(\phi_1)(s)$, $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and $\vdash \phi$ is provable, then either $\vdash (s/x)\phi_1$ is provable, or $\vdash AF_x(\phi_1)(s_i)$ is provable for each $1 \le i \le n$ but $\vdash (s/x)\phi_1$ is not provable. In the former case, by induction hypothesis, $\mathsf{cpt}(\vdash AF_x(\phi_1)(s), c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash (s/x)\phi_1, c_1, c_2') \rightsquigarrow^* c_1$ for some c_2' and c_2 has not been rewritten in these rewriting steps. In the latter case,
  1. if $\vdash AF_x(\phi_1)(s)$ labeled the root of a sub-proof of $\vdash AF_x(\phi_1)(s_i)$ for some $1 \le i \le n$, then by induction hypothesis, $\mathsf{cpt}(\vdash AF_x(\phi_1)(s), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2 since the proof of $\vdash AF_x(\phi_1)(s_i)$ contains a proof of $\vdash AF_x(\phi_1)(s)$;
  2. otherwise, prune the proof of $\vdash AF_x(\phi_1)(s)$ such that all nodes in the proof tree are labeled by different sequents, and $\vdash (s/x)\phi_1$ is not provable. By induction hypothesis, for all $1 \le i \le n$, $\mathsf{cpt}(\vdash AF_x(\phi_1)(s_i), c_1, c_2)$ rewrites to c_1 before it rewrites to c_2, then $\mathsf{cpt}(AF_x(\phi_1)(s) \vdash AF_x(\phi_1)(s_i), c_1, c_2)$ must rewrite to c_1 before it rewrites to c_2. This is because otherwise, if $\mathsf{cpt}(AF_x(\phi_1)(s) \vdash AF_x(\phi_1)(s_i), c_1, c_2)$ rewrites to c_2 before it rewrites to c_1, then according to the rewrite rules, there exists an infinite path $s, s_i, ..., s...$ such that for each state s' in this path, $\vdash (s'/x)\phi_1$ is not provable, then according to the soundness and completeness of the proof system, $\vdash AF_x(\phi_1)(s_i)$ cannot be provable. Thus,
     $\mathsf{cpt}(\vdash AF_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$
     $\mathsf{cpt}(\vdash (s/x)\phi_1, c_1, \mathsf{cpt}(AF_x(\phi_1)(s_1), \mathsf{cpt}(...\mathsf{cpt}($
     $AF_x(\phi_1)(s_n), c_1, c_2)...), c_2)) \rightsquigarrow^*$
     $\mathsf{cpt}(AF_x(\phi_1)(s_1), \mathsf{cpt}(...\mathsf{cpt}(AF_x(\phi_1)(s_n), c_1, c_2$
     $)...), c_2) \rightsquigarrow^* \mathsf{cpt}(AF_x(\phi_1)(s_n), c_1, c_2) \rightsquigarrow^* c_1$, and

c_2 has not been rewritten in these rewriting steps.

- If the last rule in the proof tree is **EU**-R_1 or **EU**-R_2, then suppose $\phi = EU_{x,y}(\phi_1,\phi_2)(s)$, $\{s_1,...,s_n\} = \mathsf{Next}(s)$, and $\vdash \phi$ is provable, then either $\vdash (s/y)\phi_2$ is provable, or $\vdash (s/y)\phi_2$ is not provable but both $\vdash (s/x)\phi_1$ and $\vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$ is provable for some $1 \leq i \leq n$. In the former case, by induction hypothesis, $\mathsf{cpt}(\vdash EU_{x,y}(\phi_1,\phi_2)(s),c_1,c_2) \rightsquigarrow \mathsf{cpt}(\vdash (s/y)\phi_2,c_1,c_2') \rightsquigarrow^* c_1$ for some c_2' and c_2 has not been rewritten in these rewriting steps. In the latter case,

  1. if $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$ labeled the root of a subproof of $\vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$, then by induction hypothesis, $\mathsf{cpt}(\vdash EU_{x,y}(\phi_1,\phi_2)(s),c_1,c_2)$ rewrites to c_1 before it rewrites to c_2 since we can find a smaller proof of $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$ than that of $\vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$;
  2. otherwise, prune the proof tree of $\vdash EU_{x,y}(\phi_1,\phi_2)(s)$ such that all nodes in the proof tree are labeled by different sequents. Then, there exists $1 \leq i \leq n$ such that $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_i),c_1,c_2)$ rewrites to c_1 before it rewrites to c_2. This is because otherwise, if for all $1 \leq i \leq n$, $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_i),c_1,c_2)$ rewrites to c_2 before it rewrites to c_1, then according to the rewrite rules, each branch of states searched is either of the form $s,s_i,...,s$ where $\vdash (s'/y)\phi_2$ is not provable for each state s' of the path, or of the form $s,s_i,...,s'$ where $\vdash (s'/x)\phi_1$ is not provable and $\vdash (s''/y)\phi_2$ is not provable for every state s'' of the path, and thus according to the soundness and completeness of the proof system, $\vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$ cannot be provable for each $1 \leq i \leq n$. Thus, there exists $1 \leq i \leq n$ such that $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_i),c_1,c_2)$ rewrites to c_1 before it rewrites to c_2. Then,
     $\mathsf{cpt}(\vdash EU_{x,y}(\phi_1,\phi_2)(s),c_1,c_2) \rightsquigarrow$
     $\mathsf{cpt}(\vdash (s/y)\phi_2,c_1,\mathsf{cpt}(\vdash (s/x)\phi_1,\mathsf{cpt}($
     $EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_1),c_1,\mathsf{cpt}(...$
     $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_n),c_1,c_2)...$
     $)),c_2)) \rightsquigarrow^*$
     $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_1),c_1,$
     $\mathsf{cpt}(...\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_n),$
     $c_1,c_2)...)) \rightsquigarrow^*$
     $\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_i),c_1,$
     $\mathsf{cpt}(...\mathsf{cpt}(EU_{x,y}(\phi_1,\phi_2)(s) \vdash EU_{x,y}(\phi_1,\phi_2)(s_n),$
     $c_1,c_2)...)) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.

- If the last rule in the proof tree is **AR**-R_1, **AR**-R_2, or **AR**-merge, then suppose $\phi = AR_{x,y}(\phi_1,\phi_2)(s)$, $\{s_1,...,s_n\} \in \mathsf{Next}(s)$, and $\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s)$ is provable, then at least one of the following three assertions holds, where $\Gamma' = \Gamma \cup \{AR_{x,y}(\phi_1,\phi_2)(s)\}$.

  1. $AR_{x,y}(\phi_1,\phi_2)(s) \in \Gamma$. In this case, $\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s),c_1,c_2) \rightsquigarrow c_1$.
  2. Both $\vdash (s/x)\phi_1$ and $\vdash (s/y)\phi_2$ are provable. In this case, according to the induction hypothesis,
     $\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s),c_1,c_2) \rightsquigarrow^*$
     $\mathsf{cpt}(\vdash (s/y)\phi_2,\mathsf{cpt}(\vdash (s/x)\phi_1,c_1,$
     $\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_1),\mathsf{cpt}(...\mathsf{cpt}($
     $\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n),c_1,c_2)...),c_2)),c_2) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.
  3. $\vdash (s/y)\phi_2$ is provable, and that $\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_i)$ is provable for each $s_i \in \mathsf{Next}(s)$. In this case, according to the induction hypothesis,
     $\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s),c_1,c_2) \rightsquigarrow^*$
     $\mathsf{cpt}(\vdash (s/y)\phi_2,\mathsf{cpt}(\vdash (s/x)\phi_1,c_1,$
     $\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_1),\mathsf{cpt}(...\mathsf{cpt}($
     $\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n),c_1,c_2)...),c_2)),c_2) \rightsquigarrow^*$
     $\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_1),\mathsf{cpt}(...\mathsf{cpt}($
     $\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n),c_1,c_2)...),c_2) \rightsquigarrow^*$
     $\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n),c_1,c_2) \rightsquigarrow^* c_1$, and c_2 has not been rewritten in these rewriting steps.

Thus, $\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s),c_1,c_2)$ rewrites to c_1 before it rewrites to c_2.

□