An Optimal On-the-fly Tableau-based Decision Procedure for PDL-Satisfiability

Rajeev Goré[1] and Florian Widmann[2]

[1] Logic and Computation Group, The Australian National University
Canberra, ACT 0200, Australia, Rajeev.Gore@anu.edu.au

[2] Logic and Computation Group and NICTA* The Australian National University
Canberra, ACT 0200, Australia, Florian.Widmann@anu.edu.au

Abstract. We give an optimal (EXPTIME), sound and complete tableau-based algorithm for deciding satisfiability for propositional dynamic logic. Our main contribution is a sound method to track unfulfilled eventualities "on the fly" which allows us to detect "bad loops" sooner rather than in multiple subsequent passes. We achieve this by propagating and updating the "status" of nodes throughout the underlying graph as soon as is possible. We give sufficient details to enable an easy implementation by others. Preliminary experimental results from our unoptimised OCaml implementation indicate that our algorithm is feasible.

1 Introduction

Propositional dynamic logic (PDL) is an important logic for reasoning about programs [1]. Its formulae consist of traditional Boolean formulae plus "action modalities" built from a finite set of atomic programs using sequential composition (;), non-deterministic choice ($\cup$), repetition ($*$), and test (?). The satisfiability problem for PDL is EXPTIME-complete [2]. Unlike EXPTIME-complete description logics with algorithms exhibiting good average-case behaviour, no decision procedures for PDL-satisfiability are satisfactory from both a theoretical (soundness, completeness, optimality) and practical (average case behaviour) viewpoint, as we briefly explain next.

Fischer and Ladner's method [1] for PDL is impractical because it first constructs the set of all consistent subsets of the set of all subformulae of the given formula, which always requires exponential time. Pratt's optimal method [2] for PDL initially builds a "pseudo-model" (graph) and then checks whether the graph is a real model by making multiple passes that prune inconsistent nodes, and prune nodes containing "eventualities" which cannot be fulfilled by the current graph. Since an eventuality is detected as unfulfilled only in the pruning phase, it can do needless work, as we show shortly. LoTREC, which is primarily an educational tool, implements such a multi-pass method for PDL, but it

* 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.

is suboptimal (2EXPTIME) because it treats disjunctions naively. Baader's [3] tableau-based decision procedure for essentially PDL without "test" is suboptimal (2EXPTIME) and we cannot see how to extend it to "test". DLP implements this method restricted to test-free formulae where $*$ applies only to atomic programs (`http://ect.bell-labs.com/who/pfps/dlp/`). De Giacomo and Massacci [4] give a 2EXPTIME algorithm for deciding converse PDL-satisfiability, discuss ways to obtain optimality, but do not give an actual EXPTIME algorithm. The prover pdl-tableau (`http://www.cs.man.ac.uk/~schmidt/pdl-tableau/`) implements a variation of this method restricted to formulae without nested star.

Theorem provers based on optimal automata-based methods [5] are still in their infancy because good optimisations are not known [6]. Optimal game-theoretic methods for fix-point logics are known [7], but the proof of decidability relies heavily on non-determinism since the main goal is to prove a complexity bound. Brünnler and Lange [8] give "focused" cut-free sequent calculi based on these games, give proofs for PLTL and CTL in detail, and state without giving details that their calculi extend to PDL. The obvious decision procedures obtainable from their completeness proofs are suboptimal (2EXPTIME) since their underlying structure is a tree. We know of no resolution methods for PDL.

Here, we give an optimal, sound and complete tableau-based decision procedure for PDL-satisfiability. Our main contribution is a sound method to track unfulfilled eventualities "on the fly" which allows us to detect "bad loops" sooner rather than in multiple subsequent passes. Essentially, we interleave the graph-building and graph-pruning phases of Pratt's method by propagating and updating the "status" of nodes throughout the underlying graph as soon as is possible, significantly extending a similar method for description logic ALC [9]. The additional technicalities are non-trivial. We present pseudo code rather than traditional tableau rules because the "on the fly" nature of our algorithm gives it a non-local flavour. Thus a set of traditional local tableau "completion rules" would be cluttered by side-conditions to enforce the non-local aspects or would require a complicated strategy of rule applications. Preliminary experimental results from our unoptimised OCaml implementation (`http://rsise.anu.edu.au/~rpg/PDLGraphProver/`) indicate that our algorithm is feasible. Further work is to add the extensive array of optimisations which have proved successful for practical tableau-based methods for description logics.

To see how Pratt's method can do needless work, consider a formula $\langle a\rangle\varphi \wedge \langle b\rangle\psi$ where φ is explored first, φ is unsatisfiable because of some unfulfillable eventuality, and ψ is a huge formula. Pratt's method can only recognise unfulfillable eventualities in the pruning phase. Thus it *must* expand ψ even though it is unnecessary. By detecting unfulfilled eventualities "on the fly", our algorithm can recognise that φ is unsatisfiable before exploring ψ.

2 Syntax and Semantics

Definition 1. *Let* AFml *and* APrg *be two disjoint and countably infinite sets of propositional variables and* atomic programs*, respectively. The set* Fml *of all*

formulae and the set Prg *of all* programs *are defined mutually inductively as follows where* $a \in \mathrm{APrg}$ *and* $p \in \mathrm{AFml}$*:*

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

A $\langle \mathrm{ap} \rangle$*-formula is any formula* $\langle\gamma\rangle\varphi$ *where* $\gamma \in \mathrm{APrg}$ *is an atomic program.*

Implication ($\rightarrow$) and equivalence ($\leftrightarrow$) are not part of the core language but can be defined as usual. The size of a formula or a program is defined inductively by adding the sizes of its direct subformulae and subprograms and adding one. In the rest of the paper, let $p \in \mathrm{AFml}$ and $a \in \mathrm{APrg}$.

Definition 2. *A* transition frame *is a pair* (W, R) *where* W *is a non-empty set of worlds and* $R : \mathrm{APrg} \to W \times W$ *is a function mapping each atomic program* $a \in \mathrm{APrg}$ *to a binary relation* R_a *over* W*. A* model (W, R, V) *is a transition frame* (W, R) *and a valuation function* $V : \mathrm{AFml} \to 2^W$ *mapping each propositional variable* $p \in \mathrm{AFml}$ *to a set* $V(p)$ *of worlds.*

Definition 3. *Let* $M = (W, R, V)$ *be a model. The functions* $\tau_M : \mathrm{Fml} \to 2^W$ *and* $\rho_M : \mathrm{Prg} \to 2^{W \times W}$ *are defined inductively as follows:*

$$\begin{array}{llll} \tau_M(p) & := V(p) & \tau_M(\neg\varphi) & := W \setminus \tau_M(\varphi) \\ \tau_M(\varphi \wedge \psi) & := \tau_M(\varphi) \cap \tau_M(\psi) & \tau_M(\varphi \vee \psi) & := \tau_M(\varphi) \cup \tau_M(\psi) \end{array}$$

$$\tau_M(\langle\gamma\rangle\varphi) := \{w \mid \exists v \in W.\ (w, v) \in \rho_M(\gamma)\ \&\ v \in \tau_M(\varphi)\}$$

$$\tau_M([\gamma]\varphi) := \{w \mid \forall v \in W.\ (w, v) \in \rho_M(\gamma) \ \Rightarrow\ v \in \tau_M(\varphi)\}$$

$$\rho_M(a) := R_a$$

$$\rho_M(\gamma \cup \delta) := \rho_M(\gamma) \cup \rho_M(\delta) \qquad \rho_M(\varphi?) := \{(w, w) \mid w \in \tau_M(\varphi)\}$$

$$\rho_M(\gamma;\delta) := \{(w, v) \mid \exists u \in W.\ (w, u) \in \rho_M(\gamma)\ \&\ (u, v) \in \rho_M(\delta)\}$$

$$\rho_M(\gamma*) := \big\{(w, v) \mid \exists k \in \mathbb{N}_0. \exists w_0, \ldots, w_k \in W. \big(w_0 = w\ \&\ w_k = v\ \& \ \forall i \in \{0, \ldots, k-1\}.\ (w_i, w_{i+1}) \in \rho_M(\gamma)\big)\big\} \ .$$

For $w \in W$ *and* $\varphi \in \mathrm{Fml}$*, we write* $M, w \Vdash \varphi$ *iff* $w \in \tau_M(\varphi)$*.*

Definition 4. *A formula* $\varphi \in \mathrm{Fml}$ *is* satisfiable *iff there exists a model* $M = (W, R, V)$ *and a world* $w \in W$ *such that* $M, w \Vdash \varphi$*. A formula* $\varphi \in \mathrm{Fml}$ *is* valid *iff* $\neg\varphi$ *is unsatisfiable.*

Definition 5. *A formula* $\varphi \in \mathrm{Fml}$ *is in* negation normal form *if the symbol* $\neg$ *appears only immediately before propositional variables. For every* $\varphi \in \mathrm{Fml}$*, we can obtain a formula* $\mathrm{nnf}(\varphi)$ *in negation normal form by pushing negations inward as far as possible such that* $\varphi \leftrightarrow \mathrm{nnf}\,\varphi$ *is valid. We define* ${\sim}\varphi := \mathrm{nnf}(\neg\varphi)$*.*

We categorise formulae as α- or β-formulae as shown in Table 1.

Proposition 6. *In the notation of Table 1, the formulae of the form* $\alpha \leftrightarrow \alpha_1 \wedge \alpha_2$ *and* $\beta \leftrightarrow \beta_1 \vee \beta_2$ *are valid.*

Definition 7. *For a given* $\varphi \in \mathrm{Fml}$ *the (infinite) set* $\mathrm{pre}(\varphi)$ *is defined as below. Using it, we define the set* Ev *of all* eventualities *as:*

$$\mathrm{pre}(\varphi) := \{\psi \in \mathrm{Fml} \mid \exists k \in \mathbb{N}_0.\ \exists \gamma_1, \ldots, \gamma_k \in \mathrm{Prg}.\ \psi = \langle\gamma_1\rangle \ldots \langle\gamma_k\rangle\varphi\}$$

$$\mathrm{Ev} := \bigcup_{\varphi \in \Delta} \mathrm{pre}(\varphi) \ \text{ where } \Delta := \{\langle\gamma*\rangle\psi \mid \gamma \in \mathrm{Prg}\ \&\ \psi \in \mathrm{Fml}\} \ .$$

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

α	$\varphi \wedge \psi$	$[\gamma \cup \delta]\varphi$	$[\gamma*]\varphi$	$\langle\psi?\rangle\varphi$	$\langle\gamma;\delta\rangle\varphi$	$[\gamma;\delta]\varphi$	β	$\varphi \vee \psi$	$\langle\gamma \cup \delta\rangle\varphi$	$\langle\gamma*\rangle\varphi$	$[\psi?]\varphi$
α_1	φ	$[\gamma]\varphi$	φ	φ	$\langle\gamma\rangle\langle\delta\rangle\varphi$	$[\gamma][\delta]\varphi$	β_1	φ	$\langle\gamma\rangle\varphi$	φ	φ
α_2	ψ	$[\delta]\varphi$	$[\gamma][\gamma*]\varphi$	ψ			β_2	ψ	$\langle\delta\rangle\varphi$	$\langle\gamma\rangle\langle\gamma*\rangle\varphi$	$\sim\psi$

Definition 8. *Let X and Y be sets. We define $X^? := X \uplus \{\bot\}$ where $\bot$ indicates the undefined value. 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 $x' \neq x$.*

3 An Overview and Our Algorithm

Our algorithm starts at a root containing a given formula ϕ and builds an and-or tree in a depth-first and left to right manner to try to build a model for ϕ. The rules are based on the semantics of PDL and either add formulae to the current world, or create a new world in the underlying model and add the appropriate formulae to it. For a node x, the attribute Γ_x carries this set of formulae.

The strategy for rule applications is the usual one where we "saturate" a node using the α/β-rules until they are no longer applicable, giving a "state" node s, and then, for each $\langle a\rangle\xi$ in s, we create an a-successor node containing $\{\xi\} \cup \Delta$, where $\Delta = \{\psi \mid [a]\psi \in s\}$. These successors are saturated to produce new states using the α/β-rules, and we create the successors of these new states, and so on.

Our strategy can produce infinite branches as the same node can be created repeatedly on the same branch. We therefore "block" a node from being created if this node exists already on any previous branch. For example, in Fig. 1, if the node y' already exists in the tree, say as node y, then we create a "backward" edge from x to y (as shown) and do not create y'. If y' does not duplicate an existing node then we create y' and add a "forward" edge from x to y'. Thus our tableau is a tree of forward edges, with backward edges that either point upwards from a node to a "forward-ancestor", or point leftwards from one branch to another. Cycles can arise only via backward edges to a forward-ancestor.

Our tableau must "fulfil" every formula of the form $\langle\delta\rangle\varphi$ in a node but only eventualities, as defined in Def. 7, cause problems. If $\langle\delta\rangle\varphi$ is not an eventuality, the α/β-rules reduce the size of the principal formula, ensuring fulfilment. If $\langle\delta\rangle\varphi$ is an eventuality, the main problem is the β-rule for formulae of the form $\langle\gamma*\rangle\varphi$. Its left child reduces $\langle\gamma*\rangle\varphi$ to a strict subformula φ, but the right child "reduces" it to $\langle\gamma\rangle\langle\gamma*\rangle\varphi$. If the left child is always inconsistent, this rule can "procrastinate" an eventuality $\langle\gamma*\rangle\varphi$ indefinitely and never find a world which makes φ true. This non-local property must be checked globally by tracking eventualities.

Consider Fig. 1, and suppose the current node x contains an eventuality e_x. We distinguish three cases. The first is that some path from x fulfils e_x in the existing tree. Else, the second case is that some path from x always procrastinates the fulfilment of e_x and hits a forward-ancestor of x on the current branch: *e.g.* the path x, y, v, u, w, z. The forward-ancestor z contains some "reduction" e_z

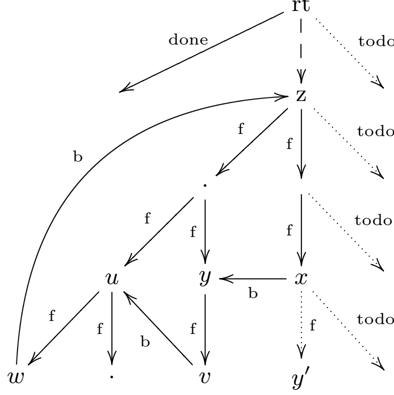

Fig. 1. Tree constructed by our algorithm using forward (f) and backward edges (b)

of e_x. The path from the root to the current node x contains the only currently existing nodes which may need further expansion, and may allow z to fulfil e_z at a later stage, and hence fulfil e_x. We call the pair (z, e_z) a "potential rescuer" of e_x in Γ_x. The only remaining case is that $e_x \in \Gamma_x$ is unfulfilled, has no potential rescuers, and hence can never become fulfilled later, so x can be "closed". The machinery to distinguish these three cases and compute, if needed, all currently existing potential rescuers of every eventuality in Γ_x is described next.

A tableau node x also contains a status sts_x. The value of sts_x is the constant `closed` if the node x is closed. Otherwise, the node is "open" and sts_x contains a function prs which maps each eventuality $e_x \in \Gamma_x$ to $\bot$ or to a set of pairs (v, e) where v is a forward-ancestor of x and e is an eventuality. The status of a node is determined from those of its children once they have all been processed. A closed child's status is propagated as usual, but the propagation of the function prs from open children is more complicated. We give details later, but the intuition is that we must preserve the following invariant for each eventuality $e_x \in \Gamma_x$:

> if e_x is fulfilled in the tree to the left of the path from the root to the node x then $\text{prs}_x(e_x) := \bot$, else $\text{prs}_x(e_x)$ is exactly the set of all potential rescuers of e_x in the current tableau.

An eventuality $e_x \in \Gamma_x$ whose $\text{prs}_x(e_x)$ becomes the empty set can never become fulfilled later, so $\text{sts}_x :=$ `closed`, thus covering the three cases as desired.

Whenever a node n gets a status `closed`, we interrupt the depth-first and left-to-right traversal and invoke a separate procedure which explicitly propagates this status transitively throughout the and-or graph rooted at n. For example, if z gets closed then so will its backward-parent w, which may also close u and so on. This update may break the invariant for some eventuality e in this subgraph by interrupting the path from e to a node that fulfils e or to a potential rescuer of e. We must therefore ensure that the update procedure re-establishes the invariant in these cases by changing the appropriate prs entries. At the end of

the update procedure, we resume the usual depth-first and left-to-right traversal of the tree by returning the status of n to its forward-parent. This "on-the-fly" nature guarantees that unfulfilled eventualities are detected as early as possible.

When our algorithm terminates, formula ϕ is satisfiable iff the root is open.

3.1 The Algorithm

Our algorithm builds a directed graph G consisting of nodes and three types of directed edges: forward-, backward-, and update-edges. We first explain the structure of G in more detail.

Definition 9. *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 three attributes: $\Gamma_x \subseteq \mathrm{Fml}$, $\mathrm{ann}_x : \mathrm{Ev} \to \mathrm{Fml}^?$, and $\mathrm{sts}_x \in \mathfrak{S}^?$ where $\mathfrak{S} := \{\texttt{closed}\} \cup \{\texttt{open}(\mathrm{prs}) \mid \mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?\}$. Each directed edge $e \in E$ is either a forward- or a backward- or an update-edge. If e is a forward- or backward-edge then it is labelled with a label $l_e \in \mathrm{Fml}^?$.*

When we say that x is a forward-ancestor of y we mean that x is an ancestor of y when only the forward-edges of G are considered. Similarly for other graph concepts like child and parent, and other edge types.

As indicated in Def. 9, some attributes of x may be undefined initially. Once an attribute becomes defined in x, however, it will never become undefined again.

The attributes Γ_x and ann_x of a node $x \in G$ are initialised at the creation of x and are not changed afterwards. Together, they uniquely identify x, so no two nodes in G have the same values for *both* attributes. The finite set Γ_x contains the formulae which are assigned to x. The attribute ann_x *annotates* each eventuality $\varphi \in \Gamma_x$, as long as it is not a $\langle \mathrm{ap} \rangle$-formula. The value $\mathrm{ann}_x(\varphi) = \bot$ indicates that φ is not expanded in x, and $\mathrm{ann}_x(\varphi) = \varphi'$ indicates that φ has already been "reduced" to $\varphi' \in \Gamma_x$. These annotations identify the fulfilling path for eventualities. Note that ann_x is defined only for some eventualities in (the finite set) Γ_x. Hence we can test whether ann_x and ann_y (for $y \in V$) are equal.

The attribute sts_x describes the *status* of x. It is initially undefined but becomes defined during the algorithm. Unlike all other attributes, its value may be modified several times, but once it becomes `closed`, it will never change. The value `closed` indicates that the node is not "annotated satisfiable", an extension of satisfiability taking annotations into account. If no formula in a set is annotated, "annotated satisfiable" and satisfiable coincide. A value $\texttt{open}(\mathrm{prs}_x)$ indicates that hope still exists that x is "annotated satisfiable". The function prs_x contains information about each eventuality $\varphi \in \Gamma_x$ as explained in the overview. That is, $\mathrm{prs}_x(\varphi)$ is either $\bot$ or is the finite set of all *potential rescuers* for φ. If $(y, \psi) \in \mathrm{prs}_x(\varphi)$ then y is a forward-ancestor of x and $\psi \in \Gamma_y$. Like ann_x, the function prs_x is defined only for some eventualities in (the finite set) Γ_x.

As the algorithm proceeds, we might have to update sts_x. For example, if the status of a node $y \in G$ is changed to `closed`, an eventuality $\varphi \in \Gamma_x$ may no longer be fulfilled in G. Moreover, if $(y, \psi) \in \mathrm{prs}_x(\varphi)$ then (y, ψ) must be removed from $\mathrm{prs}_x(\varphi)$ since it now cannot help to fulfil φ and is therefore no

longer a potential rescuer of φ. If $\mathrm{prs}_x(\varphi)$ becomes empty, we know that φ cannot be fulfilled in G ever. Hence x is unsatisfiable and its status is set to `closed`.

We insert forward- and backward-edges between two nodes x and y as explained in the overview. Note that y might be a forward- and backward-child of x. In this case, the algorithm takes y as a forward-child of x. To track eventualities, we label a forward- or backward-edge between a state and its child by the $\langle \mathrm{ap} \rangle$-formula $\langle a \rangle \chi$ which creates this child. An update-edge from x to y indicates that a status change of x might affect the status of y.

Definition 10. *Let $x, y \in G$ be nodes and $\varphi \in \mathrm{Fml}$. We call x* closed *iff it has* $\mathrm{sts}_x = \texttt{closed}$ *and* open *iff it has* $\mathrm{sts}_x = \texttt{open}(\mathrm{prs})$ *for some* $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$. *In the latter case, we define* $\mathrm{prs}_x := \mathrm{prs}$.

Note that a node is either open, closed, or has an undefined status. Thus "not closed" is not really equal to "open" as we pretended in the overview.

Definition 11. *Let* $\mathrm{ann}^\perp : \mathrm{Ev} \to \mathrm{Fml}^?$ *and* $\mathrm{prs}^\perp : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$ *be the functions which are undefined everywhere. For a node $x \in V$ and a label $l \in \mathrm{Fml}^?$, let* $\mathrm{getChild}(x, l)$ *be the node $y \in V$ (if existent and unique) such that there exists a forward- or backward-edge $e \in E$ from x to y with $l_e = l$. For a function* $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$, *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 $\mathrm{defer} : V \times \mathrm{Ev} \to \mathrm{Fml}^?$ *is defined as follows:*

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

The function $\mathrm{getChild}(x, l)$ retrieves a particular forward- or backward-child of x. It is easy to see that we will only use it in the algorithm if it is well-defined.

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

Intuitively, the function $\mathrm{defer}(x, \varphi)$ follows the "ann_x-chain". That is, it computes $\varphi_1 := \mathrm{ann}_x(\varphi)$, $\varphi_2 := \mathrm{ann}_x(\varphi_1)$, 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 $\mathrm{ann}_x(\varphi_k) = \perp$. Consequently, we cannot follow the "ann_x-chain" any more. In this case we stop and return $\mathrm{defer}(x, \varphi) := \varphi_k$. The second

Procedure `is-sat(`ϕ`)` for testing whether a formula is satisfiable
Input: a formula $\phi \in$ Fml in negation normal form **Output**: true iff ϕ is satisfiable $G :=$ a new empty graph $r :=$ `build-graph`$(\{\phi\}, \text{ann}^{\perp}, \perp, \perp)$ **return** $\text{sts}_r \neq$ `closed`

outcome is that we can follow the "ann_x-chain" indefinitely. Then, as Γ_x is finite, there must exist a cycle $\varphi_0, \ldots, \varphi_n, \varphi_0$ 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 $\phi \in$ Fml in negation normal form is satisfiable. It initialises the global variable G as the empty graph and invokes `build-graph` with the singleton set $\{\phi\}$ and no annotations. This is the only invocation of `build-graph` that is not initiated while processing a node in G, so its final two arguments are $\perp$. It returns "satisfiable" iff the resulting node $r \in G$, with $\Gamma_r = \{\phi\}$, is not closed in the final graph.

Procedure `build-graph`$(\Gamma, \text{ann}, p, l)$ builds the graph G in a tree-like fashion as explained in the overview and calls the procedures which compute the status of the nodes. Remember that G is a global variable. The arguments of `build-graph` are a set Γ, an annotation ann, the parent node p which invoked it, and a label l. Note that p is undefined for the very first invocation. If there already exists a node x in G with Γ and ann, we insert a backward-edge labelled with l from p to x (if p is defined) and return x. Otherwise we create the desired node $x \in G$, insert a forward-edge labelled with l from p to x (if p is defined), and process x as described next. Each node has at most one forward-edge pointing to it.

If Γ_x contains an "at a world" cycle or a contradiction, we close x. For the other cases, we assume implicitly that Γ_x does not contain either of these.

If Γ contains an α-formula α whose decompositions are not in Γ, or which is an unannotated eventuality, we call x an *α-node*. We create a new set Γ' by adding all decompositions of α to Γ. If α is an eventuality, we also create a new annotation extending ann by mapping α to α_1. Then we invoke `build-graph` recursively and determine and set the status of x. Note that Γ' is strictly bigger than Γ or α is an eventuality which is annotated in ann' but not in ann.

If x is not an α-node and Γ contains a β-formula β such that neither of its immediate subformulae is in Γ, 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 set Γ_i by adding β_i to Γ. If β is an eventuality, we also create a new annotation which extends ann by mapping β to β_i. Note that Γ_i is strictly bigger than Γ or β is an eventuality which is annotated in ann' but not in ann. We then invoke `build-graph` recursively. In the end we determine and set the status of x.

If x is neither an α-node nor a β-node, it must be fully saturated and we call it a *state*. For each $\langle\text{ap}\rangle$-formula $\langle a_i \rangle \varphi_i$ we create a new set Γ_i which contains φ_i

Procedure `build-graph`$(\Gamma, \mathrm{ann}, p, l)$ for building the graph

Input: a set $\Gamma \subseteq \mathrm{Fml}$, a function $\mathrm{ann} : \mathrm{Ev} \to \mathrm{Fml}^?$, a node $p \in V^?$, and a label $l \in \mathrm{Fml}^?$

Output: a node $x \in V$

if $\exists x \in V.\, \Gamma_x = \Gamma \;\&\; \mathrm{ann}_x = \mathrm{ann}$ **then** (∗ annotated set already exists in G ∗)
- **if** $p \neq \bot$ **then** insert a backward-edge from p to x labelled with l in G
- **return** x

else (∗ annotated set not in G yet ∗)
- create new node x with $\Gamma_x := \Gamma$, $\mathrm{ann}_x := \mathrm{ann}$, and $\mathrm{sts}_x := \bot$
- insert x in G
- **if** $p \neq \bot$ **then** insert a forward-edge from p to x labelled with l in G
- **if** $\exists \varphi \in \mathrm{Fml}.(\varphi \in \mathrm{Ev} \;\&\; \mathrm{defer}(x, \varphi) = \bot)$ *or* $\{\varphi, \sim\varphi\} \subseteq \Gamma$ **then**
  - $\mathrm{sts}_x :=$ `closed`
- **else if** $\exists \alpha \in \Gamma.\, \{\alpha_1, \dots, \alpha_k\} \not\subseteq \Gamma$ *or* $(\alpha \in \mathrm{Ev} \;\&\; \mathrm{ann}(\alpha) = \bot)$ **then**
  - $\Gamma' := \Gamma \cup \{\alpha_1, \dots, \alpha_k\}$
  - $\mathrm{ann}' :=$ **if** $\alpha \in \mathrm{Ev}$ **then** $\mathrm{ann}[\alpha \mapsto \alpha_1]$ **else** ann
  - `build-graph`$(\Gamma', \mathrm{ann}', x, \bot)$
  - $\mathrm{sts}_x :=$ `det-sts-`$\beta(x)$
- **else if** $\exists \beta \in \Gamma.\, \{\beta_1, \beta_2\} \cap \Gamma = \emptyset$ *or* $(\beta \in \mathrm{Ev} \;\&\; \mathrm{ann}(\beta) = \bot)$ **then**
  - **for** $i \longleftarrow 1$ **to** 2 **do**
    - $\Gamma_i := \Gamma \cup \{\beta_i\}$
    - $\mathrm{ann}_i :=$ **if** $\beta \in \mathrm{Ev}$ **then** $\mathrm{ann}[\beta \mapsto \beta_i]$ **else** ann
    - `build-graph`$(\Gamma_i, \mathrm{ann}_i, x, \bot)$
  - $\mathrm{sts}_x :=$ `det-sts-`$\beta(x)$
- **else** (∗ x is a state ∗)
  - let $\langle a_1 \rangle \varphi_1, \dots, \langle a_k \rangle \varphi_k$ be all of the $\langle \mathrm{ap} \rangle$-formulae in Γ
  - **for** $i \longleftarrow 1$ **to** k **do**
    - $\Gamma_i := \{\varphi_i\} \cup \{\psi \mid [a_i]\psi \in \Gamma\}$
    - `build-graph`$(\Gamma_i, \mathrm{ann}^{\bot}, x, \langle a_i \rangle \varphi_i)$
  - $\mathrm{sts}_x :=$ `det-sts-state`(x)
- **if** $\mathrm{sts}_x =$ `closed` **then**
  - let $y_1, \dots, y_k$ be all the nodes that are backward- or update-parents of x
  - **for** $i \longleftarrow 1$ **to** k **do** `update`(y_i)
- **return** x

and all ψ such that $[a_i]\psi \in \Gamma$. We then invoke `build-graph` recursively. As none of the eventualities in Γ_i is expanded, there are no annotations. In order to relate the resulting node y to $\langle a_i \rangle \varphi_i$, we label the edge from x to y with $\langle a_i \rangle \varphi_i$. We call y the *successor* of $\langle a_i \rangle \varphi_i$. In the end we determine and set the status of x.

If x is closed, we update all nodes that depend on the status of x; except p, whose status is undefined and which will use the result later. Finally we return x. Note that if `build-graph` creates x via the main "else" then sts_x must be either open or closed but not $\bot$. In particular, this applies to node r in `is-sat`.

Procedure `det-sts-`$\beta(x)$ computes the status of an α- or a β-node $x \in G$. For this task, an α-node can be seen as a β-node with exactly one child. If all children of x are closed then x must also be closed. Otherwise we compute the set

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

Input: an α- or a β-node $x \in V$
Output: the new status of x

let $y_1, \ldots, y_k \in G$ be all the nodes that are forward- or backward-children of x
if $\forall i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `closed` **then return** `closed`
else (∗ at least one child is not closed ∗)
    let $y'_1, \ldots, y'_l$ $(1 \leq l \leq k)$ be all the children of x that are not closed
    $\mathrm{prs} := \mathrm{prs}^{\perp}$
    foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ **do**
        for $i \longleftarrow 1$ **to** l **do**
            $\Lambda_{\varphi,i}$:= **if** y'_i *is a forward-child of* x **then** $\mathrm{prs}_{y'_i}(\varphi)$
                    else `det-prs-child`(x, y'_i, φ)
        Λ_{φ} := **if** $\exists i \in \{1, \ldots, l\}.\ \Lambda_{\varphi,i} = \perp$ **then** $\perp$ **else** $\bigcup_{i=1}^{l} \Lambda_{\varphi,i}$
        $\mathrm{prs} := \mathrm{prs}[\varphi \mapsto \Lambda_{\varphi}]$
    return `filter`(x, prs)

of potential rescuers for each eventuality φ in Γ_x as follows. For each open child y'_i of x we determine the potential rescuers of φ which result from following y'_i. We do this by distinguishing whether y'_i is a forward- or backward-child of x. If y'_i is a forward-child of x then $\mathrm{prs}_{y'_i}(\varphi)$ is just passed on. If y'_i is a backward-child of x then we invoke `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 undefined. Otherwise φ cannot 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 treat potential rescuers 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 $\langle\mathrm{ap}\rangle$-formulae in Γ_x. If any successor is closed then x is closed. Else we compute the potential rescuers for each eventuality in Γ_x as follows. For each $\langle\mathrm{ap}\rangle$-formula $\langle a_i \rangle \varphi_i$ which is an eventuality, we obtain its set of potential rescuers by distinguishing whether its successor y_i is a forward- or backward-child of x. If y_i is a forward-child of x then $\mathrm{prs}_{y_i}(\varphi_i)$ is passed on to $\langle a_i \rangle \varphi_i$. If y is a backward-child of x, we invoke `det-prs-child`. For every other eventuality φ, we determine $\varphi' := \mathrm{defer}(x, \varphi)$. Note that φ' is defined because the state 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 undefined. If φ' is an eventuality, it must be a $\langle\mathrm{ap}\rangle$-formula as x is a state, so we set $\mathrm{prs}(\varphi) := \mathrm{prs}(\varphi')$. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$.
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. We assume that y and φ can "play a part" in fulfilling ψ. First, if there is no edge from x to y in G, we insert an update-edge from x to y in G because a status change of y might affect the status of x. If y is closed, it cannot help to fulfil ψ as indicated by the empty set. If $x = y$ or y is a forward-

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

```
Input: a state x ∈ V
Output: the new status of x
let ⟨a_1⟩φ_1, ..., ⟨a_k⟩φ_k be all of the ⟨ap⟩-formulae in Γ_x
for i ⟵ 1 to k do y_i := getChild(x, ⟨a_i⟩φ_i)
if ∃i ∈ {1, ..., k}. sts_{y_i} = closed then return closed
else (* all children (if any) are not closed *)
    prs := prs^⊥
    for i ⟵ 1 to k do
        if φ_i ∈ Ev then
            Λ_i := if y_i is a forward-child of x then prs_{y_i}(φ_i)
                   else det-prs-child(x, y_i, φ_i)
            prs := prs[⟨a_i⟩φ_i ↦ Λ_i]
    foreach φ ∈ Γ_x ∩ Ev such that φ is not a ⟨ap⟩-formula do
        φ' := defer(x, φ)
        if φ' ∈ Ev then prs := prs[φ ↦ prs(φ')]
    return filter(x, prs)
```

ancestor of 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.

Each invocation of `det-prs-child` can be uniquely assigned to the invocation of `det-sts-`β or `det-sts-state` which (possibly indirectly) invoked it. To meet our complexity bound, we require that under the same invocation of `det-sts-`β or `det-sts-state`, 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. The second invocation would return the same result and would not modify the graph.

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 be strict forward-ancestors of x, we cannot keep (x, ψ) in $\text{prs}(\varphi)$; but we cannot just ignore them 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

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

Input: two nodes $x, y \in V$ such that $x = y$ or y is a forward-ancestor of x or the status of y is defined already; and a formula $\varphi \in \Gamma_y \cap \mathrm{Ev}$
Output: undefined or a set of node-formula pairs
Remark: if `det-prs-child`(x, y, φ) has already been invoked before with exactly the same arguments and *under the same invocation of* `det-sts-`β *or* `det-sts-state`, 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 *there is no edge (of any type) from x to y in G* **then**
  insert an update-edge from x to y in G
if $\mathrm{sts}_y =$ `closed` **then return** $\emptyset$
else if *y is a forward-ancestor of x or $y = x$* **then return** $\{(y, \varphi)\}$
else (* y is open because $\mathrm{sts}_y \neq$ `closed` and its status is defined already *)
  if $\mathrm{prs}_y(\varphi) = \bot$ **then return** $\bot$
  else (* $\mathrm{prs}_y(\varphi)$ is defined *)
    let $(z_1, \varphi_1), \dots, (z_k, \varphi_k)$ 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 $\exists j \in \{1, \dots, k\}.\ \Lambda_j = \bot$ **then return** $\bot$ **else return** $\bigcup_{i=1}^{k} \Lambda_i$

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

Input: a node $x \in V$ and a function $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$
Output: the new status of x

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

Procedure `update`(x) for propagating the status of nodes

Input: a node $x \in V$ that has a defined status

if $\mathrm{sts}_x \neq$ `closed` **then**
  $\mathrm{sts} :=$ **if** *x is an α- or a β-node* **then** `det-sts-`$\beta(x)$ **else** `det-sts-state`(x)
  if $\mathrm{sts}_x \neq \mathrm{sts}$ **then**
    $\mathrm{sts}_x := \mathrm{sts}$
    let $y_1, \dots, y_k$ be all the nodes that are forward-, backward- or update-parents of x
    for $i \longleftarrow 1$ **to** k **do** `update`(y_i)

Table 2. Some definitions used in the example

$\varphi_1 := \langle (a*; b*)* \rangle p$	$\varphi_2 := \langle a*; b* \rangle \varphi_1$
$\varphi_3 := \langle a* \rangle \langle b* \rangle \varphi_1$	$\varphi_4 := \langle b* \rangle \varphi_1$
$\Delta := \{ [(a \cup b)*]\neg p,\ p,\ [a \cup b][(a \cup b)*]\neg p,\ [a][(a \cup b)*]\neg p,\ [b][(a \cup b)*]\neg p \}$	

detects all self-loops from x to itself which are relevant for fulfilling φ. We add φ as it is not in $\mathrm{reach}(\mathrm{prs}, x, \varphi)$. 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 which contain proper forward-ancestors of x.

If an eventuality $\varphi \in \Gamma_x$ has the empty set of potential rescuers, x is closed. **Procedure** `update`(x) propagates status changes through G. It recomputes the status of a node $x \in V$. If the new status differs from its old one, it updates sts_x and invokes `update` recursively on all nodes whose status may be affected by this change. A node that is not closed is either an α/β-node or a state.

Theorem 12 (Soundness, Completeness and Termination). *Let $\phi \in \mathrm{Fml}$ 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.*

4 A Fully Worked Example

Consider the valid formula $\langle (a*; b*)* \rangle p \to \langle (a \cup b)* \rangle p$. The full tableau for its negation does not fit on one page, but its core subgraph is the tableau for the unsatisfiable formula $\phi := \langle b \rangle \langle b* \rangle \langle (a*; b*)* \rangle p \wedge [(a \cup b)*]\neg p$. We therefore consider the tableau for ϕ. To save space, we use the definitions in Table 2.

Figure 2 (almost) shows the corresponding graph G just before setting the status of node (2). To save space, we (recursively) perform multiple α-expansions inside nodes. Thus, there are no α-nodes in G. For example, the root node contains ϕ, as well as its decompositions $\langle b \rangle \langle b* \rangle \varphi_1$ and $[(a \cup b)*]\neg p$, the decompositions of $[(a \cup b)*]\neg p$, and so on. Incidentally, Δ, and in particular $\{\neg p\}$, is a subset of the Γ-components of all nodes, reflecting the semantics of $[(a \cup b)*]\neg p$.

The function P_{13} maps φ_3 and $\langle a \rangle \langle a* \rangle \langle b* \rangle \varphi_1$ to $\{(8, \varphi_3)\}$ and is undefined elsewhere. All other P_i in Fig. 2 map each eventuality in their corresponding set to $\{(2, \langle b* \rangle \varphi_1)\}$ and are undefined elsewhere. The nodes are labelled in creation order. The annotation ann is given using "$\rightsquigarrow$" in Γ. For example, in node (3), we have $\Gamma_3 = \{ \varphi_4,\ \langle (a*; b*)* \rangle p \} \cup \Delta$, and ann_3 maps the eventuality φ_4 to $\langle (a*; b*)* \rangle p$ and is undefined elsewhere. The bottom line of a node contains its status. Solid arrows represent forward-edges and dashed arrows represent backward-edges. There are no update-edges in this example. The label of a forward- and backward-edge is only given if it is a formula and not $\perp$.

Nodes (1), (2), (3), and (4) are created first, and (4) is closed because it contains p and $\neg p$. Then (5) and (6) are created, but (6) is closed as it contains an "at a world" cycle. Next, nodes (7) to (11) are created. Like (4), node (11) is closed because of a contradiction. When trying to create the second child of (10),

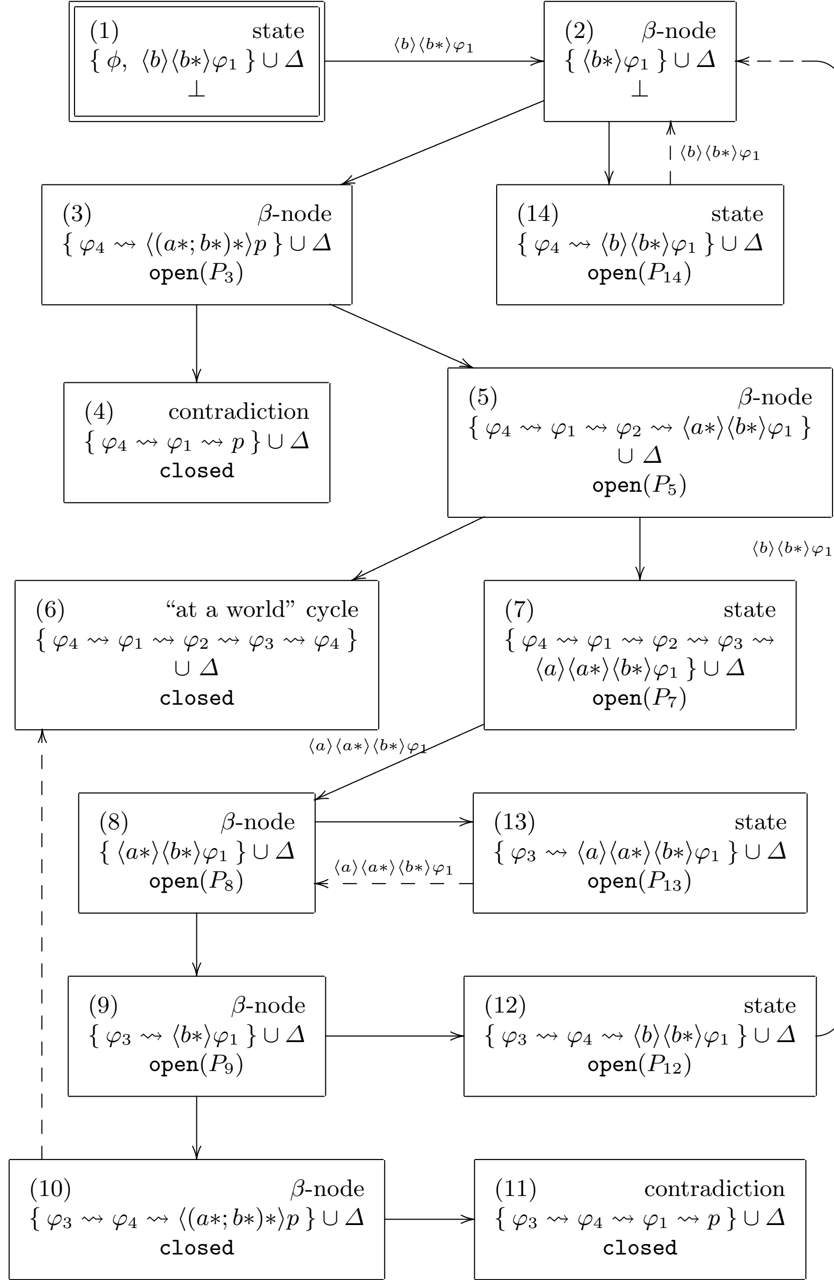

Fig. 2. An example: The graph G just before setting the status of node (2)

we find that the requested node is already contained in G as (6). Hence we insert a backward-edge from (10) to (6). Since both children of (10) are closed, node (10) itself is closed too via `det-sts-`β(10). Next, we create (12), which is a state. Its requested child is already in G as (2), so we insert a backward-edge from (12) to (2). The status of (2) is $\bot$, in particular it is not closed, and (2) is a forward-ancestor of (12). So we set the status of (12) to `open`(P_{12}) via `det-sts-state`(12) and `det-prs-child`(12, 2, $\langle b*\rangle\varphi_1$). Remember that P_{12} maps all eventualities in Γ_{12} to $\{(2, \langle b*\rangle\varphi_1)\}$. The status is then propagated to (9).

Node (13) is conceptually similar to (12). When determining the status of (8) via `det-sts-`β(8), node (8) "inherits" two potential rescuers for each eventuality: $(2, \langle b*\rangle\varphi_1)$ from (9) and $(8, \varphi_3)$ from (13). But $(8, \varphi_3)$ is removed by `filter` since a node cannot be part of a potential rescuer of itself. The status is then propagated to (7), (5), and (3). Node (14) is conceptually similar to (12). As stated before, this is the moment at which G is shown in Fig. 2. When determining the status of (2) via `det-sts-`β(2), node (2) "inherits" only the potential rescuer $(2, \langle b*\rangle\varphi_1)$ from its children. Since $(2, \langle b*\rangle\varphi_1)$ is removed by `filter`, the eventuality $\langle b*\rangle\varphi_1$ has no potential rescuers. Hence we know that $\langle b*\rangle\varphi_1$ cannot be fulfilled in G now or in the future, so (2) is closed via `filter`.

The subsequent invocation of `update`(2) closes all non-closed nodes in the subgraph rooted at (2). Finally, the root (1) is closed via `det-sts-state`(1).

5 Implementation Issues and Experiments

For clarity, the description of our algorithm omits some immediate optimisations. For example, once a state has a closed forward- or backward-child, creating and exploring its remaining children is moot: see $\langle a\rangle\varphi \wedge \langle b\rangle\psi$ in the introduction. Our implementation in OCaml (`http://rsise.anu.edu.au/~rpg/PDLGraphProver/`) includes this optimisation, but does not include most optimisations which have proved crucial in taming description logics [10].

As explained in the introduction, all existing implementations for PDL by other authors are either educational tools (LoTREC), or do not handle the full language (DLP, pdl-tableaux). Therefore, we compared our graph-based method with our tree-based method [11] on randomly generated formulae. Their implementations share many basic data structures and the same (limited) optimisations. Thus they should differ mostly in their tree versus graph aspects.

Each randomly generated formula of size n contained at most $n/20$ propositional variables and $n/20$ atomic programs. The formulae were created by randomly choosing a connective with equal probability and recursively creating the subformulae or subprograms. If a connective had two subformulae/subprograms, their sizes were chosen randomly so that the final formula had the desired size.

We randomly generated ten million formulae for each size 40, 50, ..., 90 and ran both provers on them. For each formula, we set a timeout of 10 seconds. If a solver timed out, we took its running time for this formula to be the timeout, that is 10 seconds. The results are given in Table 5. The lack of timeouts shows that the graph method is clearly more stable. Ignoring stability, there is not much

Table 3. Average running time per 10,000 formulae (and number of timeouts if greater than 0) for ten million randomly generated formulae of each size, shown separately for satisfiable and unsatisfiable formulae

formulae size	40	50	60	70	80	90
satisfiable (%)	91.3%	91.1%	93.6%	93.5%	94.9%	94.9%
Graph (sat)	0.6s	0.8s	1.0s	1.2s	1.4s	1.6s
Tree (sat)	0.7s (2)	1.2s (23)	1.4s (22)	2.1s (62)	2.2s (46)	3.7s (151)
Graph (unsat)	0.6s	0.8s	1.0s	1.3s	1.6s	1.9s
Tree (unsat)	1.0s	5.7s (27)	7.9s (28)	22.8s (94)	29.6s (102)	52.8s (190)

difference in the running times for satisfiable formulae. But the graph method is clearly superior for unsatisfiable formulae.

On individual handcrafted examples, we sometimes observed that the tree-based method was faster, even when it generated more nodes. We believe this is because tracking eventualities in graphs requires more bookkeeping and updating than in trees, and sometimes this bookkeeping is in vain when branches do not share nodes. Another reason might be that the tree-based method requires less space since it can discard previous branches.

References

1. Fisher, M., Ladner, R.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences **18**(2) (1979) 194–211
2. Pratt, V.R.: A near-optimal method for reasoning about action. Journal of Computer and System Sciences **20**(2) (1980) 231–254
3. Baader, F.: Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In: Proc. IJCAI-91. (1991) 446–451
4. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Inf. and Comp. **162** (2000) 117–137
5. Vardi, M., Wolper, P.: Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences **32**(2) (1986) 183–221
6. Pan, G., Sattler, U., Vardi, M.Y.: BDD-based decision procedures for the modal logic K. Journal of Applied Non-Classical Logics **16**(1–2) (2006) 169–208
7. Lange, M., Stirling, C.: Focus games for satisfiability and completeness of temporal logic. In: Proc. LICS-01, IEEE Computer Society (2001) 357–365
8. Brünnler, K., Lange, M.: Cut-free sequent systems for temporal logic. Journal of Logic and Algebraic Programming **76**(2) (2008) 216–225
9. Goré, R., Nguyen, L.A.: EXPTIME tableaux for ALC using sound global caching. In: Proc. of the International Workshop on Description Logics (DL-07). (2007)
10. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation **9**(3) (1999) 267–293
11. Abate, P., Goré, R., Widmann, F.: An on-the-fly tableau-based decision procedure for PDL-satisfiability. Electr. Notes Theor. Comput. Sci. **231** (2009) 191–209