Efficiently Deciding Mu-calculus with Converse over Finite Trees

Pierre Genevès, Nabil Layaïda, Alan Schmitt, Nils Gesbert

A

Efficiently Deciding μ-calculus with Converse over Finite Trees

Pierre Genevès, CNRS
Nabil Layaïda, INRIA
Alan Schmitt, INRIA
Nils Gesbert, Université Grenoble Alpes

We present a sound and complete satisfiability-testing algorithm and its effective implementation for an alternation-free modal μ-calculus with converse, where formulas are cycle-free and are interpreted over finite ordered trees. The time complexity of the satisfiability-testing algorithm is $2^{\mathcal{O}(n)}$ in terms of formula size n. The algorithm is implemented using symbolic techniques (BDD). We present crucial implementation techniques and heuristics that we used to make the algorithm as fast as possible in practice. Our implementation is available online, and can be used to solve logical formulas of significant size and practical value. We illustrate this in the setting of XML trees.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*modal logic*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*decision problems*; E.1 [**Data Structures**]: Trees

General Terms: Algorithms, languages, theory, verification

Additional Key Words and Phrases: Modal logic, satisfiability, implementation

ACM Reference Format:
Genevès, P., Layaïda, N., Schmitt, A. and Gesbert, N. Efficiently Deciding μ-calculus with Converse over Finite Trees. ACM Trans. Comput. Logic V, N, Article A (January YYYY), 40 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

This paper introduces a logic for reasoning over finite trees, a sound and complete decision procedure for checking the satisfiability of a formula of the logic, and an effective implementation of the decision procedure. The logic is a variant of μ-calculus adapted for finite trees and equipped with backward modalities and nominals. Specifically, the logic is an alternation-free modal μ-calculus with converse, where formulas are cycle-free and are interpreted over finite ordered trees. The time complexity of the satisfiability-testing algorithm is optimal: $2^{\mathcal{O}(n)}$ in terms of formula size n. We present crucial implementation techniques like the use of symbolic techniques (BDD) and heuristics to make the algorithm as fast as possible in practice. Our implementation is available online, and can be used to solve logical formulas of significant size.

Detailed affiliation of authors: Nils Gesbert[123], Pierre Genevès[213], Nabil Layaïda[312], Alan Schmitt[3]
[1]Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2]CNRS, LIG, F-38000 Grenoble, France
[3]Inria
Corresponding author's address: Pierre Genevès, INRIA Grenoble - Rhône-Alpes, 655 avenue de l'europe, Montbonnot, 38 334 Saint Ismier Cedex France; email: pierre.geneves@cnrs.fr.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1.1. Related Work and Motivations

The propositional μ-calculus was introduced as a logic for describing properties of graphs with labeled edges. It was invented by Dana Scott and Jaco de Bakker, and further developed by V.R. Pratt [Pratt 1981] and then Dexter Kozen into the version mostly used nowadays [Kozen 1983]. Several modal logics can be encoded in the μ-calculus, including linear temporal logic, computational tree logic [Clarke and Emerson 1981], CTL*, and propositional dynamic logic [Fischer and Ladner 1979]. In contrast with the importance and large applicative spectrum of the μ-calculus satisfiability problem, only very few actual effective implementations have been reported in the litterature. One implementation for the full μ-calculus (without converse modalities) and some variants is MLSolver [Friedmann and Lange 2010], a generic tool which implements the satisfiability problem for several fixpoint logics by reducing it to the problem of solving a parity game, which is then solved using the solver PGSolver [Friedmann and Lange 2009].

A large number of variants of the μ-calculus have been studied, but as pointed out in [Tanabe et al. 2005]: "the satisfiability testing problem for the μ-calculus is known to be decidable for a variety of extensions and subfragments, but effective implementation has not necessarily been developed for all such logics".

We review below the works on variants that are most closely related to ours in terms of supported logical features (e.g., backward modalities, nominals), models of the logic (trees), or from the point-of-view of the approach oriented toward an effective implementation (effective algorithmics). For instance, the work found in [Pan et al. 2006] pursues a goal similar to ours for the modal logic K. The approach yields effective BDD-based decision procedures for K, usable in practice. However, the expressive power of the logic is incomparable to the one of the μ-calculus since K lacks recursion (no fixpoint) and backward modalities.

Backward Modalities. In applications, we often need to follow edges not only in the forward direction but also in the backward direction. Therefore, researchers have been focusing on temporal logics that can handle both directions, or modalities, in order to reason about both the "past" and the "future". Although converse modalities do not add expressive power, they provide an advance in terms of succinctness as they offer a notation for otherwise exponentially larger formulas. Succinctness is a crucial matter when considering the combined complexity of the decision procedure. The satisfiability problem for the general μ-calculus with converse modalities (MC) is known to be EXPTIME-complete [Vardi 1998]. The decision procedure is constructed by converting the problem into the emptiness problem of the language recognized by a certain alternating tree automaton on infinite trees. In order to solve this emptiness problem, complex operations are required including determinization of parity automata [Safra 1988]; it can be done in $2^{\mathcal{O}(n^4 \cdot \log\, n)}$, where n is the size of the formula [Grädel et al. 2002].

The work found in [Tanabe et al. 2005; Tanabe et al. 2008] provides an implementation of a decision procedure for the alternation-free fragment of the μ-calculus with converse (AFMC), whose time complexity is $2^{\mathcal{O}(n \cdot \log\, n)}$. As expected, the decision procedure for the AFMC is less complex than the one for the MC. The alternation-free restriction makes much sense since the expressive power of AFMC exactly corresponds to the one of weak monadic-second order logic [Kupferman and Vardi 1999].

Trees. Applications of the satisfiability-checking problem often restrict the allowed models to sets of finite trees (see, e.g., [Zee et al. 2008]). Therefore, even if the AFMC lacks the finite model property (which is lost due to the addition of converse modalities), it makes sense to search for finite trees satisfying a given logical formula.

In this line of research, the work of [Afanasiev et al. 2005] presents a special version of Propositional Dynamic Logic (PDL) for reasoning about finite sibling-ordered trees. However, the precise expressive power of the logic is still an open problem, although the logic is subsumed by the AFMC.

In [Tanabe et al. 2005; Tanabe et al. 2008], models of the logic are Kripke structures (infinite graphs). Models can be restricted to be binary-branching finite trees through an additional logical formula that encodes König's lemma. However, the authors notice that the performance of the decision procedure may not be very attractive in this setting [Tanabe et al. 2005]. They do no give further details, but our research has given us some insights. Specifically, a first source of inefficiency of this approach comes from the fact that the decision procedure requires expensive cycle-detection for rejecting infinite derivation paths for least fixpoint formulas. A second and even more fundamental source of inefficiency is that the decision procedure of [Tanabe et al. 2005] must compute a greatest fixpoint: it starts from all possible (graph) nodes and progressively removes all inconsistent nodes until a fixpoint is reached. Then the formula is judged as satisfiable if the fixpoint contains a satisfying (tree) structure. As a consequence, and unlike the algorithm presented in this article, (1) the algorithm must always explore all nodes, and (2) it cannot terminate until full completion of the fixpoint computation (otherwise inconsistencies may remain). The present work shows how this can be avoided for finite trees. The resulting performance of our decision procedure, whose time complexity is $2^{\mathcal{O}(n)}$, is much more attractive.

In an earlier work, a logic for finite trees was presented [Tozawa 2004], but the logic is not closed under negation.

The connection with Automata. In our extended abstract [Genevès et al. 2007], we showed the decidability in time $2^{\mathcal{O}(n)}$ of the cycle-free fragment of the AFMC for finite trees. Since then, alternative and closely related approaches based on tree automata have been proposed with similar or higher complexity, but without implementation [Calvanese et al. 2008; Libkin and Sirangelo 2008; Calvanese et al. 2009; Libkin and Sirangelo 2010; Calvanese et al. 2010].

Approaches based on alternating two-way tree automata (2ATA) for infinite trees have resisted implementation, as noticed in [Calvanese et al. 2009], mainly because of complex determinization (see [Calvanese et al. 2008; 2009], in which it is also mentioned that it is practically infeasible to apply the symbolic approach in the infinite tree setting).

A simpler and more appropriate automata-based approach for finite trees is the technique based on weak alternating two-way tree automata (2WATA). However, they require a conversion to non-deterministic finite tree automata (NFTA) for testing non-emptiness. The translation given in [Calvanese et al. 2010] yields an automaton with $2^{\mathcal{O}(n^2)}$ states in terms of the number n of states of the original 2WATA.

As mentioned before, none of the works cited above provides an implementation. In fact, [Calvanese et al. 2008] remarks that a naive implementation of their technique would result in a blow-up in complexity, requiring the use of more elaborate techniques very similar to what we have done. The approach in [Libkin and Sirangelo 2010] is an alternative version of our previous work [Genevès et al. 2007] that allows the shortening of some proofs but does not simplify the implementation. The present work can thus be regarded as the only efficient implementation of the logic or, alternatively, of the 2WATA framework.

For the sake of simplicity and uniformity between the satisfiability algorithm, the proofs, and the implementation techniques, we focus in this paper on the native modal logic. This also emphasizes the fact that bottom-up construction of the finite tree model

and cycle-freeness come naturally and shows exactly why the whole approach is efficient.

1.2. Contributions

Our main result is a satisfiability-testing algorithm for a logic for finite trees whose time complexity is optimal: $2^{\mathcal{O}(n)}$ in terms of the formula size n, together with its effective implementation through BDD techniques.

The essence of our results lives in a sub-logic of the AFMC, with a syntactic restriction—cycle-freeness—on formulas, and whose models are finite trees. Such restrictions are interesting from a theoretical point of view: we prove that, under these conditions, the least and greatest fixpoint operators collapse into a single fixpoint operator. This makes our logic closed under negation and provides many opportunities to derive an efficient implementation.

The decision procedure is implemented and an online demonstration is publicly available, as detailed in §5.5.

An extended abstract of this work was presented at the ACM Conference on Programming Language Design and Implementation (PLDI), 2007 [Genevès et al. 2007]. The new material included in this article comprises the following. The notion of cycle-freeness, a fundamental aspect of our logic, and its formalization are much more detailed. Proofs have been added. A detailed run of the algorithm is described. Implementation techniques and the optimizations used to obtain a satisfiability-testing algorithm that performs well in practice are also discussed.

1.3. Outline

The paper is organized as follows. We first present our data model, trees with focus, in §2. We then introduce the logic in §3. Our satisfiability algorithm is introduced and proven correct in §4. Crucial implementation techniques are discussed in §5. Applications such as Regular Language Equivalence and XPath typing are reviewed in §6. We conclude in §7.

2. TREES WITH FOCUS

Our data model is based on binary trees. Note that it is possible and straightforward to use binary trees to represent unranked n-ary trees where the children of a node are ordered—we use this, for instance, to apply our work to XPath typing in Section 6.

In order to represent trees that are easy to navigate, we use *focused trees*, inspired by Huet's Zipper data structure [Huet 1997]. Focused trees not only describe a tree but also its context: its parent, its parent's other subtree, and its parent's context recursively. Exploring such a structure has the advantage to preserve all the information, which is quite useful when considering forward and backward navigation.

Formally, we assume an alphabet Λ of labels, ranged over by $\alpha, \beta, \ldots$. Each node of the trees bears a finite number of labels; we use L to range over finite sets of labels. The syntax of our data model is as follows.

$$
\begin{array}{rcll}
\tau & ::= & (L, st, st) & \text{binary tree} \\
st & ::= & \tau \mid \mathsf{nil} & \text{subtree} \\
c & ::= & & \text{context} \\
 & & \mathsf{Top} & \text{root of the tree} \\
 & \mid & (L, [], st)_c \mid (L, st, [])_c & \text{context node} \\
ft & ::= & \tau_c & \text{focused tree}
\end{array}
$$

A focused tree τ_c is a pair consisting of a tree τ and its context c. The context describes what is above the tree. It is either Top, meaning the current tree is actually at the root, $(L, [], st)_{c'}$, meaning that the current tree is the left child of a node labeled L

whose right child is st and whose context is c', or symmetrically $(L, st, [])_{c'}$, meaning that the current tree is the right child of a node labeled L whose left child is st and whose context is c'.

We write $\mathcal{F}$ for the set of finite focused trees, i.e., the language generated by the above grammar.

We now describe how to navigate focused trees. There are four directions, or *modalities*, that can be followed: for a focused tree ft, $ft\,\langle 1\rangle$ changes the focus to the left (first) child of the current tree, $ft\,\langle 2\rangle$ changes the focus to the right (second) child of the current tree, $ft\,\langle \bar{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a left child*, otherwise it is undefined, and $ft\,\langle \bar{2}\rangle$ changes the focus to the parent of the tree if the current tree is a right child and is undefined otherwise.

Formally, we have:

$$
\begin{aligned}
(L, \tau, st)_c\,\langle 1\rangle &\stackrel{\text{def}}{=} \tau_{(L,[],st)_c} \\
(L, st, \tau)_c\,\langle 2\rangle &\stackrel{\text{def}}{=} \tau_{(L,st,[])_c} \\
\tau_{(L,[],st)_c}\,\langle \bar{1}\rangle &\stackrel{\text{def}}{=} (L, \tau, st)_c \\
\tau_{(L,st,[])_c}\,\langle \bar{2}\rangle &\stackrel{\text{def}}{=} (L, st, \tau)_c
\end{aligned}
$$

When the focused tree does not have the required shape, these operations are not defined. These navigation operations can be combined and iterated to move from any node of a tree to any other node; a sequence of navigation steps forms a path in a tree. Since the steps have inverses, it makes sense to define the set of paths as a free group rather than as just a set of sequences.

Definition 2.1 (*Paths*). We define the set P of *paths* as the free group generated by 1 and 2. We denote the inverse with an overline bar, the neutral element with ε and the composition law with a dot $\cdot$.

The navigation operation on trees defined above for elementary paths is extended to arbitrary paths $p \in P$ in the obvious way.

The path navigation operation has the following interesting properties:

— if $ft\,\langle p\rangle$ is defined and $p \neq \varepsilon$, then $ft\,\langle p\rangle \neq ft$ (this represents the fact that a tree has no cycles);
— if $ft\,\langle p\rangle$ is defined, then either $ft\,\langle p \cdot q\rangle$ and $ft\,\langle p\rangle\,\langle q\rangle$ are both undefined, or they are both defined and are equal;
— the equivalence relation $\sim$ between focused trees defined by $ft \sim ft' \Leftrightarrow \exists p \in P, ft = ft'\,\langle p\rangle$ is the relation "be nodes of the same tree". Its classes are finite since we only consider finite trees.

This last property holds because we are considering focused trees: such trees make explicit their set of nodes, even those in the context, and the relation between these nodes.

We can also remark that some paths p are such that $ft\,\langle p\rangle$ is undefined for all ft, e.g., the path $1 \cdot \bar{2}$. Again, this is due to the topological restrictions of trees.

In the following, we will often use navigation operations on whole sets of trees; we introduce the following notation.

Notation 2.2. If E is a set of focused trees and p a path, we define:
$E\,\langle p\rangle \stackrel{\text{def}}{=} \{ft\,\langle p\rangle \mid ft \in E \wedge ft\,\langle p\rangle \text{ defined}\}$.

$$
\begin{array}{lll}
\varphi, \psi ::= & & \text{formula} \\
& \top \mid \bot & \text{true, false} \\
\mid & \alpha \mid \neg\alpha & \text{atomic proposition (negated)} \\
\mid & X & \text{variable} \\
\mid & \varphi \vee \psi & \text{disjunction} \\
\mid & \varphi \wedge \psi & \text{conjunction} \\
\mid & \langle a \rangle \varphi \mid \neg \langle a \rangle \top & \text{existential modality (negated)} \\
\mid & \mu(X_i = \varphi_i)_{i \in I} \text{ in } \psi & \text{least n-ary fixpoint} \\
\mid & \nu(X_i = \varphi_i)_{i \in I} \text{ in } \psi & \text{greatest n-ary fixpoint}
\end{array}
$$

Fig. 1. Logic formulas

Remark 2.3. Note that because the navigation operation is partial, we do not have, in general, $E\langle p \rangle \langle q \rangle = E\langle p \cdot q \rangle$, but only $E\langle p \rangle \langle q \rangle \subseteq E\langle p \cdot q \rangle$. More precisely: $E\langle p \rangle \langle q \rangle = E\langle p \cdot q \rangle \cap \mathcal{F}\langle p \rangle \langle q \rangle$.

This remark holds in particular for $q = \overline{p}$ and implies that, in the logic that we now define, the formulas φ and $\langle \mathtt{1} \rangle \langle \overline{\mathtt{1}} \rangle \varphi$ are not equivalent (but the second one implies the first). Indeed, the second formula additionally asserts that the selected tree node has a $\langle \mathtt{1} \rangle$ child.

3. THE LOGIC

We introduce the logic as a sub-logic of the alternation-free modal μ-calculus with converse. We also introduce a restriction on the formulas we consider and give an interpretation of formulas as sets of finite focused trees. We finally show that this restriction and this interpretation make the greatest and smallest fixpoint collapse, yielding a logic that is closed under negation without requiring a greatest fixpoint.

3.1. Formulas

In the following, we denote tuples of unknown size using parentheses and an indexing set, e.g., $(X_i = \varphi_i)_{i \in \{1..n\}}$ means $(X_1 = \varphi_1; X_2 = \varphi_2; \ldots; X_n = \varphi_n)$. It is implicitly assumed that the indexing set is always finite.

In the definitions, $a \in \{\mathtt{1}, \mathtt{2}, \overline{\mathtt{1}}, \overline{\mathtt{2}}\}$ are *programs*. Atomic propositions α correspond to labels from Λ. Formulas, defined in Fig. 1, include the truth and falsehood predicates, atomic propositions (indicating the tree at focus should bear the corresponding label), disjunction and conjunction of formulas, formulas under an existential modality (denoting the existence of a subtree satisfying the sub-formula), and least and greatest n-ary fixpoints. We chose to include a n-ary version of fixpoints because regular types are often defined as a set of mutually recursive definitions, making their translation in our logic more direct and succinct. When there is no need to distinguish between least and greatest fixpoints, we write ι for either μ or ν. In the following, we write "$\iota X.\varphi$" for "$\iota(X = \varphi)$ in X".

We consider μ and ν as binders for the variables and define the notions of free and bound variables and of open and closed formulas as usual. The language $\mathcal{L}_\mu$ we consider is the set of closed formulas.

In general, $\neg\varphi$ is not part of the syntax. It is defined, for closed formulas, as an abbreviation in Fig. 2. This definition uses an auxiliary function $\mathrm{neg}(\cdot)$ which is defined inductively on possibly open formulas, but $\mathrm{neg}(\varphi)$ is in general not the negation of φ if φ is open (in particular, $\mathrm{neg}(X) = X$).

$$\neg\varphi \stackrel{\text{def}}{=} \mathrm{neg}(\varphi) \text{ if } \varphi \text{ is closed and } \varphi \neq \langle a\rangle \top$$

$$\begin{aligned}
\mathrm{neg}(\top) &\stackrel{\text{def}}{=} \bot & \mathrm{neg}(\alpha) &\stackrel{\text{def}}{=} \neg\alpha\\
\mathrm{neg}(\bot) &\stackrel{\text{def}}{=} \top & \mathrm{neg}(X) &\stackrel{\text{def}}{=} X\\
\mathrm{neg}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \mathrm{neg}(\varphi) \wedge \mathrm{neg}(\psi) & \mathrm{neg}(\neg\varphi) &\stackrel{\text{def}}{=} \varphi\\
\mathrm{neg}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \mathrm{neg}(\varphi) \vee \mathrm{neg}(\psi) & \mathrm{neg}(\langle a\rangle\, \varphi) &\stackrel{\text{def}}{=} \neg\langle a\rangle \top \vee \langle a\rangle\, \mathrm{neg}(\varphi)
\end{aligned}$$

$$\begin{aligned}
\mathrm{neg}(\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi) &\stackrel{\text{def}}{=} \nu(X_i = \mathrm{neg}(\varphi_i))_{i\in I} \text{ in } \mathrm{neg}(\psi)\\
\mathrm{neg}(\nu(X_i = \varphi_i)_{i\in I} \text{ in } \psi) &\stackrel{\text{def}}{=} \mu(X_i = \mathrm{neg}(\varphi_i))_{i\in I} \text{ in } \mathrm{neg}(\psi)
\end{aligned}$$

Fig. 2. Negation of closed formulas, defined as syntactic sugar.

$$\begin{aligned}
[\![\top]\!]_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\alpha]\!]_V &\stackrel{\text{def}}{=} \{(L, st_1, st_2)_c \in \mathcal{F} \mid \alpha \in L\}\\
[\![\bot]\!]_V &\stackrel{\text{def}}{=} \emptyset & [\![\neg\alpha]\!]_V &\stackrel{\text{def}}{=} \{(L, st_1, st_2)_c \in \mathcal{F} \mid \alpha \notin L\}\\
[\![X]\!]_V &\stackrel{\text{def}}{=} V(X) & [\![\langle a\rangle\, \varphi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V\, \langle\overline{a}\rangle\\
[\![\varphi \vee \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V & [\![\neg\langle a\rangle \top]\!]_V &\stackrel{\text{def}}{=} \mathcal{F} \setminus \mathcal{F}\, \langle\overline{a}\rangle\\
[\![\varphi \wedge \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V
\end{aligned}$$

$[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} [\![\psi]\!]_{V[\overline{X_i \mapsto U_i}]}$ where the U_i are defined as follows:

$$\text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, [\![\varphi_j]\!]_{V[\overline{X_i\mapsto T_i}]} \subseteq T_j\}$$
$$\text{and for all } j \in I, \text{ let } U_j = \bigcap_{(T_i)\in S} T_j$$

$[\![\nu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} [\![\psi]\!]_{V[\overline{X_i \mapsto U'_i}]}$ where the U'_i are defined as follows:

$$\text{let } S' = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, T_j \subseteq [\![\varphi_j]\!]_{V[\overline{X_i\mapsto T_i}]}\}$$
$$\text{and for all } j \in I, \text{ let } U'_j = \bigcup_{(T_i)\in S'} T_j$$

Fig. 3. Interpretation of formulas

For $\varphi = (\iota(X_i = \varphi_i)_{i\in I} \text{ in } \psi)$ we define $\exp(\varphi) \stackrel{\text{def}}{=} \psi\{({}^{\iota(X_i=\varphi_i)_{i\in I} \text{ in } \varphi_k}/_{X_k})_{k\in I}\}$, which denotes the formula ψ in which every occurrence of a X_k is replaced by $(\iota(X_i = \varphi_i)_{i\in I} \text{ in } \varphi_k)$.

3.2. Model

We define in Fig. 3 an interpretation of our formulas as subsets of $\mathcal{F}$, the set of finite focused trees. For formulas containing free variables, this interpretation is parameterized by a valuation, i.e., a mapping V from variables to subsets of $\mathcal{F}$. We use square brackets and an overline bar to denote a modified mapping, as follows:

$$V[\overline{X_i \mapsto T_i}](X) \stackrel{\text{def}}{=} \begin{cases} T_k & \text{if } X = X_k \text{ with } k \in I,\\ V(X) & \text{otherwise.}\end{cases}$$

We now comment on the interpretation of n-ary fixpoints. Let $\varphi = \iota(X_i = \varphi_i)_{i\in I}$ in ψ with $\iota = \mu$ or ν. We write $\mathcal{P}(\mathcal{F})^I$ for the set of I-indexed tuples of subsets of $\mathcal{F}$; it represents the set of all possible valuations for the tuple of variables $(X_i)_{i\in I}$. We give to this set the partial ordering defined by componentwise inclusion, i.e., $(T_i) \leqslant (T'_i)$ iff $\forall i \in I, T_i \subseteq T'_i$.

The interpretation of the tuple of formulas (φ_i) parameterized by the valuation of the tuple of variables X_i defines a function from $\mathcal{P}(\mathcal{F})^I$ to $\mathcal{P}(\mathcal{F})^I$. The final valuation used to define the interpretation of φ is a fixpoint of this function, the least one if $\iota = \mu$ and the greatest one if $\iota = \nu$, with respect to the ordering defined above. We now show that this corresponds to the definitions in Fig. 3.

LEMMA 3.1. *Let $(X_i = \varphi_i)_{i\in I}$ be a tuple of bindings and ψ a formula, and let V be a valuation.*

Let $F : \mathcal{P}(\mathcal{F})^I \to \mathcal{P}(\mathcal{F})^I$ be the function defined by $F((T_i)_{i\in I}) = (\llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}]})_{j\in I}$. Let $(U_j)_{j\in I}$ and $(U'_j)_{j\in I}$ be as defined in Fig. 3. Then $(U_j)_{j\in I}$ is the least fixpoint of F and $(U'_j)_{j\in I}$ is its greatest fixpoint.

To prove this, we use the following auxiliary lemma:

LEMMA 3.2. *Let φ be a formula, X a variable, V a valuation. Let $A = V(X)$ and $A \subseteq B$. Then $\llbracket\varphi\rrbracket_V \subseteq \llbracket\varphi\rrbracket_{V[X\mapsto B]}$.*

PROOF OF LEMMA 3.2. We prove the lemma by structural induction on φ, for any valuation V, variable X, and sets of focused trees A and B. Almost all cases are straightforward (note that X cannot appear under a negation, which is the key point). We only detail the cases of μ and ν:

— $\varphi = \mu(X_i = \varphi_i)_{i\in I}$ in ψ. If X is one of the X_i, we immediately have $\llbracket\varphi\rrbracket_V = \llbracket\varphi\rrbracket_{V[X\mapsto B]}$. We thus assume X is different from every X_i and we let them commute freely in the valuation. Let $(T_i)_{i\in I} \in S_B = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, \llbracket\varphi_j\rrbracket_{V[X\mapsto B][\overline{X_i\mapsto T_i}]} \subseteq T_j\}$. By induction hypothesis, for any $j \in I$, we have $\llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}]} \subseteq \llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}][X\mapsto B]}$, thus $(T_i)_{i\in I} \in S_A = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, \llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}]} \subseteq T_j\}$, which implies that $S_B \subseteq S_A$. Let $j \in I$ and $t \in U^A_j = \bigcap_{(T_i)\in S_A} T_j$, we thus have $t \in T_j$ for every $(T_i) \in S_A$. As $S_B \subseteq S_A$, we have $t \in T_j$ for every $(T_i) \in S_B$. Thus $t \in U^B_j = \bigcap_{(T_i)\in S_B} T_j$, and $U^A_j \subseteq U^B_j$.
We conclude by the following computation, applying the induction hypothesis each time:

$$\llbracket\psi\rrbracket_{V[\overline{X_i\mapsto U^A_i}]} \subseteq \llbracket\psi\rrbracket_{V[\overline{X_i\mapsto U^A_i}][X_1\mapsto U^B_1]} \subseteq \cdots \subseteq \llbracket\psi\rrbracket_{V[\overline{X_i\mapsto U^A_i}][\overline{X_i\mapsto U^B_i}]} = \llbracket\psi\rrbracket_{V[\overline{X_i\mapsto U^B_i}]} \subseteq \llbracket\psi\rrbracket_{V[\overline{X_i\mapsto U^B_i}][X\mapsto B]}$$

— $\varphi = \nu(X_i = \varphi_i)_{i\in I}$ in ψ. As in the previous case, we use the induction hypothesis to deduce that $T_j \subseteq \llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}]} \subseteq \llbracket\varphi_j\rrbracket_{V[\overline{X_i\mapsto T_i}][X\mapsto B]}$, thus $S'_A \subseteq S'_B$. As we now define U' as the union of the tuples in S', we thus have for every j, $U'^A_j \subseteq U'^B_j$. We conclude as above. □

PROOF OF LEMMA 3.1. First of all, we deduce from the auxiliary lemma 3.2 that the function F is increasing (with respect to the ordering defined by componentwise inclusion, as explained above), i.e., $T \leqslant T'$ implies $F(T) \leqslant F(T')$. We then show the property for the two fixpoints.

— For the least fixpoint: the set S defined in Fig. 3 is the set of tuples T such that $F(T) \leqslant T$. This set is preserved by F: since F is increasing, $F(T) \leqslant T$ implies $F(F(T)) \leqslant F(T)$. Let $U = (U_j)_{j\in I}$. For all T in S, we have $U \leqslant T$, hence $F(U) \leqslant F(T)$ since F is increasing, and thus $F(U) \leqslant T$ for every $T \in S$. This implies $F(U) \leqslant U$ (since U is the intersection of every T in S). We thus have $U \in S$, and therefore $F(U) \in S$ (because S is preserved by F), from which we conclude $U \leqslant F(U)$ (because U is a lower bound), and finally $F(U) = U$. This fixpoint is clearly the least one, since S contains by definition all fixpoints of S and all its elements are greater than U.

— For the greatest fixpoint: the reasoning is the dual of the above. We summarize it: S' is the set of T s.t. $T \leqslant F(T)$. It is preserved by F and U' is its least upper bound. For all $T \in S, T \leqslant U'$, thus $F(T) \leqslant F(U')$, thus $T \leqslant F(U')$. This is true for all T, thus $U' \leqslant F(U')$, thus $U' \in S'$, thus $F(U') \in S'$, thus $F(U') \leqslant U'$, and finally $F(U') = U'$. This fixpoint is clearly the largest since S' contains all fixpoints. □

For closed formulas, the interpretation is clearly independent from the initial V, thus we omit it when considering only closed formulas. Furthermore, we can check by a straightforward induction that $[\![\neg\varphi]\!] = \mathcal{F} \setminus [\![\varphi]\!]$ for any closed formula φ, as expected.

A corollary of the fixpoint lemma 3.1 is that for any fixpoint formula φ we have $[\![\exp(\varphi)]\!] = [\![\varphi]\!]$.

To illustrate the interpretation of fixpoints, consider the two closed formulas $\varphi = \mu X.\, \langle 1\rangle\, X \vee \langle \bar{1}\rangle\, X$ and $\psi = \nu X.\, \langle 1\rangle\, X \vee \langle \bar{1}\rangle\, X$. Following the definition, the interpretation of φ is $[\![X]\!]_{X\mapsto U} = U$ where U is the intersection of all sets in S. Straightforwardly we have $\emptyset \in S$:

$$[\![\langle 1\rangle\, X \vee \langle \bar{1}\rangle\, X]\!]_{X\mapsto\emptyset} = \emptyset \subseteq \emptyset.$$

Thus $U = \emptyset$ and $[\![\varphi]\!] = \emptyset$. Intuitively, there is no base case in the formula, hence the smallest fixpoint is the empty one.

The interpretation of ψ is more complex: it is the set of every focused tree with at least two nodes, one being the parent of the other. We now show that the interpretation of ψ includes the focused tree $ft_1 = (a, (b, \mathsf{nil}, \mathsf{nil}), \mathsf{nil})_{\mathsf{Top}}$. Let $ft_2 = ft_1\, \langle 1\rangle$, that is the tree $(b, \mathsf{nil}, \mathsf{nil})_{(a,[],\mathsf{nil})_{\mathsf{Top}}}$. We thus have $ft_2\langle \bar{1}\rangle = ft_1$. Finally, let V be the mapping $[X \mapsto \{ft_1; ft_2\}]$. We compute as follows:

$$\begin{aligned}
&[\![\langle 1\rangle\, X \vee \langle \bar{1}\rangle\, X]\!]_V \\
&= [\![\langle 1\rangle\, X]\!]_V \cup [\![\langle \bar{1}\rangle\, X]\!]_V \\
&= \{f\, \langle \bar{1}\rangle \mid f \in [\![X]\!]_V \wedge f\, \langle \bar{1}\rangle \text{ defined}\} \cup \{f\, \langle 1\rangle \mid f \in [\![X]\!]_V \wedge f\, \langle 1\rangle \text{ defined}\} \\
&= \{ft_1\} \cup \{ft_2\}
\end{aligned}$$

thus $V(X) \subseteq [\![\langle 1\rangle\, X \vee \langle \bar{1}\rangle\, X]\!]_V$, hence $\{ft_1, ft_2\} \in S$, and thus $\{ft_1, ft_2\} \subseteq U$. We have indeed $ft_1 \in [\![\psi]\!]_\emptyset$.

3.3. Cycle-Free Formulas

As shown in §3.2, the smallest and greatest fixpoints do not coincide. This fact can be linked to a result from Mateescu [Mateescu 2002]: the two fixpoint operators coincide[4] on a given Kripke structure if and only if the structure is acyclic. Although our structures are trees, they are not acyclic Kripke structures since these trees are navigable in both directions. However, the fact they are trees means that the only possible cycles are paths which go one way and back the exact same way ($ft\, \langle p\rangle = ft$ if and only if

[4]By 'coincide' on a structure we mean formally: for any φ, the sets of nodes of that structure where $\mu X.\varphi$ holds and where $\nu X.\varphi$ holds are identical.

$p = \varepsilon$). This allows us to introduce a syntactic restriction on formulas, which we call *cycle-freeness*, making the two fixpoints collapse on our semantic domain.

To define this notion formally, we first define the syntactic graph of a formula, where we label the edges with tree paths from P (see Def. 2.1). Formally, we consider an oriented graph whose vertices are all formulas and whose edges are as follows:

— $\varphi_1 \wedge \varphi_2 \xrightarrow{\varepsilon} \varphi_1$, $\varphi_1 \wedge \varphi_2 \xrightarrow{\varepsilon} \varphi_2$
— $\varphi_1 \vee \varphi_2 \xrightarrow{\varepsilon} \varphi_1$, $\varphi_1 \vee \varphi_2 \xrightarrow{\varepsilon} \varphi_2$
— $\langle a \rangle \varphi' \xrightarrow{a} \varphi'$
— $\iota(X_i = \varphi_i)_{i \in I}$ in $\psi \xrightarrow{\varepsilon} \exp(\iota(X_i = \varphi_i)_{i \in I}$ in $\psi)$
— Formulas which are not of one of the forms above have no outgoing edges.

This graph has the property that the formulas reachable from a closed formula are all closed, thanks to the 'exp' operation.

Definition 3.3. The *Fisher-Ladner closure* $\mathrm{cl}(\psi)$ of a normalised (all bound variables have different names) closed formula ψ is the set of formulas reachable from ψ in the graph defined above, including ψ itself.

This set is finite. Indeed, most of the edges go from a formula to a subformula; the only exception is the fixpoint. When a fixpoint formula is expanded, all the new fixpoint formulas which appear have the form $\iota(X_i = \varphi_i)_{i \in I}$ in φ_k with $k \in I$. But the total number of such formulas is bounded by the number of variables in the whole initial formula, which is finite.

However, the subgraph corresponding to the closure typically contains cycles. What we want to ensure is that the cycles in the syntactic graph do not correspond to cycles in models of the formula. Intuitively, whether a subformula holds at a given node of a tree may depend on whether the same subformula holds at another node of that tree, but should not depend on itself (the same subformula holding at the same node) if we want the two fixpoints to coincide.

We call *walk* a sequence of consecutive formulas and edges in the syntactic graph. The definition below relates walks in the syntactic graph and paths in the trees.

Definition 3.4 (*Trace of a walk*). Let $w = \varphi_1 \xrightarrow{p_1} \varphi_2 \xrightarrow{p_2} \cdots \xrightarrow{p_n} \varphi_{n+1}$ be a walk in the syntactic graph. The *trace* of w is the path $tr(w) = p_1 \cdot p_2 \cdot \cdots \cdot p_n$.

We can now define cycle-freeness.

Definition 3.5 (*Cycle-free formula*). We say that a formula φ is *cycle-free* if, in the syntactic graph of its closure, no nonempty walk from a formula to itself has trace ε.

Note that cycle-freeness implies the more usual property of guardedness, which here could be defined as the absence of cycles whose edges are *all* labelled ε.

For instance, the formulas φ and ψ described at the end of §3.2 are not cycle-free. Indeed, looking at the syntactic graph of e.g. ψ, illustrated by Fig. 4, we can see that starting from the top vertex and going first along the right loop then along the left one, we obtain a cyclic walk with trace $\varepsilon \cdot \varepsilon \cdot \bar{\mathbf{1}} \cdot \varepsilon \cdot \varepsilon \cdot \mathbf{1} = \varepsilon$.

As a more elaborate example, the formula

$$\chi = \mu(X = \langle \mathbf{2} \rangle Y \vee \langle \bar{\mathbf{2}} \rangle Y \vee \langle \bar{\mathbf{1}} \rangle X,\ Y = \langle \mathbf{1} \rangle X) \text{ in } X$$

is not cycle-free. Indeed, its graph has 3 simple cycles from the same starting point $\exp(\chi)$, with traces respectively $p_1 = \mathbf{2} \cdot \mathbf{1}$, $p_2 = \bar{\mathbf{2}} \cdot \mathbf{1}$ and $p_3 = \bar{\mathbf{1}}$, and we can see that $p_2 \cdot p_3 \cdot p_1 \cdot p_3 = \varepsilon$.

We are now ready to show a first result: in the finite focused-tree interpretation, the least and greatest fixpoints coincide for cycle-free formulas. To this end, we prove

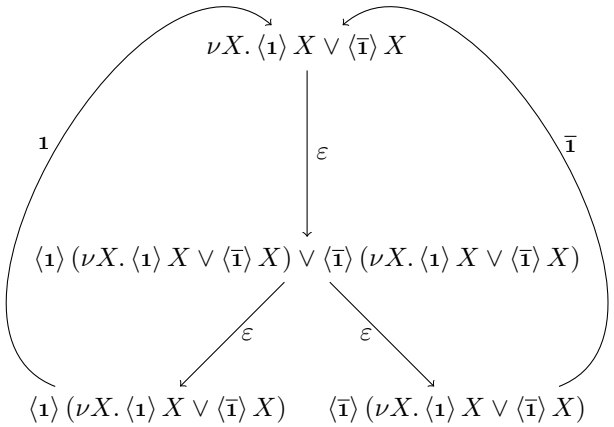


Fig. 4. Syntactic graph of $\psi = \nu X.\, \langle 1 \rangle\, X \vee \langle \overline{1} \rangle\, X$

$(\!|\varphi \vee \psi|\!)_V \stackrel{\text{def}}{=} (\!|\varphi|\!)_V \cup (\!|\psi|\!)_V$
$(\!|\varphi \wedge \psi|\!)_V \stackrel{\text{def}}{=} (\!|\varphi|\!)_V \cap (\!|\psi|\!)_V$
$(\!|\langle a \rangle\, \varphi|\!)_V \stackrel{\text{def}}{=} (\!|\varphi|\!)_V\, \langle \overline{a} \rangle$
$(\!|\iota(X_i = \varphi_i)_{i \in I} \text{ in } \psi|\!)_V \stackrel{\text{def}}{=} (\!|\psi|\!)_{V[\overline{X_i \mapsto U_i}]}$ where the U_i are defined as follows:

$$\text{let } F : \begin{array}{l} \mathcal{P}(\mathcal{F})^I \;\to\; \mathcal{P}(\mathcal{F})^I \\ (T_i)_{i \in I} \;\mapsto\; ((\!|\varphi_j|\!)_{V[\overline{X_i \mapsto T_i}]})_{j \in I} \end{array}$$
$$\text{and for all } j \in I,\ \text{let } U_j = \bigcup_{n \in \mathbb{N}} (F^n(\emptyset, \ldots, \emptyset))(j)$$

$(\!|\varphi|\!)_V = [\![\varphi]\!]_V$ in all other cases.

Fig. 5. Finite interpretation of formulas

a stronger result: the interpretation of a cycle-free formula is equal to its *finite interpretation*, defined in Fig. 5. The finite interpretation is identical to the normal interpretation for all formulas but fixpoints; for fixpoints, the valuation of the variables is obtained by iterating the function F starting from an empty valuation. We know from Lemma 3.1 and its proof that F is an increasing function with at least one fixpoint; given this, it is straightforward to see that iterating F starting from the smallest possible tuple, i. e. $(\emptyset \cdots \emptyset)$, gives an approximation of its least fixpoint from below. Therefore for any closed φ we have $(\!|\varphi|\!) \subseteq [\![\varphi]\!]$. We now prove that for cycle-free formulas, the converse inclusion also holds.

LEMMA 3.6. *Let φ be a cycle-free formula. Then $[\![\varphi]\!]_V = (\!|\varphi|\!)_V$ for any V.*

We organise the proof in auxiliary lemmas: first we show in Lemma 3.7 that $[\![\varphi]\!]_V$ can always be put into a sort of normal form, with constant parts and parts which depend on V, where the variable parts are of the form $V(X)\, \langle \overline{p} \rangle$, with p the trace of a walk from φ to X in the syntactic graph.

Then in Lemma 3.8 we show that for a fixpoint formula, by iterating that property, $F^n(U)$ can be put into a normal form where the variable parts are of the form

$U_i \langle \overline{p_1} \rangle \cdots \langle \overline{p_n} \rangle$, with the p_k traces of walks going from one to another binding of the fixpoint in the syntactic graph.

Finally, we conclude by noticing that the number of tree nodes visited by the sequences of paths $\langle \overline{p_1} \rangle \cdots \langle \overline{p_n} \rangle$ is unbounded if the formula is cycle-free. This implies that any given focused tree which belongs to $F^n(U)$ for all n, as is the case if U is a fixpoint of F, must fall in the constant part of the normal form for n large enough, which means it also belongs to $F^n(\emptyset, \dots \emptyset)$.

We now present the detail of this proof.

LEMMA 3.7. *Let φ be a formula with free variables $\{X_i \mid i \in I\}$. Then $(\!|\varphi|\!)_V$ is of the form $\bigcup_{k \in K} \left(A_k \cap \bigcap_{l \in L_k} V(X_{j_l}) \langle \overline{p_l} \rangle \right)$, where :*

— *K, the A_k, the L_k, the j_l and the p_l do not depend on V, and the L_k are finite. We also assume the L_k are all disjoint.*
— *for each l, there is a (possibly empty) walk w_l from φ to X_{j_l} such that $tr(w_l) = p_l$.*

We do not require K to be finite, and we consider an empty intersection to be equal to $\mathcal{F}$ and an empty union to $\emptyset$.

PROOF OF LEMMA 3.7. By structural induction on φ.

— If $\varphi = \top, \alpha, \neg\alpha$ or $\neg \langle a \rangle \top$, then $(\!|\varphi|\!)_V$ does not depend on V, so the result is immediate.

— If $\varphi = X$ then X is one of the X_i, and φ and X are the same vertex in the syntactic graph, so X is reachable from φ by following the empty walk. We have $(\!|\varphi|\!)_V = V(X) \langle \varepsilon \rangle$, so the result is again immediate.

— If $\varphi = \varphi_1 \vee \varphi_2$, the induction hypothesis tells us that $(\!|\varphi_1|\!)_V$ and $(\!|\varphi_2|\!)_V$ are of the appropriate union of intersections form. The union of these two unions is a larger union, of the form $\bigcup_{k \in K_1 \cup K_2} \left(A_k \cap \bigcap_{l \in L_k} V(X_{j_l}) \langle \overline{p_l} \rangle \right)$ where for each l there exists a walk w_l which starts either at φ_1 or at φ_2, which terminates at X_{j_l}, and whose trace is p_l. Since there are edges labelled ε from φ to both φ_1 and φ_2, by following the appropriate one and then w_l we obtain, for each l, a walk from φ to X_{j_l} whose trace is $\varepsilon \cdot p_l = p_l$.

— If $\varphi = \varphi_1 \wedge \varphi_2$, we use the induction hypothesis again and use distributivity to get $(\!|\varphi|\!)_V$ as a union of intersections again: we get $(\!|\varphi|\!)_V = \bigcup_{(k,k') \in K_1 \times K_2} \left(A_k \cap A_{k'} \cap \bigcap_{l \in L_k \cup L_{k'}} V(X_{j_l}) \langle \overline{p_l} \rangle \right)$. It is thus of the appropriate form and the reasoning on walks is then the same as in the previous case.

— If $\varphi = \langle a \rangle \psi$, then there is an edge from φ to ψ labelled a. By induction hypothesis we have $(\!|\psi|\!)_V = \bigcup_{k \in K} \left(A_k \cap \bigcap_{l \in L_k} V(X_{j_l}) \langle \overline{p_l} \rangle \right)$ where the p_l are traces of walks from ψ to X_{j_l}. This yields $(\!|\varphi|\!)_V = (\!|\psi|\!)_V \langle \overline{a} \rangle = \bigcup_{k \in K} \left(A_k \langle \overline{a} \rangle \cap \bigcap_{l \in L_k} V(X_{j_l}) \langle \overline{p_l} \rangle \langle \overline{a} \rangle \right)$. We conclude by remarking that :
— $V(X_{j_l}) \langle \overline{p_l} \rangle \langle \overline{a} \rangle = V(X_{j_l}) \langle \overline{p_l} \cdot \overline{a} \rangle \cap \mathcal{F} \langle \overline{p_l} \rangle \langle \overline{a} \rangle$ (see Remark 2.3).
— The sets $\mathcal{F} \langle \overline{p_l} \rangle \langle \overline{a} \rangle$ do not depend on V, nor do the $A_k \langle \overline{a} \rangle$. We can define $A'_k = A_k \langle \overline{a} \rangle \cap \bigcap_{l \in L_k} \mathcal{F} \langle \overline{p_l} \rangle \langle \overline{a} \rangle$.
— $\overline{p_l} \cdot \overline{a} = \overline{a \cdot p_l}$ and $a \cdot p_l$ is the trace of a walk from φ to X_{j_l}.

— If $\varphi = \iota(Y_i = \varphi_i)_{i \in J}$ in ψ, we consider, for all $i \in J$, the formulas $\psi_i = \iota(Y_j = \varphi_j)_{j \in J}$ in φ_i. They have the following property: $\exp(\psi_i) = \varphi_i\{({}^{\psi_k}/_{Y_k})_{k \in J}\}$. Thus, each walk from φ_i to an X_k straightfowardly translates into a walk from $\exp(\psi_i)$ to X_k, and each walk from φ_i to an Y_k straightforwardly translates into a walk from $\exp(\psi_i)$ to ψ_k. Furthermore, there is an ε-labelled edge from ψ_i to $\exp(\psi_i)$, so these walks can be prolongated to start from ψ_i while keeping the same trace.
We now proceed in several steps.

(1) We first apply the induction hypothesis to all the φ_i. The Y_i are free in these subformulas, so we get :

$$(\!|\varphi_i|\!)_V = \bigcup_{k\in K_i}\left(A_k \cap \bigcap_{l\in L_k} V(X_{j_l})\,\langle\overline{p_l}\rangle \cap \bigcap_{m\in M_k} V(Y_{j_m})\,\langle\overline{q_m}\rangle\right)$$

where for every $i \in J$ and every $k \in K_i$ there is:
— for every $l \in L_k$, a walk from φ_i to X_{j_l}, and hence also a walk from ψ_i to X_{j_l}, whose trace is p_l;
— for every $m \in M_k$, a walk from φ_i to Y_{j_m}n and hence also a walk from ψ_i to ψ_{j_m}, whose trace is q_m.
(2) We then show by induction on n that for all $i \in J$, $(F^n(\emptyset,\dots,\emptyset))(i)$ is of the form $\bigcup_{k'\in {K'}_i^n}\left(C_{k'} \cap \bigcap_{h\in H_{k'}} V(X_{j_h})\,\langle\overline{r_{i,h}}\rangle\right)$, where each $r_{i,h}$ is the trace of a walk from ψ_i to X_{j_h}:
For $n = 0$, we just take ${K'}_i^0 = \emptyset$ (the empty union yields the empty set). We now suppose that the property is true for n and for all i and pick $j \in J$. By definition, $(F^{n+1}(\emptyset,\dots,\emptyset))(j) = (\!|\varphi_j|\!)_{V[\overline{Y_i\mapsto(F^n(\emptyset,\dots,\emptyset))(i)}]}$. Combining step (1) with the induction hypothesis on n, we have that this set is of the form:

$$\bigcup_{k\in K_j}\left(A_k \cap \bigcap_{l\in L_k} V(X_{j_l})\,\langle\overline{p_l}\rangle \cap \bigcap_{m\in M_k}\bigcup_{k'\in {K'}_{j_m}^n}\left(C_{k'}\,\langle\overline{q_m}\rangle \cap \bigcap_{h\in H_{k'}} V(X_{j_h})\,\langle\overline{r_{j_m,h}}\rangle\,\langle\overline{q_m}\rangle\right)\right)$$

where:
— for each $k \in K_j$ and $l \in L_k$, there is a walk w_l from ψ_j to X_{j_l} such that $tr(w_l) = p_l$;
— for each $k \in K_j$ and $m \in M_k$, there is a walk v_m from ψ_j to ψ_{j_m} such that $tr(v_m) = q_m$;
— for each $i \in J$, $k' \in {K'}_i^n$ and $h \in H_{k'}$, there is a walk $u_{i,h}$ from ψ_i to X_{j_h} such that $tr(u_{i,h}) = r_{i,h}$.
Similarly to the case of the modality formula above, we use the equality $V(X_{j_h})\,\langle\overline{r_{j_m,h}}\rangle\,\langle\overline{q_m}\rangle = V(X_{j_h})\,\langle\overline{q_m\cdot r_{j_m,h}}\rangle \cap \mathcal{F}\,\langle\overline{r_{j_m,h}}\rangle\,\langle\overline{q_m}\rangle$.
We do some distributivity work. For $k \in K_j$, we define $K''_{k,n} = \prod_{m\in M_k} {K'}_{j_m}^n$. For a tuple $k'' \in K''_{k,n}$, we write $k''(m)$ for its component whose index is m, and we define $B_{k''} = \bigcap_{m\in M_k}\left(C_{k''(m)}\,\langle\overline{q_m}\rangle \cap \bigcap_{h\in H_{k''(m)}} \mathcal{F}\,\langle\overline{r_{j_m,h}}\rangle\,\langle\overline{q_m}\rangle\right)$. We obtain that $(F^{n+1}(\emptyset,\dots,\emptyset))(j)$ is of the correct form:

$$\bigcup_{k\in K_j\wedge k''\in K''_{k,n}}\left(A_k \cap B_{k''} \cap \bigcap_{l\in L_k} V(X_{j_l})\,\langle\overline{p_l}\rangle \cap \bigcap_{m\in M_k\wedge h\in H_{k''(m)}} V(X_{j_h})\,\langle\overline{q_m\cdot r_{j_m,h}}\rangle\right)$$

Furthermore, for each l we know that $p_l = tr(w_l)$ where w_l is a walk from ψ_j to X_{j_l}, and for each pair (m,h) we can obtain a walk from ψ_j to X_{j_h} whose trace is $q_m \cdot r_{j_m,h}$ by concatenating v_m and $u_{j_m,h}$. This concludes the induction on n.
(3) We conclude from this induction that the tuple $(U_j)_{j\in J}$ as defined in Fig. 5 is such that each U_j is of the form $\bigcup_{k\in\mathcal{K}_j}\left(C_k \cap \bigcap_{h\in H_k} V(X_{j_h})\,\langle\overline{r_{j,h}}\rangle\right)$, with $r_{j,h}$ the trace of a walk from ψ_j to X_{j_h} (by setting $\mathcal{K}_j = \bigcup_{n\in\mathbb{N}} {K'}_j^n$).
(4) We now apply the main induction hypothesis to ψ, to obtain:

$$(\!|\psi|\!)_V = \bigcup_{k\in K}\left(A_k \cap \bigcap_{l\in L_k} V(X_{j_l})\,\langle\overline{p_l}\rangle \cap \bigcap_{m\in M_k} V(Y_{j_m})\,\langle\overline{q_m}\rangle\right)$$

By applying the definition of $(\!|\bullet|\!)$ for a fixpoint formula, we obtain that $(\!|\varphi|\!)_V$ is:

$$\bigcup_{k\in K}\left(A_k \cap \bigcap_{l\in L_k} V(X_{j_l})\,\langle\overline{p_l}\rangle \cap \bigcap_{m\in M_k}\bigcup_{k'\in\mathcal{K}_{j_m}}\left(C_k\,\langle\overline{q_m}\rangle \cap \bigcap_{h\in H_{k'}} V(X_{j_h})\,\langle\overline{r_{j_m,h}}\rangle\,\langle\overline{q_m}\rangle\right)\right)$$

To conclude, we apply distributivity exactly as in step 2 above, noticing that:

— in the syntactic graph, there is an edge labelled ε from φ to $\psi\{({}^{\psi_i}/_{Y_i})_{i\in J}\}$. Therefore, for each m, since there is a walk from ψ to Y_{j_m} with trace q_m, there is also a walk w_m from φ to ψ_{j_m} with trace q_m;

— for each pair (m,h), concatenating this walk w_m with a walk from ψ_{j_m} to X_{j_h} whose trace is $r_{j_m,h}$ yields a walk from φ to X_{j_h} whose trace is $q_m \cdot r_{j_m,h}$;

— similarly, for each l, the walk from ψ to X_{j_l} with trace p_l yields a walk from φ to X_{j_l} with the same trace p_l. □

LEMMA 3.8. *Let $\varphi = \iota(X_i = \varphi_i)_{i\in I}$ in ψ be a closed fixpoint formula and let F be the function defined in Fig. 5. For each $j \in I$, we write $\psi_j \stackrel{def}{=} \iota(X_i = \varphi_i)_{i\in I}$ in φ_j. The ψ_j are closed formulas. For each pair $(i,j) \in I^2$, we consider the set of nonempty walks, in the syntactic graph of closed formulas, which go from ψ_i to ψ_j, and we write P_i^j for the corresponding set of* traces.

Then for any $n \in \mathbb{N}$ and for any $U \in \mathcal{P}(\mathcal{F})^I$, $F^n(U)$ is of the form :

$$\left(\bigcup_{k\in K_i} A_k \cap \bigcap_{l\in L_k} U_{j_l}\,\langle q_1^l\rangle\,\langle q_2^l\rangle\cdots\langle q_n^l\rangle\right)_{i\in I}$$

where :

— *the K_i, the A_k, the L_k and the q_m^l do not depend on U;*

— *for each l, there is a sequence of indices $(i_0^l,\ldots,i_n^l) \in I^n$, with $i_0^l = j_l$ and $i_n^l = i$, such that for all m between 1 and n, $\overline{q_m^l} \in P_{i_m^l}^{i_{m-1}^l}$.*

PROOF OF LEMMA 3.8. We prove the result by induction on n.

For $n = 0$, we have $F^0(U) = U$, which can be written $(\mathcal{F} \cap U_i)_{i\in I}$.

We now suppose the result true for n and show it for $n+1$. Let $j \in I$. We have $F^{n+1}(U)(j) = (\!|\varphi_j|\!)_{\{X_i\mapsto F^n(U)(i)\,|\,i\in I\}}$. Using Lemma 3.7, we have that this set is of the form $\bigcup_{k\in K}\left(A_k \cap \bigcap_{l\in L_k} F^n(U)(j_l)\,\langle\overline{p_l}\rangle\right)$, with p_l the trace of a possibly empty walk w_l from φ_j to X_{j_l}, which translates straightforwardly into a walk w_l' from $\varphi_j\{({}^{\psi_i}/_{X_i})_{i\in I}\}$ to ψ_{j_l}. Since there is an ε-labelled edge, in the syntactic graph, which goes from ψ_j to $\varphi_j\{({}^{\psi_i}/_{X_i})_{i\in I}\}$, we can construct a *nonempty* walk w_l'' from ψ_j to ψ_{j_l} by taking this edge first and then following w_l', and this walk still has trace p_l; hence $p_l \in P_j^{j_l}$.

Using now the induction hypothesis, we get:

$$\bigcup_{k\in K}\left(A_k \cap \bigcap_{l\in L_k}\bigcup_{k'\in K'_{j_l}}\left(A_{k'}\,\langle\overline{p_l}\rangle \cap \bigcap_{l'\in L_{k'}} U_{j_{l'}}\left\langle q_1^{l'}\right\rangle\left\langle q_2^{l'}\right\rangle\cdots\left\langle q_n^{l'}\right\rangle\langle\overline{p_l}\rangle\right)\right)$$

with, for all l', $i_0^{l'} = j_{l'}$ and $i_n^{l'} = j_l$ (with the i_m^l defined as in the lemma's statement).

We use distributivity to put this expression in the correct form: for each $k \in K$, let $K_k'' = \prod_{l\in L_k} K'_{j_l}$; for a tuple $k'' \in K_k''$, let $k''(l)$ be its component whose index is l. We

obtain that $F^{n+1}(U)(j)$ is:

$$\bigcup_{k\in K\wedge k''\in K''_k} A_k \cap \left(\bigcap_{l\in L_k} A_{k''(l)}\,\langle\overline{p_l}\rangle\right) \cap \bigcap_{l\in L_k\wedge l'\in L_{k''(l)}} U_{j_{l'}}\left\langle q_1^{l'}\right\rangle\left\langle q_2^{l'}\right\rangle\cdots\left\langle q_n^{l'}\right\rangle\langle\overline{p_l}\rangle$$

The big intersections are now indexed by pairs (l,l') such that $l\in L_k$ and $l'\in L_{k''(l)}$. We have to check that the property on paths is true for each such pair. Let $k\in K$ and let $l\in L_k$; $k''(l)$ is an element of K'_{j_l}, thus for $l'\in K_{k''(l)}$ we have $i_n^{l'}=j_l$; we also know that, for $m\leqslant n$, $\overline{q_m^{l'}}\in P_{i_m^{l'}}^{i_{m-1}^{l'}}$, and that $i_0^{l'}=j_{l'}$, which is what we want. The last component, p_l, has to belong to $P_j^{i_n^{l'}}$. We know from what precedes that $i_n^{l'}=j_l$ and that $p_l\in P_j^{j_l}$, so we can conclude the induction. □

PROOF OF LEMMA 3.6. We can now prove our main lemma. We do it by structural induction on φ. All cases are completely straightforward except the fixpoints, since the definitions of $[\![\varphi]\!]_V$ and $(\!|\varphi|\!)_V$ only differ for them. Thus we assume that φ is of the form $\iota(X_i=\varphi_i)_{i\in I}$ in ψ with $\iota=\mu$ or ν and that the property is true for ψ and the φ_i. The fact that it is true for the φ_i means that the function F defined in Fig. 5 is the same as the one defined in Lemma 3.1. This lemma furthermore tells us that we have $[\![\varphi]\!]_V=[\![\psi]\!]_{V[\overline{X_i\mapsto U_i}]}$ where $(U_i)_{i\in I}$ is a fixpoint of F, either the least or the greatest. By induction hypothesis, we have $[\![\psi]\!]_{V[\overline{X_i\mapsto U_i}]}=(\!|\psi|\!)_{V[\overline{X_i\mapsto U_i}]}$. What remains to prove is that for all $i\in I$ such that X_i appears free in ψ we have $U_i=\bigcup_{n\in\mathbb{N}}(F^n(\emptyset,\ldots,\emptyset))(i)$. (This is not necessarily true for *all* $i\in I$ because it is possible that some of the ψ_i are unreachable from φ in the syntactic graph, if the fixpoint formula contains unused bindings, in which case the constraint that φ is cycle-free does not impact the corresponding variables.)

As remarked earlier, the inclusion from right to left is easy. Since F is increasing, (U_i) is a fixpoint and $(\emptyset,\ldots,\emptyset)\leqslant(U_i)$, by a straightforward induction we have $F^n(\emptyset,\ldots,\emptyset)\leqslant(U_i)$ for any n.

We now have to prove that $U_i\subseteq\bigcup_{n\in\mathbb{N}}(F^n(\emptyset,\ldots,\emptyset))(i)$, or in other words, that for all $ft\in U_i$, there exists n such that $ft\in(F^n(\emptyset,\ldots,\emptyset))(i)$. So, let $ft\in U_i$. It is a finite tree, with $m>0$ nodes. This means that there are no more than m different tree paths p in P such that $ft\,\langle p\rangle$ is defined. Let $n=m\mathrm{Card}\ I$. Since (U_i) is a fixpoint of F, we have $ft\in(F^n(U))(i)$. From Lemma 3.8, we have both:

— $(F^n(U))(i)=\bigcup_{k\in K_i}A_k\cap\bigcap_{l\in L_k}U_{j_l}\left\langle q_1^l\right\rangle\left\langle q_2^l\right\rangle\cdots\left\langle q_n^l\right\rangle$ where the q_m^l have the properties described in the lemma, and
— $(F^n(\emptyset,\ldots,\emptyset))(i)=\bigcup_{k\in K_i|L_k=\emptyset}A_k$. Indeed, $\emptyset\,\langle q\rangle$ is always empty and the only nonempty intersection of empty sets is the empty intersection.

Thus, to show that $ft\in(F^n(\emptyset,\ldots,\emptyset))(i)$, it suffices to prove that for any l, $ft\notin U_{j_l}\left\langle q_1^l\right\rangle\left\langle q_2^l\right\rangle\cdots\left\langle q_n^l\right\rangle$.

Suppose we do in fact have $ft\in U_{j_l}\left\langle q_1^l\right\rangle\left\langle q_2^l\right\rangle\cdots\left\langle q_n^l\right\rangle$ for some l. We prove that in that case, φ is not cycle-free. For h between 0 and n, let $p_h=q_{h+1}^l\cdot q_{h+2}^l\cdots q_n^l$ (we adopt the convention that $p_n=\varepsilon$). Then for any h, $ft\,\langle\overline{p_h}\rangle$ must be defined. Let $(i_0^l,\ldots,i_n^l)$ be the sequence of indices associated to the q_h^l as per Lemma 3.8. The length of this sequence is $n+1$. Since there are only $\mathrm{Card}\ I$ possible indices and $n=m\mathrm{Card}\ I$, at least one of the indices, say j, appears $m+1$ times in the sequence, say at $h_0\ldots h_m$ in increasing order. Since all the $ft\,\langle\overline{p_h}\rangle$ are defined and ft has only m nodes, two of the p_{h_x} must be

equal (since there are $m+1$ of them). Suppose they are p_{h_x} and p_{h_y} with $x < y$. We have $p_{h_x} = q^l_{h_x+1} \cdot q^l_{h_x+2} \cdots q^l_{h_y} \cdot p_{h_y}$; therefore $q^l_{h_x+1} \cdot q^l_{h_x+2} \cdots q^l_{h_y} = \varepsilon$. But this path is the trace of a nonempty (since $x < y$) walk from ψ_j to ψ_j, so ψ_j is not cycle-free. Furthermore, ψ_j is reachable, in the syntactic graph, from ψ_i since it appears in the sequence, and ψ_i is reachable from φ since we assumed X_i appeared free in ψ. Thus φ is not cycle-free either. □

An important consequence of Lemma 3.6 is that the negation of a cycle-free least fixpoint formula is itself a least fixpoint, so that the subset of the logic consisting of closed cycle-free formulas with only least fixpoints is closed under negation. In the rest of the paper, we only consider this subset.

4. SATISFIABILITY-TESTING ALGORITHM

In this section, we present an algorithm to test the satisfiability of a cycle-free closed formula of $\mathcal{L}_\mu$. To simplify the approach, we restrict ourselves to checking whether a formula is satisfiable by a focused tree *whose context is* Top. This can be done without loss of generality, as the following lemma shows.

LEMMA 4.1. *Let φ be a cycle-free closed formula. Let $\psi = \mu X.\varphi \vee \langle \mathbf{1} \rangle X \vee \langle \mathbf{2} \rangle X$. Then $[\![\varphi]\!] \neq \emptyset$ if and only if there exists a focused tree $\tau_{\mathsf{Top}} \in [\![\psi]\!]$.*

PROOF. Suppose φ is unsatisfiable (i.e. $[\![\varphi]\!] = \emptyset$). Then $\emptyset$ is a fixpoint of the function F associated to ψ, hence $[\![\psi]\!] = \emptyset$.

Conversely, suppose φ is satisfiable. By definition of $[\![\psi]\!]$, we have: $[\![\varphi]\!] \subseteq [\![\psi]\!]$, and $\forall ft \in [\![\psi]\!], (ft\,\langle \bar{\mathbf{1}} \rangle \text{ defined } \Rightarrow ft\,\langle \bar{\mathbf{1}} \rangle \in [\![\psi]\!]) \wedge (ft\,\langle \bar{\mathbf{2}} \rangle \text{ defined } \Rightarrow ft\,\langle \bar{\mathbf{2}} \rangle \in [\![\psi]\!])$. Since φ is satisfiable, $[\![\psi]\!]$ is not empty. Let $ft \in [\![\psi]\!]$. Because of the structure of focused trees, either the context of ft is Top, or one of $ft\,\langle \bar{\mathbf{1}} \rangle$ or $ft\,\langle \bar{\mathbf{2}} \rangle$ is defined, which means $[\![\psi]\!]$ contains an element with a strictly smaller context than ft. By induction, since contexts are finite, $[\![\psi]\!]$ contains at least one element with context Top. □

In the rest of this section, we always assume ψ to be the 'plunged formula' built in this way from the formula φ whose satisfiability we want to decide. The algorithm only looks for models of ψ whose context is Top.

4.1. Preliminary Definitions

We call $\Sigma(\psi)$ the set of atomic propositions α used in ψ.

We consider the Fisher-Ladner closure of ψ, $\mathbf{cl}(\psi)$ (see Def. 3.3). Every formula $\varphi \in \mathbf{cl}(\psi)$ can be seen as a Boolean combination of formulas of a set called the Lean of ψ, inspired from [Pan et al. 2006]. We define it as follows:

$$\mathbf{Lean}(\psi) = \{\langle a \rangle \top \mid a \in \{\mathbf{1}, \mathbf{2}, \bar{\mathbf{1}}, \bar{\mathbf{2}}\}\} \cup \Sigma(\psi) \cup \{\langle a \rangle \varphi \mid \langle a \rangle \varphi \in \mathbf{cl}(\psi)\}$$

A *ψ-type* (or simply a "*type*") (Hintikka set in the temporal logic literature) is a set $t \subseteq \mathbf{Lean}(\psi)$ such that:

— $\forall \langle a \rangle \varphi \in \mathbf{Lean}(\psi), \langle a \rangle \varphi \in t \Rightarrow \langle a \rangle \top \in t$ (modal consistency);
— $\langle \bar{\mathbf{1}} \rangle \top \notin t \vee \langle \bar{\mathbf{2}} \rangle \top \notin t$ (a tree node cannot be both a first child and a second child).

We call $\mathbf{Types}(\psi)$ the set of ψ-types. For a ψ-type t, the *complement* of t is the set $\mathbf{Lean}(\psi) \setminus t$ (which is usually not itself a type).

A type intuitively represents the conjunction of all formulas in t together with the negations of all formulas in its complement (however this conjunction formula is never built or even considered as such in the algorithm: only the sets are manipulated).

$$\frac{}{\top \Longleftarrow (\emptyset,\emptyset)} \qquad \frac{\varphi \in \mathrm{Lean}(\psi)}{\varphi \Longleftarrow (\{\varphi\},\emptyset)} \qquad \frac{\varphi_1 \Longleftarrow (T_1,F_1) \qquad \varphi_2 \Longleftarrow (T_2,F_2)}{\varphi_1 \wedge \varphi_2 \Longleftarrow (T_1 \cup T_2, F_1 \cup F_2)}$$

$$\frac{\varphi_1 \Longleftarrow (T_1,F_1)}{\varphi_1 \vee \varphi_2 \Longleftarrow (T_1,F_1)} \qquad \frac{\varphi_2 \Longleftarrow (T_2,F_2)}{\varphi_1 \vee \varphi_2 \Longleftarrow (T_2,F_2)} \qquad \frac{\varphi \in \mathrm{Lean}(\psi)}{\neg\varphi \Longleftarrow (\emptyset,\{\varphi\})}$$

$$\frac{\mathbf{exp}(\mu(X_i = \varphi_i) \text{ in } \psi) \Longleftarrow (T,F)}{\mu(X_i = \varphi_i) \text{ in } \psi \Longleftarrow (T,F)}$$

Fig. 6. Truth assignment of a formula

Fig. 6 gives derivation rules defining a binary relation between formulas $\varphi \in \mathrm{cl}(\psi)$ and pairs (T,F) of subsets of $\mathrm{Lean}(\psi)$. The meaning of $\varphi \Longleftarrow (T,F)$ is that whenever all formulas in T are true and all formulas in F are false, φ is true.

Because we have restricted ourselves to cycle-free formulas, the derivations in Fig. 6 are finite. Indeed, the formulas in the premises are always reachable from the formula in the conclusion following an ε-labelled edge in the syntactic graph. It can therefore not loop, and we know that $\mathrm{cl}(\psi)$ is finite.

Definition 4.2. Let $\varphi \in \mathrm{cl}(\psi)$, and let $t \in \mathrm{Types}(\psi)$. We say that φ is true at type t or that t implies φ, written $\varphi \mathrel{\dot\in} t$, if there exist (T,F) such that $\varphi \Longleftarrow (T,F)$ and $T \subseteq t \subseteq \mathrm{Lean}(\psi) \setminus F$.

$\varphi \mathrel{\dot\in} t$ means that whenever all formulas in t are true and all formulas in its complement are false, φ is true.

We next define a compatibility relation between types to state that two types are related according to a modality.

Definition 4.3 (*Compatibility relation*). Two types t and t' are *compatible* under $a \in \{\mathbf{1},\mathbf{2},\overline{\mathbf{1}},\overline{\mathbf{2}}\}$, written $\Delta_a(t,t')$, iff

$$\forall \langle a\rangle \varphi \in \mathrm{Lean}(\psi), \langle a\rangle \varphi \in t \Leftrightarrow \varphi \mathrel{\dot\in} t'$$
$$\forall \langle \overline{a}\rangle \varphi \in \mathrm{Lean}(\psi), \langle \overline{a}\rangle \varphi \in t' \Leftrightarrow \varphi \mathrel{\dot\in} t$$

Note that we have $\Delta_a(t,t') \Leftrightarrow \Delta_{\overline{a}}(t',t)$. Only Δ_1 and Δ_2 are used in the algorithm.

4.2. The Algorithm

4.2.1. General idea. Recall that the models of formulas are *focused trees* consisting of a tree and a context. We say that a formula is 'partially satisfied' by a tree if, informally, the tree would satisfy the formula provided an appropriate context is added (such a context may however not exist).

The algorithm works by enumerating all partially satisfiable ψ-types, in a bottom-up order: it first considers all types which are partially satisfied by leaves, then incrementally uses the already known partially satisfiable types and the compatibility relation Δ to find types which require a deeper tree to be partially satisfied, as illustrated by Fig. 7.

Whenever a type that does not contain $\langle\overline{\mathbf{2}}\rangle \top$ nor $\langle\overline{\mathbf{1}}\rangle \top$ is found to be partially satisfiable, then it is actually satisfiable (by adding Top as the context). If such a type furthermore implies ψ ($\psi \mathrel{\dot\in} t$), then ψ is satisfied by at least a focused tree with context Top, which is what we want.

If no such type is found, the algorithm stops when it has enumerated all partially satisfiable types and concludes that ψ does not have a model of the form τ_{Top}.

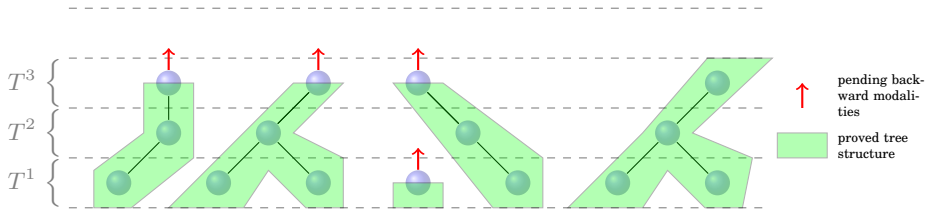


Fig. 7. Algorithm's principle: progressive bottom-up reasoning.

4.2.2. Formal description. The main iteration is as follows:

$X \leftarrow \emptyset$
repeat
  $X' \leftarrow X$
  $X \leftarrow \texttt{Upd}(X')$
  if $\texttt{FinalCheck}(X)$ **then**
    return "ψ is satisfiable"
until $X = X'$
return "ψ is unsatisfiable"

where X is the set of partially satisfiable types found so far and the update operation $\texttt{Upd}(\cdot)$ and success check operation $\texttt{FinalCheck}(\cdot)$ are defined as follows.

$$\texttt{Upd}(X) \stackrel{\text{def}}{=} X \cup \left\{ t \in \textbf{Types}(\psi) \;\middle|\; \begin{array}{l} \langle \mathbf{1} \rangle \top \in t \;\Rightarrow\; \exists t_1 \in X, \Delta_1(t, t_1) \wedge \\ \langle \mathbf{2} \rangle \top \in t \;\Rightarrow\; \exists t_2 \in X, \Delta_2(t, t_2) \end{array} \right\}$$

$$\texttt{FinalCheck}(X) \stackrel{\text{def}}{=} \exists t \in X, \psi \,\dot{\in}\, t \wedge \langle \bar{\mathbf{1}} \rangle \top \notin t \wedge \langle \bar{\mathbf{2}} \rangle \top \notin t$$

The update operation constructs new types from the existing ones, using the Δ relations. At each step of the algorithm, $\texttt{FinalCheck}(X)$ verifies whether the tested formula is implied by newly added types without pending (unproved) backward modalities, so that the algorithm may terminate as soon as a satisfying tree is found.

We call X^i the set X after i iterations, and T^i the set of types added to X at the ith iteration (so $T^i = X^i \setminus X^{i-1}$). Remark that the types in T^i can only be partially satisfied by trees of depth at least i (otherwise they would have been added earlier).

4.2.3. Example Run of the Algorithm. To illustrate the algorithm, imagine we want to test the satisfiability of the formula:

$$\varphi = \langle \bar{\mathbf{1}} \rangle \, \alpha \wedge \neg \langle \mathbf{1} \rangle \, \alpha \wedge \langle \mathbf{1} \rangle \, \mu Z.\alpha \vee \langle \mathbf{2} \rangle \, Z.$$

We write η for the subformula $\mu Z.\alpha \vee \langle \mathbf{2} \rangle \, Z$.

First, $\neg \langle \mathbf{1} \rangle \, \alpha$ is converted into the core syntax and the plunged formula ψ is computed:

$$\psi = \mu X. \, \langle \mathbf{1} \rangle \, X \vee \langle \mathbf{2} \rangle \, X \vee (\langle \bar{\mathbf{1}} \rangle \, \alpha \wedge (\langle \mathbf{1} \rangle \, \neg\alpha \vee \neg \langle \mathbf{1} \rangle \, \top) \wedge \langle \mathbf{1} \rangle \, \eta).$$

Then $\text{Lean}(\psi)$ is computed:

$$\text{Lean}(\psi) = \{\langle \mathbf{1} \rangle \top, \langle \mathbf{2} \rangle \top, \langle \bar{\mathbf{1}} \rangle \top, \langle \bar{\mathbf{2}} \rangle \top, \alpha, \langle \mathbf{1} \rangle \, \psi, \langle \mathbf{2} \rangle \, \psi, \langle \bar{\mathbf{1}} \rangle \, \alpha, \langle \mathbf{1} \rangle \, \neg\alpha, \langle \mathbf{1} \rangle \, \eta, \langle \mathbf{2} \rangle \, \eta\}$$

We then enter the main iteration with an initially empty set X^0:

$\texttt{Upd}(X^0)$: The types added during the first pass are all those which do not contain $\langle \mathbf{1} \rangle \top$ nor $\langle \mathbf{2} \rangle \top$. Because we only consider types and not arbitrary subsets of the lean, they also do not contain any formula starting with $\langle \mathbf{1} \rangle$ or $\langle \mathbf{2} \rangle$; furthermore, a type cannot contain $\langle \bar{\mathbf{1}} \rangle \alpha$ without $\langle \bar{\mathbf{1}} \rangle \top$, and cannot contain both $\langle \bar{\mathbf{1}} \rangle \top$ and $\langle \bar{\mathbf{2}} \rangle \top$. Given all this, we can find 8 types that are partially satisfied by leaves:

$$X^1 = \{\emptyset, \{\alpha\}, \{\langle \bar{\mathbf{1}} \rangle \top\}, \{\langle \bar{\mathbf{2}} \rangle \top\}, \{\alpha, \langle \bar{\mathbf{1}} \rangle \top\}, \{\alpha, \langle \bar{\mathbf{2}} \rangle \top\}, \{\langle \bar{\mathbf{1}} \rangle \alpha, \langle \bar{\mathbf{1}} \rangle \top\}, \{\alpha, \langle \bar{\mathbf{1}} \rangle \alpha, \langle \bar{\mathbf{1}} \rangle \top\}\}$$

$\texttt{FinalCheck}(X^1)$: We have:

$$\psi \mathrel{\dot\in} t \text{ iff either } \begin{cases} \langle \mathbf{1} \rangle \psi \in t \text{ or} \\ \langle \mathbf{2} \rangle \psi \in t \text{ or} \\ \langle \bar{\mathbf{1}} \rangle \alpha \in t \text{ and } \langle \mathbf{1} \rangle \eta \in t \text{ and [either] } \langle \mathbf{1} \rangle \neg\alpha \in t \text{ [or } \langle \mathbf{1} \rangle \top \notin t\text{]}^5 \end{cases}$$

Given this, we can see that there are no types t in X^1 such that $\psi \mathrel{\dot\in} t$, so we continue iterating.

$\texttt{Upd}(X^1)$: Since X is not empty anymore, we now need to compute the Δ relations. We have:

$$\eta \mathrel{\dot\in} t \text{ iff either } \begin{cases} \alpha \in t \text{ or} \\ \langle \mathbf{2} \rangle \eta \in t \end{cases}$$

As remarked before, no types in X^1 are such that $\psi \mathrel{\dot\in} t$. However there are types such that $\eta \mathrel{\dot\in} t$: they are exactly all the types containing α.

One example of type in T^2 is $t_2 = \{\langle \mathbf{2} \rangle \top, \langle \mathbf{2} \rangle \alpha, \langle \mathbf{2} \rangle \eta, \langle \bar{\mathbf{1}} \rangle \top\}$. Indeed, $t_1 = \{\alpha, \langle \bar{\mathbf{2}} \rangle \top\}$ is such that $\Delta_2(t_2, t_1)$ holds.

$\texttt{FinalCheck}(X^2)$: X^2 is too big to write here, but from what we remarked about X^1, we know that no type in X^2 can contain $\langle \mathbf{1} \rangle \psi$ or $\langle \mathbf{2} \rangle \psi$ or both $\langle \mathbf{1} \rangle \eta$ and $\langle \mathbf{1} \rangle \neg\alpha$; hence there is still no type in X^2 such that $\psi \mathrel{\dot\in} t$.

$\texttt{Upd}(X^2)$: At the third iteration, one of the types which will be added is $t_3 = \{\langle \mathbf{1} \rangle \top, \langle \mathbf{1} \rangle \neg\alpha, \langle \mathbf{1} \rangle \eta, \langle \bar{\mathbf{1}} \rangle \top, \langle \bar{\mathbf{1}} \rangle \alpha\}$. Indeed, we have $\Delta_1(t_3, t_2)$.

$\texttt{FinalCheck}(X^3)$: We can check that $\psi \mathrel{\dot\in} t_3$; however, t_3 contains a pending backward modality $\langle \bar{\mathbf{1}} \rangle \top$, so we cannot conclude yet. Since ψ was not true at any type in X^2, there are no types in X^3 which contain either $\langle \mathbf{1} \rangle \psi$ or $\langle \mathbf{2} \rangle \psi$, so `FinalCheck` still fails.

$\texttt{Upd}(X^3)$: Finally, at the fourth iteration, the type $t_4 = \{\alpha, \langle \mathbf{1} \rangle \top, \langle \mathbf{1} \rangle \psi\}$ will be added since $\Delta_1(t_4, t_3)$ holds. This type has now been proved partially satisfiable, and since it has no pending backward modalities, it is satisfiable.

$\texttt{FinalCheck}(X^4)$: We also have $\psi \mathrel{\dot\in} t_4$ and therefore the algorithm now concludes that ψ is satisfiable, and thus the initial formula φ as well.

4.3. Correctness and Complexity

In this section we prove the correctness of the satisfiability testing algorithm, and show that its time complexity is $2^{O(|\text{Lean}(\psi)|)}$.

THEOREM 4.4 (CORRECTNESS). *The algorithm decides satisfiability of cycle-free closed $\mathcal{L}_\mu$ formulas over finite focused trees.*

[5] the case between square brackets is actually incompatible with $\langle \mathbf{1} \rangle \eta \in t$ since t is a type.

4.3.1. Termination. For $\psi \in \mathcal{L}_\mu$, since $\mathbf{cl}(\psi)$ is a finite set, $\text{Lean}(\psi)$ and $\mathcal{P}(\text{Lean}(\psi))$ are also finite. Furthermore, $\texttt{Upd}(\cdot)$ is monotonic and $X \subseteq \mathcal{P}(\text{Lean}(\psi))$, therefore the algorithm terminates.

To finish the proof, it thus suffices to prove that: whenever the algorithm answers SAT, the formula is indeed satisfiable (soundness); and whenever the formula is satisfiable, the algorithm correctly answers SAT (completeness).

4.3.2. Soundness. We first relate formally the $\Longleftarrow$ relation to our semantics.

PROPOSITION 4.5.
For $\varphi \in cl(\psi)$, if $\varphi \Longleftarrow (T, F)$, then we have $\bigcap_{\chi \in T} [\![\chi]\!] \setminus \bigcup_{\chi \in F} [\![\chi]\!] \subseteq [\![\varphi]\!]$.

PROOF. Immediate by induction on the derivation yielding $\varphi \Longleftarrow (T, F)$. □

Remark 4.6. Whenever $\varphi \Longleftarrow (T, F)$ holds, F does not contain any formula of the form $\langle a \rangle \chi$ with $\chi \neq \top$; indeed, it would imply that φ contains a subformula of the form $\neg \langle a \rangle \chi$, which is not part of the core syntax (it is only defined as syntactic sugar).

We call 'negatable' the formulas in $\text{Lean}(\psi)$ which are not of this form, i.e. the formulas in $\{\langle a \rangle \top \mid a \in \{\mathbf{1}, \mathbf{2}, \overline{\mathbf{1}}, \overline{\mathbf{2}}\}\} \cup \Sigma(\psi)$.

We now define two structures, trees of types and dependency graphs, which will allow us to reason about the sets of types the algorithm builds, by making explicit the links between the types induced by the Δ relations.

A *forward path* is a path that only mentions forward modalities.

We define a tree of types $\mathcal{T}$ as a binary tree whose nodes are types. We write $\mathcal{T}(\cdot)$ for the type at the root of the tree; $\mathcal{T}\langle \mathbf{1} \rangle$ and $\mathcal{T}\langle \mathbf{2} \rangle$ are respectively the left and right subtrees if they exist, and more generally, for a forward path p, $\mathcal{T}\langle p \rangle$ is the subtree reached by navigating in $\mathcal{T}$ from the root following path p. A tree of types is *consistent* iff for every forward path p and every program $a \in \{\mathbf{1}, \mathbf{2}, \overline{\mathbf{1}}, \overline{\mathbf{2}}\}$:

— $\langle a \rangle \top \in \mathcal{T}\langle p \rangle(\cdot)$ implies that $\mathcal{T}\langle p \cdot a \rangle$ is defined, and
— if $\mathcal{T}\langle p \rangle$ and $\mathcal{T}\langle p \cdot a \rangle$ are both defined then $\Delta_a(\mathcal{T}\langle p \rangle(\cdot), \mathcal{T}\langle p \cdot a \rangle(\cdot))$ holds.

The first condition notably implies that if $\mathcal{T}$ is consistent, then its root node $\mathcal{T}(\cdot)$ contains neither $\langle \overline{\mathbf{1}} \rangle \top$ nor $\langle \overline{\mathbf{2}} \rangle \top$.

This structure is related to the result of the algorithm in the following way:

LEMMA 4.7. *Let $t \in X^n$ such that $\langle \overline{\mathbf{1}} \rangle \top \notin t$ and $\langle \overline{\mathbf{2}} \rangle \top \notin t$. Then there exists a consistent tree of types $\mathcal{T}$ such that $\mathcal{T}(\cdot) = t$.*

PROOF. We can show by a straightforward induction on n the slightly different property that *for all* $t \in X^n$, there exists a tree of types $\mathcal{T}$, such that $\mathcal{T}(\cdot) = t$, which satisfies all the consistency conditions *except* possibly, at the root node and for $a \in \{\mathbf{1}, \mathbf{2}\}$, the one which says $\mathcal{T}\langle \overline{a} \rangle$ must exist if $\langle \overline{a} \rangle \top \in \mathcal{T}(\cdot)$. This corresponds to the possible 'pending backward modalities'.

Then we just have to notice that the hypotheses $\langle \overline{\mathbf{1}} \rangle \top \notin t$ and $\langle \overline{\mathbf{2}} \rangle \top \notin t$ make that last consistency condition satisfied. □

Given a consistent tree of types $\mathcal{T}$, we define a dependency graph whose nodes are all pairs (p, φ) where p is a path such that $\mathcal{T}\langle p \rangle$ is defined and φ is either a formula in $t = \mathcal{T}\langle p \rangle(\cdot)$ or the negation of a negatable formula in the complement of t. The directed edges of the graph are labelled with modalities consistent with the tree. For every (p, φ) in the nodes we build the following outgoing edges:

— $\varphi \in \Sigma(\psi) \cup \neg\Sigma(\psi) \cup \{\langle a \rangle \top, \neg \langle a \rangle \top\}$: no edge

— $\varphi = \langle a \rangle \varphi'$, with $\varphi' \neq \top$: let $t = \mathcal{T}\langle p \rangle(\cdot)$. We have $\langle a \rangle \varphi' \in t$ and t is a type, therefore $\langle a \rangle \top \in t$; thus, since $\mathcal{T}$ is consistent, $\mathcal{T}\langle p \cdot a \rangle$ is defined: let $t' = \mathcal{T}\langle p \cdot a \rangle(\cdot)$. We have $\Delta_a(t, t')$, which implies $\varphi' \dot{\in} t'$, hence there exist T, F such that $\varphi' \Longleftarrow (T, F)$ with T a subset of t', and F a set of negatable formulas not in t'. For every $\varphi_T \in T$ we add an a-labelled edge to $(p \cdot a, \varphi_T)$, and for every $\varphi_F \in F$ we add an a-labelled edge to $(p \cdot a, \neg\varphi_F)$.

LEMMA 4.8. *The dependency graph of a consistent tree of ψ-types (where ψ is cycle-free) is cycle-free.*

PROOF. First notice that whenever we have $\varphi \Longleftarrow (T, F)$, all the formulas in T and all the $\neg\chi$ for $\chi \in F$ are reachable from φ, in the syntactic graph, following only ε-labelled edges. Thus any walk in the dependency graph corresponds to a walk with the same trace in the syntactic graph.

Furthermore, whenever there is a walk in the dependency graph from (φ, p) to (φ', p'), its trace is $\overline{p} \cdot p'$ (straightforward induction).

Thus if there were a cycle in the dependency graph, i.e. a nonempty walk from (φ, p) to (φ, p), then there would be a nonempty walk in the syntactic graph from φ to φ with trace ε, thus φ would not be cycle-free. But all φ in the dependency graph belong to $\mathsf{cl}(\psi)$ and ψ is cycle-free, hence they must be cycle-free as well; thus there cannot be a cycle in the dependency graph. □

LEMMA 4.9. *Let $\mathcal{T}$ be a consistent tree of ψ-types. Then there exists a focused tree ft whose context is* Top *and such that, for all nodes (p, φ) in the dependency graph of $\mathcal{T}$, $ft\langle p \rangle$ is defined and $ft\langle p \rangle \in [\![\varphi]\!]$.*

PROOF. We first describe how ft is built: first, inductively, we associate a tree to each node of $\mathcal{T}$, starting from the leaves. The tree associated with $\mathcal{T}\langle p \rangle$ is (L, st_1, st_2), where:

— $L = \mathcal{T}\langle p \rangle(\cdot) \cap \Lambda$
— st_a is either the tree associated with $\mathcal{T}\langle p \cdot a \rangle$ if it is defined or nil if it is not.

We then add context Top to the tree associated with $\mathcal{T}$ to obtain our focused tree ft.

We now show that this ft satisfies the condition we want, by induction on the dependency graph, which we know is finite and has no cycles. More precisely, we show by a case analysis on φ that if the property is true of all nodes directly reachable from (p, φ), then it is true of (p, φ) as well:

— $\varphi = \alpha$. In this case, $\alpha \in \mathcal{T}\langle p \rangle(\cdot) \cap \Lambda$, thus by construction of ft we have $ft\langle p \rangle \in [\![\alpha]\!]$.
— $\varphi = \neg\alpha$. In this case, $\alpha \notin \mathcal{T}\langle p \rangle(\cdot) \cap \Lambda$, thus again by construction we have $ft\langle p \rangle \notin [\![\alpha]\!]$.
— $\varphi = \langle a \rangle \top$ (resp. $\varphi = \neg\langle a \rangle \top$). Then by consistency of $\mathcal{T}$, $\mathcal{T}\langle p \cdot a \rangle$ exists (resp. does not exist), and by construction of ft, so does $ft\langle p \cdot a \rangle$.
— $\varphi = \langle a \rangle \varphi'$. Then $\mathcal{T}\langle p \rangle(\cdot)$ must contain $\langle a \rangle \top$ as well, and by the argument above $\mathcal{T}\langle p \cdot a \rangle$ and $ft\langle p \cdot a \rangle$ exist. We know from the construction of the dependency graph that there exist T and F such that $\varphi' \Longleftarrow (T, F)$ and that (p, φ) has edges going to all nodes $(p \cdot a, \chi)$ for $\chi \in T$ and all nodes $(p \cdot a, \neg\chi)$ for $\chi \in F$. By induction hypothesis, for all χ in T we have $ft\langle p \cdot a \rangle \in [\![\chi]\!]$, and for all χ in F we have $ft\langle p \cdot a \rangle \notin [\![\chi]\!]$. By Proposition 4.5, this implies $ft\langle p \cdot a \rangle \in [\![\varphi']\!]$, hence $ft\langle p \rangle \in [\![\varphi]\!]$. □

LEMMA 4.10 (SOUNDNESS). *If there exists t in X^n such that $\psi \dot{\in} t$, $\langle \overline{1} \rangle \top \notin t$ and $\langle \overline{2} \rangle \top \notin t$, then there exists a focused tree ft, whose context is* Top*, such that $ft \in [\![\psi]\!]$.*

PROOF. We know from Lemma 4.7 that there exists a consistent tree of ψ-types $\mathcal{T}$ such that $\mathcal{T}(\cdot) = t$. Furthermore, we know from Lemma 4.9 that there exists a focused tree ft whose context is Top and such that for all nodes (p, φ) in the dependency graph

of $\mathcal{T}$, $ft\,\langle p\rangle \in [\![\varphi]\!]$. We consider the particular case $p = \varepsilon$: there are nodes (ε, φ) for all φ in t and for the negations of all negatable formulas in $\mathrm{Lean}(\psi) \setminus t$, and for all these nodes we have $ft \in [\![\varphi]\!]$. Since we have $\psi \mathrel{\dot\in} t$, we can conclude from Proposition 4.5. □

4.3.3. Completeness

LEMMA 4.11 (COMPLETENESS). *If there exists a tree τ such that $\tau_{\mathsf{Top}} \in [\![\psi]\!]$, then there exists n such that $\mathtt{FinalCheck}(X^n)$ holds.*

PROOF. Let $N = \{\tau_{\mathsf{Top}}\,\langle p\rangle \mid p \in P\}$, the set of nodes in τ. For all ft in N, we define $\mathtt{type}(ft) \stackrel{\text{def}}{=} \{\varphi \in \mathrm{Lean}(\psi) \mid ft \in [\![\varphi]\!]\}$. Straightforwardly, $\mathtt{type}(ft)$ is a type. We call *depth* of a focused tree ft the length of the longest forward path p such that $ft\,\langle p\rangle$ is defined. We prove the lemma in several steps:

(1) We first show that for all $\varphi \in \mathrm{cl}(\psi)$ and all $ft \in N$, if $ft \in [\![\varphi]\!]$ then $\varphi \mathrel{\dot\in} \mathtt{type}(ft)$. For this, we consider the syntactic graph of ψ and remove all edges not labelled ε (i.e. which leave a formula of the form $\langle a\rangle\,\varphi$). The resulting graph is typically no longer connected, but has no cycles since ψ is cycle-free, hence we can reason by induction, showing that the property is true for a formula if it is true for all formulas directly reachable from it in this subgraph. We reason by cases on φ:
— if $\varphi = \top$, the result is immediate since $\varphi \mathrel{\dot\in} t$ is always true.
— if $\varphi = \bot$, the result is trivially true since there is no $ft \in [\![\varphi]\!]$.
— if φ is an atomic proposition or of the form $\langle a\rangle\,\varphi'$, then $\varphi \in \mathrm{Lean}(\psi)$, by definition of $\mathrm{Lean}(\psi)$. Then $ft \in [\![\varphi]\!]$ implies $\varphi \in \mathtt{type}(ft)$, by definition of this set, and we also have $\varphi \Longleftarrow (\{\varphi\}, \emptyset)$, thus $\varphi \mathrel{\dot\in} \mathtt{type}(ft)$.
— if $\varphi = \neg\varphi'$, with φ' either an atomic proposition or of the form $\langle a\rangle\,\top$, then $\varphi' \in \mathrm{Lean}(\psi)$ and $\varphi \Longleftarrow (\emptyset, \{\varphi'\})$. If $ft \notin [\![\varphi']\!]$, we have $\varphi' \notin \mathtt{type}(ft)$, thus $\varphi \mathrel{\dot\in} \mathtt{type}(ft)$.
— the other cases are straightforward from the induction hypothesis.

(2) We then show that for all $ft \in N$ and $a \in \{\mathtt{1}, \mathtt{2}\}$, if $ft\,\langle a\rangle$ is defined then $\Delta_a(\mathtt{type}(ft), \mathtt{type}(ft\,\langle a\rangle))$ holds. Because of the symmetric way Δ is defined, it is equivalent to show that for all $ft \in N$ and all $a \in \{\mathtt{1}, \mathtt{2}, \overline{\mathtt{1}}, \overline{\mathtt{2}}\}$, if $ft\,\langle a\rangle$ is defined then: $\forall\,\langle a\rangle\,\varphi \in \mathrm{Lean}(\psi), \langle a\rangle\,\varphi \in \mathtt{type}(ft) \Leftrightarrow \varphi \mathrel{\dot\in} \mathtt{type}(ft\,\langle a\rangle)$. We prove the two directions of the equivalence:
— If $\langle a\rangle\,\varphi \in \mathtt{type}(ft)$, then $ft \in [\![\langle a\rangle\,\varphi]\!]$. Therefore, $ft\,\langle a\rangle \in [\![\varphi]\!]$. From step (1) we then conclude $\varphi \mathrel{\dot\in} \mathtt{type}(ft\,\langle a\rangle)$.
— If $\varphi \mathrel{\dot\in} \mathtt{type}(ft\,\langle a\rangle)$, then let (T, F) be such that $\varphi \Longleftarrow (T, F)$ and $T \subseteq \mathtt{type}(ft\,\langle a\rangle) \subseteq \mathrm{Lean}(\psi) \setminus F$. By definition of $\mathtt{type}(ft\,\langle a\rangle)$, we have $ft\,\langle a\rangle \in [\![\chi]\!]$ for all χ in T and $ft\,\langle a\rangle \notin [\![\chi]\!]$ for all χ in F. From this, Proposition 4.5 allows us to deduce $ft\,\langle a\rangle \in [\![\varphi]\!]$, which implies $ft \in [\![\langle a\rangle\,\varphi]\!]$, which in turn implies $\langle a\rangle\,\varphi \in \mathtt{type}(ft)$ since $\langle a\rangle\,\varphi \in \mathrm{Lean}(\psi)$.

(3) We now show by induction on n that all $ft \in N$ of depth strictly less than n are such that $\mathtt{type}(ft) \in X^n$. This property is trivially true for $n = 0$. Suppose now that it is true of n and let $ft \in N$ be of depth n (or less). Then for $a \in \{\mathtt{1}, \mathtt{2}\}$, either $ft\,\langle a\rangle$ is undefined or it has depth strictly less than n. In the first case, we have $ft \notin [\![\langle a\rangle\,\top]\!]$, thus $\mathtt{type}(ft)$ does not contain $\langle a\rangle\,\top$ and therefore trivially satisfies the condition relative to a in $\mathtt{Upd}(X^n)$. In the second case, by induction hypothesis we have $\mathtt{type}(ft\,\langle a\rangle) \in X^n$, and by step (2) above we have $\Delta_a(\mathtt{type}(ft), \mathtt{type}(ft\,\langle a\rangle))$, so the condition is fulfilled as well. This is true for both as, so in the end we have $\mathtt{type}(ft) \in \mathtt{Upd}(X^n) = X^{n+1}$.

(4) Finally, if n is the depth of τ plus one and $t = \mathtt{type}(\tau_{\mathsf{Top}})$, we have $t \in X^n$. Since both $\tau_{\mathsf{Top}}\,\langle\overline{\mathtt{1}}\rangle$ and $\tau_{\mathsf{Top}}\,\langle\overline{\mathtt{2}}\rangle$ are undefined, t contains neither $\langle\overline{\mathtt{1}}\rangle\,\top$ nor $\langle\overline{\mathtt{2}}\rangle\,\top$. Furthermore, since $\tau_{\mathsf{Top}} \in [\![\psi]\!]$, we have from step (1) that $\psi \mathrel{\dot\in} t$. Therefore $\mathtt{FinalCheck}(X^n)$ holds. □

4.3.4. Complexity. We now present one of the main contributions of this paper: the complexity of our algorithm is $2^{\mathcal{O}(n)}$ where n is the formula size. It is well-known that

$\mathrm{cl}(\psi)$ is a finite set and its size is linear with respect to the size of ψ (i.e., the number of operators and propositional variables appearing in ψ) [Kozen 1983]. Therefore $|\mathrm{Lean}(\psi)|$ is also trivially linear with respect to the size of ψ.

THEOREM 4.12 (COMPLEXITY). *For $\psi \in \mathcal{L}_\mu$, closed and cycle-free, the satisfiability problem $[\![\psi]\!]_\emptyset \neq \emptyset$ is decidable in time $2^{\mathcal{O}(n)}$ where $n = |Lean(\psi)|$.*

PROOF. $|\mathrm{Types}(\psi)|$ is bounded by $|\mathcal{P}(\mathrm{Lean}(\psi))|$ which is $2^{\mathcal{O}(n)}$. During each iteration, the algorithm adds at least one new type (otherwise it terminates), thus it performs at most $2^{\mathcal{O}(n)}$ iterations. We now detail what it does at each iteration. For each type that may be added (there are $2^{\mathcal{O}(n)}$ of them), there are two traversals of the set of types at the previous step to collect witnesses. Hence there are $2 * 2^{\mathcal{O}(n)} * 2^{\mathcal{O}(n)} = 2^{\mathcal{O}(n)}$ witness tests at each iteration. Each witness test involves a membership test and a Δ_a test. In the implementation these are precomputed: for every formula $\langle a \rangle \varphi$ in the lean, the subsets (T, F) of the lean that must be true and false respectively for φ to be true are precomputed, so testing $\varphi \mathrel{\dot\in} t$ are simple inclusion and disjunction tests. The `FinalCheck` condition tests at most $2^{\mathcal{O}(n)}$ ψ-types and each test takes at most $2^{\mathcal{O}(n)}$. Therefore, the worst case global time complexity of the algorithm does not exceed $2^{\mathcal{O}(n)}$. □

5. IMPLEMENTATION TECHNIQUES

This section describes the main techniques used for implementing an effective $\mathcal{L}_\mu$ decision procedure. Our implementation is publicly available and usable through a web interface [Genevès et al. 2014].

5.1. Implicit Representation of Sets of ψ-Types

Our implementation relies on a symbolic representation and manipulation of sets of ψ-types using Binary Decision Diagrams (BDDs) [Bryant 1986]. BDDs provide a canonical representation of Boolean functions. Experience has shown that this representation is very compact for very large Boolean functions. Their effectiveness is notably well known in the area of formal verification of systems [Clarke et al. 1999].

We introduce a bit-vector representation of ψ-types: for $\mathrm{Lean}(\psi) = \{\varphi_1, ..., \varphi_m\}$, we represent a subset $t \subseteq \mathrm{Lean}(\psi)$ by a vector $\vec{t} = \langle t_1, ..., t_m \rangle \in \{0,1\}^m$ such that $\varphi_i \in t$ iff $t_i = 1$.

A BDD with m variables is then used to represent a set of such bit vectors.

We define auxiliary predicates for programs $a \in \{1, 2\}$:

— $\mathrm{isparent}_a(\vec{t})$ is read "$\vec{t}$ is a parent for program a" and is true iff the bit for $\langle a \rangle \top$ is true in $\vec{t}$
— $\mathrm{ischild}_a(\vec{t})$ is read "$\vec{t}$ is a child for program a" and is true iff the bit for $\langle \overline{a} \rangle \top$ is true in $\vec{t}$

For a set $T \subseteq \mathcal{P}(\mathrm{Lean}(\psi))$, we note χ_T its corresponding characteristic function.

Encoding $\chi_{\mathrm{Types}(\psi)}$ is straightforward with the previous definitions. We define the equivalent of $\dot\in$ on the bit-vector representation:

$$\mathrm{status}_\varphi(\vec{t}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \varphi = \top \\ 0 & \text{if } \varphi = \bot \\ t_i & \text{if } \varphi \in \mathrm{Lean}(\psi) \\ \mathrm{status}_{\varphi'}(\vec{t}) \wedge \mathrm{status}_{\varphi''}(\vec{t}) & \text{if } \varphi = \varphi' \wedge \varphi'' \\ \mathrm{status}_{\varphi'}(\vec{t}) \vee \mathrm{status}_{\varphi''}(\vec{t}) & \text{if } \varphi = \varphi' \vee \varphi'' \\ \neg \mathrm{status}_{\varphi'}(\vec{t}) & \text{if } \varphi = \neg\varphi' \\ \mathrm{status}_{\exp(\varphi)}(\vec{t}) & \text{if } \varphi = \mu(X_i = \varphi_i) \text{ in } \psi \end{cases}$$

We use $a \rightarrow b$ to denote the implication and $a \leftrightarrow b$ to denote the equivalence of two Boolean formulas a and b over bit vectors. We can now construct the BDD of the relation Δ_a for $a \in \{\mathtt{1}, \mathtt{2}\}$.

This BDD relates all pairs $(\vec{x}, \vec{y})$ that are consistent w.r.t the program a, i.e., such that $\vec{y}$ supports all of $\vec{x}$'s $\langle a \rangle\, \varphi$ formulas, and vice-versa $\vec{x}$ supports all of $\vec{y}$'s $\langle \overline{a} \rangle\, \varphi$ formulas:

$$\Delta_a(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq m} \begin{cases} x_i \leftrightarrow \text{status}_\varphi(\vec{y}) & \text{if } \varphi_i = \langle a \rangle\, \varphi \\ y_i \leftrightarrow \text{status}_\varphi(\vec{x}) & \text{if } \varphi_i = \langle \overline{a} \rangle\, \varphi \\ \top \quad \text{otherwise} \end{cases}$$

For $a \in \{\mathtt{1}, \mathtt{2}\}$, we define the set of witnessed vectors:

$$\chi_{\mathtt{Wit}_a(T)}(\vec{x}) \stackrel{\text{def}}{=} \text{isparent}_a(\vec{x}) \rightarrow \exists \vec{y}\ [\ \chi_T(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y})\ ]$$

Then, the BDD of the fixpoint computation is initially set to the false constant, and the main function $\mathtt{Upd}(\cdot)$ is implemented as:

$$\chi_{\mathtt{Upd}(T)}(\vec{x}) \stackrel{\text{def}}{=} \chi_T(\vec{x}) \vee \left(\chi_{\mathtt{Types}(\psi)}(\vec{x}) \wedge \bigwedge_{a \in \{\mathtt{1},\mathtt{2}\}} \chi_{\mathtt{Wit}_a(T)}(\vec{x}) \right)$$

Finally, the solver is implemented as iterations over the sets $\chi_{\mathtt{Upd}(T)}$ until a fixpoint is reached. The final satisfiability condition consists of checking whether ψ is present in a ψ-type of this fixpoint with no unfulfilled upward eventuality:

$$\exists \vec{t} \left[\chi_T(\vec{t}) \wedge \bigwedge_{a \in \{\mathtt{1},\mathtt{2}\}} \neg \text{ischild}_a(\vec{t}) \wedge \text{status}_\psi(\vec{t}) \right]$$

5.2. Satisfying Model Reconstruction

The implementation keeps a copy of each intermediate set of types computed by the algorithm, so that whenever a formula is satisfiable, a minimal satisfying model can be extracted. The top-down (re)construction of a satisfying model starts from a root (a ψ-type for which the final satisfiability condition holds), and repeatedly attempts to find successors. In order to minimize model size, only required left and right branches are built. Furthermore, for minimizing the maximal depth of the model, left and right successors of a node are successively searched in the intermediate sets of types, in the order they were computed by the algorithm.

5.3. Conjunctive Partitioning and Early Quantification

The BDD-based implementation involves computations of *relational products* of the form:

$$\exists \vec{y}\ [\ \chi_T(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y})\ ] \tag{1}$$

It is well-known that such a computation may be quite time and space consuming, because the BDD corresponding to the relation Δ_a may be quite large.

One famous optimization technique is *conjunctive partitioning* [Clarke et al. 1999] combined with *early quantification* [Pan et al. 2006]. The idea is to compute the relational product without ever building the full BDD of the relation Δ_a. This is possible by taking advantage of the form of Δ_a along with properties of existential quantification. By definition, Δ_a is a conjunction of n equivalences relating $\vec{x}$ and $\vec{y}$ where n is the number of $\langle b \rangle\, \varphi$ formulas in Lean(ψ) where $\varphi \neq \top$ and $b \in \{a, \overline{a}\}$:

$$\Delta_a(\vec{x}, \vec{y}) = \bigwedge_{i=1}^{n} R_i(\vec{x}, \vec{y})$$

If a variable y_k does not occur in the clauses $R_{i+1}, ..., R_n$ then the relational product (1) can be rewritten as:

$$\exists_{y_1, ..., y_{k-1}, y_{k+1}, ..., y_m} \left[\exists y_k \left[\chi_T(\vec{y}) \wedge \bigwedge_{1 \leq j \leq i} R_j(\vec{x}, \vec{y}) \right] \wedge \bigwedge_{i+1 \leq l \leq n} R_l(\vec{x}, \vec{y}) \right]$$

This allows to apply existential quantification on intermediate BDDs and thus to compose smaller BDDs. Of course, there are many ways to compose the $R_i(\vec{x}, \vec{y})$. Let ρ be a permutation of $\{0, ..., n-1\}$ which determines the order in which the partitions $R_i(\vec{x}, \vec{y})$ are combined. For each i, let D_i be the set of variables y_k with $k \in \{1, ..., m\}$ that $R_i(\vec{x}, \vec{y})$ depends on. We define E_i as the set of variables contained in $D_{\rho(i)}$ that are not contained in $D_{\rho(j)}$ for any j larger than i:

$$E_i = D_{\rho(i)} \setminus \bigcup_{j=i+1}^{n-1} D_{\rho(j)}$$

The E_i are pairwise disjoint and their union contains all the variables. The relational product (1) can be computed by starting from:

$$h_1(\vec{x}, \vec{y}) = \exists_{y_k \in E_0} \left[\chi_T(\vec{y}) \wedge R_{\rho(0)}(\vec{x}, \vec{y}) \right]$$

and successively computing h_{p+1} defined as follows:

$$h_{p+1}(\vec{x}, \vec{y}) = \begin{cases} \exists_{y_k \in E_p} \left[h_p(\vec{x}, \vec{y}) \wedge R_{\rho(p)}(\vec{x}, \vec{y}) \right] & \text{if } E_p \neq \emptyset \\ h_p(\vec{x}, \vec{y}) \wedge R_{\rho(p)}(\vec{x}, \vec{y}) & \text{if } E_p = \emptyset \end{cases}$$

until reaching h_n which is the result of the relational product. The ordering ρ determines how early in the computation variables can be quantified out. This directly impact the sizes of BDDs constructed and therefore the global efficiency of the decision procedure. It is thus important to choose ρ carefully. The overall goal is to minimize the size of the largest BDD created during the elimination process. We use a heuristic taken from [Clarke et al. 1999] which seems to provide the best approximation and in practice has the best performance. It defines the cost of eliminating a variable y_k as the sum of the sizes of all the D_i containing y_k:

$$\sum_{1 \leq i \leq n, y_k \in D_i} |D_i|$$

The ordering ρ on the relations R_i is then defined in such a way that variables can be eliminated in the order given by a greedy algorithm which repeatedly eliminates the variable of minimum cost.

5.4. BDD Variable Ordering

The cost of BDD operations is very sensitive to variable ordering. Finding the optimal variable ordering is known to be NP-complete [Hojati et al. 1996], however several heuristics are known to perform well in practice [Clarke et al. 1999]. Choosing a good initial order of Lean(ψ) formulas does significantly improve performance. We found out that preserving locality of the initial problem is essential. Experience has shown that the variable order determined by the breadth-first traversal of the formula ψ to solve, which keeps sister subformulas in close proximity, yields better results in practice.

Table I. Concrete syntax used in the online solver

	abstract syntax	concrete syntax
Atomic proposition	α	`_a`
Node name	σ	`a`
Variable	X	`$X`
True, False	$\top, \bot$	`T`, `F`
Disjunction	$\varphi \vee \psi$	φ `|` ψ
Conjunction	$\varphi \wedge \psi$	φ `&` ψ
Negation	$\neg\varphi$	`~`φ
Backward modality	$\langle\overline{a}\rangle\,\varphi$	`<-a>`φ
Fixpoint	$\mu(X = \varphi, Y = \psi)$ in χ	`let $X=`φ`, $Y=`ψ `in` χ

5.5. Online Implementation

The system has been implemented as a web application. Interaction with the system is offered through a user interface in a web browser. The tool is available online from:

`http://wam.inrialpes.fr/websolver/`

Table I indicates how the syntax used in this paper translates into the syntax understood by our solver, which we also use in some examples in the next section.

6. EXAMPLES AND EXPERIMENTS

In this section we report on practical experiments that we have made using the solver implementation. These experiments can be tried with the online implementation described above.

6.1. Syntax Extensions

6.1.1. Standard Syntactic Sugar. We already defined as syntactic sugar the negation of an arbitrary closed formula, in Section 3. We add equivalence $\Leftrightarrow$, implication $\Rightarrow$ and the universal modalities $[a]$ as follows:

$$\varphi \Rightarrow \psi \stackrel{\text{def}}{=} \neg\varphi \vee \psi$$
$$\varphi \Leftrightarrow \psi \stackrel{\text{def}}{=} (\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$$
$$[a]\,\varphi \stackrel{\text{def}}{=} \neg\,\langle a\rangle\,\top \vee \langle a\rangle\,\neg\varphi$$

6.1.2. Node names. A very common constraint in tree-like data models is that each node should be named, i.e. bear exactly one name σ from some set Σ. In order to state properties of such structures, we need atomic propositions corresponding to these names: σ, meaning the name of the current node is σ. These name-propositions behave like label-propositions except for the additional constraint that exactly one of them must be true for each node; because any given formula contains only a finite number of name-propositions, it is possible to encode this additional constraint as a formula. When checking the satisfiability of a formula containing such name-propositions, we can thus generate the formula corresponding to this labeling constraint, conjunct it with the plunged formula and then treat all atomic propositions in the same way in the satisfiability-checking algorithm.

This is implemented as follows: given φ, let `names`(φ) be the set of all names appearing in φ, plus an additional atomic proposition σ_x not in φ, to represent all other

possible names. The labelling-requirement formula is defined as follows:

$$\texttt{lab-req}(\varphi) \stackrel{\text{def}}{=} \bigvee_{\sigma \in \texttt{names}(\varphi)} \left(\sigma \wedge \bigwedge_{\sigma' \in \texttt{names}(\varphi) \setminus \{\sigma\}} \neg \sigma' \right)$$

The formula fed to the main algorithm is then:

$$\psi \stackrel{\text{def}}{=} (\mu X.\varphi \vee \langle \mathbf{1} \rangle X \vee \langle \mathbf{2} \rangle X) \wedge (\mu Y.\texttt{lab-req}(\varphi) \wedge [\mathbf{1}] Y \wedge [\mathbf{2}] Y).$$

Note that the cost of this additional constraint on the size of the lean is only 3. Indeed, $\texttt{lab-req}(\varphi)$ contains no modality and the only atomic proposition it contains which is not already in φ is σ_x; then two modalities are added in the final formula. Thus names are not more expensive to use than regular atomic propositions.

6.1.3. Node identifiers (a.k.a. nominals). Another useful kind of constrained proposition is *nominals*, which must be true at exactly one node of any given tree. They typically allow relating formulas by constraining two nodes reached in different ways to be the same one. Similarly to node names, nominals can be implemented as regular atomic propositions provided the plunged formula is conjuncted with a global constraint stating unicity. Given a nominal n, we define the following formulas:

$$\begin{aligned}
\texttt{nowhere}(n) &\stackrel{\text{def}}{=} \mu X.\neg n \wedge [\mathbf{1}] X \wedge [\mathbf{2}] X \\
\texttt{here}(n) &\stackrel{\text{def}}{=} n \wedge [\mathbf{1}] \texttt{nowhere}(n) \wedge [\mathbf{2}] \texttt{nowhere}(n) \\
\texttt{somewhere}(n) &\stackrel{\text{def}}{=} \mu Y.\texttt{here}(n) \vee (\neg n \wedge ((\langle \mathbf{1} \rangle Y \wedge [\mathbf{2}] \texttt{nowhere}(n)) \vee ([\mathbf{1}] \texttt{nowhere}(n) \wedge \langle \mathbf{2} \rangle Y)
\end{aligned}$$

To enforce the constraint, it is sufficient to conjunct ψ with the formula $\texttt{somewhere}(n)$ for each nominal n appearing in φ. This increases the size of the lean by 4 for each nominal (four modalities corresponding to the two 'nowhere's and the two 'somewhere's), thus is significantly more costly than simple atomic propositions. It is however still linear in terms of the original formula, so this extension does not modify the order of complexity of the algorithm.

6.2. Regular Language Equivalence

As a first, simple, application, we show how we can use our solver to decide the equivalence of regular languages. To this end, we translate regular expressions in formulas of our tree logic. The translation presented here is similar to the one described in [Lange 2005]; the actual translation used in the online implementation (using the `reg_exp` keyword) contains additional optimizations for conciseness.

As illustrated in Section 2, our model does not allow the empty tree. To translate regular expressions which may recognize the empty word, we add a final letter e at the end of the expression. We also need to be careful with the repetition of regular expressions, of the form R^*, if R is nullable (it accepts the empty word ε). A direct translation in this case would result in a recursion variable appearing naked (i.e., without a surrounding modality). We thus extract the non null part of R, written $R_{\overline{\varepsilon}}$, and translate R^* as $R^*_{\overline{\varepsilon}}$. We first recall how to naively extract from a regular expression R its nullable part (either ε or $\emptyset$), written R_ε, and its non null part, written $R_{\overline{\varepsilon}}$.

$$
\begin{array}{rlrl}
\varepsilon_{\varepsilon} &= \varepsilon & \varepsilon_{\overline{\varepsilon}} &= \emptyset \\
a_{\varepsilon} &= \emptyset & a_{\overline{\varepsilon}} &= a \\
(R.R')_{\varepsilon} &= R_{\varepsilon}.R'_{\varepsilon} & (R.R')_{\overline{\varepsilon}} &= R_{\varepsilon}.R'_{\overline{\varepsilon}} \vee R_{\overline{\varepsilon}}.R'_{\varepsilon} \vee R_{\overline{\varepsilon}}.R'_{\overline{\varepsilon}} \\
(R^*)_{\varepsilon} &= \varepsilon & (R^*)_{\overline{\varepsilon}} &= (R_{\overline{\varepsilon}})^+ \\
(R \vee R')_{\varepsilon} &= R_{\varepsilon} \vee R'_{\varepsilon} & (R \vee R')_{\overline{\varepsilon}} &= R_{\overline{\varepsilon}} \vee R'_{\overline{\varepsilon}}
\end{array}
$$

The translation of a regular expression R with a continuation c is written $[\![R]\!]_c$ and is defined as follows.

$$
\begin{array}{rll}
[\![a]\!]_c &= a \wedge \langle 1 \rangle\, c & \\
[\![\varepsilon]\!]_c &= c & \\
[\![R.R']\!]_c &= [\![R]\!]_{[\![R']\!]_c} & \\
[\![R^*]\!]_c &= \mu x.c \vee [\![R_{\overline{\varepsilon}}]\!]_x & \text{if } R_{\overline{\varepsilon}} \neq \emptyset \\
[\![R^*]\!]_c &= c & \text{if } R_{\overline{\varepsilon}} = \emptyset \\
[\![R \vee R']\!]_c &= [\![R]\!]_c \vee [\![R']\!]_c &
\end{array}
$$

Given a regular expression R, we translate it into our logic as the formula $[\![R]\!]_e$. To check the equivalence of two regular expressions R_1 and R_2, we need to check the *validity* of the formula $([\![R_1]\!]_e \iff [\![R_2]\!]_e)$ (i.e. we want this equivalence to hold for all focused trees). Since our solver is a *satisfiability* solver, we ask it the question $\neg([\![R_1]\!]_e \iff [\![R_2]\!]_e)$. If the formula is unsatisfiable, the languages are equivalent. If it is satisfiable, the solver will return a model of this formula, i.e. a tree representing a word in one language and not the other.

We illustrate this translation with an example. To show that the languages $(ab)^*a$ and $a(ba)^*$ are equivalent, we run the following query in the solver. (This code may be copied and pasted directly in the online demo.)

```
~
(let $X = (a & <1>e) | a & <1>(b & <1> $X) in $X)
<=>
(a & <1> (let $X = e | b & <1>(a & <1> $X) in $X))
```

We now extend our simple translation to study the equivalence of languages of Kleene Algebra with Test (KAT) [Kozen 1997].

We extend our translation with a case for boolean propositions α:

$$
\begin{aligned}
[\![\alpha]\!]_c &= \alpha \wedge c \\
[\![\neg\alpha]\!]_c &= \neg\alpha \wedge c
\end{aligned}
$$

Note that, unlike the translation for letters, we do not move to the next letter. One may thus specify several propositions that must concurrently be true.

As an example, consider the usual encoding of a while loop in KAT.

$$
[\![(\alpha a)^* \neg\alpha]\!]_e = \mu x.(\neg\alpha \wedge e) \vee (\alpha \wedge a \wedge \langle 1 \rangle\, x)
$$

We can thus use the solver to test for language equality or inequality. In the solver, boolean propositions α are represented by identifiers starting with an underscore. For instance, one may show that βq^* is different from $(\beta q)^*$ as follows.

```
~
(_b & (let $X = e | q & <1>$X in $X)
```

```
<=>
let $X = e | _b & q & <1>$X in $X)
```

The satisfying word found is `e ~_b`, which is the empty word annotated with the negation of `_b`: it belongs to the second language but not to the first.

6.3. Applications to XPath typing

Another natural application of the tree logic consists in the static analysis of programs that manipulate XML documents seen as trees. Backward modalities naturally capture XPath expressions that navigate upward in the tree in a succinct manner. The translations of XPath and XML type expressions into the logic are recalled from [Genevès and Layaïda 2006] in an appendix to make the article self-contained. We also give the semantics of XPath in terms of focused trees, and prove that the generated formulas are cycle-free. Owing to these translations, we can formulate several decision problems involving XPath expressions $e_1, ..., e_n$ and XML type expressions $T_1, ..., T_n$, for which the solver provides a decision procedure. In particular, the following basic problems are of special interest:

— XPath containment: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (if the formula is unsatisfiable then all nodes selected by e_1 under type constraint T_1 are selected by e_2 under type constraint T_2)
— XPath emptiness: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]}$
— XPath overlap: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$
— XPath coverage: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \bigwedge_{2 \leq i \leq n} \neg E^{\rightarrow}[\![e_i]\!]_{[\![T_i]\!]}$
— Static type checking of an annotated XPath expression: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg[\![T_2]\!]$ (if the formula is unsatisfiable then all nodes selected by e_1 under type constraint T_1 are included in the type T_2.)
— XPath equivalence under type constraints: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \neg E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ and $\neg E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge E^{\rightarrow}[\![e_2]\!]_{[\![T_2]\!]}$ (This test can be used to check that the nodes selected after a modification of a type T_1 by T_2 and an XPath expression e_1 by e_2 are the same, typically when an input type changes and the corresponding XPath expression has to change as well.)

We carried out extensive tests with the implementation[6] [Genevès et al. 2014] , and present here only a representative sample that includes the most complex language features such as recursive forward and backward axes, intersection, large and very recursive types with a reasonable alphabet size. The tests use XPath expressions shown on Fig. 8 (where "//" is used as a shorthand for "/desc-or-self::*/") and XML types shown on Table II. Table III presents some decision problems and corresponding performance results. Times reported in milliseconds correspond to the running time of the satisfiability solver without the (negligible) time spent for parsing and translating into $\mathcal{L}_\mu$.

The first XPath containment instance was first formulated in [Miklau and Suciu 2004] as an example for which the proposed tree pattern homomorphism technique is incomplete. The e_8 example shows that the official XHTML DTD does not syntactically prohibit the nesting of anchors. For the XHTML case, we observe that the time needed is more important, but it remains practically relevant, especially for static analysis operations performed only at compile-time.

[6]Experiments have been conducted with a JAVA implementation running on a Pentium 4, 3 Ghz, with 512Mb of RAM with Windows XP.

e_1 /a[.//b[c/*//d]/b[c//d]/b[c/d]]
e_2 /a[.//b[c/*//d]/b[c/d]]

e_3 a/b//c/foll-sibling::d/e
e_4 a/b//d[prec-sibling::c]/e
e_5 a/c/following::d/e
e_6 a/b[//c]/following::d/e ∩ a/d[preceding::c]/e

e_7 *//switch[ancestor::head]//seq//audio[prec-sibling::video]

e_8 descendant::a[ancestor::a]
e_9 /descendant::*
e_{10} html/(head | body)
e_{11} html/head/descendant::*
e_{12} html/body/descendant::*

Fig. 8. XPath Expressions Used in Experiments.

Table II. Types Used in Experiments.

DTD	Symbols	Binary Type Variables
SMIL 1.0	19	11
XHTML 1.0 Strict	77	325

Table III. Some Decision Problems and Corresponding Results.

XPath Decision Problem	XML Type	Time (ms)
$e_1 \subseteq e_2$ and $e_2 \not\subseteq e_1$	none	353
$e_4 \subseteq e_3$ and $e_4 \subseteq e_3$	none	45
$e_6 \subseteq e_5$ and $e_5 \not\subseteq e_6$	none	41
e_7 is satisfiable	SMIL 1.0	157
e_8 is satisfiable	XHTML 1.0	2630
$e_9 \subseteq (e_{10} \cup e_{11} \cup e_{12})$	XHTML 1.0	2872

7. CONCLUSION

We found very interesting that the collapsing of the least and greatest fixpoints on acyclic structures [Mateescu 2002] could be extended to trees with converse modalities by restricting the language to cycle-free formulas. We were able to leverage this property to obtain an efficient implementation.

We also emphasize that the fact that the algorithm is implemented as a least fixpoint construction makes it possible to check for satisfiable formulas as soon as possible, often not requiring the full fixpoint to be computed, as opposed to algorithms based on greatest fixpoint computations that must eliminate all contradictions and therefore complete the computation of the fixpoint all the time.

The main result of our paper is a sound and complete satisfiability-testing algorithm for a sub-logic of the alternation-free modal μ-calculus with converse for finite trees. The algorithm operates in time complexity $2^{\mathcal{O}(n)}$ in the length n of a formula. It has been implemented and is available online.

As a direction for future work, we plan to study the extension of the logic with counting operators. We have started this investigation for restricted form of counting and interleaving [Barcenas et al. 2011], which we want to extend to the full logic. As an-

other perspective, notice that it is possible to configure the solver such that, instead of computing one satisfying tree for a satisfiable formula, it computes a regular tree type representation of the set of all satisfying trees. Such a representation could be used in the setting of rich type systems for programming and query languages such as XQuery [Boag et al. 2007] and CDuce [Benzaken et al. 2003].

REFERENCES

AFANASIEV, L., BLACKBURN, P., DIMITRIOU, I., GAIFFE, B., GORIS, E., MARX, M., AND DE RIJKE, M. 2005. PDL for ordered trees. *Journal of Applied Non-Classical Logics 15,* 2, 115–135.

BÁRCENAS, E., GENEVÈS, P., AND LAYAÏDA, N. 2009. On the analysis of queries with counting constraints. In *DocEng '09: Proceedings of the 9th ACM symposium on Document engineering*. ACM, New York, NY, USA, 21–24.

BARCENAS, E., GENEVÈS, P., LAYAÏDA, N., AND SCHMITT, A. 2011. Query reasoning on trees with types, interleaving and counting. In *IJCAI'11 : Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 718–723.

BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. 2003. CDuce: An XML-centric general-purpose language. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA, 51–63.

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. 2007. XQuery 1.0: An XML query language, W3C recommendation.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers 35,* 8, 677–691.

CALVANESE, D., DE GIACOMO, G., LENZERINI, M., AND VARDI, M. Y. 2008. Regular XPath: Constraints, query containment and view-based answering for xml documents. In *Proc. of the 2008 Int. Workshop on Logic in Databases (LID 2008)*.

CALVANESE, D., DE GIACOMO, G., LENZERINI, M., AND VARDI, M. Y. 2009. An automata-theoretic approach to regular XPath. In *Proc. of the 12th Int. Symposium on Database Programming Languages (DBPL 2009)*. Lecture Notes in Computer Science, vol. 5708. Springer, 18–35.

CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND VARDI, M. Y. 2010. Node selection query languages for trees. In *AAAI*, M. Fox and D. Poole, Eds. AAAI Press.

CLARK, J. AND DEROSE, S. 1999. XML path language (XPath) version 1.0, W3C recommendation. http://www.w3.org/TR/ 1999/REC-xpath-19991116.

CLARKE, E. M. AND EMERSON, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*. LNCS, vol. 131. Springer-Verlag, London, UK, 52–71.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model checking*. MIT Press, Cambridge, MA, USA.

FISCHER, M. J. AND LADNER, R. E. 1979. Propositional dynamic logic of regular programs. *JCSS 18,* 2, 194–211.

FRIEDMANN, O. AND LANGE, M. 2009. The pgsolver collection of parity game solvers. http://www2.tcs.ifi.lmu.de/pgsolver/.

FRIEDMANN, O. AND LANGE, M. 2010. A solver for modal fixpoint logics. *Electron. Notes Theor. Comput. Sci. 262*, 99–111.

GENEVÈS, P., GESBERT, N., LAYAÏDA, N., AND SCHMITT, A. 2014. A satisfiability solver for XML and XPath. http://wam.inrialpes.fr/websolver.

GENEVÈS, P. AND LAYAÏDA, N. 2006. A system for the static analysis of XPath. *ACM Trans. Inf. Syst. 24,* 4, 475–502.

GENEVÈS, P., LAYAÏDA, N., AND SCHMITT, A. 2007. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 342–351.

GRÄDEL, E., THOMAS, W., AND WILKE, T. 2002. *Automata logics, and infinite games: a guide to current research*. Springer-Verlag, New York, NY, USA.

HOJATI, R., KRISHNAN, S. C., AND BRAYTON, R. K. 1996. Early quantification and partitioned transition relations. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*. IEEE Computer Society, Washington, DC, USA, 12–19.

HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. 2005. Regular expression types for XML. *ACM Trans. Program. Lang. Syst. 27,* 1, 46–90.

HUET, G. P. 1997. The zipper. *J. Funct. Program. 7,* 5, 549–554.

KOZEN, D. 1983. Results on the propositional μ-calculus. *Theoretical Computer Science 27*, 333–354.

KOZEN, D. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 427–443.

KUPFERMAN, O. AND VARDI, M. 1999. The weakness of self-complementation. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science*. LNCS, vol. 1563. Springer, London, UK, 455–466.

LANGE, M. 2005. Weak automata for the linear time μ-calculus. In *Verification, Model Checking, and Abstract Interpretation*, R. Cousot, Ed. Lecture Notes in Computer Science, vol. 3385. Springer Berlin Heidelberg, 267–281.

LIBKIN, L. AND SIRANGELO, C. 2008. Reasoning about xml with temporal logics and automata. In *LPAR '08: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Springer-Verlag, Berlin, Heidelberg, 97–112.

LIBKIN, L. AND SIRANGELO, C. 2010. Reasoning about xml with temporal logics and automata. *J. Applied Logic 8,* 2, 210–232.

MATEESCU, R. 2002. Local model-checking of modal μ-calculus on acyclic labeled transition systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, London, UK, 281–295.

MIKLAU, G. AND SUCIU, D. 2004. Containment and equivalence for a fragment of XPath. *Journal of the ACM 51,* 1, 2–45.

MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology 5,* 4, 660–704.

OLTEANU, D., MEUSS, H., FURCHE, T., AND BRY, F. 2002. XPath: Looking forward. In *EDBT '02: Proceedings of the Worshop on XML-Based Data Management*. LNCS, vol. 2490. Springer-Verlag, London, UK, 109–127.

PAN, G., SATTLER, U., AND VARDI, M. Y. 2006. BDD-based decision procedures for the modal logic K. *Journal of Applied Non-classical Logics 16,* 1-2, 169–208.

PRATT, V. 1981. A decidable mu-calculus: Preliminary report. In *Foundations of Computer Science, 1981. SFCS '81. 22nd Annual Symposium on*. 421–427.

SAFRA, S. 1988. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 319–327.

TANABE, Y., TAKAHASHI, K., AND HAGIYA, M. 2008. A decision procedure for alternation-free modal μ-calculi. In *Advances in Modal Logic*. 341–362.

TANABE, Y., TAKAHASHI, K., YAMAMOTO, M., TOZAWA, A., AND HAGIYA, M. 2005. A decision procedure for the alternation-free two-way modal μ-calculus. In *In TABLEAUX 2005*. LNCS, vol. 3702. Springer-Verlag, London, UK, 277–291.

TOZAWA, A. 2004. On binary tree logic for XML and its satisfiability test. In *PPL '04: the Sixth JSSST Workshop on Programming and Programming Languages*. Informal Proceedings, Gamagoori, Japan.

VARDI, M. Y. 1998. Reasoning about the past with two-way automata. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK, 628–641.

VOSS, J. 2007. Wikipedia dtd. http://meta.wikimedia.org/wiki/Wikipedia_DTD.

ZEE, K., KUNCAK, V., AND RINARD, M. C. 2008. Full functional verification of linked data structures. In *PLDI*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 349–361.

A. XPATH AND REGULAR TREE LANGUAGES

A.1. XPath

XPath [Clark and DeRose 1999] is a powerful language for navigating in XML documents and selecting sets of nodes matching a predicate. In their simplest form, XPath expressions look like "directory navigation paths". For example, the XPath expression

$$/\text{child::book}/\text{child::chapter}/\text{child::section}$$

navigates from the root of a document (designated by the leading "/") through the top-level "book" node to its "chapter" child nodes and on to its child nodes named "section". The result of the evaluation of the entire expression is the set of all the "section" nodes that can be reached in this manner. The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than "child". For instance, one may use the "preceding-sibling" axis for navigating backward through nodes of the same parent, or the "ancestor" axis for navigating upward recursively. Furthermore, at each step in the navigation the selected nodes can be filtered using qualifiers: Boolean expression between brackets that can test the existence or absence of paths.

For the practical experiments, we consider an XPath fragment covering all major features of the XPath 1.0 recommendation [Clark and DeRose 1999] with the exception of counting and comparisons between data values.

Fig. 9 gives the syntax of XPath expressions.

$\mathcal{L}_{\text{XPath}} \ni e$	$::=$		XPath expression
		$/p$	absolute path
	$\mid$	p	relative path
	$\mid$	$e_1 \cup e_2$	union
	$\mid$	$e_1 \cap e_2$	intersection
Path p	$::=$		path
		p_1/p_2	path composition
	$\mid$	$p[q]$	qualified path
	$\mid$	$a::\sigma$	step with node test
	$\mid$	$a::*$	step
Qualif q	$::=$		qualifier
		q_1 and q_2	conjunction
	$\mid$	q_1 or q_2	disjunction
	$\mid$	not q	negation
	$\mid$	p	path
Axis a	$::=$		tree navigation axis
		child | self | parent	
	$\mid$	descendant | desc-or-self	
	$\mid$	ancestor | anc-or-self	
	$\mid$	foll-sibling | prec-sibling	
	$\mid$	following | preceding	

Fig. 9. XPath Abstract Syntax.

A.2. Focused Tree Interpretation of XPath

XML documents have the structure of unranked n-ary trees where the children of a node are ordered. This structure can be represented by a binary tree if we relate a

node directly only to its first child and its next sibling, if any. To formalise this, we give below a syntax for n-ary trees:

$$\begin{array}{rcll} tl & ::= & & \text{Tree list} \\ & & \varepsilon & \text{Empty list} \\ & | & t :: tl & \text{Cons cell} \\ t & ::= & (L, tl) & \text{n-ary tree} \end{array}$$

We now define the `bintree` translation from tree lists tl to subtrees st, and notice that the translation of a nonempty list is always a nonempty subtree:

$$\texttt{bintree}(\varepsilon) \stackrel{\text{def}}{=} \mathsf{nil}$$

$$\texttt{bintree}((L, tl_1) :: tl_2) \stackrel{\text{def}}{=} (L, \texttt{bintree}(tl_1), \texttt{bintree}(tl_2))$$

The translation of a single n-ary tree t can then be defined as:

$$\texttt{bintree}(t) \stackrel{\text{def}}{=} \texttt{bintree}(t :: \varepsilon).$$

Note that this is not a bijection since the root of the resulting binary tree never has a right child.

We now describe how XPath expressions can be interpreted in terms of focused trees. XPath expressions can be absolute, starting from the root of the document, or relative, starting from a set of nodes called *context nodes*. In order to determine things such as equivalence or containment of possibly relative XPath expressions, we mark all the context nodes with a distinguished label Ⓢ which we call the 'start mark'. The precise data domain we consider is thus the set $\mathcal{F}^*$ of focused trees whose root has no right child, where each node has exactly one name σ from Σ, and where at least one node bears the start mark Ⓢ in addition. For a focused tree $ft = (L, st_1, st_2)_c$, we write $\texttt{name}(ft)$ for the unique $\sigma \in L \cap \Sigma$.

Fig. 10 gives the interpretation of XPath expressions as functions between sets of such focused trees.

A.3. XPath Embedding

We now explain how an XPath expression can be translated into an equivalent $\mathcal{L}_\mu$ formula that performs navigation in focused trees in binary style.

Logical Interpretation of Axes. The translation of navigational primitives (namely XPath axes) is formally specified in Fig. 11. The translation function, noted "$A^{\rightarrow}[\![a]\!]_\chi$", takes an XPath axis a as input, and returns its $\mathcal{L}_\mu$ translation, parameterized by the $\mathcal{L}_\mu$ formula χ given as parameter. This parameter represents the context in which the axis occurs and is needed for formula composition in order to translate path composition. More precisely, the formula $A^{\rightarrow}[\![a]\!]_\chi$ holds for all nodes that can be accessed through the axis a from some node verifying χ.

Let us consider an example. The formula $A^{\rightarrow}[\![\text{child}]\!]_\chi$, translated as $\mu Z.\, \langle \bar{1} \rangle\, \chi \vee \langle \bar{2} \rangle\, Z$, is satisfied by children of the context χ. These nodes consist of the first child and the remaining children. From the first child, the context must be reached immediately by going once upward via $\bar{1}$. From the remaining children, the context is reached by going upward (any number of times) via $\bar{2}$ and finally once via $\bar{1}$.

Logical Interpretation of Expressions. Fig. 12 gives the translation of XPath expressions into $\mathcal{L}_\mu$. The translation function "$E^{\rightarrow}[\![e]\!]_\chi$" takes an XPath expression e and a $\mathcal{L}_\mu$ formula χ as input, and returns the corresponding $\mathcal{L}_\mu$ translation. The translation of a relative XPath expression marks the initial context with Ⓢ. The translation of an absolute XPath takes as initial context a formula saying that we are at the root ($\neg \langle \bar{1} \rangle \top \wedge \neg \langle \bar{2} \rangle \top$) and that the previous context (Ⓢ $\wedge\, \chi$) can be anywhere below.

$$
\begin{array}{rl}
\mathcal{S}_{\mathrm{e}}[\![\cdot]\!]_{\cdot} : & \mathcal{L}_{\mathrm{XPath}} \to \mathcal{P}(\mathcal{F}^*) \to \mathcal{P}(\mathcal{F}^*) \\
\mathcal{S}_{\mathrm{e}}[\![/p]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{p}}[\![p]\!]_{\mathtt{root}(F)} \\
\mathcal{S}_{\mathrm{e}}[\![p]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{p}}[\![p]\!]_{\{(L,st_1,st_2)_c \in F \mid \textcircled{s} \in L\}} \\
\mathcal{S}_{\mathrm{e}}[\![e_1 \dashv e_2]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{e}}[\![e_1]\!]_F \cup \mathcal{S}_{\mathrm{e}}[\![e_2]\!]_F \\
\mathcal{S}_{\mathrm{e}}[\![e_1 \cap e_2]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{e}}[\![e_1]\!]_F \cap \mathcal{S}_{\mathrm{e}}[\![e_2]\!]_F \\
\hline
\mathcal{S}_{\mathrm{p}}[\![\cdot]\!]_{\cdot} : & \mathit{Path} \to \mathcal{P}(\mathcal{F}^*) \to \mathcal{P}(\mathcal{F}^*) \\
\mathcal{S}_{\mathrm{p}}[\![p_1/p_2]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{p}}[\![p_2]\!]_{(\mathcal{S}_{\mathrm{p}}[\![p_1]\!]_F)} \\
\mathcal{S}_{\mathrm{p}}[\![p[q]]\!]_F & \stackrel{\text{def}}{=} \{ft \in \mathcal{S}_{\mathrm{p}}[\![p]\!]_F \mid \mathcal{S}_{\mathrm{q}}[\![q]\!]_{ft}\} \\
\mathcal{S}_{\mathrm{p}}[\![a::\sigma]\!]_F & \stackrel{\text{def}}{=} \{ft \in \mathcal{S}_{\mathrm{a}}[\![a]\!]_F \mid \mathtt{name}(ft) = \sigma\} \\
\mathcal{S}_{\mathrm{p}}[\![a::*]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{a}}[\![a]\!]_F \\
\hline
\mathcal{S}_{\mathrm{q}}[\![\cdot]\!]_{\cdot} : & \mathit{Qualif} \to \mathcal{F}^* \to \{\mathrm{true}, \mathrm{false}\} \\
\mathcal{S}_{\mathrm{q}}[\![q_1 \text{ and } q_2]\!]_{ft} & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{q}}[\![q_1]\!]_{ft} \wedge \mathcal{S}_{\mathrm{q}}[\![q_2]\!]_{ft} \\
\mathcal{S}_{\mathrm{q}}[\![q_1 \text{ or } q_2]\!]_{ft} & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{q}}[\![q_1]\!]_{ft} \vee \mathcal{S}_{\mathrm{q}}[\![q_2]\!]_{ft} \\
\mathcal{S}_{\mathrm{q}}[\![\text{not } q]\!]_{ft} & \stackrel{\text{def}}{=} \neg\, \mathcal{S}_{\mathrm{q}}[\![q]\!]_{ft} \\
\mathcal{S}_{\mathrm{q}}[\![p]\!]_{ft} & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{p}}[\![p]\!]_{\{ft\}} \neq \emptyset \\
\hline
\mathcal{S}_{\mathrm{a}}[\![\cdot]\!]_{\cdot} : & \mathit{Axis} \to \mathcal{P}(\mathcal{F}^*) \to \mathcal{P}(\mathcal{F}^*) \\
\mathcal{S}_{\mathrm{a}}[\![a]\!]_{\emptyset} & \stackrel{\text{def}}{=} \emptyset \\
\mathcal{S}_{\mathrm{a}}[\![\text{self}]\!]_F & \stackrel{\text{def}}{=} F \\
\mathcal{S}_{\mathrm{a}}[\![\text{child}]\!]_F & \stackrel{\text{def}}{=} F\,\langle \mathbf{1}\rangle \cup \mathcal{S}_{\mathrm{a}}[\![\text{foll-sibling}]\!]_{F\langle \mathbf{1}\rangle} \\
\mathcal{S}_{\mathrm{a}}[\![\text{foll-sibling}]\!]_F & \stackrel{\text{def}}{=} F\,\langle \mathbf{2}\rangle \cup \mathcal{S}_{\mathrm{a}}[\![\text{foll-sibling}]\!]_{F\langle \mathbf{2}\rangle} \\
\mathcal{S}_{\mathrm{a}}[\![\text{prec-sibling}]\!]_F & \stackrel{\text{def}}{=} F\,\langle \overline{\mathbf{2}}\rangle \cup \mathcal{S}_{\mathrm{a}}[\![\text{prec-sibling}]\!]_{F\langle \overline{\mathbf{2}}\rangle} \\
\mathcal{S}_{\mathrm{a}}[\![\text{parent}]\!]_F & \stackrel{\text{def}}{=} F\,\langle \overline{\mathbf{1}}\rangle \cup \mathcal{S}_{\mathrm{a}}[\![\text{parent}]\!]_{F\langle \overline{\mathbf{2}}\rangle} \\
\mathcal{S}_{\mathrm{a}}[\![\text{descendant}]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{a}}[\![\text{child}]\!]_F \cup \mathcal{S}_{\mathrm{a}}[\![\text{descendant}]\!]_{(\mathcal{S}_{\mathrm{a}}[\![\text{child}]\!]_F)} \\
\mathcal{S}_{\mathrm{a}}[\![\text{desc-or-self}]\!]_F & \stackrel{\text{def}}{=} F \cup \mathcal{S}_{\mathrm{a}}[\![\text{descendant}]\!]_F \\
\mathcal{S}_{\mathrm{a}}[\![\text{ancestor}]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{a}}[\![\text{parent}]\!]_F \cup \mathcal{S}_{\mathrm{a}}[\![\text{ancestor}]\!]_{(\mathcal{S}_{\mathrm{a}}[\![\text{parent}]\!]_F)} \\
\mathcal{S}_{\mathrm{a}}[\![\text{anc-or-self}]\!]_F & \stackrel{\text{def}}{=} F \cup \mathcal{S}_{\mathrm{a}}[\![\text{ancestor}]\!]_F \\
\mathcal{S}_{\mathrm{a}}[\![\text{following}]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{a}}[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_{\mathrm{a}}[\![\text{foll-sibling}]\!]_{(\mathcal{S}_{\mathrm{a}}[\![\text{anc-or-self}]\!]_F)}\right)} \\
\mathcal{S}_{\mathrm{a}}[\![\text{preceding}]\!]_F & \stackrel{\text{def}}{=} \mathcal{S}_{\mathrm{a}}[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_{\mathrm{a}}[\![\text{prec-sibling}]\!]_{(\mathcal{S}_{\mathrm{a}}[\![\text{anc-or-self}]\!]_F)}\right)} \\
\hline
\mathtt{root}(\emptyset) & \stackrel{\text{def}}{=} \emptyset \\
\mathtt{root}(F) & \stackrel{\text{def}}{=} \{\tau_{\mathsf{Top}} \in F\} \cup \mathtt{root}(F\,\langle \overline{\mathbf{2}}\rangle) \cup \mathtt{root}(F\,\langle \overline{\mathbf{1}}\rangle)
\end{array}
$$

Fig. 10. Interpretation of XPath Expressions as Functions Between Sets of Focused Trees.

$$
\begin{aligned}
A^{\rightarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Axis} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
A^{\rightarrow}[\![\mathbf{self}]\!]_\chi &\stackrel{\text{def}}{=} \chi \\
A^{\rightarrow}[\![\mathbf{child}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \bar{\mathbf{1}} \rangle\, \chi \vee \langle \bar{\mathbf{2}} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{foll\text{-}sibling}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \bar{\mathbf{2}} \rangle\, \chi \vee \langle \bar{\mathbf{2}} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{prec\text{-}sibling}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \mathbf{2} \rangle\, \chi \vee \langle \mathbf{2} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{parent}]\!]_\chi &\stackrel{\text{def}}{=} \langle \mathbf{1} \rangle\, \mu Z.\chi \vee \langle \mathbf{2} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{descendant}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \bar{\mathbf{1}} \rangle\, (\chi \vee Z) \vee \langle \bar{\mathbf{2}} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{desc\text{-}or\text{-}self}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\chi \vee \mu Y.\, \langle \bar{\mathbf{1}} \rangle\, (Y \vee Z) \vee \langle \bar{\mathbf{2}} \rangle\, Y \\
A^{\rightarrow}[\![\mathbf{ancestor}]\!]_\chi &\stackrel{\text{def}}{=} \langle \mathbf{1} \rangle\, \mu Z.\chi \vee \langle \mathbf{1} \rangle\, Z \vee \langle \mathbf{2} \rangle\, Z \\
A^{\rightarrow}[\![\mathbf{anc\text{-}or\text{-}self}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\chi \vee \langle \mathbf{1} \rangle\, \mu Y.Z \vee \langle \mathbf{2} \rangle\, Y \\
A^{\rightarrow}[\![\mathbf{following}]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathbf{desc\text{-}or\text{-}self}]\!]_{\eta_1} \\
A^{\rightarrow}[\![\mathbf{preceding}]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathbf{desc\text{-}or\text{-}self}]\!]_{\eta_2} \\
\eta_1 &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathbf{foll\text{-}sibling}]\!]_{A^{\rightarrow}[\![\mathbf{anc\text{-}or\text{-}self}]\!]_\chi} \\
\eta_2 &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathbf{prec\text{-}sibling}]\!]_{A^{\rightarrow}[\![\mathbf{anc\text{-}or\text{-}self}]\!]_\chi}
\end{aligned}
$$

Fig. 11. Translation of XPath Axes.

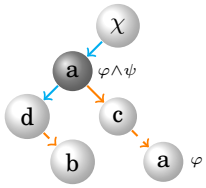
$$
\begin{aligned}
E^{\rightarrow}[\![\cdot]\!]_{\cdot} &: \mathcal{L}_{\text{XPath}} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
E^{\rightarrow}[\![/p]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p]\!]_{((\neg\langle\bar{\mathbf{1}}\rangle\top \wedge \neg\langle\bar{\mathbf{2}}\rangle\top) \wedge (\mu Y.\chi \wedge \mathbb{S} \vee \langle\mathbf{1}\rangle Y \vee \langle\mathbf{2}\rangle Y))} \\
E^{\rightarrow}[\![p]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p]\!]_{(\chi \wedge \mathbb{S})} \\
E^{\rightarrow}[\![e_1 \dashv e_2]\!]_\chi &\stackrel{\text{def}}{=} E^{\rightarrow}[\![e_1]\!]_\chi \vee E^{\rightarrow}[\![e_2]\!]_\chi \\
E^{\rightarrow}[\![e_1 \cap e_2]\!]_\chi &\stackrel{\text{def}}{=} E^{\rightarrow}[\![e_1]\!]_\chi \wedge E^{\rightarrow}[\![e_2]\!]_\chi
\end{aligned}
$$

$$
\begin{aligned}
P^{\rightarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Path} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
P^{\rightarrow}[\![p_1/p_2]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p_2]\!]_{(P^{\rightarrow}[\![p_1]\!]_\chi)} \\
P^{\rightarrow}[\![p[q]]\!]_\chi &\stackrel{\text{def}}{=} P^{\rightarrow}[\![p]\!]_\chi \wedge Q^{\leftarrow}[\![q]\!]_\top \\
P^{\rightarrow}[\![a{::}L]\!]_\chi &\stackrel{\text{def}}{=} L \wedge A^{\rightarrow}[\![a]\!]_\chi \\
P^{\rightarrow}[\![a{::}*]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![a]\!]_\chi
\end{aligned}
$$

Fig. 12. Translation of Expressions and Paths.

The parameter χ in the translation allows straightforward composition of XPath expressions. For the whole expression, the initial χ is the formula $\mu Z.(\neg\langle\bar{\mathbf{1}}\rangle\top \wedge \neg\langle\bar{\mathbf{2}}\rangle\top \wedge \neg\langle\mathbf{2}\rangle\top) \vee \langle\bar{\mathbf{1}}\rangle Z \vee \langle\bar{\mathbf{2}}\rangle Z$, which states that the root has no right child, as required by our interpretation domain $\mathcal{F}^*$.

Fig. 13 illustrates the translation of the XPath expression "child::a[child::b]". This expression selects all "a" child nodes of a given context which have at least one "b" child.



Translated Expression: child::a[child::b]

$$\underbrace{a \wedge (\mu X.\, \langle \bar{1} \rangle\, (\chi \wedge \textcircled{S}) \vee \langle \bar{2} \rangle\, X)}_{\varphi} \wedge \underbrace{\langle 1 \rangle\, \mu Y.b \vee \langle 2 \rangle\, Y}_{\psi}$$

Fig. 13. XPath Translation Example.

The translated $\mathcal{L}_\mu$ formula holds for "a" nodes which are selected by the expression. The first part of the translated formula, φ, corresponds to the step "child::a" which selects candidates "a" nodes. The second part, ψ, navigates downward in the subtrees of these candidate nodes to verify that they have at least one immediate "b" child.

Note that without converse programs we would have been unable to differentiate selected nodes from nodes whose existence is tested: we must state properties on both the ancestors and the descendants of the selected node. The fact that the $\mathcal{L}_\mu$ logic is equipped with both forward and converse programs is important for supporting XPath[7]. Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

XPath composition construct p_1/p_2 translates into formula composition in $\mathcal{L}_\mu$, such that the resulting formula holds for all nodes accessed through p_2 from those nodes accessed through p_1 from χ. The translation of the branching construct $p[q]$ significantly differs. The resulting formula must hold for all nodes that can be accessed through p and from which q holds. To preserve semantics, the translation of $p[q]$ stops the "selecting navigation" to those nodes reached by p, then filters them depending on whether q holds or not. We express this by introducing a dual formal translation function for XPath qualifiers, noted $Q^{\leftarrow}[\![q]\!].$ and defined in Fig. 14, that performs "filtering" instead of navigation. Specifically, $P^{\rightarrow}[\![\cdot]\!].$ can be seen as the "navigational" translating function: the translated formula holds for target nodes of the given path. On the opposite, $Q^{\leftarrow}[\![\cdot]\!].$ can be seen as the "filtering" translating function: it states the existence of a path *without moving to its result*. The translated formula $Q^{\leftarrow}[\![q]\!]_\chi$ (respectively $P^{\leftarrow}[\![p]\!]_\chi$) holds for nodes from which there exists a qualifier q (respectively a path p) leading to a node verifying χ.

XPath translation is based on these two translating "modes", the first one being used for paths and the second one for qualifiers. Whenever the "filtering" mode is entered, it will never be left.

The translation of paths inside qualifiers is also given in Fig. 14. It uses the translation for axes and is based on XPath symmetry: $\mathit{symmetric}(a)$ denotes the symmetric XPath axis corresponding to the axis a (for instance $\mathit{symmetric}(\text{child}) = \text{parent}$).

We may now state that our translation is correct, by relating the interpretation of an XPath formula applied to some set of trees to the interpretation of its translation, by stating that the translation of a formula is cycle-free, and by giving a bound in the size of this translation.

[7]One may ask whether it is possible to eliminate upward navigation at the XPath level but it is well known that such XPath rewriting techniques cause exponential blow-ups of expression sizes [Olteanu et al. 2002].

$$
\begin{aligned}
Q^{\leftarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Qualif} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
Q^{\leftarrow}[\![q_1 \text{ and } q_2]\!]_\chi &\stackrel{\text{def}}{=} Q^{\leftarrow}[\![q_1]\!]_\chi \wedge Q^{\leftarrow}[\![q_2]\!]_\chi \\
Q^{\leftarrow}[\![q_1 \text{ or } q_2]\!]_\chi &\stackrel{\text{def}}{=} Q^{\leftarrow}[\![q_1]\!]_\chi \vee Q^{\leftarrow}[\![q_2]\!]_\chi \\
Q^{\leftarrow}[\![\text{not } q]\!]_\chi &\stackrel{\text{def}}{=} \neg\, Q^{\leftarrow}[\![q]\!]_\chi \\
Q^{\leftarrow}[\![p]\!]_\chi &\stackrel{\text{def}}{=} P^{\leftarrow}[\![p]\!]_\chi
\end{aligned}
$$

$$
\begin{aligned}
P^{\leftarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Path} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
P^{\leftarrow}[\![p_1/p_2]\!]_\chi &\stackrel{\text{def}}{=} P^{\leftarrow}[\![p_1]\!]_{(P^{\leftarrow}[\![p_2]\!]_\chi)} \\
P^{\leftarrow}[\![p[q]]\!]_\chi &\stackrel{\text{def}}{=} P^{\leftarrow}[\![p]\!]_{(\chi \wedge Q^{\leftarrow}[\![q]\!]_\top)} \\
P^{\leftarrow}[\![a{::}L]\!]_\chi &\stackrel{\text{def}}{=} A^{\leftarrow}[\![a]\!]_{(\chi \wedge L)} \\
P^{\leftarrow}[\![a{::}*]\!]_\chi &\stackrel{\text{def}}{=} A^{\leftarrow}[\![a]\!]_\chi
\end{aligned}
$$

$$
\begin{aligned}
A^{\leftarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Axis} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
A^{\leftarrow}[\![a]\!]_\chi &\stackrel{\text{def}}{=} A^{\rightarrow}[\![\mathit{symmetric}(a)]\!]_\chi
\end{aligned}
$$

Fig. 14. Translation of Qualifiers.

We restrict the sets of trees to which an XPath formula may be applied to those that may be denoted by an $\mathcal{L}_\mu$ formula. This restriction will be justified in Section A.4 where we show that every regular tree language may be translated to an $\mathcal{L}_\mu$ formula.

PROPOSITION A.1 (TRANSLATION CORRECTNESS). *The following hold for an XPath expression e and a $\mathcal{L}_\mu$ formula φ denoting a set of focused trees, with $\psi = E^{\rightarrow}[\![e]\!]_\varphi$:*

(1) $[\![\psi]\!]_\emptyset = \mathcal{S}_{\mathrm{e}}[\![e]\!]_{[\![\varphi]\!]_\emptyset}$
(2) *ψ is cycle-free*
(3) *the size of ψ is linear in the size of e and φ*

PROOF. The proof uses a structural induction that "peels off" the compositional layers of each set of rules over focused trees. The cycle-free part follows from the fact that translated fixpoint formulas are closed and there is no nesting of modalities with converse programs between a fixpoint variable and its binder. Each XPath navigation step is cycle-free, and their composition yields a proper nesting of fixpoint formulas which is also cycle-free. Fig. 15 illustrates this on an typical example. Finally, formal translations do not duplicate any subformula of arbitrary length. □

A.4. Embedding Regular Tree Languages

Several formalisms exist for describing types of XML documents (e.g. DTD, XML Schema, Relax NG). In this paper we embed regular tree types into $\mathcal{L}_\mu$. Regular tree types gather most of the schemas occuring in practice [Murata et al. 2005][8]. We rely on a straightforward isomorphism between unranked regular tree types and binary regular tree types [Hosoya et al. 2005]. Assuming a countably infinite set of type variables

[8]Notice, however, that we do not consider counting nor interleaving features that can be found in e.g. XML Schemas. These features are beyond the scope of this paper: see [Bárcenas et al. 2009] for a preliminary work on how to integrate counting constraints in such a logic.

Translation of following-sibling::a/preceding-sibling::b
into $\mathcal{L}_\mu$: $b \wedge [\mu Y.\, \langle \mathbf{2} \rangle\, (a \wedge (\mu Z.\, \langle \bar{\mathbf{2}} \rangle\, \textcircled{S} \vee \langle \bar{\mathbf{2}} \rangle\, Z)) \vee \langle \mathbf{2} \rangle\, Y]$

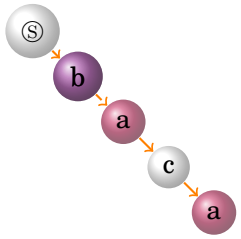


Fig. 15. Example of Back and Forth – Yet Cycle-Free – XPath Navigation.

ranged over by X, binary regular tree type expressions are defined as follows:

$$\begin{array}{rlll} \mathcal{L}_{BT} \ni T & ::= & & \text{tree type expression} \\ & & \emptyset & \text{empty set} \\ & | & \varepsilon & \text{leaf} \\ & | & T_1 \vee T_2 & \text{union} \\ & | & L(X_1, X_2) & \text{label} \\ & | & \text{let } \overline{X_i.T_i} \text{ in } T & \text{binder} \end{array}$$

We refer the reader to [Hosoya et al. 2005] for the denotational semantics of regular tree languages, and directly introduce their translation into $\mathcal{L}_\mu$:

$$\begin{aligned} [\![\cdot]\!] &: \mathcal{L}_{BT} \to \mathcal{L}_\mu \\ [\![T]\!] &\stackrel{\text{def}}{=} \bot \quad \text{for} \quad T = \emptyset, \varepsilon \\ [\![T_1 \vee T_2]\!] &\stackrel{\text{def}}{=} [\![T_1]\!] \vee [\![T_2]\!] \\ [\![L(X_1, X_2)]\!] &\stackrel{\text{def}}{=} L \wedge \mathit{succ}_1(X_1) \wedge \mathit{succ}_2(X_2) \\ [\![\text{let } \overline{X_i.T_i} \text{ in } T]\!] &\stackrel{\text{def}}{=} \mu(X_i = [\![T_i]\!]) \text{ in } [\![T]\!] \end{aligned}$$

where the function $\mathit{succ}_\cdot(\cdot)$ takes care of setting the type frontier:

$$\mathit{succ}_\alpha(X) = \begin{cases} \neg \langle \alpha \rangle \top & \textit{if } X \textit{ is bound to } \varepsilon \\ \neg \langle \alpha \rangle \top \vee \langle \alpha \rangle X & \textit{if } \mathit{nullable}(X) \\ \langle \alpha \rangle X & \textit{if not } \mathit{nullable}(X) \end{cases}$$

according to the predicate $\mathit{nullable}(X)$ which indicates whether the type $T \neq \varepsilon$ bound to X contains the empty tree. For example, Fig. 17 gives the translation of a DTD fragment of the Wikipedia encyclopedia [Voss 2007] shown on Fig. 16.

Note that the translation of a regular tree type uses only downward modalities since it describes the allowed subtrees at a given context. No additional restriction is imposed on the context from which the type definition starts. In particular, navigation is allowed in the upward direction so that we can support type constraints for which we have only partial knowledge in a given direction. However, when we know the position of the root, conditions similar to those of absolute paths are added in the form of additional formulas describing the position that need to be satisfied. This is particularly useful when a regular type is used by an XPath expression that starts its navigation at the root ($/p$) since the path will not go above the root of the type.

On the other hand, if the type is compared with another type (typically to check inclusion of the result of an XPath expression in this type), then there is no restriction

```
<!ELEMENT article (meta, (text | redirect))>
<!ELEMENT meta (title, status?, interwiki*, history?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT interwiki (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT history (edit)+>
<!ELEMENT edit (status?, interwiki*, (text | redirect)?)>
<!ELEMENT redirect EMPTY>
<!ELEMENT text (#PCDATA)>
```

Fig. 16. A Fragment of the DTD of the Wikipedia Encyclopedia.

```
let
  $X11=(interwiki & ~<1>T & (~<2>T | <2>($X11 | (history & ~<2>T & <1>$X6)))),
  $X6=(edit & (~<1>T | <1>(status & ~<1>T &
       (~<2>T | <2>($X8 | (text & ~<1>T & ~<2>T) | (redirect & ~<1>T & ~<2>T)))))
       & (~<2>T | <2>$X6)),
  $X8=(interwiki & ~<1>T & (~<2>T | <2>($X8 | (text & ~<1>T & ~<2>T) |
       (redirect & ~<1>T & ~<2>T))))
in
  (article & <1>(meta & <1>(title & ~<1>T & (~<2>T | <2>($X11 | (history &
     ~<2>T & <1>$X6) | (status & ~<1>T & (~<2>T | <2>($X11 | (history & ~<2>T &
       <1>$X6)))))) & <2>((text & ~<1>T & ~<2>T) | (redirect & ~<1>T & ~<2>T))))
```

Fig. 17. The $\mathcal{L}_\mu$ Formula for the DTD of Fig. 16.

as to where the root of the type is (our translation does not impose the chosen node to be at the root). This is particularly useful since an XPath expression usually returns a set of nodes deep in the tree which we may compare to this partially defined type.