Optimal and Cut-Free Tableaux for Propositional Dynamic Logic with Converse

Rajeev Goré[1] and Florian Widmann[1,2]

[1]Logic and Computation Group
[2,⋆] NICTA The Australian National University
Canberra, ACT 0200, Australia,
{Rajeev.Gore,Florian.Widmann}@anu.edu.au

Abstract. We give an optimal (EXPTIME), sound and complete tableau-based algorithm for deciding satisfiability for propositional dynamic logic with converse (CPDL) which does not require the use of analytic cut. Our main contribution is a sound method to combine our previous optimal method for tracking least fix-points in PDL with our previous optimal method for handling converse in the description logic $ALCI$. The extension is non-trivial as the two methods cannot be combined naively. We give sufficient details to enable an implementation by others. Our OCaml implementation seems to be the first theorem prover for CPDL.

1 Introduction

Propositional dynamic logic (PDL) is an important logic for reasoning about programs. Its formulae consist of traditional Boolean formulae plus "action modalities" built from a finite set of atomic programs using sequential composition (;), non-deterministic choice (∪), repetition (∗), and test (?). The logic CPDL is obtained by adding converse ($^-$), which allows us to reason about previous actions. The satisfiability problem for CPDL is EXPTIME-complete [1].

De Giacomo and Massacci [2] give an NEXPTIME tableau algorithm for deciding CPDL-satisfiability, and discuss ways to obtain optimality, but do not give an actual EXPTIME algorithm. The tableau method of Nguyen and Szałas [3] is optimal. Neither method has been implemented, and since both require an explicit analytic cut rule, it is not at all obvious that they can be implemented efficiently. Optimal game-theoretic methods for fix-point logics [4] can be adapted to handle CPDL [5] but involve significant non-determinism. Optimal automata-based methods [6] for fix-point logics are still in their infancy because good optimisations are not known. We know of no resolution methods for CPDL.

We give an optimal tableau method for deciding CPDL-satisfiability which does not rely on a cut rule. Our main contribution is a sound method to combine our method for tracking and detecting unfulfilled eventualities as early as possible

⋆ NICTA is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

J. Giesl and R. Hähnle (Eds.): IJCAR 2010, LNAI 6173, pp. 225–239, 2010.

Table 1. Smullyan's α- and β-notation to classify formulae

α	$\varphi \wedge \psi$	$[\gamma \cup \delta]\varphi$	$[\gamma*]\varphi$	$\langle\psi?\rangle\varphi$	$\langle\gamma;\delta\rangle\varphi$	$[\gamma;\delta]\varphi$
α_1	φ	$[\gamma]\varphi$	φ	φ	$\langle\gamma\rangle\langle\delta\rangle\varphi$	$[\gamma][\delta]\varphi$
α_2	ψ	$[\delta]\varphi$	$[\gamma][\gamma*]\varphi$	ψ		

β	$\varphi \vee \psi$	$\langle\gamma \cup \delta\rangle\varphi$	$\langle\gamma*\rangle\varphi$	$[\psi?]\varphi$
β_1	φ	$\langle\gamma\rangle\varphi$	φ	φ
β_2	ψ	$\langle\delta\rangle\varphi$	$\langle\gamma\rangle\langle\gamma*\rangle\varphi$	$\sim\psi$

in PDL [7] with our method for handling converse for *ALCI* [8]. The extension is non-trivial as the two methods cannot be combined naively.

We present a mixture of pseudo code and tableau rules rather than a set of traditional tableau rules to enable easy implementation by others. Our unoptimised OCaml implementation appears to be the first automated theorem prover for CPDL (`http://rsise.anu.edu.au/~rpg/CPDLTabProver/`). A longer version with full proofs is available at `http://arxiv.org/abs/1002.0172`.

2 Syntactic Preliminaries

Definition 1. *Let* AFml *and* APrg *be two disjoint and countably infinite sets of propositional variables and* atomic programs*, respectively. The set* LPrg *of* literal programs *is defined as* $\text{LPrg} := \text{APrg} \cup \{a^- \mid a \in \text{APrg}\}$*. The set* Fml *of all formulae and the set* Prg *of all* programs *are defined mutually inductively as follows where* $p \in \text{AFml}$ *and* $l \in \text{LPrg}$*:*

$$\begin{array}{ll} \text{Fml} & \varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\gamma\rangle\varphi \mid [\gamma]\varphi \\ \text{Prg} & \gamma ::= l \mid \gamma;\gamma \mid \gamma \cup \gamma \mid \gamma* \mid \varphi? \ . \end{array}$$

A $\langle\text{lp}\rangle$*-formula is a formula* $\langle\gamma\rangle\varphi$ *where* $\gamma \in \text{LPrg}$ *is a literal program.*

Implication ($\rightarrow$) and equivalence ($\leftrightarrow$) are not part of the core language but can be defined as usual. In the rest of the paper, let $p \in \text{AFml}$ and $l \in \text{LPrg}$.

We omit the semantics as it is a straightforward extension of PDL [7] and write $M, w \Vdash \varphi$ if $\varphi \in \text{Fml}$ holds in the world $w \in W$ of the model M.

Definition 2. *For a literal program* $l \in \text{LPrg}$*, we define* $l^{\smile}$ *as* a *if* l *is of the form* a^-*, and as* l^- *otherwise. A formula* $\varphi \in \text{Fml}$ *is in* negation normal form *if the symbol* $\neg$ *appears only directly before propositional variables. For every* $\varphi \in \text{Fml}$*, we can obtain a formula* $\text{nnf}(\varphi)$ *in negation normal form by pushing negations inward such that* $\varphi \leftrightarrow \text{nnf}\,\varphi$ *is valid. We define* $\sim\varphi := \text{nnf}(\neg\varphi)$*.*

We categorise formulae as α- or β-formulae as shown in Table 1 so that the formulae of the form $\alpha \leftrightarrow \alpha_1 \wedge \alpha_2$ and $\beta \leftrightarrow \beta_1 \vee \beta_2$ are valid. An *eventuality* is a formula of the form $\langle\gamma_1\rangle \dots \langle\gamma_k\rangle\langle\gamma*\rangle\varphi$, and Ev is the set of all eventualities. Using Table 1, the binary relation "$\rightsquigarrow$" relates a $\langle\rangle$-formulae α (respectively β), to its reduction α_1 (respectively β_1 and β_2). See [7, Def. 7] for their formal definitions.

3 An Overview of our Algorithm

Our algorithm builds an and-or graph G by repeatedly applying four rules (see Table 2) to try to build a model for a given ϕ in negation normal form. Each node x carries a formula set Γ_x, a status sts_x, and other fields to be described shortly. Rule 1 applies the usual expansion rules to a node to create its children. These expansion rules capture the semantics of CPDL. We use Smullyan's α/β-rule notation for classifying rules and nodes. As usual, a node x is a ("saturated") *state* if no α/β-rule can be applied to it. If x is a state then for each $\langle l \rangle \xi$ in Γ_x, we create a node y with $\Gamma_y = \{\xi\} \cup \Delta$, where $\Delta = \{\psi \mid [l]\psi \in \Gamma_x\}$, and add an edge from x to y labelled with $\langle l \rangle \xi$ to record that y is an l-successor of x.

If Γ_x contains an obvious contradiction during expansion, its status becomes "closed", which is irrevocable. Else, at some later stage, Rule 2 determines its status as either "closed" or "open". "Open" nodes contain additional information which depends on the status of other nodes. Hence, if a node changes its status, it might affect the status of another ("open") node. If the stored status of a node does not match its current status, the node is no longer *up-to-date*. Rule 3, which may be applied multiple times to the same node, ensures that "open" nodes are kept up-to-date by recomputing their status if necessary. Finally, Rule 4 detects eventualities which are impossible to fulfil and closes nodes which contain them. We first describe the various important components of our algorithm separately.

Global State Caching. For optimality, the graph G never contains two state nodes which carry the same set of formulae [8]. However, there may be multiple non-states which carry the same set of formulae. That is, a non-state node x carrying Γ which appears while saturating a child y of a state z is unique to y. If a node carrying Γ is required in some other saturation phase, a new node carrying Γ is created. Hence the nodes of two saturation phases are distinct.

Converse. Suppose state y is a descendant of an l-successor of a state x, with no intervening states. Call x the parent state of y since all intervening nodes are not states. We require that $\{\psi \mid [l^-]\psi \in \Gamma_y\} \subseteq \Gamma_x$, since y is then compatible with being a l-successor of x in the putative model under construction. If some $[l^-]\psi \in \Gamma_y$ has $\psi \notin \Gamma_x$ then x is "too small", and must be "restarted" as an alternative node x^+ containing all such ψ. If any such ψ is a complex formula to which an α/β-rule is applicable then x^+ is not a state and may have to be "saturated" further. The job of creating these alternatives is done by *special nodes* [8]. Each special node monitors a state and creates the alternatives when needed.

Detecting Fulfilled and Unfulfilled Eventualities. Suppose the current node x contains an eventuality e_x. There are three possibilities. The first is that e_x can be fulfilled in the part of the graph which is "older" than x. Else, it may be possible to reach a node z in the parts of the graph "newer" than x such that z contains a reduction e_z of e_x. Since this "newer" part of the graph is not fully explored yet, future expansions may enable us to fulfil e_x via z, so the pair (z, e_z) is a "potential rescuer" of e_x. The only remaining case is that e_x cannot be fulfilled in the "older" part of the graph, and has no potential rescuers. Thus future expansions of the graph cannot possibly help to fulfil e_x since it cannot reach these "newer" parts of the future graph. In this case x can be "closed". The technical machinery to maintain this information for PDL is from [7]. However, the presence of "converse" and the resulting need for alternative nodes requires a more elaborate scheme for CPDL.

4 The Algorithm

Our algorithm builds a directed graph G consisting of nodes and directed edges. We first explain the structure of G in more detail.

Definition 3. *Let X and Y be sets. We define $X^\perp := X \uplus \{\perp\}$ where $\perp$ indicates the undefined value and $\uplus$ is the disjoint union. If $f : X \to Y$ is a function and $x \in X$ and $y \in Y$ then the function $f[x \mapsto y] : X \to Y$ is defined as $f[x \mapsto y](x') := y$ if $x' = x$ and $f[x \mapsto y](x') := f(x')$ if $x' \neq x$.*

Definition 4. *Let $G = (V, E)$ be a graph where V is a set of nodes and E is a set of directed edges. Each node $x \in V$ has six attributes: $\Gamma_x \subseteq \mathrm{Fml}$, $\mathrm{ann}_x : \mathrm{Ev} \to \mathrm{Fml}^\perp$, $\mathrm{pst}_x \in V^\perp$, $\mathrm{ppr}_x \in \mathrm{LPrg}^\perp$, $\mathrm{idx}_x \in Nat^\perp$, and $\mathrm{sts}_x \in \mathfrak{S}$ where $\mathfrak{S} := \{\mathtt{unexp}, \mathtt{undef}\} \cup \{\mathtt{closed}(\mathrm{alt}) \mid \mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})\} \cup \{\mathtt{open}(\mathrm{prs}, \mathrm{alt}) \mid \mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^\perp \;\&\; \mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})\}$. Each directed edge $e \in E$ is labelled with a label $l_e \in (\mathrm{Fml} \cup \mathscr{P}(\mathrm{Fml}) \cup \{\mathrm{cs}\})^\perp$ where cs is just a constant.*

All attributes of a node $x \in V$ are initially set at the creation of x, possibly with the value $\perp$ (if allowed). Only the attributes idx_x and sts_x are changed at a later time. We use the function create-new-node(Γ, ann, pst, ppr, idx, sts) to create a new node and initialise its attributes in the obvious way.

The finite set Γ_x contains the formulae which are assigned to x. The attribute ann_x is defined for the eventualities in Γ_x at most. If $\mathrm{ann}_x(\varphi) = \varphi'$ then $\varphi' \in \Gamma_x$ and $\varphi \rightsquigarrow \varphi'$. The intuitive meaning is that φ has already been "reduced" to φ' in x. For a state (as defined below) we always have that ann_x is undefined everywhere since we do not need the attribute for states.

The node x is called a *state* iff both attributes pst_x and ppr_x are undefined. For all other nodes, the attribute pst_x identifies the, as we will ensure, unique ancestor $p \in V$ of x such that p is a state and there is no other state between p and x in G. We call p the *parent state* of x. The creation of the child of p which lies on the path from p to x (it could be x) was caused by a $\langle \mathrm{lp} \rangle$-formula $\langle l \rangle \varphi$ in Γ_p. The literal program l which we call the *parent program* of x is stored in ppr_x. Hence, for nodes which are not states, both pst_x and ppr_x are defined.

The attribute sts_x describes the *status* of x. Unlike the attributes described so far, its value may be modified several times. The value `unexp`, which is the initial value of each node, indicates that the node has not yet been expanded. When a node is expanded, its status becomes either `closed`$(\cdot)$ if it contains an immediate contradiction, or `undef` to indicate that the node has been expanded but that its "real" status is to be determined. Eventually, the status of each node is set to either `closed`$(\cdot)$ or `open`$(\cdot,\cdot)$. If the status is `open`$(\cdot,\cdot)$, it might be modified several times later on, either to `closed`$(\cdot)$ or to `open`$(\cdot,\cdot)$ (with different arguments), but once it becomes `closed`$(\cdot)$, it will never change again.

We call a node *undefined* if its status is `unexp` or `undef` and *defined* otherwise. Hence a node is undefined initially, becomes defined eventually, and then never becomes undefined again. Furthermore, we call x *closed* iff its status is $\texttt{closed}(\mathrm{alt})$ for some alt $\subseteq \mathscr{P}(\mathrm{Fml})$. In this case, we define $\mathrm{alt}_x := \mathrm{alt}$. We call x *open* iff its status is $\texttt{open}(\mathrm{prs}, \mathrm{alt})$ for some $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^\perp$ and some alt $\subseteq \mathscr{P}(\mathrm{Fml})$. In this case, we define $\mathrm{prs}_x := \mathrm{prs}$ and $\mathrm{alt}_x := \mathrm{alt}$. To avoid some clumsy case distinctions, we define $\mathrm{alt}_x := \emptyset$ if x is undefined.

The value $\texttt{closed}(\mathrm{alt})$ indicates that the node is "useless" for building an interpretation because it is either unsatisfiable or "too small". In the latter case, the set alt of *alternative sets* contains information about missing formulae. Finally, the value $\texttt{open}(\mathrm{prs}, \mathrm{alt})$ indicates that there is still hope that x is "useful" and the function prs_x contains information about each eventuality $e_x \in \Gamma_x$ as explained in the overview. Although x itself may be useful, we need its alternative sets in case it becomes closed later on. Hence it also has a set of alternative sets.

The attribute idx_x serves as a time stamp. It is set to $\perp$ at creation time of x and becomes defined when x becomes defined. When this happens, the value of idx_x is set such that $\mathrm{idx}_x > \mathrm{idx}_y$ for all nodes y which became defined earlier than x. We define $y \sqsubset x$ iff $\mathrm{idx}_y \neq \perp$ and either $\mathrm{idx}_x = \perp$ or $\mathrm{idx}_y < \mathrm{idx}_x$. Note that $y \sqsubset x$ depends on the current state of the graph. However, once $y \sqsubset x$ holds, it will do so for the rest of the time.

To track eventualities, we label an edge between a state and one of its children by the $\langle\mathrm{lp}\rangle$-formula $\langle l \rangle\varphi$ which creates this child. Additionally, we label edges from special nodes (see overview) to their <u>c</u>orresponding <u>s</u>tates with the marker cs. We also label edges from special nodes to its alternative nodes with the corresponding alternative set.

Definition 5. *Let* $\mathrm{ann}^\perp : \mathrm{Ev} \to \mathrm{Fml}^\perp$ *and* $\mathrm{prs}^\perp : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^\perp$ *be the functions which are undefined everywhere. For a node* $x \in V$ *and a label* $l \in \mathrm{Fml} \cup \mathscr{P}(\mathrm{Fml}) \cup \{\mathrm{cs}\}$*, let* $\mathrm{getChild}(x, l)$ *be the node* $y \in V$ *such that there exists an edge* $e \in E$ *from* x *to* y *with* $l_e = l$*. If* y *does not exists or is not unique, let the result be* $\perp$*. For a function* $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^\perp$*, a node* $x \in V$*, and an eventuality* $\varphi \in \mathrm{Ev}$*, we define the set* $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ *of eventualities as follows:*

$$\mathrm{reach}(\mathrm{prs}, x, \varphi) := \Big\{\psi \in \mathrm{Ev} \mid \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \dots, \varphi_k \in \mathrm{Ev}.\, \Big(\psi = \varphi_k \; \& \; (x, \varphi_0) \in \mathrm{prs}(\varphi) \; \& \; \forall i \in \{0, \dots, k-1\}.\, (x, \varphi_{i+1}) \in \mathrm{prs}(\varphi_i)\Big)\Big\} \; .$$

The function $\mathrm{defer} : V \times \mathrm{Ev} \to \mathrm{Fml}^\perp$ *is defined as follows:*

$$\mathrm{defer}(x, \varphi) := \begin{cases} \psi & \textit{if } \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \dots, \varphi_k \in \mathrm{Fml}.\, \Big(\varphi_0 = \varphi \; \& \; \varphi_k = \psi \; \& \\ & \quad \forall i \in \{0, \dots, k-1\}.\, \big(\varphi_i \in \mathrm{Ev} \; \& \; \mathrm{ann}_x(\varphi_i) = \varphi_{i+1}\big) \; \& \\ & \quad \big(\varphi_k \notin \mathrm{Ev} \textit{ or } \mathrm{ann}_x(\varphi_k) = \perp\big)\Big) \\ \perp & \textit{otherwise.} \end{cases}$$

The function $\mathrm{getChild}(x, l)$ retrieves a particular child of x. It is easy to see that, during the algorithm, the child is always unique if it exists.

Intuitively, the function $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ computes all eventualities which can be "reached" from φ inside x according to prs. If a potential rescuer (x, ψ) is

Procedure `is-sat`(ϕ) for testing whether a formula ϕ is satisfiable

Input: a formula $\phi \in \text{Fml}$ in negation normal form
Output: true iff ϕ is satisfiable

$G :=$ a new empty graph; idx := 1
let $d \in \text{APrg}$ be a dummy atomic program which does not occur in ϕ
rt := create-new-node($\{\langle d \rangle \phi\}, \text{ann}^{\perp}, \perp, \perp, \perp,$ `unexp`)
insert rt in G
while *one of the rules in Table 2 is applicable* **do**
  apply any one of the applicable rules in Table 2
if $\text{sts}_{\text{rt}} =$ `open`$(\cdot,\cdot)$ **then return** true **else return** false

Table 2. Rules used in the procedure `is-sat`

Rule 1:	Some node x has not been expanded yet.
Condition:	$\exists x \in V.\ \text{sts}_x =$ `unexp`
Action:	`expand`(x)
Rule 2:	The status of some node x is still undefined.
Condition:	$\exists x \in V.\ \text{sts}_x =$ `undef`
Action:	$\text{sts}_x :=$ `det-status`(x) & $\text{idx}_x := \text{idx}$ & $\text{idx} := \text{idx} + 1$
Rule 3:	Some open node x is not up-to-date.
Condition:	$\exists x \in V.$ `open`$(\cdot,\cdot) = \text{sts}_x \neq$ `det-status`(x)
Action:	$\text{sts}_x :=$ `det-status`(x)
Rule 4:	All nodes are up-to-date, and some x has an unfulfilled eventuality φ.
Condition:	Rule 3 is not applicable and
	$\exists x \in V.\ \text{sts}_x =$ `open`$(\text{prs}_x, \text{alt}_x)$ & $\exists \varphi \in \text{Ev} \cap \Gamma_x.\ \text{prs}_x(\varphi) = \emptyset$
Action:	$\text{sts}_x :=$ `closed`(alt_x)

contained in $\text{prs}(\varphi)$, the potential rescuers of ψ are somehow relevant for φ at x. Therefore ψ itself is relevant for φ at x. The function $\text{reach}(\text{prs}, x, \varphi)$ computes exactly the transitive closure of this relevance relation.

Intuitively, the function $\text{defer}(x, \varphi)$ follows the "ann_x-chain". That is, it computes $\varphi_1 := \text{ann}_x(\varphi)$, $\varphi_2 := \text{ann}_x(\varphi_1)$, and so on. There are two possible outcomes. The first outcome is that we eventually encounter a φ_k which is either not an eventuality or has $\text{ann}_x(\varphi_k) = \perp$. Consequently, we cannot follow the "ann_x-chain" any more. In this case we stop and return $\text{defer}(x, \varphi) := \varphi_k$. The second outcome is that we can follow the "ann_x-chain" indefinitely. Then, as Γ_x is finite, there must exist a cycle $\varphi_0, \dots, \varphi_n, \varphi_0$ of eventualities such that $\text{ann}_x(\varphi_i) = \varphi_{i+1}$ for all $0 \leq i < n$, and $\text{ann}_x(\varphi_n) = \varphi_0$. In this case we say that x (or Γ_x) contains an *"at a world" cycle* and return $\text{defer}(x, \varphi) := \perp$.

Next we comment on all procedures given in pseudocode.

Procedure `is-sat`(ϕ) is invoked to determine whether a formula $\phi \in \text{Fml}$ in negation normal form is satisfiable. It creates a root node rt and initialises the graph G to contain only rt. The dummy program d is used to make rt a state so that each node in G which is not a state has a parent state. The global variable idx is used to set the time stamps of the nodes accordingly.

Procedure `expand`(x) for expanding a node x

```
Input: a node x ∈ V with sts_x = unexp

if ∃φ ∈ Γ_x. ~φ ∈ Γ_x or (φ ∈ Ev & defer(x, φ) = ⊥) then
    idx_x := idx;   idx := idx + 1;   sts_x := closed(∅)
else (* x does not contain a contradiction *)
    sts_x := undef
    if pst_x = ⊥ then (* x is a state *)
        let ⟨l_1⟩φ_1, ..., ⟨l_k⟩φ_k be all of the ⟨lp⟩-formulae in Γ_x
        for i ⟵ 1 to k do
            Γ_i := {φ_i} ∪ {ψ | [l_i]ψ ∈ Γ_x}
            y_i := create-new-node(Γ_i, ann^⊥, x, l_i, ⊥, unexp)
            insert y_i, and an edge from x to y_i labelled with ⟨l_i⟩φ_i, into G
    else if ∃α ∈ Γ_x. {α_1, ..., α_k} ⊈ Γ_x or (α ∈ Ev & ann_x(α) = ⊥) then
        Γ := Γ_x ∪ {α_1, ..., α_k}
        ann := if α ∈ Ev then ann_x[α ↦ α_1] else ann_x
        y := create-new-node(Γ, ann, pst_x, ppr_x, ⊥, unexp)
        insert y, and an edge from x to y, into G
    else if ∃β ∈ Γ_x. {β_1, β_2} ∩ Γ_x = ∅ or (β ∈ Ev & ann_x(β) = ⊥) then
        for i ⟵ 1 to 2 do
            Γ_i := Γ_x ∪ {β_i}
            ann_i := if β ∈ Ev then ann_x[β ↦ β_i] else ann_x
            y_i := create-new-node(Γ_i, ann_i, pst_x, ppr_x, ⊥, unexp)
            insert y_i, and an edge from x to y_i, into G
    else (* x is a special node *)
        if ∃y ∈ V. Γ_y = Γ_x & pst_y = ⊥ then (* state already exists in G *)
            insert an edge from x to y labelled with cs into G
        else (* state does not exist in G yet *)
            y := create-new-node(Γ_x, ann^⊥, ⊥, ⊥, ⊥, unexp)
            insert y, and an edge from x to y labelled with cs, into G
```

While at least one of the rules in Table 2 is applicable, that is its condition is true, the algorithm applies any applicable rule. If no rules are applicable, the algorithm returns satisfiable iff rt is open.

Rule 1 picks an unexpanded node and expands it. Rule 2 picks an expanded but undefined node and computes its (initial) status. It also sets the correct time stamp. Rule 3 picks an open node whose status has changed and recomputes its status. Its meaning is, that if we compute `det-status`(x) on the current graph then its result is different from the value in sts_x, and consequently, we update sts_x accordingly. Rule 4 is only applicable if all nodes are up-to-date. It picks an open node containing an eventuality φ which is currently not fulfilled in the graph and which does not have any potential rescuers either. As this indicates that φ can never be fulfilled, the node is closed.

This description leaves several questions open, most notably: "How do we check efficiently whether Rule 3 is applicable?" and "Which rule should be taken if several rules are applicable?". We address these issues in Section 5.

Procedure `det-status`(x) for determining the status of a node x

Input: a node $x \in V$ with $\texttt{unexp} \neq \text{sts}_x \neq \texttt{closed}(\cdot)$

if *x is an α-or a β-node* **then** $\text{sts}_x := \texttt{det-sts-}\beta(x)$
else if *x is a state* **then** $\text{sts}_x := \texttt{det-sts-state}(x)$
else (∗ x is a special node, in particular $\text{pst}_x \neq \bot \neq \text{ppr}_x$ ∗)
  $\varGamma_{\text{alt}} := \{\varphi \mid [\text{ppr}_x^{\smile}]\varphi \in \varGamma_x\} \setminus \varGamma_{\text{pst}_x}$
  if $\varGamma_{\text{alt}} = \emptyset$ **then** $\text{sts}_x := \texttt{det-sts-spl}(x)$ **else** $sts_x := \texttt{closed}(\{\varGamma_{\text{alt}}\})$

Procedure `expand`(x) expands a node x. If $\varGamma_x$ contains an immediate contradiction or an "at a world" cycle then we close x and set the time stamp accordingly. For the other cases, we assume implicitly that $\varGamma_x$ does not contain either of these.

If x is a state, that is $\text{pst}_x = \bot$, then we do the following for each $\langle\text{lp}\rangle$-formula $\langle l_i\rangle\varphi_i$. We create a new node y_i whose associated set contains φ_i and all ψ such that $[l_i]\psi \in \varGamma_x$. As none of the eventualities in $\varGamma_{y_i}$ is reduced yet, there are no annotations. The parent state of y_i is obviously x and its parent program is l_i. In order to relate y_i to $\langle l_i\rangle\varphi_i$, we label the edge from x to y_i with $\langle l_i\rangle\varphi_i$. We call y_i the *successor* of $\langle l_i\rangle\varphi_i$.

If x is not a state and $\varGamma_x$ contains an α-formula α whose decompositions are not in $\varGamma_x$, or which is an unannotated eventuality, we call x an *α-node*. In this case, we create a new node y whose associated set is the result of adding all decompositions of α to $\varGamma_x$. If α is an eventuality then ann_y extends ann_x by mapping α to α_1. The parent state and the parent program of y are inherited from x. Note that pst_x and ppr_x are defined as x is not a state. Also note that $\varGamma_y \supsetneq \varGamma_x$ or α is an eventuality which is annotated in ann_y but not in ann_x.

If x is neither a state nor an α-node and $\varGamma_x$ contains a β-formula β such that neither of its immediate subformulae is in $\varGamma_x$, or such that β is an unannotated eventuality, we call x a *β-node*. For each decomposition β_i we do the following. We create a new node y_i whose associated set is the result of adding β_i to $\varGamma_x$. If β is an eventuality then ann_{y_i} extends ann_x by mapping α to β_i. The parent state and the parent program of y are inherited from x. Note that pst_x and ppr_x are defined as x is not a state. Also note that $\varGamma_{y_i} \supsetneq \varGamma_x$ or β is an eventuality which is annotated in ann_{y_i} but not in ann_x.

If x is neither a state nor an α-node nor a β-node, it must be fully saturated and we call it a *special node*. Intuitively, a special node sits between a saturation phase and a state and is needed to handle the "special" issue arising from converse programs, as explained in the overview. Like α- and β-nodes, special nodes have a unique parent state and a unique parent program. In this case we check whether there already exists a state y in G which has the same set of formulae as the special node. If such a state y exists, we link x to y; else we create such a state and link x to it. In both cases we label the edge with the marker cs since a special node can have several children (see below) and we want to uniquely identify the cs-child y of x. Note that there is only at most one state for each set of formulae and that states are always fully saturated since special nodes are.

Procedure `det-sts-`$\beta(x)$ for determining the status of an α- or a β-node

Input: an α- or a β-node $x \in V$ with $\texttt{unexp} \neq \mathrm{sts}_x \neq \texttt{closed}(\cdot)$
Output: the new status of x

let $y_1, \ldots, y_k \in V$ be all children of x
alt $:= \bigcup_{i=1}^{k} \mathrm{alt}_{y_i}$
if $\forall i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} = \texttt{closed}(\cdot)$ **then return** $\texttt{closed}(\mathrm{alt})$
else (∗ at least one child is not closed ∗)
  prs $:= \mathrm{prs}^{\perp}$
  foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ **do**
    for $i \longleftarrow 1$ **to** k **do** $\Lambda_i := \texttt{det-prs-child}(x, y_i, \varphi)$
    $\Lambda :=$ **if** $\exists i \in \{1, \ldots, k\}.\ \Lambda_i = \perp$ **then** $\perp$ **else** $\bigcup_{i=1}^{k} \Lambda_i$
    prs $:= \mathrm{prs}[\varphi \mapsto \Lambda]$
  $\mathrm{prs}' := \texttt{filter}(x, \mathrm{prs})$
  return $\texttt{open}(\mathrm{prs}', \mathrm{alt})$

Procedure `det-status`(x) determines the current status of a node x. Its result will always be $\texttt{closed}(\cdot)$ or $\texttt{open}(\cdot,\cdot)$. If x is an α/β-node or a state, the procedure just calls the corresponding sub-procedure. If x is a special node, we determine the set Γ_{alt} of all formulae φ such that $[\mathrm{ppr}_x^{\smile}]\varphi$ is in Γ_x but φ is not in the set of the parent state of x. If there is no such formula, that is Γ_{alt} is the empty set, we say that x is *compatible* with its parent state pst_x. Note that incompatibilities can only arise because of converse programs.

If x is compatible with pst_x, all is well, so we determine its status via the corresponding sub-procedure. Else we cannot connect pst_x to a state with Γ_x assigned to it in the putative model as explained in the overview, and, thus, we can close x. That does not, however, mean that pst_x is unsatisfiable; maybe it is just missing some formulae. We cannot extend pst_x directly as this may have side-effects elsewhere; but to tell pst_x what went wrong, we remember Γ_{alt}. The meaning is that if we create an alternative node for pst_x by adding the formulae in Γ_{alt}, we might be more successful in building an interpretation.

Procedure `det-sts-`$\beta(x)$ computes the status of an α- or a β-node $x \in V$. For this task, an α-node can be seen as a β-node with exactly one child. The set of alternative sets of x is the union of the sets of alternative sets of all children. If all children of x are closed then x must also be closed. Otherwise we compute the set of potential rescuers for each eventuality φ in Γ_x as follows. For each child y_i of x we determine the potential rescuers of φ which result from following y_i by invoking `det-prs-child`. If the set of potential rescuers corresponding to some y_i is $\perp$ then φ can currently be fulfilled via y_i and $\mathrm{prs}_x(\varphi)$ is set to $\perp$. Else φ cannot currently be fulfilled in G, but each child returned a set of potential rescuers, and the set of potential rescuers for φ is their union. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`.

Procedure `det-sts-state`(x) computes the status of a state $x \in V$. We obtain the successors for all $\langle\mathrm{lp}\rangle$-formulae in Γ_x. If any successor is closed then x is closed with the same set of alternative sets. Else the set of alternative sets of x is the union of the sets of alternative sets of all children and we compute the potential

Procedure `det-sts-state`(x) for determining the status of a state

Input: a state $x \in V$ with $\texttt{unexp} \neq \mathrm{sts}_x \neq \texttt{closed}(\cdot)$
Output: the new status of x

let $\langle l_1\rangle\varphi_1, \dots, \langle l_k\rangle\varphi_k$ be all of the $\langle \mathrm{lp}\rangle$-formulae in Γ_x
for $i \longleftarrow 1$ **to** k **do** $y_i := \mathrm{getChild}(x, \langle l_i\rangle\varphi_i)$
if $\exists i \in \{1,\dots,k\}.\ \mathrm{sts}_{y_i} = \texttt{closed}(\mathrm{alt})$ **then return** $\texttt{closed}(\mathrm{alt})$
else (∗ no child is closed ∗)
  $\mathrm{alt} := \bigcup_{i=1}^{k} \mathrm{alt}_{y_i}$
  $\mathrm{prs} := \mathrm{prs}^{\perp}$
  for $i \longleftarrow 1$ **to** k **do**
    if $\varphi_i \in \mathrm{Ev}$ **then**
      $\Lambda := \texttt{det-prs-child}(x, y_i, \varphi_i)$
      $\mathrm{prs} := \mathrm{prs}[\langle l_i\rangle\varphi_i \mapsto \Lambda]$
  $\mathrm{prs}' := \texttt{filter}(x, \mathrm{prs})$
  return $\texttt{open}(\mathrm{prs}', \mathrm{alt})$

rescuers for each eventuality $\langle l_i\rangle\varphi_i$ in Γ_x by invoking `det-prs-child`. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`. Note that we do not consider eventualities which are not $\langle \mathrm{lp}\rangle$-formulae. The intuitive reason is that the potential rescuers of such eventualities are determined by following the annotation chain (see below). However, different special nodes which have the same set, and hence all link to x, might have different annotations. Hence we cannot (and do not need to) fix the potential rescuer sets for eventualities in x which are not $\langle \mathrm{lp}\rangle$-formulae.

Procedure `det-sts-spl`(x) computes the status of a special node $x \in V$. First, we retrieve the state y_0 corresponding to x, namely the unique cs-child of x. For all alternative sets Γ_i of y_0 we do the following. If there does not exist a child of x such that the corresponding edge is labelled with Γ_i, we create a new node y_i whose associated set is the result of adding the formulae in Γ_i to Γ_x. The annotations, the parent state, and the parent program of y_i are inherited from x. We label the new edge from x to y_i with Γ_i. In other words we unpack the information stored in the alternative sets in alt_{y_0} into actual nodes which are all children of x. Note that each $\Gamma_i \neq \emptyset$ by construction in `det-status`. Some children of x may not be referenced from alt_{y_0}, but we consider them anyway.

The set of alternative sets of x is the union of the sets of alternative sets of all children; with the exception of y_0 since the alternative sets of y_0 are not related to pst_x but affect x directly as we have seen. If all children of x are closed then x must also be closed. Otherwise we compute the set of potential rescuers for each eventuality φ in Γ_x as follows.

First, we determine $\varphi' := \mathrm{defer}(x, \varphi)$. Note that φ' is defined because the special node x cannot contain an "at a world" cycle by definition. If φ' is not an eventuality then φ' is fulfilled in x and $\mathrm{prs}(\varphi)$ remains $\perp$. If φ' is an eventuality, it must be a $\langle \mathrm{lp}\rangle$-formula as x is a special node. We use φ' instead of φ since only $\langle \mathrm{lp}\rangle$-formula have a meaningful interpretation in prs_{y_0} (see above). For each child y_i of x we determine the potential rescuers of φ' by invoking

Procedure `det-sts-spl`(x) for determining the status of a special node

Input: a special node $x \in V$ with $\texttt{unexp} \neq \text{sts}_x \neq \texttt{closed}(\cdot)$
Output: the new status of x

$y_0 := \text{getChild}(x, \text{cs})$
let $\Gamma_1, \dots, \Gamma_j$ be all the sets in the set alt_{y_0}
for $i \longleftarrow 1$ **to** j **do**
  $y_i := \text{getChild}(x, \Gamma_i)$
  if $y_i = \bot$ **then** (∗ child does not exist ∗)
    $y_i := \text{create-new-node}(\Gamma_x \cup \Gamma_i, \text{ann}_x, \text{pst}_x, \text{ppr}_x, \bot, \texttt{unexp})$
    insert y_i, and an edge from x to y_i labelled with Γ_i, into G

let $y_{j+1}, \dots, y_k$ be all the remaining children of x
$\text{alt} := \bigcup_{i=1}^{k} \text{alt}_{y_i}$
if $\forall i \in \{0, \dots, k\}.\ \text{sts}_{y_i} = \texttt{closed}(\cdot)$ **then return** $\texttt{closed}(\text{alt})$
else (∗ at least one child is not closed ∗)
  $\text{prs} := \text{prs}^{\bot}$
  foreach $\varphi \in \Gamma_x \cap \text{Ev}$ **do**
    $\varphi' := \text{defer}(x, \varphi)$
    if $\varphi' \in \text{Ev}$ **then**
      for $i \longleftarrow 0$ **to** k **do** $\Lambda_i := \texttt{det-prs-child}(x, y_i, \varphi')$
      $\Lambda :=$ **if** $\exists i \in \{0, \dots, k\}.\ \Lambda_i = \bot$ **then** $\bot$ **else** $\bigcup_{i=0}^{k} \Lambda_i$
      $\text{prs} := \text{prs}[\varphi \mapsto \Lambda]$

  $\text{prs}' := \texttt{filter}(x, \text{prs})$
  return $\texttt{open}(\text{prs}', \text{alt})$

`det-prs-child`. If the set of potential rescuers corresponding to some y_i is $\bot$ then φ' can currently be fulfilled via y_i and so $\text{prs}_x(\varphi)$ is set to $\bot$. Otherwise φ' cannot currently be fulfilled in G, but each child returned a set of potential rescuers, and the set of potential rescuers for φ is their union. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \text{Ev}$ by calling `filter`.

Procedure `det-prs-child`(x, y, φ) determines whether an eventuality $\psi \in \Gamma_x$, which is not passed as an argument, can be fulfilled via y such that φ is part of the corresponding fulfilling path; or else which potential rescuers ψ can reach via y and φ. If y is closed, it cannot help to fulfil ψ as indicated by the empty set. If y is undefined or did not become defined before x then (y, φ) itself is a potential rescuer of x. Else, if φ can be fulfilled, i.e. $\text{prs}_y(\varphi) = \bot$, then ψ can be fulfilled too, so we return $\bot$. Otherwise we invoke the procedure recursively on all potential rescuers in $\text{prs}_y(\varphi)$. If at least one of these invocations returns $\bot$ then ψ can be fulfilled via y and φ and the corresponding rescuer in $\text{prs}_y(\varphi)$. If all invocations return a set of potential rescuers, the set of potential rescuers for ψ is their union. The recursion is well-defined because if $(z_i, \varphi_i) \in \text{prs}_y(\varphi)$ then either z_i is still undefined or z_i became defined later than y.

Each invocation of `det-prs-child` can be uniquely assigned to the invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl` which (possibly indirectly) invoked it. To meet our complexity bound, we require that under the same invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl`, the procedure `det-prs-child` is only executed at most once for each argument triple. Instead of executing it a second time with the same arguments, it uses the cached result of the first invocation. Since `det-prs-child` does not modify the graph, the second invocation would return the same result as the first one. An easy implementation of the cache is to store the result of `det-prs-child`(x, y, φ) in the node y together with φ and a unique id number for each invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl`.

Procedure `filter`(x, prs) deals with the potential rescuers for each eventuality of a node x which are of the form (x, ψ) for some $\psi \in \text{Ev}$. The second argument of `filter` is a provisional prs for x. If an eventuality $\varphi \in \Gamma_x$ is currently fulfillable in G there is nothing to be done, so let $(x, \psi) \in \text{prs}(\varphi)$. If $\psi = \varphi$ then (x, φ) cannot be a potential rescuer for φ in x and should not appear in $\text{prs}(\varphi)$. But what about potential rescuers of the form (x, ψ) with $\psi \neq \varphi$? Since we want the nodes in the potential rescuers to become defined later than x, we cannot keep (x, ψ) in $\text{prs}(\varphi)$; but we cannot just ignore the pair either.

Intuitively $(x, \psi) \in \text{prs}(\varphi)$ means that $\varphi \in \Gamma_x$ can "reach" $\psi \in \Gamma_x$ by following a loop in G which starts at x and returns to x itself. Thus if ψ can be fulfilled in G, so can φ; and all potential rescuers of ψ are also potential rescuers of φ. The function $\text{reach}(\text{prs}, x, \varphi)$ computes all eventualities in x which are "reachable" from φ in the sense above, where transitivity is taken into account. That is, it detects all self-loops from x to itself which are relevant for fulfilling φ. We add φ as it is not in $\text{reach}(\text{prs}, x, \varphi)$. If any of these eventualities is fulfilled in G then φ can be fulfilled and is consequently undefined in the resulting prs'. Otherwise we take all their potential rescuers whose nodes are not x.

Theorem 6 (Soundness, Completeness and Complexity). *Let $\phi \in \text{Fml}$ be a formula in negation normal form of size n. The procedure `is-sat`(ϕ) terminates, runs in* EXPTIME *in n, and ϕ is satisfiable iff `is-sat`(ϕ) returns true.*

5 Implementation, Optimisations, and Strategy

It should be fairly straightforward to implement our algorithm. It remains to show an efficient way to find nodes which are not up-to-date. It is not too hard to see that the status of a node x can become outdated only if its children change their status or `det-prs-child`$(x, y, \cdot)$ was invoked when x's previous status was determined and y now changes its status. If we keep track of nodes of the second kind by inserting additional "update"-edges as described in [7], we can use a queue for all nodes that might need updating. When the status of a node is modified, we queue all parents and all nodes linked by "update"-edges.

We have omitted several refinements from our description for clarity. The most important is that if a state s is closed, all non-states which have s as a parent state are ignorable since their status cannot influence any other node t unless t also has s as a parent state. Moreover, if every special node parent x of a state s' is incompatible or itself has a closed parent state, then s' and the nodes having s' as parent state are ignorable. This applies transitively, but if s' gets a new parent whose parent state is not closed then s' becomes "active" again.

Another issue is which rule to choose if several are applicable. As we have seen, it is advantageous to close nodes as early as possible. Apart from immediate contradictions, we have Rule 4 which closes a node because it contains an unfulfillable eventuality. If we can apply Rule 4 early while the graph is still small, we might prevent big parts of the graph being built needlessly later. Trying to apply Rule 4 has several consequences on the strategy of how to apply rules.

First, it is important to keep all nodes up-to-date since Rule 4 is not applicable otherwise. Second, it is preferable that a node x cannot reach open nodes which became defined (or will be defined) after x did. Hence, we should try to use Rule 2 on a node only if all children are already defined.

6 An Example

To demonstrate how the algorithm works, we invoke it on the satisfiable toy formula $\langle a\rangle\phi$ where $\phi := \langle a*\rangle[a^-]p$. To save space, Fig. 1 only shows the core subgraph of the tableau. Remember that the order of rule applications is not fixed but the example will follow some of the guidelines given in Section 5.

The nodes in Fig. 1 are numbered in order of creation. The annotation ann is given using "$\rightsquigarrow$" in Γ. For example, in node (3), we have $\Gamma_3 = \{\ \phi,\ [a^-]p\ \}$, and ann_3 maps the eventuality ϕ to $[a^-]p$ and is undefined elsewhere. The bottom line of a node contains the parent state and the parent program on the left, and the time stamp on the right. We do not show the status of a node since it changes during the algorithm, but explain it in the text. If we write $\mathrm{sts}_x = \mathtt{open}(\Lambda, \cdot)$ where $\Lambda \subseteq V \times \mathrm{Ev}$, we mean that prs_x maps all eventualities in Γ_x, with the exception of non-$\langle\mathrm{lp}\rangle$-formulae if x is a state, to Λ and is undefined elsewhere.

We only consider the core subgraph of ϕ and start by expanding node (1) which creates (2). Then we expand (2) and create (3) and (4) which are both special nodes. Next we expand (3) and create the state (5). Expanding (5) creates no new nodes since Γ_5 contains no $\langle\mathrm{lp}\rangle$-formula. Now we define (5) and then (3). This results in setting $\mathrm{sts}_5 := \mathtt{open}(\mathrm{prs}^\perp, \emptyset)$ according to `det-sts-state`, and $\mathrm{sts}_3 := \mathtt{closed}(\{p\})$ since (3) is not compatible with its parent state (1). Expanding (4) inserts the edge from (4) to (1) and defining (4) sets $\mathrm{sts}_4 := \mathtt{open}(\{(1, \langle a\rangle\phi)\}, \emptyset)$ according to `det-sts-spl`. Note that (6) does not exist yet. Next we define (2) and then (1) which results in setting $\mathrm{sts}_2 := \mathtt{open}(\{(1, \langle a\rangle\phi)\}, \{p\})$ according to `det-sts-`β and $\mathrm{sts}_1 := \mathtt{open}(\emptyset, \{p\})$ thanks to `filter`.

Note that $\langle a\rangle\phi \in \Gamma_1$ has an empty set of potential rescuers. In PDL, we could thus close (1), but converse programs complicate matters for CPDL as reflected by the fact that Rule 4 is not applicable for (1) because (4) is not up-to-date. Updating (4) creates (6) and sets $\mathrm{sts}_4 := \mathtt{open}(\{(1, \langle a\rangle\phi), (6, \langle a\rangle\phi)\}, \emptyset)$. Updating (2) and then (1) sets $\mathrm{sts}_2 := \mathtt{open}(\{(1, \langle a\rangle\phi), (6, \langle a\rangle\phi)\}, \{p\})$ and $\mathrm{sts}_1 := \mathtt{open}(\{(6, \langle a\rangle\phi)\}, \{p\})$. Now all nodes are up-to-date, but Rule 4 is not applicable for (1) because the set of potential rescuers for ϕ is no longer empty.

Next we expand (6), which creates (7), then (7), which creates (8), then (8), which creates (9) and (10), and finally (9), which creates no new nodes. Node (9)

Procedure `det-prs-child`(x, y, φ) for passing a prs-entry of a child to a parent

Input: two nodes $x, y \in V$ and a formula $\varphi \in \Gamma_y \cap \mathrm{Ev}$
Output: $\bot$ or a set of node-formula pairs
Remark: if `det-prs-child`(x, y, φ) has been invoked before with exactly the same arguments and *under the same invocation of* `det-sts-`β*,* `det-sts-state` *or* `det-sts-spl`, the procedure is not executed a second time but returns the cached result of the first invocation. We do not model this behaviour explicitly in the pseudocode.

if $\mathrm{sts}_y = \texttt{closed}(\cdot)$ **then return** $\emptyset$
else if $\mathrm{sts}_y = \texttt{unexp}$ *or* $\mathrm{sts}_y = \texttt{undef}$ *or not* $y \sqsubset x$ **then return** $\{(y, \varphi)\}$
else (∗ $\mathrm{sts}_y = \texttt{open}(\cdot,\cdot)$ & $y \sqsubset x$ ∗)
  if $\mathrm{prs}_y(\varphi) = \bot$ **then return** $\bot$
  else (∗ $\mathrm{prs}_y(\varphi)$ is defined ∗)
    let $(z_1, \varphi_1), \ldots, (z_k, \varphi_k)$ be all of the pairs in $\mathrm{prs}_y(\varphi)$
    for $i \longleftarrow 1$ **to** k **do** $\Lambda_i :=$ `det-prs-child`(x, z_i, φ_i)
    if $\exists j \in \{1, \ldots, k\}.\ \Lambda_j = \bot$ **then return** $\bot$ **else return** $\bigcup_{i=1}^{k} \Lambda_i$

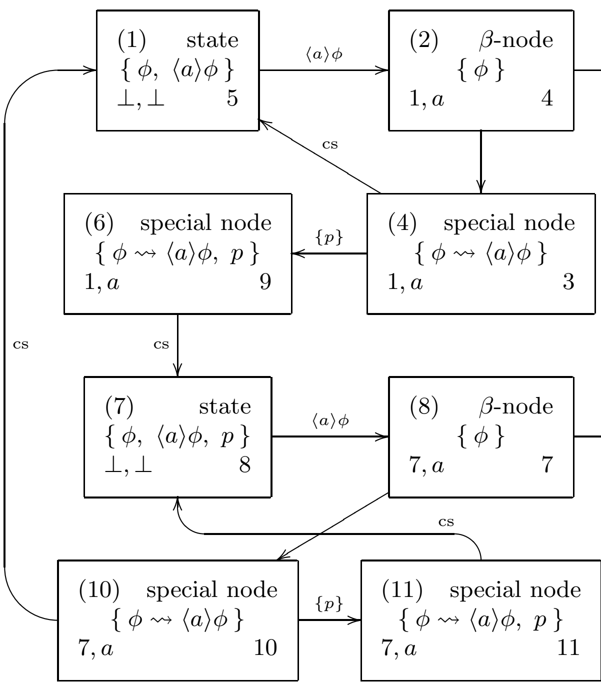

Fig. 1. An example: The graph G just before setting the status of node (2)

Procedure `filter`(x, prs) for handling self-loops in prs chains in G

Input: a node $x \in V$ and a function $\text{prs} : \text{Ev} \to (\mathscr{P}(V \times \text{Ev}))^{\perp}$
Output: prs where self-loops have been handled

$\text{prs}' := \text{prs}^{\perp}$
foreach $\varphi \in \Gamma_x \cap \text{Ev}$ *such that* $\text{prs}(\varphi) \neq \perp$ **do**
  $\Delta := \{\varphi\} \cup \text{reach}(\text{prs}, x, \varphi)$
  if *not* $\exists \chi \in \Delta.\ \text{prs}(\chi) = \perp$ **then**
    $\Lambda := \bigcup_{\chi \in \Delta} \{(z, \psi) \in \text{prs}(\chi) \mid z \neq x\}$
    $\text{prs}' := \text{prs}'[\varphi \mapsto \Lambda]$

return prs'

is similar to (3), but unlike (3), it is compatible with its parent state (7) which results in $\text{sts}_9 :=$ `open`$(\perp, \emptyset)$. Using our strategy from the last section, we would now expand (10) so that (8) can become defined after both its children became defined. Since (9) fulfils all its eventualities, we choose to define (8) instead and set $\text{sts}_8 :=$ `open`$(\perp, \emptyset)$. Next we define (7) and then (6) which sets $\text{sts}_7 :=$ `open`$(\perp, \emptyset)$ and $\text{sts}_6 :=$ `open`$(\perp, \emptyset)$. The status of (4) is not affected since (6) was defined after (4), giving "(6) $\not\sqsubset$ (4)" in `det-prs-child`(4, 6, $\langle a \rangle \phi$).

We expand (10) which inserts the edge from (10) to (1). Then we define (10) which creates (11) and sets $\text{sts}_{10} :=$ `open`$(\perp, \emptyset)$. Note that the invocation of `det-prs-child`(10, 1, $\langle a \rangle \phi$) in the invocation `det-sts-spl`(10) leads to the recursive invocation `det-prs-child`(10, 6, $\langle a \rangle \phi$). Expanding and defining (11) yields $\text{sts}_{11} :=$ `open`$(\perp, \emptyset)$. Finally, no rule is applicable in the shown subgraph.

References

1. Vardi, M.Y.: The taming of converse: Reasoning about two-way computations. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 413–424. Springer, Heidelberg (1985)
2. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Inf. and Comp. 162, 117–137 (2000)
3. Nguyen, L.A., Szałas, A.: An optimal tableau decision procedure for Converse-PDL. In: Proc. KSE-09, pp. 207–214. IEEE Computer Society, Los Alamitos (2009)
4. Lange, M., Stirling, C.: Focus games for satisfiability and completeness of temporal logic. In: Proc. LICS-01, pp. 357–365. IEEE Computer Society, Los Alamitos (2001)
5. Lange, M.: Satisfiability and completeness of Converse-PDL replayed. In: Günter, A., Kruse, R., Neumann, B. (eds.) KI 2003. LNCS (LNAI), vol. 2821, pp. 79–92. Springer, Heidelberg (2003)
6. Vardi, M., Wolper, P.: Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences 32(2), 183–221 (1986)
7. Goré, R., Widmann, F.: An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 437–452. Springer, Heidelberg (2009)
8. Goré, R., Widmann, F.: Sound global state caching for ALC with inverse roles. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 205–219. Springer, Heidelberg (2009)