Towards Combining Model Checking and Proof Checking
Ying Jiang, Jian Liu, Gilles Dowek, Kailiang Ji

To cite this version:
Ying Jiang, Jian Liu, Gilles Dowek, Kailiang Ji. Towards Combining Model Checking and Proof Checking. The Computer Journal, In press, 10.1093/comjnl/bxy112 . hal-01970274

HAL Id: hal-01970274
https://inria.hal.science/hal-01970274
Submitted on 4 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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; jkjl@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 \(EX_i(P(x))(s)\). 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 \(s_n\) and then the right shoe of the same pair at a later state \(s_r\). This property is expressed by the formula \(EF((EF_i(P(x,y))(x))(s_0))\) 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).

In this paper, we use the calculus SCTL instead of SCTL\(_P\). 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 \(\vdash A \vdash B\), we do not need hypotheses to prove implications. So we use the left hand
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 $EG_a(EG_b(P(x,y))(x))(s)$, 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 $EG_b(P(a,y))(a)$ 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 $AF_a(\neg E(x,y))(x))(s)$ in SCTL.

The 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), in the programming language OCaml. 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, a QBF-based bounded model checker Verds, two BDD-based symbolic model checkers NuSMV and NuXMV, and the verification toolbox CADP 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 logics 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 $M$, a CTL$_P$ formula is provable in SCTL if and only if it holds in $M$.

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.
#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**

In this section, we present the logic $\text{CTL}_p(\mathcal{M})$ 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 $\rightarrow$ 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 $s \rightarrow s'$,
- and a family of relations, each being a subset of $S^n$ for some natural number $n$.

We write $\text{Next}(s)$ for the set $\{s' \in S \mid s \rightarrow s'\}$ which is always finite and not empty. A path is a finite or infinite sequence of states $s_0, s_1, \ldots, s_n$ or $s_0, s_1, \ldots$ such that for each $i$, if $s_i$ is not the last element of the sequence, then $s_{i+1} \in \text{Next}(s_i)$. 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 $\text{Next}(s)$.

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 $\text{CTL}_p(\mathcal{M})$ formulae is displayed below:

$$\phi := \top \mid \bot \mid P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n) \mid \phi \land \phi \mid \phi \lor \phi \mid AX_x(\phi)(t) \mid EX_x(\phi)(t) \mid AF_x(\phi)(t) \mid EF_x(\phi)(t) \mid AR_x(\phi_1, \phi_2)(t) \mid \text{EU}_x(\phi_1, \phi_2)(t)$$

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 $EF$ bind the variable $x$ in $\phi$, and modalities $AR$ and $EU$ bind respectively the variable $x$ in $\phi_1$ and $y$ in $\phi_2$. 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 $\phi$. As usual, in presence of binders, substitution avoids variable captures.

The following abbreviations are used.

- $\phi_1 \implies \phi_2 \equiv \neg \phi_1 \lor \phi_2$;
- $EF_x(\phi)(t) \equiv EU_x(\top)(t)$;
- $AR_x(\phi_1, \phi_2)(t) \equiv EU_x(\phi_1, \phi_2)(z) \land (z/y)\phi_2) \lor EG_x(\phi_1)(t)$, where $z$ is a variable that occurs neither in $\phi_1$ nor in $\phi_2$;
- $AG_x(\phi)(t) \equiv \neg (EF_x(\neg \phi))(t)$;
- $AU_x(\phi_1, \phi_2)(t) \equiv \neg (AR_x(\neg \phi_1, \neg \phi_2))(t)$.

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 $\phi$ be a closed $\text{CTL}_p$ formula. The validity of the formula $\phi$ in the model $\mathcal{M}$ is defined by induction on $\phi$ in Figure 1.

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

**Example 1.** The formula $EF_x(EF_x(P(x,y))(x))(a)$ 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:

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

and the set $P = \{(a, c)\}$, but not in that formed with the same relation and the set $P = \{(a, d)\}$ 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 $\text{CTL}$ 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 area. 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 an 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 $\sigma$, 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 $\sigma$. This property can be easily characterized by the $\text{CTL}_p$ formula $AG_x(AF_x(\text{D}(s)(x,y))(x))(s_0)$, where $s_0$ is the landing position of the unmanned vehicle, i.e., the initial state; and atomic formula $\text{D}(s)(x,y)$ characterizes the spatial property that the distance between state $x$ and state $y$ is longer than $\sigma$. This scenario is depicted in Figure 2. This kind of properties can be easily expressed in $\text{CTL}_p$, but cannot be elegantly, even impossibly, expressed in $\text{CTL}$, as there are no mechanisms to speak about specific states in
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.

\[ \mathcal{M} = P(s_1, \ldots, s_n), \text{ if } (s_1, \ldots, s_n) \in P \text{ with } P \text{ an } n\text{-ary relation on } \mathcal{M}; \]
\[ \mathcal{M} = \neg P(s_1, \ldots, s_n), \text{ if } (s_1, \ldots, s_n) \notin P \text{ with } P \text{ an } n\text{-ary relation on } \mathcal{M}; \]
\[ \mathcal{M} = \top \text{ is always the case;} \]
\[ \mathcal{M} = \bot \text{ is never the case;} \]
\[ \mathcal{M} = \phi_1 \land \phi_2, \text{ if } \mathcal{M} \models \phi_1 \text{ and } \mathcal{M} \models \phi_2; \]
\[ \mathcal{M} = \phi_1 \lor \phi_2, \text{ if } \mathcal{M} \models \phi_1 \text{ or } \mathcal{M} \models \phi_2; \]
\[ \mathcal{M} = AX \phi(s), \text{ if for each state } s' \text{ in } \text{Next}(s), \mathcal{M} \models (s'/x) \phi; \]
\[ \mathcal{M} = EX \phi(s), \text{ if there exists a state } s' \text{ in } \text{Next}(s) \text{ such that } \mathcal{M} \models (s'/x) \phi; \]
\[ \mathcal{M} = AF \phi(s), \text{ if there exists a finite path-tree } T \text{ such that } T' \text{'s root is labeled by } s, \text{ and for each leaf } s', \mathcal{M} \models (s'/x) \phi; \]
\[ \mathcal{M} = EG \phi(s), \text{ if there exists an infinite path } \gamma_0, \gamma_1, \ldots \text{ starting from } s, \text{ such that for all natural numbers } i, \mathcal{M} \models (s'/x) \phi; \]
\[ \mathcal{M} = AEU \phi_1(s_1), \ldots, \phi_n(s_n), \text{ if there exists a path-tree } T \text{ such that } T' \text{'s root is labeled by } s, \text{ and for each node } s \in T, \mathcal{M} \models (s'/x)\phi_1, \ldots, \phi_n \text{ and for each leaf } s' \in T, \mathcal{M} \models (s'/x)\phi_1; \]
\[ \mathcal{M} = EU \phi_1(s_1), \ldots, \phi_n(s_n), \text{ if there exists a path-tree } \gamma_0, \ldots, \gamma_n \text{ starting from } s \text{ such that } \mathcal{M} \models (s'/x)\phi_1, \ldots, \phi_n \text{ and for all } i \neq n, \mathcal{M} \models (s'/x)\phi_1. \]

3. SCTL

In this section, we present SCTL(M), a proof system for SCTL(M) taking a Kripke model as the parameter. 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 SCTL(M) if and only if it is valid in the Kripke model M.

3.1. Proofs

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

Consider now the formula AF \((AF(P(x))(a))\) that contains nested modalities. To justify the validity of this formula, one needs to provide a path-tree whose root is labeled by \(x\), and at each leaf \(a\), the formula

\[ \vdash (x/x)\phi \quad \text{AF-R}_1 \]
\[ \vdash AF_1(\phi)(s_1) \ldots \vdash AF_1(\phi)(s_n) \quad \text{AF-R}_2 \]
\[ \vdash AF_1(\phi)(s) \quad [s_1, \ldots, s_n] = \text{Next}(s) \]

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

\[ a \xrightarrow{b} d \xrightarrow{c} \]

A proof of the formula AF \((P(x))(a)\) is

\[ \vdash P(b) \quad \text{atom-R} \]
\[ \vdash AF_1(P(x))(b) \quad \text{AF-R}_1 \]
\[ \vdash AF_1(P(x))(a) \quad \text{AF-R}_2 \]

where besides the rules AF-R1 and AF-R2, we use the following rule.

\[ \vdash P(x_1, \ldots, x_n) \quad \text{atom-R} \]
\[ (x_1, \ldots, x_n) \in P \]

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

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

\[ \Gamma \vdash \phi_1, \phi_2, \Delta \]
\[ \Gamma \vdash \phi_1 \lor \phi_2, \Delta \]

or in an additive way

\[ \Gamma \vdash \phi_1, \Delta \]
\[ \Gamma \vdash \phi_1 \lor \phi_2, \Delta \]

\[ \text{FIGURE 2. Possible positions of an unmanned vehicle} \]

\[ \text{FIGURE 1. Validity of a formula in CTL}_P \]

\[ \text{FIGURE 3. Possible positions of an unmanned vehicle} \]

\[ \text{FIGURE 4. Possible positions of an unmanned vehicle} \]
The sequents have therefore the form $\Gamma \vdash \phi$, with a non-empty $\Gamma$ 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 $s_0, \ldots, s_n$ be a finite sequence of states such that for all $i$ between $0$ and $n-1$, $s_i \to s_{i+1}$, and $s_n = s_p$ for some $p$ between $0$ and $n-1$. Then there exists an infinite sequence of states $s'_0, s'_1, \ldots$ such that $s_0 = s'_0$ and for all $i$, $s'_i \to s'_{i+1}$, and all the $s'_j$ are among $s_0, \ldots, s_n$.

**Proof.** Take the sequence $s_0, \ldots, s_p, s_{p+1}, \ldots, s_{n-1}, s_p, \ldots$, 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 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 $s_0, s_1, \ldots$ be an infinite sequence of states such that for all $i$, $s_i \to s_{i+1}$. Then there exists a finite sequence of states $s'_{0}, \ldots, s'_{n}$ such that for all $i$ between $0$ and $n-1$, $s'_i \to s'_{i+1}$, $s'_n = s'_p$ for some $p$ between $0$ and $n-1$, and all the $s'_j$ are among $s_0, s_1, \ldots$.

**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 $s_0, \ldots, s_n$.

---

13Intuitively, we use the left hand side of the sequent to store visited states for co-inductive modalities.
**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 Kónig’s lemma, it is finite. □

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

**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 $\text{CTL}_P$ 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 $t$ or $f$, where $t$ and $f$ are......
where $c$ 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 $\text{cpt}(\Gamma \vdash \phi, c_1, c_2)$, or visually by

$$\Gamma \vdash \phi \quad \xrightarrow{c_1} \quad \xrightarrow{c_2}$$

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 $\text{cpt}(\Gamma \vdash \phi, t, f)$, reduces to $t$ or to $f$. To do so, we analyze the form of the formula $\phi$. If, for instance, it is $\phi_1 \land \phi_2$, we transform, using one of the rewrite rules, the tree $\text{cpt}(\vdash \phi_1 \land \phi_2, t, f)$ into $\text{cpt}(\vdash \phi_1, \text{cpt}(\vdash \phi_2, t, f), f)$ which is depicted as

$$\Gamma \vdash \phi_1 \quad \xrightarrow{\phi_2} \quad \xrightarrow{f}$$

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 $f$. Then the reduction procedure will apply to $\text{cpt}(\vdash \phi_1, \text{cpt}(\vdash \phi_2, t), f)$ by considering the form of $\phi_1$, 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 $\vdash$, 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
Step 3

The above information is encoded into a single CPT that is on the right side of $\Rightarrow$. 

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 $\Rightarrow$ reduces to its left sub-tree (t-continuation) which is $t$. 

Step 5. Like at step 1 and 3, we cannot judge whether the sequent $\Gamma \vdash AF(P(x))(c)$ is provable immediately, so we encode the two steps to find successively the proofs of $\vdash P(c)$ and $\Gamma_{3} \vdash AF(P(x))(d)$ into the CPT which is on the right side of $\Rightarrow_{5}$. 

Step 6. Like at step 2 and 4, as the atomic formula $P(c)$ is provable, the CPT on the left side of $\Rightarrow_{6}$ reduces to its left sub-tree (t-continuation) which is $t$. Now, the proof search of $\vdash AF(P(x))(a)$ 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 (formulæ starting with modalities $AF$ or $EU$ etc.) in the proof system SCTL, we do need merge rules for this kind of formulæ 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 $\vdash EU_{s_{0}}(\phi_{1}, \phi_{2})(s)$, we need to find a finite path of states, say $s_{0} (= s_{1}, \ldots, s_{n})$, in the Kripke model $M$ under consideration, such that $\phi_{2}$ holds at state $s_{n}$ and $\phi_{1}$ holds at all other states $s_{i}$ (if any). However, $M$ may also contain infinite paths of states, say $s_{0} (= s_{1}, \ldots)$, such that $\phi_{1}$ holds at each state $s_{i}$ ($i \geq 0$), and $\phi_{2}$ does not hold at any of them. On such an infinite path, the proof search procedure for the sequent $\vdash EU_{s_{0}}(\phi_{1}, \phi_{2})(s)$ cannot terminate. To deal with this kind of scenarios, in the CPT rewriting system, the merge rules for inductive formulæ are introduced.

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

- If $EG(\phi)(s) \in \Gamma$, then we have $\text{cpl}(\Gamma \vdash EG(\phi)(s), c_{1}, c_{2}) \Rightarrow c_{1}$ according to the CPT rewriting system, and we proceed with the reduction of $c_{1}$, where $\text{cpl}(\Gamma \vdash EG(\phi)(s), c_{1}, c_{2})$ is a corresponding CPT of the sequent $\Gamma \vdash EG(\phi)(s)$.

- If $EG(\phi)(s) \notin \Gamma$, then one of the following cases holds:
  - if $\vdash (s/x)\phi$ is not provable in SCTL, then we have $\text{cpl}(\Gamma \vdash EG(\phi)(s), c_{1}, c_{2}) \Rightarrow c_{2}$, and proceed with the reduction of $c_{2}$, where $\text{cpl}(\Gamma \vdash AF(P(x))(a)) \vdash P(a)$ is provable by applying the $AF$-R$_{1}$ rule, that is, if $\vdash P(a)$ is provable.

- If $\Gamma \vdash AF(P(x))(b)$ is provable by applying the $AF$-R$_{2}$ rule, that is, if $\Gamma_{2} \vdash AF(P(x))(d)$ is provable.

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cpt_rewrites.png}
\caption{An illustration of CPT rewrites}
\end{figure}

- If $\Gamma_{1} \vdash AF(P(x))(a)$ is provable by applying the $AF$-R$_{1}$ rule, that is, if $\vdash P(a)$ is provable.

- If $\Gamma_{1} \vdash AF(P(x))(b)$ is provable by applying the $AF$-R$_{2}$ rule, that is, if $\Gamma_{2} \vdash AF(P(x))(d)$ is provable.

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

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

Step 3. Like at step 1, we need to decide if $\Gamma_{1} \vdash AF(P(x))(b)$ is provable, which is not known at that moment. According to the CPT which is on the left side of $\Rightarrow_{2}$, we know that if $\Gamma_{1} \vdash AF(P(x))(b)$ is provable then we need to decide if $\Gamma_{1} \vdash AF(P(x))(c)$ is provable. Then we have to decide successively:

- If $\Gamma_{1} \vdash AF(P(x))(b)$ is provable by applying the $AF$-R$_{1}$ rule, that is, if $\vdash P(b)$ is provable.

- If $\Gamma_{1} \vdash AF(P(x))(b)$ is provable by applying the $AF$-R$_{2}$ rule, that is, if $\Gamma_{2} \vdash AF(P(x))(d)$ is provable.
$EG_i(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma \vdash EG_i(\phi)(s)$;
- otherwise, $EG_i(\phi)(s)$ is added into the context, and we proceed with the reduction of a corresponding CPT of a sequent $\Gamma' \vdash EG_i(\phi)(s')$, where $\Gamma' = \Gamma \cup \{EG_i(\phi)(s)\}$ and $s' \in Next(s)$.

In the case of inductive formulæ: consider, for example, the sequent $\Gamma \vdash AF_i(\phi)(s)$.
- If $AF_i(\phi)(s) \in \Gamma$, then there exists a rewriting step $\text{cpt}(\Gamma' \vdash AF_i(\phi)(s), c_1, c_2) \Rightarrow c_2$ according to the CPT rewriting system, and we proceed with the reduction of $c_2$, where $\text{cpt}(\Gamma' \vdash AF_i(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma' \vdash AF_i(\phi)(s)$;
- If $AF_i(\phi)(s) \notin \Gamma$, then one of the following cases holds:
  - if $\vdash (s/x)\phi$ is provable in SCTL, then we have $\text{cpt}(\Gamma' \vdash AF_i(\phi)(s), c_1, c_2) \Rightarrow c_1$, and proceed with the reduction of $c_1$, where $\text{cpt}(\Gamma' \vdash AF_i(\phi)(s), c_1, c_2)$ is a corresponding CPT of the sequent $\Gamma' \vdash AF_i(\phi)(s)$;
  - otherwise, $AF_i(\phi)(s)$ is added into the context, and for all $s' \in Next(s)$, we proceed with the reduction of a corresponding CPT of the sequent $\Gamma' \vdash AF_i(\phi)(s')$, where $\Gamma' = \Gamma \cup \{AF_i(\phi)(s)\}$.

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 $Q = \{b, c\}$.

```
      a
     /  \\
    b     c
     \  /
      d
```

We apply the proof search method to the co-inductive formulæ $EG_i(P(x))(a)$ and $EG_i(Q(x))(a)$ and to the inductive formulæ $AF_i(\neg P(x))(a)$ and $AF_i(\neg Q(x))(a)$, respectively.

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

$$
\begin{align*}
\text{cpt}(\vdash EG_i(P(x))(a), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash EG_i(P(x))(c), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash EG_i(P(x))(c), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash EG_i(P(x))(c), t, f) & \Rightarrow \\
\end{align*}
$$

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

$$
\begin{align*}
\text{cpt}(\vdash AF_i(\neg Q(x))(a), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash AF_i(\neg Q(x))(b), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash AF_i(\neg Q(x))(c), t, f) & \Rightarrow \\
\text{cpt}(\Gamma' \vdash AF_i(\neg Q(x))(c), t, f) & \Rightarrow \\
\end{align*}
$$

where $\Gamma = \{AF_i(\neg Q(x))(a)\}$, $\Gamma_1 = \Gamma \cup \{AF_i(\neg Q(x))(b)\}$, and $\Gamma_2 = \Gamma \cup \{AF_i(\neg Q(x))(d)\}$.

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 $\succeq \geq$ be a quasi-ordering (i.e., a binary relation that is reflexive and transitive) on $F$. The lexicographic path ordering $\succeq_{\text{lex}}$ on the set $T(F)$ of terms over $F$ is defined recursively as follows.

$$s \succeq_{\text{lex}} t \text{ if and only if at least one of the following assertions holds:}
\begin{align*}
1. & s \succeq_{\text{lex}} t \text{ for some } t \in \{1, \ldots, m\}, \\
2. & f \succeq_{\text{lex}} g \text{ and } s \succeq_{\text{lex}} t_j \text{ for all } j \in \{1, \ldots, n\}, \\
3. & f = g \text{ and } (s_1, \ldots, s_m) \succeq_{\text{lex}} (t_1, \ldots, t_n) \text{ and } s \succeq_{\text{lex}} t_j \text{ for all } j \in \{2, \ldots, n\}, \text{ where } s \succeq_{\text{lex}} t_j \text{ is the lexicographic ordering induced by } s \succeq_{\text{lex}}.
\end{align*}$$

As usual, $f \succeq_{\text{lex}} g$ (resp. $s \succeq_{\text{lex}} t$) is shorthand for “$f \succeq g$” (resp. “$s \succeq t$”).

**PROPOSITION 4.1 (Well-foundedness of lexicographic path orderings).** Let $\succeq \geq$ be a quasi-ordering on a set $F$ of function symbols where each element has a fixed arity, and $\succeq_{\text{lex}}$ be the lexicographic path ordering on the set $T(F)$ of terms over $F$. Then $\succeq_{\text{lex}}$ 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 $M$ with cardinal $n$ and a sequent $\Gamma \vdash \phi$ in SCTL($M$), let $|\phi|$ denote the size of $\phi$ defined in the usual way, and $|\Gamma|$ denote the cardinal of a context $\Gamma$. We define the weight of the sequent $\Gamma \vdash \phi$ as $w(\Gamma \vdash \phi) = |\phi|, (n - |\Gamma|)$.

It is 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.
PROPOSITION 4.2 (Termination). cpt(⌜φ, t, f⌝) always rewrites to t or 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 t or f. Now we prove that the rewriting of a CPT always terminates. Consider a CPT cpt(⌜φ, t, f⌝), we define first the set of functions symbols as follows:

\[ F = \{t, f, \text{cpt}\} \cup \text{Seq} \]

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

- \(\text{cpt} \succeq t\)
- \(\text{cpt} \succeq f\)
- \(Γ \vdash \phi \Rightarrow \text{cpt}\) for each sequent \(Γ \vdash φ\)
- \(Γ \vdash φ \Rightarrow Γ' \vdash φ' \iff \text{w}(Γ' \vdash φ') > \text{w}(Γ' \vdash φ')\), 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 \(\succeq_{\text{lpo}}\) on CPTs. By Proposition 4.1, \(\succeq_{\text{lpo}}\) is also well-founded. Then it is sufficient to prove that, for any CPTs \(c\) and \(c'\),

\[ c \rightsquigarrow c' \text{ implies } c \succ_{\text{lpo}} c' \]

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 \(\text{cpt}(⌜φ, c_1, c_2⌝)\).

- If \(φ = T, \bot, P(s_1, ..., s_m)\) or \(¬P(s_1, ..., s_m)\), then according to Definition 4.2 (1), we have \(c_1 \preceq c_2\), because \(c_1, c_2\) are sub-terms of \(\text{cpt}(⌜φ, c_1, c_2⌝)\).
- If \(φ = φ_1 \lor φ_2\), then according to Definition 4.2 (3), we have \(c_1 \preceq c_2\) because \(φ_1, φ_2 \Rightarrow Γ, c_1, c_2\).
- If \(φ = AX(φ_1)\), then by Definition 4.2 (3), we have \(c_1 \preceq c_2\) because \(AX(φ_1)\).
- If \(φ = EG(φ_1)\), then by Definition 4.2 (1), we have \(c_1 \preceq c_2\) because \(EG(φ_1)\).
- If \(φ = AR_x(φ_1, φ_2)\), then by Definition 4.2 (2), we have \(c_1 \preceq c_2\) because \(AR_x(φ_1, φ_2)\).

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

PROPOSITION 4.3 (Correctness). For any given closed formula \(φ\), \(\text{cpt}(⌜φ, t, f⌝) \rightsquigarrow t\) if and only if \(Γ \vdash φ\) 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\(^10\) (in the form of hash tables or BDDs) to store respectively the set of visited states and the states currently been visited as well. 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 \(2n + 2\) (\(n \geq 1\)) states: \(s_0, s_1, s_2, ..., s_{n-1}, s_n, s_0, 0, ..., s_{n-2}\), and a set \(P = \{s_0, s_1, s_2, ..., s_{n-1}, s_n, 0, ..., s_{n-2}\}\), that is the property \(P\) holds at all states of this model except at state \(s_{n}\). The graph of transition relation of this model is depicted as follows.

\(^{10}\)A more detailed explanation of the global variables in the proof search algorithm can be found in Appendix A.
When proving the formula $EG_i(\mathcal{P}(x))(s_0)$, we need to rewrite the CPT $\text{cpt}(\vdash EG_i(\mathcal{P}(x))(s_0), t, f)$. Suppose that $s_1$ is searched before $s_2$ in $\text{Next}(s_1)$, and $s_1, s_2$ is searched before $s_{2,i+1}$ in $\text{Next}(s_{2,i})$ for all $i$ between 0 and $n-1$. The corresponding rewriting steps of the CPT are depicted in Figure 7.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.png}
\caption{Rewriting steps of $\text{cpt}(\vdash EG_i(\mathcal{P}(x))(s_0), t, f)$}
\end{figure}

where

$$
\begin{align*}
\Gamma_0 &= \{ EG_i(\mathcal{P}(x))(s_0) \}, \\
\Gamma_{i+1} &= \{ EG_i(\mathcal{P}(x))(s_i), EG_i(\mathcal{P}(x))(s_{i+1}) \}.
\end{align*}
$$

In Figure 7, before rewriting $\text{cpt}(\Gamma_{i+1} \vdash EG_i(\mathcal{P}(x))(s_{i+1}), t, f)$, all states in the path $\pi = s_1, s_2, \ldots, s_{i+1}$ are already visited, and each time we rewrite $\text{cpt}(\Gamma_{i+1} \vdash EG_i(\mathcal{P}(x))(s_{i+1}), t, f)$, all states in $\pi$ 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 $\pi$ can be avoided. Indeed, $M$ stores all visited state $s$ for which $EG_i(\mathcal{P}(x))(s)$ does not hold, and before we rewrite $\text{cpt}(\Gamma_{i+1} \vdash EG_i(\mathcal{P}(x))(s_{i+1}), t, f)$, all states in $\pi$ 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 $\text{SCTL}$.
- $\text{c}$ denotes the CPT to be rewritten.
- $\text{pt}$ and $\text{ct}$ denote the proof tree and counterexample to be constructed during the proof search, respectively.
- $\text{M}$ (resp. $\text{M}'$) is used to store the set of visited states on which a sub-formula of $\psi$ holds (resp. does not hold), and they are used to avoid visiting the same state repeatedly.
- $\text{visited}$ is used to store temporarily the visited states by sub-formula of $\psi$, starting with $\text{E}U$ or $\text{A}R$ (if any), and these states will be added into either $\text{M}$ or $\text{M}'$ in later rewriting steps.
- We associate a set of actions A to each CPT $\text{c}$, written as $c^A$. When rewriting a CPT $\text{c}$, each action in $A$ needs to be performed. For instance, for a CPT $\text{c} = \text{cpt}(\Gamma \vdash \phi, c_1, c_2)$, three sets of actions $A_0, A_1, A_2$ are associated to $c, c_1, c_2$, respectively. $c^{A_0}$ is written as $\text{cpto}(\Gamma \vdash \phi, c_1^{A_1}, c_2^{A_2})$. When rewriting $\text{c}$, the actions in $A_0$ need to be performed. The actions in $A_2$ 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 $\text{c}$, until either $e$ or $f$ is reached. For the sake of brevity, only the subroutine $\text{ProveAnd}$ is presented (Figure 9), where $ac(t, parent, children)$ represents the operation of adding each element in children as a child to the node parent in the tree $t$. 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 $\text{cpt}(\vdash \phi, f, t)$, each sub-formula of $\phi$ appears, in the worst case, $|M|$ times at the roots of CPTs that appear in the rewriting steps starting with $\text{cpt}(\vdash \phi, f, t)$, where $|M|$ is the number of states in the Kripke model under consideration. Therefore, the time complexity of the proof search algorithm is $O(|\phi| \times |M|)$, where $|\phi|$ is the size of the formula $\phi$ to be proved.

4.3 SCTLProV

In this section, we develop a new automated theorem prover called SCTLProV. As 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.

\begin{tabular}{c}
\hline
\textbf{THE COMPUTER JOURNAL, Vol. ??, No. ??, ??} \\
\hline
\end{tabular}

\footnote{https://github.com/sctlpov/sctlpov_code}
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 = cpt(ψ, pt, φ₁, φ₂)
3: let pt = ce = (tree with a single node: !ψ)
4: let M₀ = M¹ = visited = {empty hash table}
5: while c has the form cpt(ψ, Γ ⊨ φ, c₁, c₂) do
6: ∀a ∈ A₀, perform action a
7: case φ
8: T: c := A₁
9: L: c := A₂
10: P(s₁, s₂, ...): P := (s₁, s₂, ...)
11: if (s₁, ..., sₙ) ∈ P then c := c₁ else c := c₂ end if
12: if (s₁, ..., sₙ) ∈ P then c := c₁ else c := c₂ end if
13: φ₁ ∨ φ₂: ProveOr(!φ₁ ∨ φ₂)
14: EX₁(φ₁)(s): ProveEX(Γ ⊨ EX₁(φ₁)(s))
15: AX₁(φ₁)(s): ProveAX(Γ ⊨ AX₁(φ₁)(s))
16: EG₁(φ₁)(s): ProveEG(Γ ⊨ EG₁(φ₁)(s))
17: AF₁(φ₁)(s): ProveAF(Γ ⊨ AF₁(φ₁)(s))
18: EUₓ₁(φ₁)(φ₂)(s): ProveEU(Γ ⊨ EUₓ₁(φ₁)(φ₂)(s))
19: ARₓ₁(φ₁)(φ₂)(s): ProveARE(Γ ⊨ ARₓ₁(φ₁)(φ₂)(s))
20: end case
21: end while
22: if c = tₕ then
23: ∀a ∈ A, perform a
24: return (true, pt)
25: end if
26: if c = tₕ then
27: ∀a ∈ A, perform a
28: return (false, ce)
29: end if
30: end function

FIGURE 8. The proof search algorithm

1: let A₁₂ = A₁ ∪ {ac(φ₁ ∧ φ₂, {1 ⊢ φ₁ ∧ φ₂})}
2: let A₁₃ = A₁ ∪ {ac(φ₁ ∧ φ₂, {1 ⊢ φ₁})}
3: let A₁₄ = A₁ ∪ {ac(φ₁ ∧ φ₂, {1 ⊢ φ₂})}
4: let c' = cpt(φ₁, cpt(φ₁ ∨ φ₂, c₁, c₂), c₂)
5: c := c'

FIGURE 9. ProveAnd(!φ₁ ∧ φ₂)

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 M with the initial state s₀ and a transition relation T. To check whether M, s₀ ⊨ EFφ holds in such a model checker, say NuSMV, first one needs to calculate the least fixed point lfp = μY(φ ∧ EXY), then check whether s₀ ∈ lfp [29, 30]. Calculating lfp may consume a lot of time and stack space during the verification process.

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

![FIGURE 10. A general work flow of SCTLProV](image-url)
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 \(M, s_0 \models \varphi\), 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]:

\[
|M, E\varphi|_{k+1} : = \bigwedge_{i=0}^{k} T(s_{i}, s_{i+1}) \land \bigvee_{j=0}^{k} \varphi(s_{j})
\]

To avoid exploring unnecessary states in \(M\), 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 \(\vdash E\varphi_i\varphi(s_0)\), SCTLProV unfolds the transition relation \(T\) and the formula \(E\varphi_i\varphi(s_0)\) as

\[
\text{unfold}(S, E\varphi_i\varphi(s_0)) := 
\varphi(s_0) \lor ((s_0 \notin S) \land T(s_0, s_1) \land \text{unfold}(S \cup \{s_0\}, E\varphi_i\varphi(s_1)))
\]

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 \(\varphi \in C\), \(\varphi\) is valid infinitely often on this path. An SCTL formula \(EG_x\varphi(t)\) under fairness constraint \(C\), written as \(ECG_x\varphi(t)\), 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, \((s/x)\varphi\) is valid. Similarly, an SCTL formula \(AF_x\varphi(t)\) under fairness constraint \(C\), written as \(ACF_x\varphi(t)\), 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 \((s/x)\varphi\) is valid.

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

\[
egin{align*}
ECX_x\varphi(t) &= EX_x(\varphi \land ECG_{\top}(\varphi)(s))(t) \\
ACX_x\varphi(t) &= AX_x(\varphi \lor ACF_{\bot}(\varphi)(s))(t) \\
ECU_x(\varphi_1, \varphi_2)(t) &= EU_x(\varphi_1, \varphi_2 \land ECG_{\top}(\varphi_1)(s))(t) \\
ACR_x(\varphi_1, \varphi_2)(t) &= AR_x(\varphi_1, \varphi_2 \lor ACF_{\bot}(\varphi_2)(s))(t)
\end{align*}
\]

Since SCTL is sound and complete, to prove \(ECG_x\varphi(t)\) is equivalent to prove \(EG_x\varphi(t)\) only if only fair paths are considered, i.e., to prove the existence of a fair path on which \(\varphi\) is always provable. Similarly, to prove \(ACF_x\varphi(t)\) is equivalent to prove \(AF_x\varphi(t)\) only if only fair paths are considered, i.e., to prove the absence of a fair path on which \(\varphi\) 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 \(s_i \rightarrow s_{i+1}\) for all \(i\). If every element of \(C\) is valid infinitely often on \(\sigma\), 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 \(\sigma\),
- \(s_j \rightarrow s_{j+1}\) for all \(0 \leq j \leq n-1\),
- \(s_0 = s_{n}\) for some \(0 \leq p \leq n-1\),
- for every \(\varphi \in C\), \(\varphi\) is valid at some state \(s_q\) where \(p \leq q \leq n\).

**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 \(\sigma\), and each state not in \(S\) appears finitely often in \(\sigma\). Then, each formula in \(C\) must be valid at some state in \(S\), this is because otherwise, if there exists a formula \(\varphi \in C\) which is not valid on any state in \(S\), then \(\varphi\) must be valid only finitely often on \(\sigma\). Assume \(S = \{s_0, s_1, \ldots, s_k\}\) such that \(s_0 \leq s_1 \leq \ldots \leq s_k\); then let \(\sigma_f\) be a prefix of \(\sigma\): \(\sigma_f = s_0, s_1, \ldots, s_q = s_0, \ldots, s_1, \ldots, s_{q-1}, s_q = s_0, \ldots, s_1, \ldots, s_{q-1}, s_q, \ldots, s_k\). 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
• $s_i \rightarrow s_{i+1}$ 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 $s_i'$ is a state on $\sigma$,
• $s_i' \rightarrow s_{i+1}'$ for all $i \geq 0$,
• every formula in $C$ is valid infinitely often on $\sigma$.

Proof. It is sufficient to take

$$\sigma = s_0, s_1, s_2, \ldots$$

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. 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. 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.

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

\[ id : \text{sequent} [id_1, \ldots, id_n] \]

where $\text{sequent}$ 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.

\[ /* 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;
Towards Combining Model Checking and Proof Checking

This proof tree can be visualized by a visualization tool VMDV\(^2\) (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 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.

\(\text{verifying on the model mutual... find_bug: EU(x, y, TRUE, bug(y), ini) find_bug is false.}\

\(\text{https://github.com/terminatorlxj/VMDV}\

**FIGURE 13.** The output file “output.out”

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

**FIGURE 15.** A simple solution of the mutual exclusion problem
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 \(\land\) and \(\lor\) by \(\lor\) and \(\land\), 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 \(b\).

<table>
<thead>
<tr>
<th>(P)</th>
<th>benchmarks #1</th>
<th>benchmarks #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_{1})</td>
<td>1816 (63.1%)</td>
<td>736 (12.8%)</td>
</tr>
<tr>
<td>(P_{2})</td>
<td>2230 (77.4%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{3})</td>
<td>2741 (95.2%)</td>
<td>2763 (95.9%)</td>
</tr>
<tr>
<td>(P_{4})</td>
<td>2748 (44.7%)</td>
<td>2741 (95.2%)</td>
</tr>
<tr>
<td>(P_{5})</td>
<td>2870 (99.7%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{6})</td>
<td>2935 (98.2%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{7})</td>
<td>2999 (99.2%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{8})</td>
<td>3073 (99.7%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{9})</td>
<td>3148 (99.9%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{10})</td>
<td>3223 (100%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{11})</td>
<td>3298 (100%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{12})</td>
<td>3373 (100%)</td>
<td>2823 (98.0%)</td>
</tr>
<tr>
<td>(P_{13})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{14})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{15})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{16})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{17})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{18})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{19})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{20})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{21})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{22})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{23})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{24})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(P)</th>
<th>benchmarks #1</th>
<th>benchmarks #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_{13})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{14})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{15})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{16})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{17})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{18})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{19})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{20})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{21})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{22})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{23})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
<tr>
<td>(P_{24})</td>
<td>4441 (77.1%)</td>
<td>4441 (77.1%)</td>
</tr>
</tbody>
</table>

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. 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.

![Figure 17](image1.png)  
**FIGURE 17.** Average verification time in benchmark #1 and #2

![Figure 18](image2.png)  
**FIGURE 18.** Average memory usage in benchmark #1 and #2

<table>
<thead>
<tr>
<th>Bench</th>
<th>SCTLProV</th>
<th>SCTLProV&lt;sub&gt;R&lt;/sub&gt;</th>
<th>(SCTLProV &lt; SCTLProV&lt;sub&gt;R&lt;/sub&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>2842 (99.4%)</td>
<td>2082 (93.7%)</td>
<td>760 (90.2%)</td>
</tr>
<tr>
<td>#2</td>
<td>2646 (97.8%)</td>
<td>5300 (98.4%)</td>
<td>6541 (97.7%)</td>
</tr>
</tbody>
</table>

**TABLE 3.** SCTLProV vs. SCTLProV<sub>R</sub>

### 6.3. Programs with fairness constraints

In this part, first we evaluate benchmark #3, which models mutual exclusion algorithms and ring algorithms. 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, nom, (resp. try<sub>i</sub>, cri<sub>i</sub>) 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 \( P_0 \) and \( P_3 \).

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

---

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

22http://lcs.ioa.ac.cn/~zwh/verds/verds_code/bpi2.rar
There are 194 (74.0%) test cases on which SCTLProV and NuXMV algorithms (Table 5). In these formulae, there are 66 (25.2%) test cases that can only be solved by Verds and 21 (65.6%) test cases by NuSMV, respectively. There are 71 (27.1%) test cases that can only be solved by NuXMV or by SCTLProV.

There is no test case that can only be solved by SCTLProV. There are 211 (80.2%) test cases (Table 6). In these formulae, there are 66 (25.2%) test cases that can only be solved by Verds, 20 (62.5%) test cases that can only be solved by NuSMV, and 20 (62.5%) test cases that can only be solved by NuXMV, respectively.

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 #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).

There are 23 (9.1%) test cases in benchmark #4 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 NuSMV, respectively.

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

<table>
<thead>
<tr>
<th>Programs</th>
<th>Verds</th>
<th>NuSMV</th>
<th>NuXMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutual exclusion</td>
<td>181 (86.3%)</td>
<td>191 (83.9%)</td>
<td>191 (83.9%)</td>
</tr>
<tr>
<td>Ring</td>
<td>7 (3.7%)</td>
<td>20 (62.5%)</td>
<td>20 (62.5%)</td>
</tr>
<tr>
<td>Sum</td>
<td>194 (74.0%)</td>
<td>211 (80.2%)</td>
<td>211 (80.2%)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cases</th>
<th>u(SCTLProV) &lt; u(CADP)</th>
<th>m(SCTLProV) &lt; m(CADP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadlock</td>
<td>7 (27.1%)</td>
<td>3 (11.1%)</td>
</tr>
<tr>
<td>livelock</td>
<td>22 (85.0%)</td>
<td>6 (18.0%)</td>
</tr>
</tbody>
</table>

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

6.4. The VLTS benchmark

In this part, we evaluate benchmark #4, which is also called the VLTS (Very Large Transition Systems) benchmark, which was originally proposed as a part of the CADP (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).
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 SCTLProV25.

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

25https://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\textsubscript{P} which extends CTL with polyadic predicate symbols. CTL\textsubscript{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\textsubscript{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 iPProver 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 iPProver 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\textsubscript{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

Towards Combining Model Checking and Proof Checking


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 SCTL(\(M\)) formula \(\phi\) with modality \(AF, EG, AR\), or EU, a schema \(\phi^0\) of \(\phi\) is defined by replacing the state constant in \(\phi\) by \(\_\). For instance, the schema of the formula \(EU_{x_1}(\phi_1,\phi_2)(s)\) is \(EU_{x_1}(\_1,\_2)(\_),\) and the schema of the formula \(AF_{x_1}(\phi_1)(s')\) is \(AF_{x_1}(\_1)(\_).\) A formula schema can be viewed as a context of a state constant in an SCTL(\(M\)) formula.

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

- \(\vdash\) 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. \(M^e\)) is used to store the set of visited states on which a sub-formula of \(\Psi\) holds (resp. does not hold), and they are used to avoid visiting the same state repeatedly. Note that \(M^t\) and \(M^e\) 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, \(M^t_{EG,\phi}(\_|\_)\) to the set of states \(S\) such that \(\forall s \in S, EG,\_\phi(s)\) holds, and \(M^t_{EG,\phi}(\_|\_)\) to the set of states \(S\) such that \(\forall s \in S, EG,\_\phi(s)\) does not hold.

- visited is used to store temporally the visited states by sub-formula of \(\Psi\), starting with EU or AR (if any), and these states will be added into either \(M^t\) or \(M^e\) in later rewriting steps. Like \(M^t\) and \(M^e\), visited can also be seen as a hash table. For instance, we denote visited \(EU_{x_1}(\_1,\_2)(\_)\) 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 \(c = \text{cpt}(\Gamma \vdash \phi, c_1, c_2)\), three sets of actions \(A_0, A_1, A_2\) are associated to \(c, c_1, c_2\), respectively. \(c^A_0\) is written as \(\text{cpt}^A_0(\Gamma \vdash \phi, c^A_1, c^A_2)\). 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 $\phi$ is an atomic formula or the negation of an atomic formula is presented, and the explanation for other cases are presented separately in the following Subsections.

### Appendix A.2. ProveAnd ($\neg \phi_1 \land \phi_2$)

In the case of ProveEX, we let $\{s_1, ..., s_n\} = \text{Next}(s)$, 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 $\emptyset$ case in the future steps, then $\vdash (s_i/x)\phi_1$ should be added as the children of $\vdash \text{EX}_{n}(\phi_1)(s)$ in the proof tree (Line 4), and otherwise, when $c$ rewrites to $c_2$, then $\vdash (s_1/x)\phi_1, ..., (s_n/x)\phi_1$ should be added as the children of $\vdash \text{EX}_{n}(\phi_1)(s)$ in the counterexample (Line 3).

### Appendix A.3. ProveOr ($\neg \phi_1 \lor \phi_2$)

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

### Appendix A.4. ProveEX and ProveAX

The pseudo code for the ProveAnd case is depicted in Figure A.2, where $ac(t, parent, children)$ 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 $c_4$ 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 $\vdash \neg \phi_1 \land \phi_2$ 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 $\vdash \phi_1 \land \phi_2$ in the counterexample.
- Line 5: Rewrite $c$ to $c'$.

![FIGURE A.1. The proof search algorithm](image)

![FIGURE A.2. ProveAnd ($\neg \phi_1 \land \phi_2$)](image)

![FIGURE A.3. ProveOr ($\neg \phi_1 \lor \phi_2$)](image)

![FIGURE A.4. ProveEX ($\neg \text{EX}_n(\phi_1)(s)$)](image)

![FIGURE A.5. ProveAX ($\neg \text{AX}_n(\phi_1)(s)$)](image)
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 \(\{s_1,\ldots,s_n\} = \text{Next}(s)\), and \(\Gamma = \Gamma \cup \{\text{EG}_s(\phi_1)(s)\}\). The pseudo code for this case is explained as follows.

- Line 3 – 4: If \(\text{EG}_s(\phi_1)(s)\) does not hold, then just rewrite \(c\) to \(c_2\).
- Line 5 – 7: If \(\text{EG}_s(\phi_1)(s)\) holds, or \(\text{EG}_s(\phi_1)(s) \in \Gamma\), then rewrite \(c\) to \(c_1\), and every formula in \(\Gamma\) holds and its corresponding state is thus added into \(M^i_{\text{EG}_s(\phi_1)(\cdot)}\).
- 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 \(\vdash (s/x) \phi_2\) is not provable, and thus \(\vdash (s/x) \phi_1\) will be added as the evidence of \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) in the counterexample (Line 9). At the same time, since \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) is not provable, we add \(s\) into \(M^i_{\text{EG}_s(\phi_1)(\cdot)}\) (Line 13).
2. If \(c'\) rewrites to the inner \(c_2\), then \(\Gamma \vdash \text{EG}_s(\phi_1)(s_1),\ldots\), \(\Gamma \vdash \text{EG}_s(\phi_1)(s_n)\) are all added as the evidence of \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) in the counterexample (Line 10). At the same time, since \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) is not provable, we add \(s\) into \(M^i_{\text{EG}_s(\phi_1)(\cdot)}\) (Line 14).
3. If \(c'\) rewrites to the \(i\)th \(c_1\), then both \(\vdash (s/x) \phi_1\) and \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) are provable, and thus should be added as the evidence of proving \(\Gamma \vdash \text{EG}_s(\phi_1)(s)\) in the proof tree (Line 11 – 12).
- Line 17: Rewrite \(c\) to \(c'\).

1: /* For notation purpose, here we refer “k” to AF\(_s(\phi_1)(\cdot)\) and “k(s)” to AF\(_s(\phi_1)(s)\). */
2: "k(s)" to AF\(_s(\phi_1)(s)\).
3: if \(s \in M^i_0\) then
4: \(c := c_2\)
5: else if \(s \in M^i_0\) or \(k(s) \in \Gamma\) then
6: \(c := c_3\)
7: \(M^i_k := M^i_{\cdot} \cup \text{states}(\Gamma)\)
8: else
9: let \(A_2 = A_2 \cup \{\text{ace,} \Gamma \vdash k(s), \{\vdash (s/x) \phi_1\}\}\}
10: \(A_2 = A_2 \cup \{\text{ace,} \Gamma \vdash k(s), \{\Gamma \vdash \text{EG}_s(\phi_1)(s)\}\}\}
11: \forall i \in \{1,\ldots,n\}, let
12: \(A_2 = A_2 \cup \{\text{ace,} \Gamma \vdash k(s), \{\vdash (s/x) \phi_1, \Gamma \vdash k(s)\}\}\}
13: for all \(k(s) \in \Gamma\) do
14: \(M^i_k := M^i_{\cdot} \cup \{\cdot\}\)
15: \(c' := c'\)
16: \(c := c'\)
17: end if

FIGURE A.6. ProveEG(\(\Gamma \vdash \text{EG}_s(\phi_1)(s)\))

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

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 \(\{s_1,\ldots,s_n\} = \text{Next}(s)\), \(\Gamma' = \Gamma \cup \{\text{EU}_s(\phi_1,\phi_2)(\cdot)\}\), and use reachable(S) to denote the set of visited states that can reach some state in S, where S represents the states 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 \(\text{EU}_s(\phi_1,\phi_2)(s)\) holds, then just rewrite \(c\) to \(c_1\). At the same time, all states in reachable(states(\(\Gamma\))) are added into \(M^i_{\text{EU}_s(\phi_1,\phi_2)(\cdot)}\) and removed from \(M^i_{\text{EU}_s(\phi_1,\phi_2)(\cdot)}\).
- Line 7 – 9: If \(s\) has been visited, then just rewrite \(c\) to \(c_2\). At the same time, all states in states(\(\Gamma\)) are added into \(M^i_{\text{EU}_s(\phi_1,\phi_2)(\cdot)}\) 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 \(\Gamma \vdash \text{EU}_s(\phi_1,\phi_2)(s)\) (Line 12). At the same time, if \((s/y) \phi_2\) holds, then for each state \(s'\) that can reach \(s\), \(\text{EU}_s(\phi_1,\phi_2)(s')\) also holds, so we add reachable(states(\(\Gamma\) \cup \{\cdot\}) into \(M^i_{\text{EU}_s(\phi_1,\phi_2)(\cdot)}\) and remove all states in reachable(states(\(\Gamma\) \cup \{\cdot\}) from \(M^i_{\text{EU}_s(\phi_1,\phi_2)(\cdot)}\)) (Line 13 – 14).
2. If \(c'\) rewrites to the \(i\)th other \(c_1\), then both

FIGURE A.7. ProveAF(\(\Gamma \vdash \text{AF}_s(\phi_1)(s)\))
1. For notation purpose, here we refer "k" to AR_\kappa_\phi_1_\phi_2(s) and "k(s)" to AR_\kappa_\phi_1_\phi_2(s).
2. "k(s)" to AR_\kappa_\phi_1_\phi_2(s).
3. if s ∈ M_k then
   4. c := c_k
   5. M_k := M_k \cup \text{reachable}(\text{states}(\Gamma))
   6. end if

1. "k" to AR_\kappa_\phi_1_\phi_2(s) and "k(s)" to AR_\kappa_\phi_1_\phi_2(s).
2. if s ∈ M_k then
   3. c := c_k
   4. M_k := M_k \cup \text{reachable}(\text{states}(\Gamma))
   5. end if

\text{if} \text{c}\text{'} \text{rewrites to the outer c}_2, \text{then both} \vdash (s/x)\phi_1 \text{and} \vdash (s/y)\phi_2 \text{should be added to the counterexample as the children of} \Gamma \vdash \text{EU}_x(y)(\phi_1, \phi_2)(s) \text{(Line 16). At the same time, all states in states(\Gamma) \cup \{s\} should be added into} M_k^\phi_1_\phi_2(s) \text{to avoid being visited again (Line 17).}

4. If \text{c} \text{'} \text{rewrites to the inner c}_2, then \Gamma^\prime \vdash \text{EU}_x(y)(\phi_1, \phi_2)(s_1), ..., \Gamma^\prime \vdash \text{EU}_x(y)(\phi_1, \phi_2)(s_n) \text{should all be added to the counterexample as the children of} \Gamma \vdash \text{EU}_x(y)(\phi_1, \phi_2)(s) \text{(Line 18).}

- Line 28: Rewrite \text{c} to \text{c} \text{'}.

**Figure A.9. ProveAR(\Gamma \vdash AR_\kappa_\phi_1_\phi_2(s))**

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 other four 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.

![TABLE B.1. Solvable cases in five tools](image)

<table>
<thead>
<tr>
<th>Programs</th>
<th>SCTLProV</th>
<th>Verds</th>
<th>NuSMV</th>
<th>NuXMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>100%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S2</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S3</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S4</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S5</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Programs</th>
<th>SCTLProV</th>
<th>Verds</th>
<th>NuSMV</th>
<th>NuXMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>100%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S2</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S3</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S4</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
<tr>
<td>S5</td>
<td>100.0%</td>
<td>99.0%</td>
<td>98.0%</td>
<td>99.2%</td>
</tr>
</tbody>
</table>
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.

### Table B.3. Solvable cases in five tools

<table>
<thead>
<tr>
<th>Programs</th>
<th>iProver Modulo</th>
<th>NuXMV</th>
<th>NuSMV</th>
<th>SCTLProV</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 9 \times 40 )</td>
<td>396/406 (98.5%)</td>
<td>113/115 (99.1%)</td>
<td>113/115 (99.1%)</td>
<td>486/502 (97.4%)</td>
</tr>
<tr>
<td>( 9 \times 60 )</td>
<td>354/356 (100%)</td>
<td>110/110 (100%)</td>
<td>110/110 (100%)</td>
<td>396/402 (98.5%)</td>
</tr>
<tr>
<td>( 9 \times 72 )</td>
<td>282/284 (99.3%)</td>
<td>92/93 (98.9%)</td>
<td>92/93 (98.9%)</td>
<td>286/288 (99.3%)</td>
</tr>
<tr>
<td>( 9 \times 60 )</td>
<td>354/356 (100%)</td>
<td>110/110 (100%)</td>
<td>110/110 (100%)</td>
<td>396/402 (98.5%)</td>
</tr>
<tr>
<td>( 9 \times 72 )</td>
<td>282/284 (99.3%)</td>
<td>92/93 (98.9%)</td>
<td>92/93 (98.9%)</td>
<td>286/288 (99.3%)</td>
</tr>
<tr>
<td>( 9 \times 24 )</td>
<td>228/229 (99.6%)</td>
<td>448/449 (99.7%)</td>
<td>448/449 (99.7%)</td>
<td>228/229 (99.6%)</td>
</tr>
<tr>
<td>( 9 \times 32 )</td>
<td>188/189 (99.5%)</td>
<td>228/229 (99.6%)</td>
<td>228/229 (99.6%)</td>
<td>188/189 (99.5%)</td>
</tr>
<tr>
<td>( 9 \times 24 )</td>
<td>228/229 (99.6%)</td>
<td>448/449 (99.7%)</td>
<td>448/449 (99.7%)</td>
<td>228/229 (99.6%)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Programs</th>
<th>iProver Modulo</th>
<th>NuXMV</th>
<th>NuSMV</th>
<th>SCTLProV</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 9 \times 40 )</td>
<td>396/406 (98.5%)</td>
<td>113/115 (99.1%)</td>
<td>113/115 (99.1%)</td>
<td>486/502 (97.4%)</td>
</tr>
<tr>
<td>( 9 \times 60 )</td>
<td>354/356 (100%)</td>
<td>110/110 (100%)</td>
<td>110/110 (100%)</td>
<td>396/402 (98.5%)</td>
</tr>
<tr>
<td>( 9 \times 72 )</td>
<td>282/284 (99.3%)</td>
<td>92/93 (98.9%)</td>
<td>92/93 (98.9%)</td>
<td>286/288 (99.3%)</td>
</tr>
<tr>
<td>( 9 \times 60 )</td>
<td>354/356 (100%)</td>
<td>110/110 (100%)</td>
<td>110/110 (100%)</td>
<td>396/402 (98.5%)</td>
</tr>
<tr>
<td>( 9 \times 72 )</td>
<td>282/284 (99.3%)</td>
<td>92/93 (98.9%)</td>
<td>92/93 (98.9%)</td>
<td>286/288 (99.3%)</td>
</tr>
<tr>
<td>( 9 \times 24 )</td>
<td>228/229 (99.6%)</td>
<td>448/449 (99.7%)</td>
<td>448/449 (99.7%)</td>
<td>228/229 (99.6%)</td>
</tr>
<tr>
<td>( 9 \times 32 )</td>
<td>188/189 (99.5%)</td>
<td>228/229 (99.6%)</td>
<td>228/229 (99.6%)</td>
<td>188/189 (99.5%)</td>
</tr>
<tr>
<td>( 9 \times 24 )</td>
<td>228/229 (99.6%)</td>
<td>448/449 (99.7%)</td>
<td>448/449 (99.7%)</td>
<td>228/229 (99.6%)</td>
</tr>
</tbody>
</table>

### 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.

### 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 \( L = (s_0, S, Act, \rightarrow) \) 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. \( L \) is translated into a Kripke model \( M = (s_0', S', \rightarrow, P) \) by performing the following transformation, where \( S' = (S \cup \{s_0\}) \times (Act \cup \{\}) \), \( s_0' \in S' \), \( \rightarrow \subseteq S' \times S' \), and \( P = \{P, Q\} \).

- Let \( s_0' \) be \((s_0, \cdot)\), where \( \cdot \notin Act \) is a special action symbol;  
- Add \((s_d, \cdot) \) to \( S' \) and \((s_d, \cdot) \) to the transition relation of \( M \), 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 \( M \):
  - For all \( a \in Act \), if \( s_1 \xrightarrow{a} s_2 \) is in \( L \), then add a transition \((s_1, b) \xrightarrow{} (s_2, a) \) to \( M \) for all \( (s_1, b) \in S' \) where \( b \in Act \cup \{\} \), and add state \((s_d, a) \) into \( S' \) for all state \((s, a) \in S' \), if \( s \) has no successor in \( L \), then add a
transitions \((s, a) \rightarrow (s_{d'})\) 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, we add a special transition \((s, a) \rightarrow (s_{d'}, \cdot)\) to the transformed Kripke model; and to make the transition relation \(\rightarrow\) total, we add another special transition \((s_{d'}, \cdot) \rightarrow (s_{d'}, \cdot)\) 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 \(L\) is a state which has no successor. When detecting deadlocks of an LTS \(L\) in SCTLProV, we first translate \(L\) into a Kripke model \(\mathcal{M}\) using the above transformation. It can be observed that a deadlock state is reachable from \(s_0\) in \(L\) if and only if \((s_{d'}, \cdot)\) is reachable from \((s_0, \cdot)\) in \(\mathcal{M}\).

Thus, we can detect deadlocks in an LTS \(L\) by performing the proof search of the formula

\[
EF_s(P(x))(s_{d'}, \cdot)
\]

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

We perform the deadlock detection both in SCTLProV and CDP on this benchmark. For 40 test cases in this benchmark, both SCTLProV and CDP can solve all test cases. Moreover, SCTLProV uses less time than CDP in 33 (82.5\%) test cases, and uses less memory than CDP 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 \(\tau\) 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 \(L\) and the Kripke model \(\mathcal{M}\) by performing the above transformation on \(L\), a livelock exists in \(L\) 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 \(\tau\) transitions in \(L\), then the cycle has the form \(s_0 \overset{\tau}{\rightarrow} s_{p+1} \overset{\tau}{\rightarrow} \cdots \overset{\tau}{\rightarrow} s_n \overset{\tau}{\rightarrow} s_{p}\), where \(s_p = s_0\), or there exists a path \(s_0 \overset{a_0}{\rightarrow} \cdots \overset{a_{p-1}}{\rightarrow} s_p\). Then according to the transformation, there exists a cycle of the form \((s_p, a) \rightarrow (s_{p+1}, \tau) \rightarrow \cdots \rightarrow (s_n, \tau) \rightarrow (s_p, \tau) \rightarrow (s_{p+1}, \tau)\) in \(\mathcal{M}\), where \((s_0, \cdot) \rightarrow^* (s_p, a)\), and \(a = a_{p-1}\).

**TABLE B.6.** Deadlock detection in SCTLProV and CDP

<table>
<thead>
<tr>
<th>Name</th>
<th>Deadlocks</th>
<th>SCTLProV</th>
<th>CDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>vasy_2_1</td>
<td>No</td>
<td>0.13</td>
<td>26.48</td>
</tr>
<tr>
<td>vasy_2_2</td>
<td>No</td>
<td>0.13</td>
<td>27.71</td>
</tr>
<tr>
<td>vasy_2_4</td>
<td>No</td>
<td>0.14</td>
<td>27.83</td>
</tr>
<tr>
<td>vasy_2_14</td>
<td>Yes</td>
<td>0.29</td>
<td>25.79</td>
</tr>
<tr>
<td>vasy_2_9</td>
<td>No</td>
<td>0.14</td>
<td>25.70</td>
</tr>
<tr>
<td>vasy_2_24</td>
<td>No</td>
<td>0.17</td>
<td>28.90</td>
</tr>
<tr>
<td>vasy_2_38</td>
<td>Yes</td>
<td>0.14</td>
<td>25.76</td>
</tr>
<tr>
<td>vasy_2_56</td>
<td>No</td>
<td>0.22</td>
<td>29.78</td>
</tr>
<tr>
<td>vasy_2_73</td>
<td>No</td>
<td>0.24</td>
<td>31.68</td>
</tr>
<tr>
<td>vasy_2_25</td>
<td>Yes</td>
<td>0.97</td>
<td>33.52</td>
</tr>
<tr>
<td>vasy_2_60</td>
<td>No</td>
<td>0.21</td>
<td>29.42</td>
</tr>
<tr>
<td>vasy_2_318</td>
<td>No</td>
<td>0.39</td>
<td>41.09</td>
</tr>
<tr>
<td>vasy_2_263</td>
<td>No</td>
<td>1.41</td>
<td>77.02</td>
</tr>
<tr>
<td>vasy_2_802</td>
<td>No</td>
<td>0.89</td>
<td>24.92</td>
</tr>
<tr>
<td>vasy_2_220</td>
<td>No</td>
<td>0.23</td>
<td>27.97</td>
</tr>
<tr>
<td>vasy_2_325</td>
<td>No</td>
<td>0.21</td>
<td>27.96</td>
</tr>
<tr>
<td>vasy_2_368</td>
<td>No</td>
<td>0.67</td>
<td>32.27</td>
</tr>
<tr>
<td>vasy_2_325</td>
<td>No</td>
<td>0.28</td>
<td>28.33</td>
</tr>
<tr>
<td>vasy_2_297</td>
<td>No</td>
<td>0.18</td>
<td>27.14</td>
</tr>
<tr>
<td>vasy_2_167</td>
<td>No</td>
<td>2.53</td>
<td>48.39</td>
</tr>
<tr>
<td>vasy_2_651</td>
<td>No</td>
<td>0.29</td>
<td>31.19</td>
</tr>
<tr>
<td>vasy_2_684</td>
<td>No</td>
<td>0.39</td>
<td>34.63</td>
</tr>
<tr>
<td>vasy_2_641</td>
<td>No</td>
<td>1.36</td>
<td>40.41</td>
</tr>
<tr>
<td>vasy_2_171</td>
<td>No</td>
<td>2.14</td>
<td>74.11</td>
</tr>
<tr>
<td>vasy_2_998</td>
<td>No</td>
<td>0.38</td>
<td>37.98</td>
</tr>
<tr>
<td>vasy_2_136</td>
<td>No</td>
<td>21.56</td>
<td>272.11</td>
</tr>
<tr>
<td>vasy_2_390</td>
<td>No</td>
<td>0.23</td>
<td>28.49</td>
</tr>
<tr>
<td>vasy_2_390</td>
<td>No</td>
<td>10.2</td>
<td>89.81</td>
</tr>
<tr>
<td>vasy_2_373</td>
<td>No</td>
<td>10.31</td>
<td>166.14</td>
</tr>
<tr>
<td>vasy_2_766</td>
<td>No</td>
<td>3.19</td>
<td>87.61</td>
</tr>
<tr>
<td>vasy_2_1442</td>
<td>No</td>
<td>2.90</td>
<td>74.11</td>
</tr>
<tr>
<td>vasy_2_1944</td>
<td>No</td>
<td>2.85</td>
<td>89.50</td>
</tr>
<tr>
<td>vasy_2_1966</td>
<td>No</td>
<td>3.41</td>
<td>96.21</td>
</tr>
<tr>
<td>vasy_2_1953</td>
<td>No</td>
<td>29.41</td>
<td>413.40</td>
</tr>
<tr>
<td>vasy_2_1031</td>
<td>No</td>
<td>2.35</td>
<td>82.57</td>
</tr>
<tr>
<td>vasy_2_1901</td>
<td>No</td>
<td>14.92</td>
<td>108.67</td>
</tr>
<tr>
<td>vasy_2_2453</td>
<td>No</td>
<td>7.85</td>
<td>809.74</td>
</tr>
<tr>
<td>vasy_2_2466</td>
<td>No</td>
<td>17.48</td>
<td>149.80</td>
</tr>
<tr>
<td>vasy_2_2467</td>
<td>No</td>
<td>5.40</td>
<td>164.73</td>
</tr>
<tr>
<td>vasy_2_25318</td>
<td>No</td>
<td>296.36</td>
<td>2159.58</td>
</tr>
</tbody>
</table>

**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
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 \( \phi \) is closed, and the sequent \( \vdash \phi \) has a proof \( \pi \) in \( \text{SCTL}^\text{M} \), then \( \models \pi \) for the given Kripke model \( \mathcal{M} \).

**Proof.** By induction on the structure of the proof \( \pi \).

- If the last rule of \( \pi \) is \( \text{atom-R} \), then the proved sequent has the form \( \vdash P(s_1, \ldots, s_n) \), hence \( \mathcal{M} \models P(s_1, \ldots, s_n) \).
- If the last rule of \( \pi \) is \( \neg\text{-R} \), then the proved sequent has the form \( \vdash \neg P(s_1, \ldots, s_n) \), hence \( \mathcal{M} \models \neg P(s_1, \ldots, s_n) \).
- If the last rule of \( \pi \) is \( \top\text{-R} \), the proved sequent has the form \( \vdash \top \) and hence \( \mathcal{M} \models \top \).
- If the last rule of \( \pi \) is \( \land\text{-R} \), then the proved sequent has the form \( \vdash \phi_1 \land \phi_2 \). By induction hypothesis \( \mathcal{M} \models \phi_1 \) and \( \mathcal{M} \models \phi_2 \), hence \( \mathcal{M} \models \phi_1 \land \phi_2 \).
- If the last rule of \( \pi \) is \( \lor\text{-R}_1 \) or \( \lor\text{-R}_2 \), then the proved sequent has the form \( \vdash \phi_1 \lor \phi_2 \). By induction hypothesis \( \mathcal{M} \models \phi_1 \) or \( \mathcal{M} \models \phi_2 \), hence \( \mathcal{M} \models \phi_1 \lor \phi_2 \).
- If the last rule of \( \pi \) is \( \text{AX-R} \), then the proved sequent has the form \( \vdash AX_i(\phi_i)(s) \). By induction hypothesis, for each \( s' \) in \( \text{Next}(s) \), \( \mathcal{M} \models (s'/x)\phi_i \), hence \( \mathcal{M} \models AX_i(\phi_i)(s) \).
- If the last rule of \( \pi \) is \( \text{EX-R} \), then the proved sequent has the form \( \vdash EX_i(\phi_i)(s) \). By induction hypothesis, for some \( s' \) in \( \text{Next}(s) \), \( \mathcal{M} \models (s'/x)\phi_i \), hence \( \mathcal{M} \models EX_i(\phi_i)(s) \).
- If the last rule of \( \pi \) is \( \text{AF-R}_1 \) or \( \text{AF-R}_2 \), then the proved sequent has the form \( \vdash AF_i(\phi_i)(s) \). We associate a finite path-tree \( [\pi] \) to the proof \( \pi \) by induction in the following way.

  - If the proof \( \pi \) ends with the \( \text{AF-R}_1 \) rule with a sub-proof \( \rho \) of the sequent \( \vdash (s'/x)\phi_i \), then the path-tree contains a single node \( s \).
  - If the proof \( \pi \) ends with the \( \text{AF-R}_2 \) rule, with sub-proofs \( \pi_1, \ldots, \pi_n \) of the sequent \( \vdash AF_i(\phi_i)(s_1), \ldots, AF_i(\phi_i)(s_n) \), respectively, then \( [\pi] \) is the path-tree \( s(\pi_1, \ldots, \pi_n) \).

The path-tree \( [\pi] \) has root \( s \), and for each leaf \( s' \) of \( [\pi] \), the sequent \( \vdash (s'/x)\phi_i \) has a proof smaller than \( \pi \). By induction hypothesis, for each leaf \( s' \) of \( [\pi] \), \( \mathcal{M} \models (s'/x)\phi_i \). Hence \( \mathcal{M} \models AF_i(\phi_i)(s) \).

- If the last rule of \( \pi \) is \( \text{EG-R} \), then the proved sequent has the form \( \vdash EG_i(\phi_i)(s') \). We associate a finite sequence \( [\pi] \) to the proof \( \pi \) by induction in the following way.

  - If the proof \( \pi \) ends with the \( \text{EG-merge} \) rule, then the sequence contains a single element \( s \).
  - If the proof \( \pi \) ends with the \( \text{EG-R} \) rule, with sub-proofs \( \rho \) and \( \pi_1 \) of the sequents \( \vdash (s'/x)\phi_1 \) and \( \Gamma, EG_i(\phi_i)(s) \), respectively, then \( [\pi] \) is the sequence \( s|\pi_1| \).

The sequent \( [\pi] = s_0, s_1, \ldots, s_n \) is such that \( s_0 = s \); for all \( i \) between \( 0 \) and \( n-1 \), \( s_i \rightarrow s_{i+1} \); for all \( i \) between \( 0 \) and \( n \), the sequent \( \vdash (s_i/x)\phi_1 \) has a proof smaller than \( \pi \), and \( s_n \) is equal to \( s' \), 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 \( s'_0, s'_1, \ldots, \) such that for all \( i \), we have \( s'_i \rightarrow s'_{i+1} \), and \( \mathcal{M} \models (s'/x)\phi_1 \). Hence, \( \mathcal{M} \models EG_i(\phi_i)(s) \).

- If the last rule of \( \pi \) is \( \text{AR-R}_1 \) or \( \text{AR-R}_2 \), then the proved sequent has the form \( \vdash AR_i(\phi_i)(s_1) \). We associate a finite path-tree \( [\pi] \) to the proof \( \pi \) by induction in the following way.

  - If the proof \( \pi \) ends with the \( \text{AR-R}_3 \) rule with sub-proofs \( \rho_1 \) and \( \rho_2 \) of the sequents \( \vdash (s'/x)\phi_2 \) and \( \vdash (s'/x)\phi_2 \), respectively, or with the \( \text{AR-merge} \) rule, then the path-tree contains a single node \( s \).
  - If the proof \( \pi \) ends with the \( \text{AR-R}_2 \) rule, with sub-proofs \( \rho, \pi_1, \ldots, \pi_n \) of the sequents \( \vdash (s'/x)\phi_2 \), \( \Gamma, AR_i(\phi_1)(s_1), \ldots, AR_i(\phi_1)(s_1) \), respectively, then \( [\pi] \) is the path-tree \( s(\pi_1, \ldots, [\pi_n]) \).

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 \( \pi \), and for each leaf \( s' \), either the sequent \( \vdash (s'/x)\phi_2 \) has a proof smaller than \( \pi \), 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 $M \models (s'/y)\varphi_2$ and for each leaf $s'$, either $M \models (s'/x)\varphi_1$ 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 $T'$, $M \models (s'/y)\varphi_2$, and for each leaf $s'$ of $T'$, $M \models (s'/x)\varphi_1$. Thus, $M \models AR_{s}(s_1, \varphi_2)(s)$.

- If the last rule of $\pi$ is $EU-R_1$ or $EU-R_2$, then the proved sequent has the form $\vdash \mathit{EU}_{\mathit{xy}}(\varphi_1, \varphi_2)(s)$. We associate a finite sequence $|\pi|$ to the proof $\pi$ by induction in the following way.

- If the proof $\pi$ ends with the $EU-R_1$ rule with a sub-proof $\rho$ of the sequent $\vdash (s'/y)\varphi_2$, then the sub-sequent contains a single element $s$.

- If the proof $\pi$ ends with the $EU-R_2$ rule, with sub-proofs $\rho$ and $\varphi_1$ of the sequents $\vdash (s'/x)\varphi_1$ and $\vdash \mathit{EU}_{\mathit{xy}}(\varphi_1, \varphi_2)(s')$, respectively, then $|\pi|$ is the sequence $s|\pi|$.

The sequent $|\pi| = s_0, \ldots, s_n$ is such that $s_0 = s$; for each $i$ between 0 and $n - 1$, $s_i \rightarrow s_{i+1}$; for each $i$ between 0 and $n - 1$, the sequent $\vdash (s_i'/x)\varphi_1$ has a proof smaller than $\pi$; and the sequent $\vdash (s_n'/y)\varphi_2$ has a proof smaller than $\pi$. By induction hypothesis, for each $i$ between 0 and $n - 1$, $M \models (s_i'/x)\varphi_1$ and $M \models (s_n'/y)\varphi_2$. Hence, $M \models \mathit{EU}_{\mathit{xy}}(\varphi_1, \varphi_2)(s)$.

- The last rule cannot be a merge rule.

\[ \square \]

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

**Proof.** By induction over the size of $\phi$.

- If $\phi = P(s_1, \ldots, s_n)$, then as $M \models P(s_1, \ldots, s_n)$, the sequent $\vdash P(s_1, \ldots, s_n)$ is provable with the rule $\mathit{atom-R}$.

- If $\phi = \neg P(s_1, \ldots, s_n)$, then as $M \models \neg P(s_1, \ldots, s_n)$, the sequent $\vdash \neg P(s_1, \ldots, s_n)$ 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 $M \models \bot$.

- If $\phi = \varphi_1 \wedge \varphi_2$, then as $M \models \varphi_1 \wedge \varphi_2$, $M \models \varphi_1$ and $M \models \varphi_2$. By induction hypothesis, the sequents $\vdash \varphi_1$ and $\vdash \varphi_2$ are provable. Thus the sequent $\vdash \varphi_1 \wedge \varphi_2$ is provable with the $\wedge$-R rule.

- If $\phi = \varphi_1 \vee \varphi_2$, as $M \models \varphi_1 \vee \varphi_2$, $M \models \varphi_1$ or $M \models \varphi_2$. By induction hypothesis, the sequent $\vdash \varphi_1$ or $\vdash \varphi_2$ is provable and the sequent $\vdash \varphi_1 \vee \varphi_2$ is provable with the $\lor$-R$_1$ or $\lor$-R$_2$ rule, respectively.

- If $\phi = AX_{s}(\varphi_1)(s)$, as $M \models AX_{s}(\varphi_1)(s)$, for each state $s'$ in $\mathit{Next}(s)$, we have $M \models (s'/x)\varphi_1$. By induction hypothesis, for each $s'$ in $\mathit{Next}(s)$, the sequent $\vdash (s'/x)\varphi_1$ is provable. Using these proofs and the $\mathit{AX}$-R rule, we build a proof of the sequent $\vdash AX_{s}(\varphi_1)(s)$.

- If $\phi = EX_{s}(\varphi_1)(s)$, as $M \models EX_{s}(\varphi_1)(s)$, there exists a state $s'$ in $\mathit{Next}(s)$ such that $M \models (s'/x)\varphi_1$. By induction hypothesis, the sequent $\vdash (s'/x)\varphi_1$ is provable. With this proof and the $\mathit{EX}$-R rule, we build a proof of the sequent $\vdash EX_{s}(\varphi_1)(s)$.

- If $\phi = AF_{s}(\varphi_1)(s)$, as $M \models AF_{s}(\varphi_1)(s)$, 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 $\mathit{Next}(s')$, and for each leaf $s'$, $M \models (s'/x)\varphi_1$. By induction hypothesis, for every leaf $s'$, the sequent $\vdash (s'/x)\varphi_1$ is provable. Then, to each subtree $T'$ of $T$, we associate a proof $[T']$ of the sequent $\vdash AF_{s}(\varphi_1)(s')$ 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 $\mathit{AF}$-R$_1$ rule from the proof of $\vdash (a/x)\varphi_1$ given by the induction hypothesis.

- If $T' = s'(T_1, \ldots, T_n)$, then the proof $[T]$ is built with the $\mathit{AF}$-R$_2$ rule from the proofs $[T_1, \ldots, T_n]$ of the sequents $\vdash AF_{s}(\varphi_1)(s_1), \ldots \vdash AF_{s}(\varphi_1)(s_n)$, respectively, where $s_1, \ldots, s_n$ are the elements of $\mathit{Next}(s')$.

This way, the proof $[T]$ is a proof of the sequent $\vdash AF_{s}(\varphi_1)(s)$.

- If $\phi = EG_{s}(\varphi_1)(s)$, as $M \models EG_{s}(\varphi_1)(s)$, there exists a path $s_0, s_1, \ldots$ such that $s_0 = s$ and for all $i$, $M \models (s_i/x)\varphi_1$. By induction hypothesis, all the sequents $\vdash (s_i/x)\varphi_1$ are provable. Using Proposition 3.3, there exists a finite sequence $T = s_0, \ldots, s_n$ such that for all $i$, $s_i \rightarrow s_{i+1}$, the sequent $\vdash (s_i/x)\varphi_1$ is provable and $s_0$ is some $s_p$ for $p < n$. We associate a proof $[s_0, \ldots, s_n]$ of the sequent $EG_{s}(\varphi_1)(s_0), \ldots EG_{s}(\varphi_1)(s_{p-1}) \vdash EG_{s}(\varphi_1)(s_p)$ to each suffix of $T$ by induction as follows.

- The proof $[s_n]$ is built with the $\mathit{EG}$-merge rule.

- If $i \leq n - 1$, then the proof $[s_0, \ldots, s_n]$ is built with the $\mathit{EG}$-R rule from the proof of $\vdash (s_i/x)\varphi_1$ given by the induction hypothesis and the proof $[s_0, \ldots, s_{n-1}]$ of the sequent $EG_{s}(\varphi_1)(s_0), \ldots EG_{s}(\varphi_1)(s_{i-1}) \vdash EG_{s}(\varphi_1)(s_i)$ to each suffix of $T$ by induction as follows.

This way, the proof $[s_0, \ldots, s_n]$ is a proof of the sequent $\vdash EG_{s}(\varphi_1)(s)$.

- If $\phi = AR_{s}(\varphi_1, \varphi_2)(s)$, as $M \models AR_{s}(\varphi_1, \varphi_2)(s)$, there exists a possibly infinite path-tree such that the root of this tree is $s$, for each node $s'$, $M \models (s'/y)\varphi_2$ and for each leaf $s'$, $M \models (s'/x)\varphi_1$. By induction hypothesis, for each node $s'$ of the path-tree, the sequent $\vdash (s'/y)\varphi_2$ is provable and for each leaf $s'$ of the path-tree, the sequent $\vdash (s'/x)\varphi_1$ is provable. Using Proposition 3.4, there exists a finite path-tree $T$ such that for each node $s'$ of $T$, the sequent $\vdash (s'/y)\varphi_2$ is provable, and for each leaf $s'$, either the sequent $\vdash (s'/x)\varphi_1$ 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 $AR_{s}(\varphi_1, \varphi_2)(s_1), \ldots AR_{s}(\varphi_1, \varphi_2)(s_m) \vdash AR_{s}(\varphi_1, \varphi_2)(s')$ where $s'$ is the root of $T'$ and $s_1, \ldots, s_m$ is the sequence of nodes in $T$ from the root of $T$ to the root of $T'$. 

\[ \square \]
APPENDIX D. PROOF OF THE CORRECTNESS OF THE PROOF SEARCH METHOD

PROPOSITION D.1. Given a closed formula φ, \( \text{cpt}(\Gamma \vdash \phi, t, f) \rightsquigarrow t \) if \( \Gamma \vdash \phi \) is provable.

Proof. The proof is split into two cases: given a closed formula \( \phi \),

\[
\Rightarrow: \text{cpt}(\Gamma \vdash \phi, t, f) \rightarrow t \text{ implies that } \Gamma \vdash \phi \text{ is provable;}
\]

\[
\Leftarrow: \Gamma \vdash \phi \text{ implies that } \text{cpt}(\Gamma \vdash \phi, t, f) \rightarrow^* t.
\]

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 \( \phi \) is a closed formula, \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) rewrites to \( c_1 \) before it rewrites to \( c_2 \). Suppose \( T \) 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 \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) where \( c_1 \neq c_2 \), and \( \phi \) is a closed formula, if \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) rewrites to \( c_1 \) before it rewrites to \( c_2 \), then \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) 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 \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) to \( c_1 \), neither \( c_1 \) nor \( c_2 \) is examined until \( c_1 \) is rewritten. Thus, we can safely replace \( c_1 \) by \( c_1' \), and \( c_2 \) by \( c_2' \) in the rewriting steps from \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) to \( c_1 \), and obtain the rewriting steps from \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) to \( c_1' \). Analogously, if \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) rewrites to \( c_2 \) before it rewrites to \( c_1 \), then \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) rewrites to \( c_1' \) 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.

\[ \Gamma \vdash \phi \Rightarrow T, \text{ trivial.} \]

\[ \Gamma \vdash \phi \Rightarrow \neg P(s_1, \ldots, s_n) \text{ where } P(s_1, \ldots, s_n) \text{ is an atomic formula, then the only way that } \text{cpt}(\neg P(s_1, \ldots, s_n), c_1, c_2) \text{ rewrites to } c_1 \text{ before it rewrites to } c_2 \text{ is that } \text{cpt}(\neg P(s_1, \ldots, s_n), c_1, c_2) \Rightarrow c_1, \text{ in which case } \{s_1, \ldots, s_n\} \notin P. \text{ Then, it is sufficient to take the following proof tree as } T. \]

\[ \neg P(s_1, \ldots, s_n) \text{ atom-R} \]

\[ \neg P(s_1, \ldots, s_n) \text{ \neg-R} \]

\[ \Gamma \vdash \phi \Rightarrow \neg P(s_1, \ldots, s_n) \text{ where } P(s_1, \ldots, s_n) \text{ is an atomic formula, then the only way that } \text{cpt}(\neg P(s_1, \ldots, s_n), c_1, c_2) \text{ rewrites to } c_1 \text{ before it rewrites to } c_2 \text{ is that } \text{cpt}(\neg P(s_1, \ldots, s_n), c_1, c_2) \Rightarrow c_1, \text{ in which case } \{s_1, \ldots, s_n\} \notin P. \text{ Then, it is sufficient to take the following proof tree as } T. \]

\[ \Gamma \vdash \phi \Rightarrow \phi_1 \text{ \land-R} \]

\[ \Gamma \vdash \phi \Rightarrow \phi_1 \text{ \lor-R} \]

\[ \Gamma \vdash \phi \Rightarrow \phi_1 \text{ \land-R} \]

\[ \Gamma \vdash \phi \Rightarrow \phi_1 \text{ \lor-R} \]
then it is sufficient to take the following proof tree as $T$.

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

2. $\text{cpt}(\vdash \phi_1 \lor \phi_2, c_1, c_2) \rightsquigarrow \text{cpt}(\vdash \phi_1, c_1, \text{cpt}(\vdash \phi_2, c_1, c_2))$, $\text{cpt}(\vdash \phi_1, c_1, \text{cpt}(\vdash \phi_2, c_1, c_2))$ rewrites to $\text{cpt}(\vdash \phi_2, c_1, c_2)$ before it rewrites to $c_1$, and $\text{cpt}(\vdash \phi_2, c_1, c_2)$ 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 $T$.

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

- If $\Gamma \vdash \phi = \vdash \text{AX}_j(\psi_1)(s)$, and $\{s_1, \ldots, s_n\} = \text{Next}(s)$, then $\text{cpt}(\vdash \text{AX}_j(\psi_1)(s), c_1, c_2)$ rewrites to $c_1$ before it rewrites to $c_2$ implies that there exists $s_i \in \text{Next}(s)$ such that $\vdash (s_i/x)\psi_1$ is provable. This is because otherwise, if for all $s_j \in \text{Next}(s)$, $\vdash (s_j/x)\psi_1$ is not provable, then according to the induction hypothesis, $\text{cpt}(\vdash \text{AX}_j(\psi_1)(s), c_1, c_2) \rightsquigarrow \text{cpt}(\vdash (s_1/x)\psi_1, c_1, \text{cpt}(\vdash (s_0/x)\psi_1, c_1, c_2, \ldots)) \rightsquigarrow^* (s/n)\psi_1, c_1, c_2)$, and $c_1$ has not been rewritten in the above rewriting steps. Suppose $T'$ is the proof tree constructed for $\vdash (s/x)\psi_1$, then it is sufficient to take the following proof tree as $T$.

$$
\frac{T'}{\vdash \text{AX}_j(\psi_1)(s)} \quad \text{EX-R}
$$

- If $\Gamma \vdash \phi = \vdash \text{EG}_i(\psi_1)(s)$, then $\text{cpt}(\vdash \text{EG}_i(\psi_1)(s), c_1, c_2)$ rewrites to $c_1$ before it rewrites to $c_2$ implies that one of the following two cases holds.

1. $\text{EG}_i(\psi_1)(s) \in \Gamma$;
2. $\text{EG}_i(\psi_1)(s) \not\in \Gamma$, and

\[
\text{cpt}(\vdash (s/x)\psi_1, \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_1)), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_2)), c_1, c_2, \ldots)) \rightsquigarrow^* \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_1), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_2)), c_1, c_2, \ldots)) \rightsquigarrow^* \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_1), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EG}_i(\psi_1)(s_2)), c_1, c_2, \ldots)) \ rightsquigarrow^* c_1, \text{ where } \Gamma = \Gamma \cup \{\text{EG}_i(\psi_1)(s)\} \text{ and } 1 \leq i \leq n.
\]

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

$$
\frac{\Gamma \vdash \text{EG}_i(\psi_1)(s)}{\vdash \text{EG}_i(\psi_1)(s)} \quad \text{EG-merge}
$$

If case 2 holds, then by induction hypothesis, both $\vdash (s/x)\psi_1$ and $\Gamma \vdash \text{EG}_i(\psi_1)(s_i)$ are provable. Suppose $T_1$ and $T_2$ are the constructed proof trees of $\vdash (s/x)\psi_1$ and $\Gamma \vdash \text{EG}_i(\psi_1)(s_i)$, respectively, then it is sufficient to take the following proof tree as $T$.

$$
\frac{T_1 \quad T_2}{\vdash \text{EG}_i(\psi_1)(s)} \quad \text{EG-R}
$$

- If $\Gamma \vdash \phi = \vdash \text{AF}_j(\psi_1)(s)$, as the dual case of $\text{EG}$, the analysis for the $\text{AF}$ case is analogous to that of the $\text{EG}$ case.
- If $\Gamma \vdash \phi = \Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s)$, then $\text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s), c_1, c_2)$ rewrites to $c_1$ before it rewrites to $c_2$ implies that one of the following two cases holds.

1. $\text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s), c_1, c_2) \rightsquigarrow^* \text{cpt}(\vdash (s/y)\psi_2, c_1, \text{cpt}(\vdash (s/x)\psi_1, \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_1), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_2)), c_1, c_2, \ldots)), c_2)) \rightsquigarrow^* c_1$;
2. $\text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s), c_1, c_2) \rightsquigarrow^* \text{cpt}(\vdash (s/y)\psi_2, c_1, \text{cpt}(\vdash (s/x)\psi_1, \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_1), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_2)), c_1, c_2, \ldots)), c_2)) \rightsquigarrow^* \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_1), c_1, \text{cpt}(\ldots \text{cpt}(\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s_2)), c_1, c_2, \ldots)) \rightsquigarrow^* c_1$ where $1 \leq i \leq n$.

If case 1 holds, then by induction hypothesis, $\vdash (s/y)\psi_2$ is provable. Suppose $T'$ is the constructed proof tree of $\vdash (s/y)\psi_2$, then it is sufficient to take the following proof tree as $T$.

$$
\frac{T'}{\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s)} \quad \text{EU-R}_1
$$

If case 2 holds, then by induction hypothesis, both $\vdash (s/x)\psi_1$ and $\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s)$ are provable. Let $T_1$ and $T_2$ be the constructed proof trees of $\vdash (s/x)\psi_1$ and $\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s)$, respectively, and it is sufficient to take the following proof tree as $T$.

$$
\frac{T_1 \quad T_2}{\Gamma \vdash \text{EU}_{x,y}(\psi_1, \psi_2)(s)} \quad \text{EU-R}_2
$$

- If $\Gamma \vdash \phi = \Gamma \vdash \text{AR}_{x,y}(\psi_1, \psi_2)(s)$, as the dual case of $\text{EU}$, the analysis for the $\text{AR}$ case is analogous to that of the $\text{EU}$ case.

For the “$\vdash$” case, we also prove a generalized proposition: for two different CPTs $c_1$ and $c_2$, and a closed formula $\phi$, if $\phi$ is an inductive formula, then $\vdash \phi$ is provable implies that $\text{cpt}(\vdash \phi, c_1, c_2)$ rewrites to $c_1$ before it rewrites to $c_2$.

---

26In the rules of the proof system, the context of an $\text{EU}$ sequent is always empty. However, to make the proof easier to follow, the contexts of $\text{EU}$ sequents are kept. The contexts of $\text{EU}$ 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 $\text{EU}$ case is explained in Section 4.1.1.
• if \( \phi \) is not an inductive formula, then \( \Gamma \vdash \phi \) is provable implies that \( \text{cpt}(\Gamma \vdash \phi, c_1, c_2) \) 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 \( \phi \) is an inductive formula).

• If the last rule in the proof tree is atom-\( R \), then suppose \( \phi = P(s_1, \ldots, s_n) \) where \( P(s_1, \ldots, s_n) \) is an atomic formula, and \( \vdash P(s_1, \ldots, s_n) \) is provable, then \( \langle s_1, \ldots, s_n \rangle \in P \), then \( \text{cpt}(\vdash P(s_1, \ldots, s_n), c_1, c_2) \) \( \rightsquigarrow \) \( c_1 \).

• If the last rule in the proof tree is \( \neg-\text{R} \), then suppose \( \phi = \neg P(s_1, \ldots, s_n) \) where \( P(s_1, \ldots, s_n) \) is an atomic formula, and \( \neg \vdash P(s_1, \ldots, s_n) \) is provable, then \( \langle s_1, \ldots, s_n \rangle \notin P \), then \( \text{cpt}(\neg \vdash P(s_1, \ldots, s_n), c_1, c_2) \) \( \rightsquigarrow \) \( c_1 \).

• If the last rule in the proof tree is \( \land-\text{R} \), then suppose \( \phi = \phi_1 \land \phi_2 \) 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, \( \text{cpt}(\vdash \phi_1 \land \phi_2, c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash \phi_1, c_1, c_2), \text{cpt}(\vdash \phi_2, c_1, c_2) \) \( \rightsquigarrow^{*} \) \( c_1 \), and \( c_2 \) has not been rewritten in these rewriting steps.

• If the last rule in the proof tree is \( \lor-\text{R} \), then suppose \( \phi = \phi_1 \lor \phi_2 \) 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, \( \text{cpt}(\vdash \phi_1 \lor \phi_2, c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash \phi_1, c_1, c_2), \text{cpt}(\vdash \phi_2, c_1, c_2) \) \( \rightsquigarrow^{*} \) \( c_1 \), 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 \( \text{cpt}(\vdash \phi_1 \lor \phi_2, c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash \phi_1, c_1, c_2), \text{cpt}(\vdash \phi_2, c_1, c_2) \) \( \rightsquigarrow^{*} \) \( c_1 \), or \( \text{cpt}(\vdash \phi_1 \lor \phi_2, c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash \phi_1, c_1, c_2), \text{cpt}(\vdash \phi_2, c_1, c_2) \) \( \rightsquigarrow^{*} \) \( c_1 \), and \( c_2 \) has not been rewritten in these rewriting steps.

• If the last rule in the proof tree is \( \neg \land-\text{R} \), then suppose \( \phi = EX_i(\phi_1)(s), \{s_1, \ldots, s_n\} = \text{Next}(s) \), and \( \vdash \phi \) is provable, then according to the rules of the proof system, there exists \( s_i \in \text{Next}(s) \) such that \( \vdash (s_i / x) \phi_1 \) is provable, and for all \( j \) such that \( 1 \leq j < i \), \( \vdash (s_j / x) \phi_1 \) is not provable. Then, according to the induction hypothesis, either \( \text{cpt}(\vdash EX_i(\phi_1)(s), c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash (s_1 / x) \phi_1, c_1, c_2), \text{cpt}(\vdash \ldots \text{cpt}(\vdash (s_n / x) \phi_1, c_1, c_2)) \) \( \rightsquigarrow^{*} \) \( c_1 \) and \( c_2 \) has not been rewritten in these rewriting steps.

1. if \( \vdash AF_i(\phi_1)(s) \) labeled the root of a sub-proof of \( \vdash AF_i(\phi_1)(s) \) for some \( 1 \leq i \leq n \), then by induction hypothesis, \( \text{cpt}(\vdash AF_i(\phi_1)(s), c_1, c_2) \) rewrites to \( c_1 \) before it rewrites to \( c_2 \), then \( \text{cpt}(\vdash AF_i(\phi_1)(s) \rightarrow AF_i(\phi_1)(s), c_1, c_2) \) must rewrite to \( c_1 \) before it rewrites to \( c_2 \). This is because otherwise, if \( \text{cpt}(\vdash AF_i(\phi_1)(s) \rightarrow AF_i(\phi_1)(s), c_1, c_2) \) rewrites to \( c_2 \) before it rewrites to \( c_1 \), then according to the rewrite rules, there exists an infinite path \( s, s_1, \ldots, s_n \) such that for each state \( s_i \) in this path, \( \vdash (s_i / x) \phi_1 \) is not provable, then according to the soundness and completeness of the proof system, \( \vdash AF_i(\phi_1)(s) \) cannot be provable. Thus, \( \text{cpt}(\vdash AF_i(\phi_1)(s), c_1, c_2) \) \( \rightsquigarrow \) \( \text{cpt}(\vdash (s_1 / x) \phi_1, c_1, c_2), \text{cpt}(\vdash AF_i(\phi_1)(s), c_1, c_2), \text{cpt}(\vdash AF_i(\phi_1)(s), c_1, c_2) \) \( \rightsquigarrow^{*} \) \( c_1 \), and \( c_2 \) has not been rewritten in these rewriting steps.
\( c_2 \) has not been rewritten in these rewriting steps.

- If the last rule in the proof tree is \( \text{EU-R}_3 \) or \( \text{EU-R}_2 \), then suppose \( \phi = EU_{x,y}(\phi_1, \phi_2)(s), \{s_1, ..., s_n\} = \text{Next}(s) \), and \( \phi \) 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 \( \vdash EU_{x,y}(\phi_1, \phi_2)(s) \) is provable for some \( 1 \leq i \leq n \). In the former case, by induction hypothesis, \( \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow \text{cpt}(\vdash (s/y)\phi_2, c_1, c_2) \Rightarrow c_1 \) 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, \( \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \) 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 sequences. Then, there exists \( 1 \leq i \leq n \) such that \( \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s) \Rightarrow EU_{x,y}(\phi_1, \phi_2)(s_i), c_1, c_2) \) rewrites to \( c_1 \) before it rewrites to \( c_2 \). This is because otherwise, for all \( 1 \leq i \leq n \), \( \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s) \Rightarrow EU_{x,y}(\phi_1, \phi_2)(s_i), c_1, c_2) \) 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 \( s, s_1, ..., s \) where \( \vdash (s'/y)\phi_2 \) is not provable for each state \( s' \) of the path, or of the form \( s, s_1, ..., s' \) 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
  \[
  \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s) \Rightarrow EU_{x,y}(\phi_1, \phi_2)(s_i), c_1, c_2) \Rightarrow c_1 \Rightarrow \text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s) \Rightarrow EU_{x,y}(\phi_1, \phi_2)(s_i), c_1, c_2) \Rightarrow c_2.
  \]

- If the last rule in the proof tree is \( \text{AR-R}_1 \), \( \text{AR-R}_2 \), or \( \text{AR-merge} \), then suppose \( \phi = AR_{x,y}(\phi_1, \phi_2)(s), \{s_1, ..., s_n\} \in \text{Next}(s) \), 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, \( \text{cpt}(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1 \).
  2. Both \( \vdash (s/x)\phi_1 \) and \( \vdash (s/y)\phi_2 \) are provable. In this case, according to the induction hypothesis,
     \[
     \text{cpt}(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1 \]
     \[
     \text{cpt}(\vdash (s/y)\phi_2, \text{cpt}(\vdash (s/y)\phi_1, c_1), \text{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1, c_2 \text{ has not been rewritten in these rewriting steps})
     \]
     \[
     \text{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1 \text{, and } c_2 \text{ has not been rewritten in these rewriting steps}
     \]
  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 \( s_i \in \text{Next}(s) \). In this case, according to the induction hypothesis,
     \[
     \text{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1 \]
     \[
     \text{cpt}(\vdash (s/y)\phi_2, \text{cpt}(\vdash (s/y)\phi_1, c_1), \text{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1)
     \]
     \[
     \text{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \Rightarrow c_1 \text{, and } c_2 \text{ has not been rewritten in these rewriting steps}
     \]

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