Efficiently Deciding Mu-calculus with Converse over Finite Trees

Pierre Genevès, Nabil Layaïda, Alan Schmitt

A

Efficiently Deciding μ-calculus with Converse over Finite Trees

Pierre Genevès, CNRS
Nabil Layaïda, INRIA
Alan Schmitt, INRIA

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 which is 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 practically significant size.

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. and Layaïda, N. and Schmitt A. 2011. Efficiently Deciding μ-calculus with Converse over Finite Trees. ACM Trans. Comput. Logic V, N, Article A (January YYYY), 39 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 as well as its effective implementation. 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 which is 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 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 practically significant size.

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 Dexter Kozen into the version mostly used nowadays [Kozen 1983]. Several modal logics can be encoded in the μ-calculus, including linear temporal logic,

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@inria.fr.

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

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. The work found in [Tanabe et al. 2005] points this out neatly: "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 that are most closely related 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 a research effort has been focusing on temporal logics that can handle both directions of edges as modalities in order to reason about both the "past" and the "future". Although converse modalities do not provide additional 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 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 the emptiness problem, complex operations are required including determinization of parity automata [Safra 1988]. No implementation is reported.

The best known complexity for deciding MC is obtained through reduction to the emptiness problem of alternating tree automata on infinite trees, which can be done in $2^{\mathcal{O}(n^4 \cdot \log\, n)}$, where n is the size of the formula [Grädel et al. 2002]. Again, no actual implementation has been reported.

A notable exception is the work found in [Tanabe et al. 2005; Tanabe et al. 2008], that provides an implementation of a decision procedure for the alternation-free fragment of the μ-calculus with converse (AFMC), whose time complexity is $2^{\mathcal{O}(n \cdot \log\, n)}$, noticing in passing that the decision procedure for the AFMC is less complex than the one for the MC, as expected. 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. In applications of the satisfiability-checking problem, relevant models often consist only of the set 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 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). Owing to an additional logical formula that encodes König's lemma, models can be restricted to be binary-branching finite trees. However, the au-

thors notice that the performance of the decision procedure may not be very attractive in this setting [Tanabe et al. 2005]. The authors do not comment on the reasons, 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. Finally, if the fixpoint contains a satisfying (tree) structure then the formula is judged as satisfiable. 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. As a consequence, the resulting performance of the decision procedure proposed in this article, 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].

Automata-based 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 and parity games (see [Calvanese et al. 2008; 2009], in which it is also mentioned that it is practically infeasible to apply the symbolic approach in the the infinite tree setting).

A more appropriate automata version for finite trees is called weak alternating two-way tree automata (2WATA) which are simpler when compared to infinite tree automata-theoretic techniques. However, they require a conversion to nondeterministic finite tree automata (NTFA) 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.

In fact, neither of the papers [Libkin and Sirangelo 2008; Calvanese et al. 2008; 2009; Libkin and Sirangelo 2010; Calvanese et al. 2010] provide an implementation. [Calvanese et al. 2008] even 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.

In [Libkin and Sirangelo 2010], the authors acknowledge the fact that they provide an alternative version of our pioneering work described in our extended abstract [Genevès et al. 2007]. 2WATA are interesting to shorten some proofs but they do not simplify the implementation.

The present work can be regarded as the pioneering and 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, in the whole present paper we focus on the native modal logic in the finite case. This also emphasizes the fact that bottom-up construction of the finite tree model and cycle-freeness come naturally and show 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 restrictions called 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 in a single fixpoint operator. This makes our logic closed under negation, and also provide 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

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 previous siblings and its parent, including its parent context recursively. Exploring such a structure has the advantage to preserve all information, which is quite useful when considering forward and backward navigation.

Formally, we assume an alphabet Σ of labels, ranged over by σ. The syntax of our data model is as follows.

t	$::=$	$\sigma[tl]$	tree
tl	$::=$		list of trees
		ϵ	empty list
	$\mid$	$t :: tl$	cons cell
c	$::=$		context
		(tl, Top, tl)	root of the tree
	$\mid$	$(tl, c[\sigma], tl)$	context node
f	$::=$	(t, c)	focused tree

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be Top if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

$\mathcal{L}_\mu \ni \varphi, \psi ::=$		formula
	$\top$	true
$\mid$	$\sigma \mid \neg\sigma$	atomic proposition (negated)
$\mid$	$\textcircled{\scriptsize S} \mid \neg\textcircled{\scriptsize S}$	start proposition (negated)
$\mid$	X	variable
$\mid$	$\varphi \vee \psi$	disjunction
$\mid$	$\varphi \wedge \psi$	conjunction
$\mid$	$\langle a \rangle \varphi \mid \neg \langle a \rangle \top$	existential (negated)
$\mid$	$\mu \overline{X_i = \varphi_i}$ in ψ	least n-ary fixpoint
$\mid$	$\nu \overline{X_i = \varphi_i}$ in ψ	greatest n-ary fixpoint

Fig. 1. Logic formulas

In order to deal with decision problems such as containment of queries (i.e. binary relations over tree nodes), we need to represent in a focused tree the place where the evaluation of a query was started. To this end, we use a *start mark*, often simply called "mark" in the following. We thus consider focused trees where a single tree or a single context node is marked, as in $\sigma^{\textcircled{\scriptsize S}}[tl]$ or $(tl, c[\sigma^{\textcircled{\scriptsize S}}], tl)$. When the presence of the mark is unknown, we write it as $\sigma^\circ[tl]$. We write $\mathcal{F}$ for the set of finite focused trees containing a single mark. The *name* of a focused tree is defined as $\mathtt{nm}(\sigma^\circ[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions, or *modalities*, that can be followed: for a focused tree f, $f\,\langle 1\rangle$ changes the focus to the first child of the current tree, $f\,\langle 2\rangle$ changes the focus to the next sibling of the current tree, $f\,\langle \overline{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\,\langle \overline{2}\rangle$ changes the focus to the previous sibling.

Formally, we have:

$$(\sigma^\circ[t :: tl], c)\,\langle 1\rangle \stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma^\circ], tl))$$
$$(t, (tl_l, c[\sigma^\circ], t' :: tl_r))\,\langle 2\rangle \stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma^\circ], tl_r))$$
$$(t, (\epsilon, c[\sigma^\circ], tl))\,\langle \overline{1}\rangle \stackrel{\text{def}}{=} (\sigma^\circ[t :: tl], c)$$
$$(t', (t :: tl_l, c[\sigma^\circ], tl_r))\,\langle \overline{2}\rangle \stackrel{\text{def}}{=} (t, (tl_l, c[\sigma^\circ], t' :: tl_r))$$

When the focused tree does not have the required shape, these operations are not defined.

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.

3.1. Formulas

In the following, we use an overline bar to denote tuples. For instance, we write $\overline{X_i = \varphi_i}$ for $(X_1 = \varphi_1; X_2 = \varphi_2; \ldots; X_n = \varphi_n)$. Tuples of variables, such as $\overline{X_i}$, are often identified to sets.

In the following definitions, $a \in \{1, 2, \overline{1}, \overline{2}\}$ are *programs*. Atomic propositions σ correspond to labels from Σ. We also assume that $\overline{\overline{a}} = a$. Formulas defined in Figure 1 include the truth predicate, atomic propositions (denoting the name of the tree in focus), start propositions (denoting the presence of the start mark), disjunction and conjunc-

tion of formulas, formulas under an existential (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 "$\iota\overline{X = \varphi}$ in φ".

3.1.1. Normal form. We define *recursions* of a formula φ as the set of recursive sub-formulas in φ.

$$\begin{aligned}
\mathcal{R}(\varphi) &\stackrel{\text{def}}{=} \emptyset \qquad \text{for } \varphi = \top, X, \sigma, \neg\sigma, \textcircled{\scriptsize S}, \neg\textcircled{\scriptsize S}, \neg\langle a\rangle\top \\
\mathcal{R}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \mathcal{R}(\varphi) \cup \mathcal{R}(\psi) \\
\mathcal{R}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \mathcal{R}(\varphi) \cup \mathcal{R}(\psi) \\
\mathcal{R}(\iota\overline{X_i = \varphi_i} \text{ in } \varphi_j) &\stackrel{\text{def}}{=} \{\iota\overline{X_i = \varphi_i} \text{ in } \varphi_j\} \cup \bigcup_i \mathcal{R}(\varphi_i)
\end{aligned}$$

As in [Kozen 1983], we assume formulas are written such that every bound variable is distinct, and we denote a sub-formula $\iota\overline{X_i = \varphi_i}$ in φ_j by ιX_j. The set of *recursive identifiers* of a formula φ, written $\mathcal{I}(\varphi)$ is:

$$\mathcal{I}(\varphi) \stackrel{\text{def}}{=} \bigcup_{\iota\overline{X_i=\varphi_i} \text{ in } \varphi_j \in \mathcal{R}(\varphi)} \bigcup_i \{\iota X_i\}$$

For $\varphi = (\iota\overline{X_i = \varphi_i} \text{ in } \varphi_j)$ we define $\text{exp}(\varphi) \stackrel{\text{def}}{=} \varphi_j\{\overline{\iota\overline{X_i=\varphi_i} \text{ in } \varphi_i}/_{X_i}\}$ which denotes the formula φ_j in which every occurrence of a X_i is replaced by $(\iota\overline{X_i = \varphi_i} \text{ in } \varphi_i)$. This operation is extended homomorphically to every other operator (e.g., $\text{exp}(\varphi \vee \psi) = \text{exp}(\varphi) \vee \text{exp}(\psi)$ or $\text{exp}(\langle a\rangle\varphi) = \langle a\rangle\,\text{exp}(\varphi)$).

We say that φ is in normal form iff $\mathcal{R}(\varphi) = \mathcal{R}(\text{exp}(\varphi))$. As described by [Kozen 1983], one reaches a normal form by expanding each fixpoint once, starting from the outer ones; moreover, it is shown that the set of distinct sub-formulas of a normal form is no larger than the initial formula.

3.2. Model

We define in Figure 2 an interpretation of our formulas as subsets of $\mathcal{F}$, the set of finite focused trees with a single start mark. The interpretation of the n-ary fixpoints first compute the smallest or largest interpretation for each φ_i, bind the resulting sets T_i to the variables X_i, then returns the interpretation of ψ.

To illustrate the interpretation of fixpoints, consider the two following formulas $\varphi = \mu X.\langle 1\rangle X \vee \langle\overline{1}\rangle X$ and $\psi = \nu X.\langle 1\rangle X \vee \langle\overline{1}\rangle X$, which respectively expand to $\mu\overline{X = \langle 1\rangle X \vee \langle\overline{1}\rangle X}$ in $\langle 1\rangle X \vee \langle\overline{1}\rangle X$ and $\nu\overline{X = \langle 1\rangle X \vee \langle\overline{1}\rangle X}$ in $\langle 1\rangle X \vee \langle\overline{1}\rangle X$.

The interpretation of φ is straightforward: associating the empty set to X, we have

$$[\![\langle 1\rangle X \vee \langle\overline{1}\rangle X]\!]_{[\emptyset/X]} \subseteq \emptyset$$

thus $[\![\varphi]\!] = \emptyset$. 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 $f_1 = (a[b[\epsilon]], T)$, where T is the top-level context $(\epsilon, Top, \epsilon)$. We do not specify the position of the mark as it is not used in the query: it could be anywhere. Let $f_2 = f_1\langle 1\rangle$, that is the tree $(b[\epsilon], (\epsilon, T[a], \epsilon))$. We thus have $f_2\langle\overline{1}\rangle = f_1$.

$$\begin{aligned}
[\![\top]\!]_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\sigma]\!]_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = \sigma\}\\
[\![X]\!]_V &\stackrel{\text{def}}{=} V(X) & [\![\neg\sigma]\!]_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq \sigma\}\\
[\![\varphi \vee \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V & [\![\text{Ⓢ}]\!]_V &\stackrel{\text{def}}{=} \left\{f \mid f = (\sigma^{\text{Ⓢ}}[tl], c)\right\}\\
[\![\varphi \wedge \psi]\!]_V &\stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V & [\![\neg\text{Ⓢ}]\!]_V &\stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\}
\end{aligned}$$

$$\begin{aligned}
[\![\langle a\rangle\, \varphi]\!]_V &\stackrel{\text{def}}{=} \{f\,\langle\overline{a}\rangle \mid f \in [\![\varphi]\!]_V \wedge f\,\langle\overline{a}\rangle \text{ defined}\}\\
[\![\neg\,\langle a\rangle\, \top]\!]_V &\stackrel{\text{def}}{=} \{f \mid f\,\langle a\rangle \text{ undefined}\}\\
[\![\mu\overline{X_i = \varphi_i} \text{ in } \psi]\!]_V &\stackrel{\text{def}}{=} \text{let } \overline{T_i} = \left(\bigcap\left\{\overline{T_i \subseteq \mathcal{F}} \mid \overline{[\![\varphi_i]\!]_{V[\overline{T_i/X_i}]} \subseteq T_i}\right\}\right)_i\\
&\qquad \text{in } [\![\psi]\!]_{V[\overline{T_i/X_i}]}\\
[\![\nu\overline{X_i = \varphi_i} \text{ in } \psi]\!]_V &\stackrel{\text{def}}{=} \text{let } \overline{T_i} = \left(\bigcup\left\{\overline{T_i \subseteq \mathcal{F}} \mid \overline{T_i \subseteq [\![\varphi_i]\!]_{V[\overline{T_i/X_i}]}}\right\}\right)_i\\
&\qquad \text{in } [\![\psi]\!]_{V[\overline{T_i/X_i}]}
\end{aligned}$$

Fig. 2. Interpretation of formulas

Finally, let V be the mapping $[\{f_1; f_2\}/X]$. We compute as follow:

$$\begin{aligned}
&[\![\langle 1\rangle\, X \vee \langle\overline{1}\rangle\, X]\!]_V\\
&= [\![\langle 1\rangle\, X]\!]_V \cup [\![\langle\overline{1}\rangle\, X]\!]_V\\
&= \left\{f\,\langle\overline{1}\rangle \mid f \in [\![X]\!]_V \wedge f\,\langle\overline{1}\rangle \text{ defined}\right\} \cup \{f\,\langle 1\rangle \mid f \in [\![X]\!]_V \wedge f\,\langle 1\rangle \text{ defined}\}\\
&= \{f_1\} \cup \{f_2\}
\end{aligned}$$

thus $V(X) \subseteq [\![\langle 1\rangle\, X \vee \langle\overline{1}\rangle\, X]\!]_V$, hence by definition of the largest fixpoint, we have $f_1 \in [\![\psi]\!]_\emptyset$.

As shown in [Kozen 1983], for any valuation V, if ψ is a normal form of φ, then $[\![\varphi]\!]_V = [\![\psi]\!]_V$.

3.3. Cycle-Free Formulas

As shown in §3.2, the smallest and greatest fixpoints do not coincide. We now introduce a restriction that will make them collapse, requiring formulas to be *cycle-free*. To define this notion, we first need to introduce the set of *paths* of a formula. Given a formula φ, the set of its paths $\mathcal{P}(\varphi)$ is the set of sequential chains of modalities contained in the formula. Writing ϵ for the empty path, we have the following.

$$\begin{aligned}
\mathcal{P}(\langle a\rangle\, \varphi) &= \{\langle a\rangle\, p \mid p \in \mathcal{P}(\varphi)\}\\
\mathcal{P}(\varphi \vee \psi) &= \mathcal{P}(\varphi) \cup \mathcal{P}(\psi)\\
\mathcal{P}(\varphi \wedge \psi) &= \mathcal{P}(\varphi) \cup \mathcal{P}(\psi)\\
\mathcal{P}(\varphi) &= \epsilon \qquad \text{otherwise}
\end{aligned}$$

Note that this notion is very syntactic, and unfolding a fixpoint in a formula may change its set of paths.

A *modality cycle* in a path is a sub-sequence of the form $\langle a\rangle\, \langle\overline{a}\rangle$. We new define *cycle-free formulas* as formulas for which there is a bound in the number of modality cycles of their paths, independent on the unfolding.

$$\frac{\varphi = \top, \sigma, \neg\sigma, \text{Ⓢ}, \text{ or } \neg\text{Ⓢ}}{\Delta \parallel \Gamma \vdash_I^R \varphi} \qquad \frac{\Delta \parallel \Gamma \vdash_I^R \varphi \qquad \Delta \parallel \Gamma \vdash_I^R \psi}{\Delta \parallel \Gamma \vdash_I^R \varphi \vee \psi} \qquad \frac{\Delta \parallel \Gamma \vdash_I^R \varphi \qquad \Delta \parallel \Gamma \vdash_I^R \psi}{\Delta \parallel \Gamma \vdash_I^R \varphi \wedge \psi}$$

$$\frac{}{\Delta \parallel \Gamma \vdash_I^R \neg \langle a \rangle \top} \qquad \frac{\Delta \parallel (\Gamma \lhd \langle a \rangle) \vdash_I^R \varphi}{\Delta \parallel \Gamma \vdash_I^R \langle a \rangle \varphi}$$

$$\frac{\forall X_j \in \overline{X_i}. \left((\Delta + \overline{X_i : \varphi_i}) \parallel (\Gamma + \overline{X_i : \_}) \vdash_{I \setminus \overline{X_i}}^{R \setminus \overline{X_i}} \varphi_j \right) \qquad \Delta \parallel \Gamma \vdash_{I \cup \overline{X_i}}^{R \setminus \overline{X_i}} \psi}{\Delta \parallel \Gamma \vdash_I^R \mu \overline{X_i = \varphi_i} \text{ in } \psi}$$

$$\frac{\forall X_j \in \overline{X_i}. \left((\Delta + \overline{X_i : \varphi_i}) \parallel (\Gamma + \overline{X_i : \_}) \vdash_{I \setminus \overline{X_i}}^{R \setminus \overline{X_i}} \varphi_j \right) \qquad \Delta \parallel \Gamma \vdash_{I \cup \overline{X_i}}^{R \setminus \overline{X_i}} \psi}{\Delta \parallel \Gamma \vdash_I^R \nu \overline{X_i = \varphi_i} \text{ in } \psi}$$

$$\text{NoRec}\ \frac{X \in R \qquad \Gamma(X) = \langle a \rangle}{\Delta \parallel \Gamma \vdash_I^R X} \qquad \text{Rec}\ \frac{X \notin R \qquad \Delta \parallel \Gamma \vdash_I^{R \cup \{X\}} \Delta(X)}{\Delta \parallel \Gamma \vdash_I^R X} \qquad \text{Ign}\ \frac{X \in I}{\Delta \parallel \Gamma \vdash_I^R X}$$

Fig. 3. Deciding Cycle-free Formulas

DEFINITION 3.1 (CYCLE-FREE FORMULA). *A formula φ is cycle-free iff there exists an integer n such that for any unfolding $\psi \in unf(\varphi)$, for any path $p \in \mathcal{P}(\psi)$, the number of modality cycles in p is strictly smaller than n.*

For instance, the formula "$\mu\overline{X = \langle 1 \rangle (\top \vee \langle \overline{1} \rangle X)}$ in X" is not cycle free: for any integer n, there is an unfolding of the formula such that a path with n modality cycles exists. Similarly, the formulas φ and ψ in §3.2 are also not cycle free. On the other hand, the formula "$\mu\overline{X = \langle 1 \rangle (X \vee Y),\ Y = \langle \overline{1} \rangle (Y \vee \top)}$ in X" is cycle free: there is at most one modality cycle for each path, independently of the number of unfoldings of its fixpoint.

Cycle-free formulas have a very interesting property, which we now describe. To test whether a tree satisfies a formula, one may define a straightforward inductive relation between trees and formulas that only holds when the root of the tree satisfies the formula, unfolding fixpoints if necessary. Given a tree, if a formula φ is cycle free, then every node of the tree will be tested a finite number of time against any given subformula of φ. The intuition behind this property, which holds a central role in the proof of lemma 3.4, is the following. If a tree node is tested an infinite number of times against a subformula, then there must be a cycle in the navigation in the tree, corresponding to some modalities occurring in the subformula, between one occurrence of the test and the next one. As we consider trees, the cycle implies there is a modality cycle in the formula (as unbalanced cycles of the form $\langle 1 \rangle \langle 2 \rangle \langle \overline{1} \rangle \langle \overline{2} \rangle$ cannot occur). Hence the number of modality cycles in any expansion of φ is unbounded, thus the formula is not cycle free.

Although it provides the correct intuition, Definition 3.1 is not very practical. We give in Figure 3 a decision procedure, in the form of an inductive relation, that ensures that a formula is cycle free. In the judgement $\Delta \parallel \Gamma \vdash_I^R \varphi$ of Figure 3, Δ is an environment binding some recursion variables to their formulas, Γ binds variables to modalities, R is a set of variables that have already been expanded (see below), and I is a set of variables already checked.

The environment Γ used to derive the judgement consists of bindings from variables (from enclosing fixpoint operators) to modalities. A modality may be $\_$ (no information is known about the variable), $\langle a \rangle$ (the last modality taken $\langle a \rangle$ was consistent), or $\bot$ (a cycle has been detected). A formula is not cycle free if an occurrence of a variable under a fixpoint operator is either not under a modality (in this case $\Gamma(X) = \_$), or is under a cycle ($\Gamma(X) = \bot$). Cycle detection uses an auxiliary operator to detect modality cycles:

$$\Gamma \lhd \langle a \rangle \stackrel{\text{def}}{=} \{X : (\Gamma(X) \lhd \langle a \rangle)\}$$

where

$\cdot \lhd \cdot$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \overline{1} \rangle$	$\langle \overline{2} \rangle$
$\_$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \overline{1} \rangle$	$\langle \overline{2} \rangle$
$\langle 1 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\bot$	$\langle \overline{2} \rangle$
$\langle 2 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \overline{1} \rangle$	$\bot$
$\langle \overline{1} \rangle$	$\bot$	$\langle 2 \rangle$	$\langle \overline{1} \rangle$	$\langle \overline{2} \rangle$
$\langle \overline{2} \rangle$	$\langle 1 \rangle$	$\bot$	$\langle \overline{1} \rangle$	$\langle \overline{2} \rangle$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

To check that mutually recursive formulas are cycle-free, we proceed the following way. When a mutually recursive formula is encountered, for instance $\mu \overline{X_i = \varphi_i}$ in ψ, we check every recursive binding. Because of mutual recursion, we cannot check formulas independently and we need to expand a variable the first time it is encountered (rule REC). However there is no need to expand it a second time (rule NOREC). When checking ψ, as the formulas bound to the enclosing recursion have been checked to be cycle free, there is no need to further check these variables (rule IGN). To account for shadowing of variables, we make sure that newly bound recursion variables are removed from I and R when checking a recursion. One may easily prove that if $\Delta \parallel \Gamma \vdash^R_I \varphi$ holds, then $I \cap R = \emptyset$.

This relation detects when a formula is not cycle free because, in this case, there must be a recursive binding of X_i to φ_i such that $\varphi_i\{^{\varphi_i}/_{X_i}\}\{^{\overline{\varphi_j}}/_{\overline{X_j}}\}$ exhibits a modality cycle above X_i, where the X_j are other recursion variables already defined (either in the recursion defining X_i or in an enclosing recursion definition). Cycles are thus detected unfolding every recursive definition once in every formula.

Note that we may wrongly detect a formula as having a cycle. For instance, the formula $\mu X. \langle 1 \rangle \langle \overline{1} \rangle X$ in $\top$ is said to include a cycle even though the variable on which the cycle occurs never needs to be expanded. We have found that in practice this approximation is precise enough to check formulas entered by hand. We state that our approximation is correct.

LEMMA 3.2. *Let φ be a formula. If $\emptyset \parallel \emptyset \vdash^\emptyset_\emptyset \varphi$, then φ is cycle-free.*

Proof: [Sketch] We proceed by contraposition: we assume φ is not cycle-free, and show that we cannot derive $\emptyset \parallel \emptyset \vdash^\emptyset_\emptyset \varphi$. As φ is not cycle-free, it is because a modality and its inverse are under a recursion (either directly, or through a conjunction or disjunction), or because a recursion variable is not guarded by a modality. Focusing on this fixpoint, we can show that after expanding the variable (rule REC), when we encounter the variable again (rule NOREC), we then either have $\Gamma(X) = \bot$ or $\Gamma(X) = \_$, thus the derivation is not possible. □

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 a stronger result that states that a given focused tree is in the interpretation of a cycle-free formula φ if it is in the interpretation of a finite unfolding of the formula $unf_f(\varphi)$.

The definition of finite unfolding below is very similar to Definition 3.1.1. The only difference is in the handling of a fixpoint: the fixpoint itself is not included in the set of unfoldings. As a consequence, formulas in $unf_f(\varphi)$ do not contain any fixpoint operator and correspond to the finite unfoldings of φ followed by the erasure of its fixpoints. Note that if there is no base case to a fixpoint of a formula, as in $\mu X.\, \langle 1\rangle \left\langle \overline{1}\right\rangle X$ in X, then the finite unfolding of this formula will be the empty set.

DEFINITION 3.3 (FINITE INTERPRETATION). *Let φ be a formula in normal form, and N a mapping from $\mathcal{I}(\varphi)$ to $\mathbb{N}$. The* finite interpretation *of a φ is inductively defined as follows.*

$$
\begin{aligned}
unf_f(\varphi) &\stackrel{def}{=} \{\varphi\} \textit{ for } \varphi = \top, \sigma, \neg\sigma, \text{Ⓢ}, \neg\text{Ⓢ}, X, \neg\langle a\rangle \top \\
unf_f(\varphi \vee \psi) &\stackrel{def}{=} \{\varphi' \vee \psi' \mid \varphi' \in unf_f(\varphi), \psi' \in unf_f(\psi)\} \\
unf_f(\varphi \wedge \psi) &\stackrel{def}{=} \{\varphi' \wedge \psi' \mid \varphi' \in unf_f(\varphi), \psi' \in unf_f(\psi)\} \\
unf_f(\langle a\rangle \varphi) &\stackrel{def}{=} \{\langle a\rangle \varphi' \mid \varphi' \in unf_f(\varphi)\} \\
unf_f(\mu\overline{X_i = \varphi_i} \textit{ in } \psi) &\stackrel{def}{=} unf_f(\psi\{^{\mu\overline{X_i=\varphi_i} \textit{ in } \varphi_i}/_{X_i}\}) \\
unf_f(\nu\overline{X_i = \varphi_i} \textit{ in } \psi) &\stackrel{def}{=} unf_f(\psi\{^{\nu\overline{X_i=\varphi_i} \textit{ in } \varphi_i}/_{X_i}\})
\end{aligned}
$$

LEMMA 3.4. *Let φ a cycle-free formula, then we have the following.*

$$[\![\varphi]\!]_V = \bigcup_{\psi \in unf_f(\varphi)} [\![\psi]\!]_V$$

The intuition why this lemma holds is the following. Given a tree satisfying φ, we deduce from the hypothesis that φ is cycle free the fact that every node of the tree will be tested a finite number of times against every subformula of φ. As the tree and the number of subformulas are finite, the satisfaction derivation is finite hence only a finite number of unfolding is necessary to prove that the tree satisfies the formula. As least and greatest fixpoints coincide when only a finite number of unfolding is required, this is sufficient to show that they collapse. Note that this would not hold if infinite trees were allowed: the formula $\mu X.\, \langle 1\rangle X$ is cycle free, but its interpretation is empty, whereas the interpretation of $\nu X.\, \langle 1\rangle X$ includes every tree with an infinite branch of $\langle 1\rangle$ children.

PROOF OF LEMMA 3.4. Let f in $[\![\varphi]\!]_V$, we show that it is in $[\![\psi]\!]_V$ for some finite unfolding $\psi \in unf_f(\varphi)$. As recursive definitions are never negated, the converse is immediate.

As hinted above, the result is a consequence of the fact that a sub-formula is never confronted twice to the same node of f as there is no cycle in the formula. It is thus possible to annotate occurrences of ν and μ with the direction the formula is exploring for each variable, as in Figure 3, and prove the result by induction on the size of f in this direction.

First, we unfold every formula once, to guarantee that the sub-formulas of the shape $\mu\overline{X_i = \varphi_i}$ in ψ are in fact of the shape $\mu\overline{X_i = \varphi_i}$ in φ_j.

Then, we associate each recursion variable in every μ and ν of the initial formula with a unique identifier. (From now on, we do not distinguish between smallest and largest fixed points, as we handle them identically.) For every recursive formula $\mu\overline{X_i = \varphi_i}$ in ψ, we annotate every modality $\langle a\rangle \xi$ in every φ_j where X_i is free in ξ with the variable X_i. Note that modalities may be annotated with more than one variable.

We now detail how recursion identifiers and annotations are updated upon unfolding and encountering modalities.

— Upon unfolding a recursive formula for the first time, the recursion identifiers are recorded and associated with the $\_$ direction. Moreover, they are also associated with an integer, the size of the tree f.
— Upon encountering a modality $\langle a \rangle$ annotated with identifiers, the direction of the identifiers is updated with the modality according to the $\cdot \lhd \langle a \rangle$ operator. As the formula is cycle-free, the resulting direction cannot be $\bot$.
— Upon unfolding a recursive formula $\mu\overline{X_i = \varphi_i}$ in φ_j whose identifiers have been already recorded, the integer associated to X_j is updated to be the longest path, defined below, of the current focused tree in X_j's direction. As the formula is cycle-free, a direction must have been recorded for every identifier.

We now define the *longest path* of a focused tree in a given direction. Given a tree f and a direction $\langle a \rangle$, we define the longest path as the longest *cycle-free* path of f compatible with the direction, i.e. that does not start in the $\langle \overline{a} \rangle$ direction. By definition of the trees, if $\langle a \rangle$ is $\langle 1 \rangle$ or $\langle 2 \rangle$, then the path is only made of $\langle 1 \rangle$ and $\langle 2 \rangle$ steps. If $\langle a \rangle$ is $\langle \overline{1} \rangle$ or $\langle \overline{2} \rangle$, then the path is a sequence of $\langle \overline{1} \rangle$ or $\langle \overline{2} \rangle$ steps followed by a sequence of $\langle 1 \rangle$ and $\langle 2 \rangle$ steps joined by either a $\langle \overline{1} \rangle \langle 2 \rangle$ or a $\langle \overline{2} \rangle \langle 1 \rangle$ sequence.

We may now prove the property that f belongs to the interpretation of a finite unfolding ψ of φ by progressively building it, relying on an induction on the lexical order of:

(1) the number of identifiers in ψ not yet annotated with a direction and an integer;
(2) the sum of the integers of every annotated identifier in ψ;
(3) the size of ψ.

The base cases are for the true formula, the atomic proposition and its negation, the start proposition and its negation, and the negation of the existential formula. The result is immediate for all these results as they do not involve recursive formulas.

For the inductive case, we proceed by case on the syntax of ψ. The interesting cases are recursive formulas (in every other case, the size of the formula decreases while leaving the other induction metrics unchanged as annotations are updated only when unfolding formulas). In the case of a formula involving unannotated identifiers, they become annotated (thus decreasing the number of unannotated identifiers) and associated to the size of the tree, and we conclude by induction. In the case of an annotated formula recursion $\psi = \mu\overline{X_i = \varphi_i}$ in φ_j, this formula may only have been produced by a previous expansion where X_j was replaced by ψ. As the formula is cycle-free, at least one modality has been encountered and it was annotated by X_j, since X_j was free in the formula before the previous expansion. Moreover, every modality encountered since the previous unfolding was also annotated by X_j, and as the formula is cycle-free these modalities are all compatible. Thus the longest path of f in X_j's direction has decreased by at least one, and as the other identifiers may only have decreased, after expansion the sum has decreased, and we conclude by induction. □

In the rest of the paper, we only consider least fixpoints. An important consequence of Lemma 3.4 is that the logic restricted in this way is closed under negation using De Morgan's dualities, extended to eventualities and fixpoints as follows:

$$\neg \langle a \rangle \varphi \stackrel{\text{def}}{=} \neg \langle a \rangle \top \vee \langle a \rangle \neg\varphi$$

$$\neg\mu\overline{X_i = \varphi_i} \text{ in } \psi \stackrel{\text{def}}{=} \mu\overline{X_i = \neg\varphi_i\{\overline{X_i}/_{\neg X_i}\}} \text{ in } \neg\psi\{\overline{X_i}/_{\neg X_i}\}$$

4. SATISFIABILITY-TESTING ALGORITHM

In this section we present our algorithm, show that it is sound and complete, and prove a time complexity boundary. To check a formula φ, our algorithm builds satisfiable

formulas out of some subformulas (and their negation) of φ, then checks whether φ was produced. We first describe how to extract the subformulas from φ.

4.1. Preliminary Definitions

We define the *Fisher-Ladner closure* $\mathrm{cl}(\psi)$ of a formula ψ as the set of all subformulas of ψ where fixpoint formulas are additionally unwound once. Specifically, we define the relation $\rightarrow_e \subseteq \mathcal{L}_\mu \times \mathcal{L}_\mu$ as the least relation that satisfies the following:

— $\varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_2$
— $\varphi_1 \vee \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \vee \varphi_2 \rightarrow_e \varphi_2$
— $\langle a \rangle \varphi' \rightarrow_e \varphi'$
— $\mu\overline{X_i = \varphi_i}$ in $\psi \rightarrow_e \exp(\mu\overline{X_i = \varphi_i}$ in $\psi)$

The closure $\mathrm{cl}(\psi)$ is the smallest set S that contains ψ and is closed under the relation $\rightarrow_e$, i.e. if $\varphi_1 \in S$ and $\varphi_1 \rightarrow_e \varphi_2$ then $\varphi_2 \in S$.

We call $\Sigma(\psi)$ the set of atomic propositions σ used in ψ along with another name, σ_x, that does not occur in ψ to represent atomic propositions not occurring in ψ.

We define $\mathrm{cl}^*(\psi) = \mathrm{cl}(\psi) \cup \{\neg\varphi \mid \varphi \in \mathrm{cl}(\psi)\}$. Every formula $\varphi \in \mathrm{cl}^*(\psi)$ can be seen as a Boolean combination of formulas of a set called the Lean of ψ, inspired from [Pan et al. 2006]. We note this set $\mathrm{Lean}(\psi)$ and define it as follows:

$$\mathrm{Lean}(\psi) \quad = \quad \left\{\langle a \rangle \top \mid a \in \{1, 2, \overline{1}, \overline{2}\}\right\} \ \cup \ \Sigma(\psi) \ \cup \ \{\text{Ⓢ}\} \ \cup \ \{\langle a \rangle \varphi \mid \langle a \rangle \varphi \in \mathrm{cl}(\psi)\}$$

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

— $\forall \langle a \rangle \varphi \in \mathrm{Lean}(\psi), \langle a \rangle \varphi \in t \Rightarrow \langle a \rangle \top \in t$ (modal consistency);
— $\left\langle \overline{1} \right\rangle \top \notin t \vee \left\langle \overline{2} \right\rangle \top \notin t$ (a tree node cannot be both a first child and a second child);
— exactly one atomic proposition $\sigma \in t$; we use the function $\sigma(t)$ to return the atomic proposition of a type t;
— Ⓢ may belong to t.

We call $\mathrm{Types}(\psi)$ the set of ψ-types. For a ψ-type t, the *complement* of t is the set $\mathrm{Lean}(\psi) \setminus t$.

A type determines a truth assignment of every formula in $\mathrm{cl}^*(\psi)$ with the relation $\dot{\in}$ defined in Figure 4. Note that such derivations are finite because the number of naked $\mu\overline{X_i = \varphi_i}$ in ψ (that do not occur under modalities) strictly decreases after each expansion.

We often write $\varphi \mathrel{\dot{\in}} t$ if there are some T, F such that $\varphi \mathrel{\dot{\in}} t \Longrightarrow (T, F)$. We say that a formula φ is true at a type t iff $\varphi \mathrel{\dot{\in}} t$.

We now relate a formula to the truth assignment of its ψ-types.

PROPOSITION 4.1. *If $\varphi \mathrel{\dot{\in}} t \Longrightarrow (T, F)$, then we have $T \subseteq t$, $F \subseteq \mathrm{Lean}(\varphi) \setminus t$, and $\bigwedge_{\psi \in T} \psi \wedge \bigwedge_{\psi \in F} \neg\psi$ implies φ (every tree in the interpretation of the first formula is in the interpretation of the second). If $\varphi \mathrel{\dot{\notin}} t \Longrightarrow (T, F)$, then we have $T \subseteq t$, $F \subseteq \mathrm{Lean}(\varphi) \setminus t$, and $\bigwedge_{\psi \in T} \psi \wedge \bigwedge_{\psi \in F} \neg\psi$ implies $\neg\varphi$.*

Proof: Immediate by induction on the derivations. □

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

$$\frac{}{\top \mathrel{\dot\in} t \Longrightarrow (\emptyset,\emptyset)} \qquad \frac{\varphi \in \mathbf{Lean}(\psi) \qquad \varphi \in t}{\varphi \mathrel{\dot\in} t \Longrightarrow (\{\varphi\},\emptyset)} \qquad \frac{\varphi_1 \mathrel{\dot\in} t \Longrightarrow (T_1,F_1) \qquad \varphi_2 \mathrel{\dot\in} t \Longrightarrow (T_2,F_2)}{\varphi_1 \wedge \varphi_2 \mathrel{\dot\in} t \Longrightarrow (T_1 \cup T_2, F_1 \cup F_2)}$$

$$\frac{\varphi_1 \mathrel{\dot\in} t \Longrightarrow (T_1,F_1)}{\varphi_1 \vee \varphi_2 \mathrel{\dot\in} t \Longrightarrow (T_1,F_1)} \qquad \frac{\varphi_2 \mathrel{\dot\in} t \Longrightarrow (T_2,F_2)}{\varphi_1 \vee \varphi_2 \mathrel{\dot\in} t \Longrightarrow (T_2,F_2)} \qquad \frac{\varphi \mathrel{\dot\notin} t \Longrightarrow (T,F)}{\neg\varphi \mathrel{\dot\in} t \Longrightarrow (T,F)}$$

$$\frac{\mathbf{exp}(\mu\overline{X_i = \varphi_i} \text{ in } \psi) \mathrel{\dot\in} t \Longrightarrow (T,F)}{\mu\overline{X_i = \varphi_i} \text{ in } \psi \mathrel{\dot\in} t \Longrightarrow (T,F)} \qquad \frac{\varphi \in \mathbf{Lean}(\psi) \qquad \varphi \notin t}{\varphi \mathrel{\dot\notin} t \Longrightarrow (\emptyset,\{\varphi\})}$$

$$\frac{\varphi_1 \mathrel{\dot\notin} t \Longrightarrow (T_1,F_1) \qquad \varphi_2 \mathrel{\dot\notin} t \Longrightarrow (T_2,F_2)}{\varphi_1 \vee \varphi_2 \mathrel{\dot\notin} t \Longrightarrow (T_1 \cup T_2, F_1 \cup F_2)} \qquad \frac{\varphi_1 \mathrel{\dot\notin} t \Longrightarrow (T_1,F_1)}{\varphi_1 \wedge \varphi_2 \mathrel{\dot\notin} t \Longrightarrow (T_1,F_1)}$$

$$\frac{\varphi_2 \mathrel{\dot\notin} t \Longrightarrow (T_2,F_2)}{\varphi_1 \wedge \varphi_2 \mathrel{\dot\notin} t \Longrightarrow (T_2,F_2)} \qquad \frac{\varphi \mathrel{\dot\in} t \Longrightarrow (T,F)}{\neg\varphi \mathrel{\dot\notin} t \Longrightarrow (T,F)} \qquad \frac{\mathbf{exp}(\mu\overline{X_i = \varphi_i} \text{ in } \psi) \mathrel{\dot\notin} t \Longrightarrow (T,F)}{\mu\overline{X_i = \varphi_i} \text{ in } \psi \mathrel{\dot\notin} t \Longrightarrow (T,F)}$$

Fig. 4. Truth assignment of a formula

DEFINITION 4.2 (COMPATIBILITY RELATION). *Two types t and t' are* compatible *under $a \in \{1,2\}$, written $\Delta_a(t,t')$, iff*

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

4.2. The Algorithm

The algorithm works on sets of triples of the form (t, w_1, w_2) where t is a type, and w_1 and w_2 are sets of types which represent every witness for t according to relations $\Delta_1(t,\cdot)$ and $\Delta_2(t,\cdot)$.

The algorithm proceeds in a bottom-up approach, repeatedly adding new triples until a satisfying model is found (i.e. a triple whose first component is a type implying the formula), or until no more triple can be added. Each iteration of the algorithm builds types representing deeper trees (in the 1 and 2 direction) with pending backward modalities that will be fulfilled at later iterations. Types with no backward modalities are satisfiable, and if such a type implies the formula being tested, then it is satisfiable. The main iteration is as follows:

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

where $X \subseteq \text{Types}(\psi) \times 2^{\text{Types}(\psi)} \times 2^{\text{Types}(\psi)}$ and the update operation $\mathtt{Upd}(\cdot)$ and success check operation $\mathtt{FinalCheck}(\cdot,\cdot)$ are defined on Figure 5. The update operation requires four almost identical cases to ensure that the optional mark remains *unique*. The first case corresponds to the absence of the mark, the second case to the presence of the

$$\begin{aligned}
\mathtt{Upd}(X) \stackrel{\text{def}}{=} X \cup & \left\{ (t, \mathtt{w}_1(t, X^\circ), \mathtt{w}_2(t, X^\circ)) \mid \begin{array}{l} \textcircled{\scriptsize S} \notin t \subseteq \mathbf{Types}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathtt{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathtt{w}_2(t, X^\circ) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_1(t, X^\circ), \mathtt{w}_2(t, X^\circ))^{\textcircled{\scriptsize S}} \mid \begin{array}{l} \textcircled{\scriptsize S} \in t \subseteq \mathbf{Types}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathtt{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathtt{w}_2(t, X^\circ) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_1(t, X^{\textcircled{\scriptsize S}}), \mathtt{w}_2(t, X^\circ))^{\textcircled{\scriptsize S}} \mid \begin{array}{l} \textcircled{\scriptsize S} \notin t \subseteq \mathbf{Types}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathtt{w}_1(t, X^{\textcircled{\scriptsize S}}) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathtt{w}_2(t, X^\circ) \neq \emptyset \end{array} \right\} \\
\cup & \left\{ (t, \mathtt{w}_1(t, X^\circ), \mathtt{w}_2(t, X^{\textcircled{\scriptsize S}}))^{\textcircled{\scriptsize S}} \mid \begin{array}{l} \textcircled{\scriptsize S} \notin t \subseteq \mathbf{Types}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathtt{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathtt{w}_2(t, X^{\textcircled{\scriptsize S}}) \neq \emptyset \end{array} \right\}
\end{aligned}$$

$$\mathtt{w}_a(t, X) \stackrel{\text{def}}{=} \{\mathtt{type}(x) \mid x \in X \wedge \langle \overline{a} \rangle \top \in \mathtt{type}(x) \wedge \Delta_a(t, \mathtt{type}(x))\}$$

$$\mathtt{type}((t, w_1, w_2)) \stackrel{\text{def}}{=} t$$

$$\mathtt{FinalCheck}(\psi, X) \stackrel{\text{def}}{=} \exists x \in X, \mathtt{dsat}(x, \psi) \wedge \forall a \in \{\overline{1}, \overline{2}\}, \langle a \rangle \top \notin \mathtt{type}(x)$$

$$\mathtt{dsat}((t, w_1, w_2), \psi) \stackrel{\text{def}}{=} \psi \doteq t \vee \exists x', \mathtt{dsat}(x', \psi) \wedge (x' \in w_1 \vee x' \in w_2)$$

$$X^{\textcircled{\scriptsize S}} \stackrel{\text{def}}{=} \left\{x \in X \mid x = (\_, \_, \_)^{\textcircled{\scriptsize S}}\right\}$$

$$X^\circ \stackrel{\text{def}}{=} \{x \in X \mid x = (\_, \_, \_)\}$$

Fig. 5. Operations used by the Algorithm.

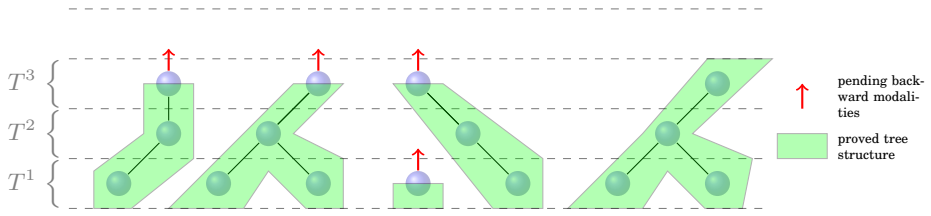
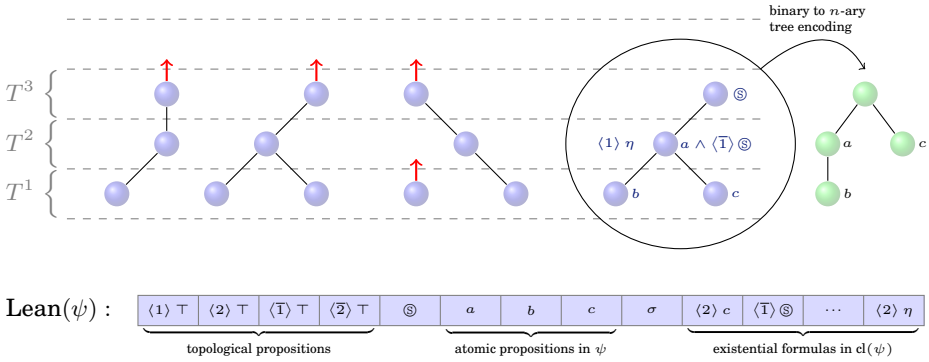


Fig. 6. Algorithm's principle: progressive bottom-up reasoning.

mark at the top level, the third case to the presence of the mark deeper in the first child, and the last case to the presence of the mark deeper in the second child.

At each step of the algorithm, $\mathtt{FinalCheck}(\cdot, \cdot)$ 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 note X^i the set of triples and T^i the set of types after i iterations: $T^i = \left\{\mathtt{type}(x) \mid x \in X^i\right\}$. Note that T^{i+1} is the set of types for which at least one witness belongs to T^i.



$$\psi = \varphi_1 \wedge \neg\varphi_2$$
$$\varphi_1 = a \wedge (\mu Y.\, \langle 2\rangle\, (c \wedge \theta) \vee \langle 2\rangle\, Y) \wedge \langle 1\rangle\, \eta$$
$$\varphi_2 = c \wedge \underbrace{\mu X.\, \langle \overline{1}\rangle\, Ⓢ \vee \langle \overline{2}\rangle\, X}_{\theta} \wedge \langle 1\rangle\, \underbrace{\mu Z.b \vee \langle 2\rangle\, Z}_{\eta}$$

Fig. 7. Run of the algorithm for a sample formula.

4.3. Example Run of the Algorithm

In a sense, the algorithm performs a kind of progressive bottom-up reasoning while ensuring partial (forward) satisfiability of subformulas, as illustrated by Figure 6.

More specifically, Figure 7 illustrates a run of the algorithm for a sample formula ψ. Lean(ψ) is computed, and the fixpoint computation starts: the set of types T^1 contains all possible leaves. Each type added in T^i ($i \geq 2$) requires at least one witness type found in T^{i-1} (else it would have been added at some previous step $j < i$). In this example, a satisfying binary tree of depth 3 is found (as shown on Figure 7), therefore the algorithm stops just after computing T^3. The first XPath query is not contained in the second one: a counter-example tree is provided to the user (see Figure 7).

4.4. 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 4.3 (CORRECTNESS). *The algorithm decides satisfiability of $\mathcal{L}_\mu$ formulas over finite focused trees.*

Termination. For $\psi \in \mathcal{L}_\mu$, since cl(ψ) is a finite set, Lean(ψ) and $2^{\text{Lean}(\psi)}$ are also finite. Furthermore, $\texttt{Upd}(\cdot)$ is monotonic and each X^i is included in the finite set $\text{Types}(\psi) \times 2^{\text{Types}(\psi)} \times 2^{\text{Types}(\psi)}$, therefore the algorithm terminates. To finish the proof, it thus suffices to prove soundness and completeness.

Preliminary Definitions for Soundness. First, we introduce a notion of partial satisfiability for a formula, where backward modalities are only checked up to a given level. A formula φ is partially satisfied iff $[\![\varphi]\!]^0_V \neq \emptyset$ as defined in Figure 8.

$$
\begin{aligned}
[\![\top]\!]^n_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![X]\!]^n_V &\stackrel{\text{def}}{=} V(X)\\
[\![\varphi \vee \psi]\!]^n_V &\stackrel{\text{def}}{=} [\![\varphi]\!]^n_V \cup [\![\psi]\!]^n_V & [\![p]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = p\}\\
[\![\varphi \wedge \psi]\!]^n_V &\stackrel{\text{def}}{=} [\![\varphi]\!]^n_V \cap [\![\psi]\!]^n_V & [\![\neg p]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq p\}\\
[\![\langle \overline{1} \rangle \varphi]\!]^0_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![Ⓢ]\!]^n_V &\stackrel{\text{def}}{=} \left\{f \mid f = (\sigma^{Ⓢ}[tl], c)\right\}\\
[\![\langle \overline{2} \rangle \varphi]\!]^0_V &\stackrel{\text{def}}{=} \mathcal{F} & [\![\neg Ⓢ]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\langle \overline{1} \rangle \varphi]\!]^{n>0}_V &\stackrel{\text{def}}{=} \left\{f\,\langle 1\rangle \mid f \in [\![\varphi]\!]^{n-1}_V \wedge f\,\langle 1\rangle \text{ defined}\right\}\\
[\![\langle \overline{2} \rangle \varphi]\!]^{n>0}_V &\stackrel{\text{def}}{=} \left\{f\,\langle 2\rangle \mid f \in [\![\varphi]\!]^{n-1}_V \wedge f\,\langle 2\rangle \text{ defined}\right\}\\
[\![\langle 1 \rangle \varphi]\!]^{n}_V &\stackrel{\text{def}}{=} \left\{f\,\langle \overline{1}\rangle \mid f \in [\![\varphi]\!]^{n+1}_V \wedge f\,\langle \overline{1}\rangle \text{ defined}\right\}\\
[\![\langle 2 \rangle \varphi]\!]^{n}_V &\stackrel{\text{def}}{=} \left\{f\,\langle \overline{2}\rangle \mid f \in [\![\varphi]\!]^{n+1}_V \wedge f\,\langle \overline{2}\rangle \text{ defined}\right\}\\
[\![\neg\langle a\rangle \top]\!]^n_V &\stackrel{\text{def}}{=} \{f \mid f\,\langle a\rangle \text{ undefined}\}\\
[\![\mu\overline{X_i = \varphi_i} \text{ in } \psi]\!]^n_V &\stackrel{\text{def}}{=} \text{let } T_i = \left(\bigcap \left\{\overline{T_i} \subseteq \overline{\mathcal{F}} \mid [\![\overline{\varphi_i}]\!]^n_{V[\overline{T_i/X_i}]} \subseteq \overline{T_i}\right\}\right)_i\\
&\qquad \text{in } [\![\psi]\!]^n_{V[\overline{T_i/X_i}]}
\end{aligned}
$$

Fig. 8. Partial satisfiability

For a type t, we note $\varphi_c(t)$ its *most constrained formula*, where atoms are taken from $\text{Lean}(\psi)$. In the following, $\circ$ stands for Ⓢ if $Ⓢ \in t$, and for $\neg Ⓢ$ otherwise.

$$\varphi_c(t) = \sigma(t) \wedge \bigwedge_{\sigma \in \Sigma, \sigma \notin t} \neg\sigma \wedge \circ \wedge \bigwedge_{\langle a\rangle\varphi \in t} \langle a\rangle\,\varphi \wedge \bigwedge_{\langle a\rangle\varphi \notin t} \neg\langle a\rangle\,\varphi$$

We now introduce a notion of *paths*, written ρ which are concatenations of modalities: the empty path is written ϵ, and path concatenation is written ρa.

Every path may be given a *depth*:

$$
\begin{aligned}
\mathit{depth}(\epsilon) &\stackrel{\text{def}}{=} 0\\
\mathit{depth}(\rho a) &\stackrel{\text{def}}{=} \mathit{depth}(\rho) + 1 \quad \text{if } a \in \{1, 2\}\\
\mathit{depth}(\rho a) &\stackrel{\text{def}}{=} \mathit{depth}(\rho) - 1 \quad \text{if } a \in \{\overline{1}, \overline{2}\}
\end{aligned}
$$

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

We define a tree of types $\mathcal{T}$ as a tree whose nodes are types, $\mathcal{T}(\cdot) = t$, with at most two children, $\mathcal{T}\,\langle 1\rangle$ and $\mathcal{T}\,\langle 2\rangle$. The navigation in trees of types is trivially extended to forward paths. A tree of types is *consistent* iff for every forward path ρ and for every child a of $\mathcal{T}\,\langle\rho\rangle$, we have $\mathcal{T}\,\langle\rho\rangle\,(\cdot) = t$, $\mathcal{T}\,\langle\rho a\rangle\,(\cdot) = t'$ implies $\langle a\rangle \top \in t$, $\langle \overline{a}\rangle \top \in t'$, and $\Delta_a(t, t')$.

Given a consistent tree of types $\mathcal{T}$, we now define a dependency graph whose nodes are pairs of a forward path ρ and a formula in $t = \mathcal{T}\,\langle\rho\rangle\,(\cdot)$ or the negation of a formula in the complement of t. The directed edges of the graph are labeled with modalities consistent with the tree. This graph corresponds to what the algorithm ultimately builds, as every iteration discovers longer forward paths. For every (ρ, φ) in the nodes we build the following edges:

— $\varphi \in \Sigma(\psi) \cup \neg\Sigma(\psi) \cup \{Ⓢ, \neg Ⓢ, \langle a\rangle \top, \neg\langle a\rangle \top\}$: no edge
— $\rho = \epsilon$ and $\varphi = \langle \overline{a}\rangle\,\varphi'$ with $a \in \{1, 2\}$: no edge

— $\rho = \rho' a$ and $\varphi = \langle a' \rangle \varphi'$: let $t = \mathcal{T} \langle \rho \rangle (\cdot)$.
We first consider the case where $a' \in \{1, 2\}$ and let $t' = \mathcal{T} \langle \rho a' \rangle (\cdot)$. As $\mathcal{T}$ is consistent, we have $\varphi' \dot{\in} t'$ hence there are T, F such that $\varphi' \dot{\in} t' \Longrightarrow (T, F)$ with T a subset of t', and F a subset of the complement of t'. For every $\varphi_T \in T$ we add an edge a' to $(\rho a', \varphi_T)$, and for every $\varphi_F \in F$ we add an edge a' to $(\rho a', \neg\varphi_F)$.
We now consider the case where $a' \in \{\overline{1}, \overline{2}\}$ and first show that we have $a' = \overline{a}$. As $\mathcal{T}$ is consistent, we have $\langle \overline{a} \rangle \top$ in t. Moreover, as t is a tree type, it must contain $\langle a' \rangle \top$. As a' is a backward modality, it must be equal to $\overline{a}$ as at most one may be present. Hence we have $\rho' a a' = \rho'$ and we let $t' = \mathcal{T} \langle \rho' \rangle (\cdot)$. By consistency, we have $\varphi' \dot{\in} t'$, hence $\varphi' \dot{\in} t' \Longrightarrow (T, F)$ and we add edges as in the previous case: to (ρ', φ_T) and to $(\rho', \neg\varphi_F)$.
— $\rho = \rho' a$ and $\varphi = \neg \langle a' \rangle \varphi'$: let $t = \mathcal{T} \langle \rho \rangle (\cdot)$. If $\langle a' \rangle \top$ is not in t then no edge is added. Otherwise, we proceed as in the previous case. For downward modalities, we let $t' = \mathcal{T} \langle \rho a' \rangle (\cdot)$ and we compute $\varphi' \dot{\notin} t' \Longrightarrow (T, F)$, which we know to hold by consistency. We then add edges to $(\rho a', \varphi_T)$ and to $(\rho a', \neg\varphi_F)$ as before. For upward modalities, as we have $\langle a' \rangle \top$ in t, we must have $a' = \overline{a}$ and we let $t' = \mathcal{T} \langle \rho' \rangle (\cdot)$. We compute $\varphi' \dot{\notin} t' \Longrightarrow (T, F)$ and we add the edges to (ρ', φ_T) and to $(\rho', \neg\varphi_F)$ as before.

LEMMA 4.4. *The dependency graph of a consistent tree of types of a cycle-free formula is cycle free.*

Proof: The proof proceeds by induction on the depth of the cycle, relying on the fact that the dependency graph is consistent with the tree structure (i.e., if a 1 edge reaches a node, no $\overline{2}$ edge may leave this node). The induction case is trivial: if there is a cycle of depth n, there must be a cycle of depth $n - 1$, a contradiction.

The base case is for a cycle of depth 1. We describe one case, where the cycle is $(\rho, \langle 1 \rangle \varphi) \longrightarrow^1 (\rho 1, \langle \overline{1} \rangle \psi) \longrightarrow^{\overline{1}} (\rho, \langle 1 \rangle \varphi)$. As φ must be a subformula of ψ and ψ a subformula of φ, they are both recursive formula. An analysis of the shape of φ, based on the derivations $\varphi \dot{\in} t \Longrightarrow (T, F)$ and $\psi \dot{\in} t' \Longrightarrow (T', F')$ with $\langle 1 \rangle \psi \in T$ and $\langle \overline{1} \rangle \varphi \in T'$ then shows that φ is not a cycle-free formula, a contradiction. □

LEMMA 4.5 (SOUNDNESS). *Let T be the result set of the algorithm. For any type $t \in T$ and any φ such that $\varphi \dot{\in} t$, then $[\![\varphi]\!]^0_\emptyset \neq \emptyset$.*

Proof:
The proof proceeds by induction on the number of steps of the algorithm. For every t in T^n and every witness tree $\mathcal{T}$ rooted at t built from X^n, we show that $\mathcal{T}$ is a consistent tree type and we build a focused tree f that is rooted (i.e. of the shape $(\sigma^\circ[tl], (\epsilon, Top, tl'))$). The tree f is in the partial interpretation of $\varphi_c(t)$: $f \langle \rho \rangle \in [\![\varphi_c(\mathcal{T} \langle \rho \rangle (\cdot))]\!]^{depth(\rho)}_\emptyset$ for any path ρ whose depth is 0 or more, and f contains the start mark only if Ⓢ occurs in $\mathcal{T}$. We then show that for all $\varphi \dot{\in} t$, we have $f \in [\![\varphi]\!]^0_\emptyset$.

The base case is trivial by the shape of t: it may only contain backward modalities (trivially satisfied at level 0), one atomic proposition, and one start proposition. Moreover there is only one tree of witnesses to consider, the tree whose only node is t. If the atomic proposition is σ, then the focused tree returned is either $(\sigma^{Ⓢ}[\epsilon], (\epsilon, Top, \epsilon))$ or $(\sigma[\epsilon], (\epsilon, Top, \epsilon))$ depending on the start proposition.

In the inductive case, we consider every witness types for both downward modalities, t_1 and t_2. For each of them, we consider every tree type $\mathcal{T}_1$ and $\mathcal{T}_2$ and build a tree type rooted at t which is consistent by definition of the algorithm. By induction, we have f_1 and f_2 such that $f_1 \langle \rho \rangle \in [\![\varphi_c(\mathcal{T} \langle 1\rho \rangle (\cdot))]\!]^{depth(\rho)}_\emptyset$ and $f_2 \langle \rho \rangle \in [\![\varphi_c(\mathcal{T} \langle 2\rho \rangle (\cdot))]\!]^{depth(\rho)}_\emptyset$ for any path ρ whose depth is 0 or more. If either $\mathcal{T}_1$ or $\mathcal{T}_2$ contains Ⓢ, then f_1 or f_2 contains

$$\frac{}{f \Vdash_\rho \top} \qquad \frac{\mathtt{nm}(f) = \sigma}{f \Vdash_\rho \sigma} \qquad \frac{\mathtt{nm}(f) \neq \sigma}{f \Vdash_\rho \neg\sigma} \qquad \frac{}{(\sigma^{\textcircled{S}}[tl], c) \Vdash_\rho \textcircled{S}} \qquad \frac{}{(\sigma[tl], c) \Vdash_\rho \neg\textcircled{S}}$$

$$\frac{f \Vdash_\rho \varphi}{f \Vdash_\rho \varphi \vee \psi} \qquad \frac{f \Vdash_\rho \psi}{f \Vdash_\rho \varphi \vee \psi} \qquad \frac{f \Vdash_\rho \varphi \qquad f \Vdash_\rho \psi}{f \Vdash_\rho \varphi \wedge \psi} \qquad \frac{f\langle 1\rangle \Vdash_{\rho 1} \varphi}{f \Vdash_\rho \langle 1\rangle \varphi} \qquad \frac{f\langle 2\rangle \Vdash_{\rho 2} \varphi}{f \Vdash_\rho \langle 2\rangle \varphi}$$

$$\frac{f\langle \overline{1}\rangle \Vdash_{\rho\overline{1}} \varphi}{f \Vdash_\rho \langle \overline{1}\rangle \varphi} \qquad \frac{f\langle \overline{2}\rangle \Vdash_{\rho\overline{2}} \varphi}{f \Vdash_\rho \langle \overline{2}\rangle \varphi} \qquad \frac{f\langle a\rangle \text{ undefined}}{f \Vdash_\rho \neg\langle a\rangle \top} \qquad \frac{f \Vdash_\rho \exp(\mu\overline{X_i = \varphi_i} \text{ in } \psi)}{f \Vdash_\rho \mu\overline{X_i = \varphi_i} \text{ in } \psi}$$

Fig. 9. Satisfiability relation

the start mark by induction. Moreover, by definition of the algorithm, it is the case for only one of them and $\textcircled{S}$ is not in t.

Let f_1 be $(\sigma_1^\circ[tl_1], (\epsilon, Top, tr_1))$ and f_2 be $(\sigma_2^\circ[tl_2], (\epsilon, Top, tr_2))$. Let f be the tree $(\sigma(t)^\circ[\sigma_1^\circ[tl_1] :: tr_1], (\epsilon, Top, \sigma_2^\circ[tl_2] :: tr_2))$ where $\sigma(t)^\circ$ is $\sigma(t)^{\textcircled{S}}$ if $\textcircled{S} \in t$, and $\sigma(t)$ otherwise. Note that f contains exactly one start mark iff $\textcircled{S} \in \mathcal{T}$.

We next show that if $f_1\langle\rho\rangle$ is in $[\![\varphi_c(\mathcal{T}\langle 1\rho\rangle(\cdot))]\!]_\emptyset^{depth(\rho)}$, then the tree $f\langle 1\rho\rangle$ is in $[\![\varphi_c(\mathcal{T}\langle 1\rho\rangle(\cdot))]\!]_\emptyset^{depth(\rho)}$, and the same for the other modality. This is shown by induction on the depth of the path, remarking that every backward modality at level 0 is trivially satisfied.

We then proceed to show that f satisfies $\varphi_c(t)$ at level 0. To do so, we need a further induction on the dependency tree. Let ρ be a path of the dependency tree and ψ be a formula at that path in the dependency tree, we show that $f\langle\rho\rangle \in [\![\psi]\!]_V^{depth(\rho)}$. To do so, we rely on $f\langle\rho\rangle \in [\![\psi]\!]_V^{depth(\rho)-1}$ if $depth(\rho) \neq 0$. In the base case at depth 0, the result is by construction as the formula is either a backward modality or an atomic formula. In the base case at another depth, the case is immediate by induction as the formula has to be an atomic formula whose interpretation does not depend on the depth. In the induction case, we conclude by the inductive hypothesis and by definition of partial satisfiability.

We conclude the proof by noticing that the final selected type has no backward modality, hence $[\![\varphi_c(t)]\!]_0^\emptyset = [\![\varphi_c(t)]\!]_\emptyset$. □

LEMMA 4.6 (COMPLETENESS). *For a cycle-free closed formula $\varphi \in \mathcal{L}_\mu$, if $[\![\varphi]\!]_\emptyset \neq \emptyset$ then the algorithm terminates with a set of triples X such that* `FinalCheck`(φ, X).

Proof: Let $f \in [\![\varphi]\!]_\emptyset$ be a smallest focused tree validating the formula such that the names occurring in f are either also occurring in φ or are a single other name σ_x. By Lemma 3.4, there is a finite unfolding of φ such that f belongs to its interpretation. Hence there is a finite satisfiability derivation, defined in Figure 9, of $f \Vdash_\epsilon \varphi$.

In the satisfiability derivation, we assume the paths are normalized ($1\overline{1} = \epsilon$). Hence every path is a concatenation of a (possibly empty) backward path ρ_b followed by a forward path ρ_f.

This derivation has the following property, immediate by induction: let f the initial focused tree, then $f' \Vdash_\rho \varphi$ implies $f' = f\langle\rho\rangle$. Hence if $f_1 \Vdash_\rho \varphi_1$ and $f_2 \Vdash_\rho \varphi_2$, then $f_1 = f_2$.

We next use the satisfiability derivation to construct a run of the algorithm that concludes that φ is satisfiable. We first associate each path to a type, which we then saturate (adding formulas that are true even though the satisfiability relation does not mention them at that path). We next show that every formula at a path in the

satisfiability relation is implied by the type at that path, and that types are consistent according to the $\Delta_a(t, t')$ relation. We then conclude that the types are created by a run of the algorithm by induction on the paths.

More precisely, we first describe how we build t_ρ. Let Φ_ρ the set of formulas at path ρ. We first add every formula of Φ_ρ that is in Lean(φ), then we complete this set to yield a correct type: if $\langle a \rangle \psi \in \Phi_\rho$ then we add $\langle a \rangle \top$; for every modality a for which $f \langle a \rangle$ is defined we add $\langle a \rangle \top$; if there is no atomic proposition in Φ_ρ then we add $\mathtt{nm}(f \langle \rho \rangle)$; finally if $f \langle \rho \rangle$ has the start mark we add $\textcircled{\scriptsize S}$.

We next saturate the types. For every path t_ρ if $t_{\rho a}$ exists, if $\langle a \rangle \psi \in$ Lean(φ), and if $\psi \mathrel{\dot\in} t_{\rho a}$ then we add $\langle a \rangle \psi$ to t_ρ. This procedure is repeated until it does not change any type. Termination is a consequence of the finite size of the lean and of the number of paths. The resulting types are satisfiable as they are before saturation (since a focused tree satisfies them) and each formula added during saturation is first checked to be implied by the type.

We next show (*): for any given path ρ, if $\varphi_\rho \in \Phi_\rho$ then $\varphi_\rho \mathrel{\dot\in} t_\rho$, by induction on the satisfiability derivation. Base cases with no negation are immediate by definition of t_ρ as these are formulas of the lean. For base cases with negation, we rely on the fact that $f \langle \rho \rangle$ satisfies the formula, hence we cannot for instance have σ and $\neg\sigma$ in Φ_ρ. If $\neg \langle a \rangle \top \in \Phi_\rho$ then we cannot also have $\langle a \rangle \psi \in \Phi_\rho$ as ρa is not a valid path, hence $\langle a \rangle \top$ is not in t_ρ thus $\neg \langle a \rangle \top \mathrel{\dot\in} t_\rho$. The inductive cases of this induction (disjunction, conjunction, recursion) are immediate as they correspond to the definition of $\cdot \mathrel{\dot\in} \cdot$.

We next show that for every type t_ρ and $t_{\rho a}$ where a is a forward modality, we have $\langle \overline{a} \rangle \top \in t_{\rho a}$ and $\Delta_a(t_\rho, t_{\rho a})$. (Note that, by path normalization, the types considered may be $t_{\overline{12}}$ and $t_{\overline{1}}$ for modality 2.) The first condition is immediate by construction of $t_{\rho a}$ as $f \langle \rho a \rangle$ is defined. For the second condition, let $\langle a \rangle \psi \in t_\rho$. If $\langle a \rangle \psi \in \Phi_\rho$, then it occurs in the satisfiability derivation with an hypothesis $f_{\rho a} \Vdash_{\rho a} \psi$. In this case we have $\psi \mathrel{\dot\in} t_{\rho a}$ by (*). If $\langle a \rangle \psi \notin \Phi_\rho$ then it was added during saturation and the result is immediate by construction. Conversely, if $\psi \mathrel{\dot\in} t_{\rho a}$ then by saturation we have $\langle a \rangle \psi \in t_\rho$. We now consider the case $\langle \overline{a} \rangle \psi \in t_{\rho a}$. The proof goes exactly as before, distinguishing the case where the formula is in $\Phi_{\rho a}$ and the case where it was added by saturation.

We now show that there is a run of the algorithm that produces these types. We proceed by induction on the paths in the downward direction: if $t_{\rho a}$ has been proven for a partial run for $a \in \{1, 2\}$, then t_ρ is proven for the next step of the algorithm. Moreover, we show that $(t_\rho, \{t_{\rho 1}\}, \{t_{\rho 2}\})$ is marked iff a forward subtree of $f \langle \rho \rangle$ contains the start mark. The base case is for paths with no descendants, hence no witness is required. The algorithm then adds $(t_\rho, \emptyset, \emptyset)$ to its set of types, with a mark iff $\textcircled{\scriptsize S} \in t_\rho$, iff $f \langle \rho \rangle$ is marked.

We now consider the inductive case. By induction, a partial run of the algorithm returns $t_{\rho 1}$ and/or $t_{\rho 2}$. We first show that t_ρ is returned in the next step of the algorithm, taking these two types as witnesses. We first remark that if either witness is marked then the other is not and the mark is not at $f \langle \rho \rangle$, since there is only one start mark in f, and if the mark is at $f \langle \rho \rangle$, then neither witness is marked. For each child $a \in \{1, 2\}$ we have $\Delta_a(t_\rho, t_{\rho a})$ and $\langle \overline{a} \rangle \top \in t_{\rho a}$, hence the triple (t_ρ, W_1, W_2) with $t_{\rho 1} \in W_1$ and $t_{\rho 2} \in W_2$ is added by the algorithm.

We may now conclude. At the end of the induction, the last path considered, ρ_0, has no predecessor, hence it is the longest backward only path. Since $f \langle \rho_0 \rangle$ is the root of the tree, we have $\langle \overline{1} \rangle \top \notin t_{\rho_0}$ and $\langle \overline{2} \rangle \top \notin t_{\rho_0}$. Moreover, as the start mark is somewhere in f, it is in a forward subtree of $f \langle \rho_0 \rangle$, hence the final type is marked. Finally, t_ϵ is in the witness tree of the final type, and since $f \Vdash_\epsilon \varphi$, we have $\varphi \mathrel{\dot\in} t_\epsilon$. □

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.[1]

THEOREM 4.7 (COMPLEXITY). *For $\psi \in \mathcal{L}_\mu$ the satisfiability problem $[\![\psi]\!]_\emptyset \neq \emptyset$ is decidable in time $2^{\mathcal{O}(n)}$ where $n = |\mathit{Lean}(\psi)|$.*

Proof: $|\mathrm{Types}(\psi)|$ is bounded by $\left|2^{\mathrm{Lean}(\psi)}\right|$ 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 test at most $2^{\mathcal{O}(n)}$ ψ-types and each test takes at most $2^{\mathcal{O}(n)}$ (testing the formulas containing Ⓢ against ψ). 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.].

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 [Edmund M. Clarke et al. 1999].

First, we observe that the implementation can avoid keeping track of every possible witnesses of each ψ-type. In fact, for a formula φ, we can test $[\![\varphi]\!]_\emptyset \neq \emptyset$ by testing the satisfiability of the (linear-size) "plunging" formula $\psi = \mu X.\varphi \vee \langle 1 \rangle X \vee \langle 2 \rangle X$ at the root of focused trees. That is, checking $[\![\psi]\!]^0_\emptyset \neq \emptyset$ while ensuring there is no unfulfilled upward eventuality at top level 0. One advantage of proceeding this way is that the implementation only need to deal with a current set of ψ-types at each step.

We now introduce a bit-vector representation of ψ-types. Types are complete in the sense that either a subformula or its negation must belong to a type. It is thus possible for a formula $\varphi \in \mathrm{Lean}(\psi)$ to be represented using a single BDD variable. 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 $a \in \{1, 2\}$:

[1] The acute reader may notice that for large formulas, $|\mathrm{Lean}(\psi)|$ is usually smaller than the size of ψ since disjunctions, conjunctions and negations are not members of $\mathrm{Lean}(\psi)$.

— $\text{isparent}_a(\vec{t})$ is read "$\vec{t}$ is a parent for program a" and is true iff the bit for $\langle a \rangle \top$ is true in $\vec{t}$
— $\text{ischild}_a(\vec{t})$ 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 $T \subseteq 2^{\text{Lean}(\psi)}$, we note χ_T its corresponding characteristic function.

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

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

We note $a \rightarrow b$ the implication and $a \leftrightarrow b$ the equivalence of two Boolean formulas a and b over vector bits. We can now construct the BDD of the relation Δ_a for $a \in \{1, 2\}$.

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

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

where $h(\vec{y}) = \chi_T(\vec{y}) \wedge \text{ischild}_a(\vec{y})$.

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

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

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

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

.

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:

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

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* [Edmund M. 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:

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

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:

$$h_1(\vec{x}, \vec{y}) = \mathop{\exists}_{y_k \in E_0} \left[\ h(\vec{y}) \wedge R_{\rho(0)}(\vec{x}, \vec{y})\ \right]$$

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

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

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 [Edmund M. 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 [Edmund M. 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.

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/`

A screenshot of the interface is given in Figure 10. The user can either enter a formula through area (1) of Figure 10 or select from pre-loaded analysis tasks offered in area (4) of Figure 10. The level of details displayed by the solver can be adjusted in area (2) of Figure 10 and makes it possible to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 10 together with counter-examples.

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. 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 very naive; the actual translation used in the online implementation (using the `reg_exp` keyword) results in more concise formulas.

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 $R_{\overline{\epsilon}}$, and translate R^* as $R^*_{\overline{\epsilon}}$. 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 $R_{\overline{\epsilon}}$.

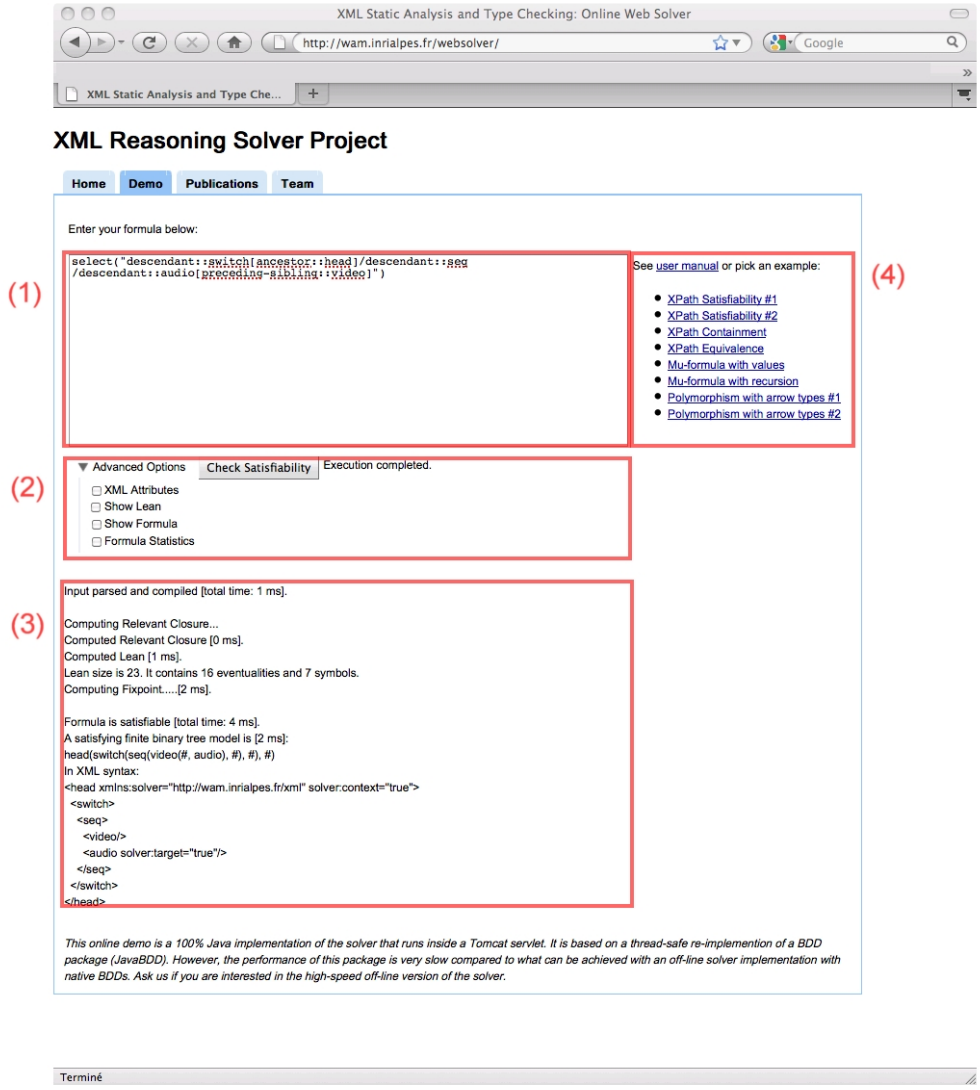


Fig. 10. Screenshot of the Solver Interface.

$$
\begin{aligned}
\epsilon_{\epsilon} &= \epsilon & \epsilon_{\overline{\epsilon}} &= \emptyset \\
a_{\epsilon} &= \emptyset & a_{\overline{\epsilon}} &= a \\
(R.R')_{\epsilon} &= R_{\epsilon}.R'_{\epsilon} & (R.R')_{\overline{\epsilon}} &= R_{\epsilon}.R'_{\overline{\epsilon}} \vee R_{\overline{\epsilon}}.R'_{\epsilon} \vee R_{\overline{\epsilon}}.R'_{\overline{\epsilon}} \\
(R^*)_{\epsilon} &= \epsilon & (R^*)_{\overline{\epsilon}} &= (R_{\overline{\epsilon}})^+ \\
(R \vee R')_{\epsilon} &= R_{\epsilon} \vee R'_{\epsilon} & (R \vee R')_{\overline{\epsilon}} &= R_{\overline{\epsilon}} \vee R'_{\overline{\epsilon}}
\end{aligned}
$$

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

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

Given a regular expression R, we translate it in our logic as the formula $[\![R]\!]_e$.

The expression `F` $\iff$ `G` is syntactic sugar for $(\mathtt{F}\&\mathtt{G})|(\neg\mathtt{F}\&\neg\mathtt{G})$. It is satisfiable if a tree can be found that has a node where either both `F` and `G` are true, or where neither is. The negation of this formula is satisfiable if there is a tree with a node where either $\mathtt{F}\&\neg\mathtt{G}$ or $\neg\mathtt{F}\&\mathtt{G}$ is true. It is *unsatisfiable* if for every tree and for every node, either $\mathtt{F}\&\mathtt{G}$ is true or $\neg\mathtt{F}\&\neg\mathtt{G}$ is true: the formulas are equivalent (they will always select the same set of nodes).

To check the equivalence of two regular expressions R_1 and R_2, we thus ask the solver 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 word in one language and not the other.

We illustrate this translation with several examples. 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 show the more complex language equivalence $(a|b)^* = a^*(ba^*)^*$. To this end, we need to circumvent a shortcoming of the solver. In the following expressions, some fixpoint variables are not syntactically guarded, yet there is no fixpoint expansions that would continue indefinitely without encountering a guard. More precisely, when unfolding a `$Z`, there will always be a `<1>` modality before reaching `$Z` again. As the more precise analysis of guardedness and cycle-freeness (see Figure 3) is not yet implemented, we annotate these recursive calls with `<0>` to let the solver know we guarantee these calls are guarded.

```
~
((let $X = e | (a & <1>$X) | (b & <1>$X) in $X)
<=>
(let $X = <0>$Y | (a & <1>$X),
     $Y = e | (b & <1>$Z),
     $Z = <0>$Y | (a & <1>$Z)
in $X))
```

Finally, we show that if the languages are not equivalent, the solver provides a counter-example: a word that is in one language but not the other. For instance, let us compare $(a|b)^*$ and $a^*(ba^*)^*b$. We thus try the following formula.

```
~
((let $X = e | (a & <1>$X) | (b & <1>$X) in $X)
<=>
(let $X = <0>$Y | (a & <1>$X),
```

```
     $Y = (b & <1>e) | (b & <1>$Z),
     $Z = <0>$Y | (a & <1>$Z)
in $X))
```

To which the solver replies "Formula is satisfiable [total time: 4 ms]. A satisfying finite binary tree model is [2 ms]: e." Indeed, the empty word is accepted by the first formula but not by the second.

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

To translate KAT formulas, we add boolean propositions, written $\_p$, to the logic. We also extend our model, annotating every node with the set of propositions which are true at that node. These extensions carry over to the algorithm straightforwardly, as we only need to add to the lean the set of boolean propositions mentioned in the formula. This extension has in fact been implemented in our solver and can be tried in the online version.

We extend our translation with a case for boolean propositions $\_p$

$$[\![\_p]\!]_c = \_p \wedge c$$
$$[\![\neg\_p]\!]_c = \neg\_p \wedge c$$

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.

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

We can thus use the solver to test for language equality or inequality. For instance, one may show that $\_bq^*$ is different from $(\_bq)^*$ 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.

6.2. 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 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: $E^{\rightarrow}[\![e_1]\!]_{[\![T_1]\!]} \wedge \bigwedge_{2\leq i\leq n} \neg E^{\rightarrow}[\![e_i]\!]_{[\![T_i]\!]}$

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 $\cap$ 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. 11. XPath Expressions Used in Experiments.

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

Fig. 12. Types Used in Experiments.

— 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[2] [Genevès et al.] , 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 Figure 11 (where "//" is used as a shorthand for "/desc-or-self::*/") and XML types shown on Table 12. Table 13 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.

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

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

Fig. 13. Some Decision Problems and Corresponding Results.

7. CONCLUSION

We found very interesting that the practical restriction to trees and cycle-free formulas led to theoretical implications such as the collapsing of the least and greatest fixpoints. 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 another 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.

EDMUND M. CLARKE, J., 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.

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. A satisfiability solver for XML and XPath. http://wam.inrialpes.fr/websolver.

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.

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.

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.

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

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.

Figure 14 gives the syntax of XPath expressions. Figure 15 and Figure 16 give an interpretation of XPath expressions as functions between sets of focused trees.

$\mathcal{L}_{\text{XPath}} \ni e$	$::=$		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. 14. XPath Abstract Syntax.

A.1. 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.

$$
\begin{aligned}
\mathcal{S}_e[\![\cdot]\!]_\cdot &: \mathcal{L}_{\text{XPath}} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_e[\![/p]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\texttt{root}(F)} \\
\mathcal{S}_e[\![p]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\{(\sigma^{\circledS}[tl],c)\in F\}} \\
\mathcal{S}_e[\![e_1 \mid e_2]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[\![e_1]\!]_F \cup \mathcal{S}_e[\![e_2]\!]_F \\
\mathcal{S}_e[\![e_1 \cap e_2]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[\![e_1]\!]_F \cap \mathcal{S}_e[\![e_2]\!]_F
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_p[\![\cdot]\!]_\cdot &: \mathit{Path} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_p[\![p_1/p_2]\!]_F &\stackrel{\text{def}}{=} \left\{ f' \mid f' \in \mathcal{S}_p[\![p_2]\!]_{(\mathcal{S}_p[\![p_1]\!]_F)} \right\} \\
\mathcal{S}_p[\![p[q]]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_p[\![p]\!]_F \wedge \mathcal{S}_q[\![q]\!]_f \} \\
\mathcal{S}_p[\![a::\sigma]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_a[\![a]\!]_F \wedge \texttt{nm}(f) = \sigma \} \\
\mathcal{S}_p[\![a::*]\!]_F &\stackrel{\text{def}}{=} \{ f \mid f \in \mathcal{S}_a[\![a]\!]_F \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_q[\![\cdot]\!]_\cdot &: \mathit{Qualif} \to \mathcal{F} \to \{\text{true}, \text{false}\} \\
\mathcal{S}_q[\![q_1 \text{ and } q_2]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[\![q_1]\!]_f \wedge \mathcal{S}_q[\![q_2]\!]_f \\
\mathcal{S}_q[\![q_1 \text{ or } q_2]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[\![q_1]\!]_f \vee \mathcal{S}_q[\![q_2]\!]_f \\
\mathcal{S}_q[\![\text{not } q]\!]_f &\stackrel{\text{def}}{=} \neg\, \mathcal{S}_q[\![q]\!]_f \\
\mathcal{S}_q[\![p]\!]_f &\stackrel{\text{def}}{=} \mathcal{S}_p[\![p]\!]_{\{f\}} \neq \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_a[\![\cdot]\!]_\cdot &: \mathit{Axis} \to 2^{\mathcal{F}} \to 2^{\mathcal{F}} \\
\mathcal{S}_a[\![\text{self}]\!]_F &\stackrel{\text{def}}{=} F \\
\mathcal{S}_a[\![\text{child}]\!]_F &\stackrel{\text{def}}{=} \texttt{fchild}(F) \cup \mathcal{S}_a[\![\text{foll-sibling}]\!]_{\texttt{fchild}(F)} \\
\mathcal{S}_a[\![\text{foll-sibling}]\!]_F &\stackrel{\text{def}}{=} \texttt{nsibling}(F) \cup \mathcal{S}_a[\![\text{foll-sibling}]\!]_{\texttt{nsibling}(F)} \\
\mathcal{S}_a[\![\text{prec-sibling}]\!]_F &\stackrel{\text{def}}{=} \texttt{psibling}(F) \cup \mathcal{S}_a[\![\text{prec-sibling}]\!]_{\texttt{psibling}(F)} \\
\mathcal{S}_a[\![\text{parent}]\!]_F &\stackrel{\text{def}}{=} \texttt{parent}(F) \\
\mathcal{S}_a[\![\text{descendant}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{child}]\!]_F \cup \mathcal{S}_a[\![\text{descendant}]\!]_{(\mathcal{S}_a[\![\text{child}]\!]_F)} \\
\mathcal{S}_a[\![\text{desc-or-self}]\!]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\![\text{descendant}]\!]_F \\
\mathcal{S}_a[\![\text{ancestor}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{parent}]\!]_F \cup \mathcal{S}_a[\![\text{ancestor}]\!]_{(\mathcal{S}_a[\![\text{parent}]\!]_F)} \\
\mathcal{S}_a[\![\text{anc-or-self}]\!]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\![\text{ancestor}]\!]_F \\
\mathcal{S}_a[\![\text{following}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_a[\![\text{foll-sibling}]\!]_{(\mathcal{S}_a[\![\text{anc-or-self}]\!]_F)}\right)} \\
\mathcal{S}_a[\![\text{preceding}]\!]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{desc-or-self}]\!]_{\left(\mathcal{S}_a[\![\text{prec-sibling}]\!]_{(\mathcal{S}_a[\![\text{anc-or-self}]\!]_F)}\right)}
\end{aligned}
$$

Fig. 15. Interpretation of XPath Expressions as Functions Between Sets of Focused Trees.

$$
\begin{aligned}
\texttt{fchild}(F) &\stackrel{\text{def}}{=} \{f\,\langle 1\rangle \mid f \in F \wedge f\,\langle 1\rangle \text{ defined}\}\\
\texttt{nsibling}(F) &\stackrel{\text{def}}{=} \{f\,\langle 2\rangle \mid f \in F \wedge f\,\langle 2\rangle \text{ defined}\}\\
\texttt{psibling}(F) &\stackrel{\text{def}}{=} \{f\,\langle \overline{2}\rangle \mid f \in F \wedge f\,\langle \overline{2}\rangle \text{ defined}\}\\
\texttt{parent}(F) &\stackrel{\text{def}}{=} \{(\sigma^{\circ}[\texttt{rev\_a}(tl_l, t :: tl_r)], c)\\
&\qquad\qquad \mid (t, (tl_l, c[\sigma^{\circ}], tl_r)) \in F\}\\
\texttt{rev\_a}(\epsilon, tl_r) &\stackrel{\text{def}}{=} tl_r\\
\texttt{rev\_a}(t :: tl_l, tl_r) &\stackrel{\text{def}}{=} \texttt{rev\_a}(tl_l, t :: tl_r)\\
\texttt{root}(F) &\stackrel{\text{def}}{=} \{(\sigma^{\textcircled{S}}[tl], (tl, \mathit{Top}, tl)) \in F\}\\
&\qquad\qquad \cup\, \texttt{root}(\texttt{parent}(F))
\end{aligned}
$$

Fig. 16. Auxiliary Functions for XPath Interpretation.

Logical Interpretation of Axes. The translation of navigational primitives (namely XPath axes) is formally specified in Figure 17. 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 $\mu Z.\, \langle \overline{1}\rangle\, \chi \vee \langle \overline{2}\rangle\, Z$, 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 $\overline{1}$. From the remaining children, the context is reached by going upward (any number of times) via $\overline{2}$ and finally once via $\overline{1}$.

Logical Interpretation of Expressions. Figure 18 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 expression navigates to the root which is taken as the initial context.

Figure 19 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. 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[3]. 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.

[3]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].

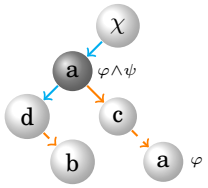
$$
\begin{aligned}
A^{\rightarrow}[\![\cdot]\!]_{\cdot} &: \mathit{Axis} \to \mathcal{L}_\mu \to \mathcal{L}_\mu \\
A^{\rightarrow}[\![\textsf{self}]\!]_\chi &\stackrel{\text{def}}{=} \chi \\
A^{\rightarrow}[\![\textsf{child}]\!]_\chi &\stackrel{\text{def}}{=} \mu Z.\, \langle \overline{1} \rangle\,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

Fig. 17. Translation of XPath Axes.

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

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

Fig. 18. Translation of Expressions and Paths.



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

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

Fig. 19. XPath Translation Example.

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 Figure 20, 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 Figure 20. 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.

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.2 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) $[\![\psi]\!]_\emptyset = \mathcal{S}_e[\![e]\!]_{[\![\varphi]\!]_\emptyset}$
(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

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

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

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

Fig. 20. Translation of Qualifiers.

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

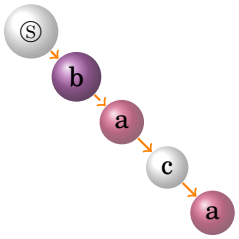


Fig. 21. Example of Back and Forth – Yet Cycle-Free – XPath Navigation.

cycle-free, and their composition yields a proper nesting of fixpoint formulas which is also cycle-free. Figure 21 illustrates this on an typical example. Finally, formal translations do not duplicate any subformula of arbitrary length. □

A.2. 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][4]. 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

[4]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.

```
<!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. 22. A Fragment of the DTD of the Wikipedia Encyclopedia.

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

$\mathcal{L}_{BT} \ni T ::=$		tree type expression
	$\emptyset$	empty set
$\mid$	ϵ	leaf
$\mid$	$T_1 \vee T_2$	union
$\mid$	$\sigma(X_1, X_2)$	label
$\mid$	let $\overline{X_i.T_i}$ in T	binder

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

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

where we use the formula $\sigma \wedge \neg\sigma$ as "false", and the function $\mathit{succ}_\cdot(\cdot)$ takes care of setting the type frontier:

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

according to the predicate $\mathit{nullable}(X)$ which indicates whether the type $T \neq \epsilon$ bound to X contains the empty tree. For example, Figure 24 gives the translation of a DTD fragment of the Wikipedia encyclopedia [Voss 2007] shown on Figure 22. The intermediate binary tree type encoding of the DTD is shown on Figure 23.

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

```
$9 ->EPSILON
        | text($Epsilon, $Epsilon)
        | redirect($Epsilon, $Epsilon)
        | interwiki($Epsilon, $9)
$6 ->EPSILON
        | text($Epsilon, $Epsilon)
        | redirect($Epsilon, $Epsilon)
        | interwiki($Epsilon, $9)
        | status($Epsilon, $9)
$5 ->edit($6, $Epsilon)
        | edit($6, $5)
$14 ->EPSILON
        | history($5, $Epsilon)
        | interwiki($Epsilon, $14)
$4 ->EPSILON
        | history($5, $Epsilon)
        | interwiki($Epsilon, $14)
        | status($Epsilon, $14)
$2 ->title($Epsilon, $4)
$17 ->text($Epsilon, $Epsilon)
        | redirect($Epsilon, $Epsilon)
$1 ->meta($2, $17)
$article ->article($1, $Epsilon)
Start Symbol is $article
9 type variables.
9 terminals.
```

Fig. 23. The Binary Encoding of the DTD of Figure 22.

```
(let_mu
  X2=((((text & ~(<1>T)) & ~(<2>T)) | ((redirect & ~(<1>T)) & ~(<2>T)))
     | ((interwiki & ~(<1>T)) & (~(<2>T) | <2>X2))),
  X3=(((((text & ~(<1>T)) & ~(<2>T)) | ((redirect & ~(<1>T)) & ~(<2>T)))
     | ((interwiki & ~(<1>T)) & (~(<2>T) | <2>X2)))
     | ((status & ~(<1>T)) & (~(<2>T) | <2>X2))),
  X4=(((edit & (~(<1>T) | <1>X3)) & ~(<2>T)) | ((edit & (~(<1>T) | <1>X3)) & <2>X4)),
  X5=(((history & <1>X4) & ~(<2>T)) | ((interwiki & ~(<1>T)) & (~(<2>T) | <2>X5))),
  X6=((((history & <1>X4) & ~(<2>T)) | ((interwiki & ~(<1>T)) & (~(<2>T)
     | <2>X5))) | ((status & ~(<1>T)) & (~(<2>T) | <2>X5))),
  X7=((title & ~(<1>T)) & (~(<2>T) | <2>X6)),
  X8=(((text & ~(<1>T)) & ~(<2>T)) | ((redirect & ~(<1>T)) & ~(<2>T))),
  X9=((meta & <1>X7) & <2>X8),
  X10=((article & <1>X9) & ~(<2>T))
in
  X10)
```

Fig. 24. The $\mathcal{L}_\mu$ Formula for the DTD of Figure 22.

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 (by adding the restriction $\mu Z.\neg \langle \overline{1} \rangle \top \vee \langle \overline{2} \rangle Z$).

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 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.