Deciding extended modal logics by combining state space generation and SAT solving

Martin Strecker[0000−0001−9953−9871]

IRIT, University of Toulouse, France

Abstract. This paper presents a method of deciding extended modal formulas that arise, in particular, when reasoning about transformations of relational structures and graphs. The method proceeds by first unwinding a structure that is an over-approximation of potential models, and then selecting effective models with the aid of a SAT solver.

Keywords: Automated Theorem Proving, Modal Logic, Graph Transformations, Program Verification

1 Introduction

Modal logics have a relatively long history in computer science, and nevertheless, they are still an active research area. This is due to the wide spectrum of variants and possible application areas of modal logics. Basic modal logics have mainly been conceived for reasoning about possibility and necessity or related modalities, such as obligation and knowledge. Temporal logics such as LTL and CTL are variants that can capture properties of a system at different time instants and, thus, characterize how a system may evolve as time passes. An important practically relevant application is the verification of concurrent and reactive systems. Description Logics, used as the foundation of semantic databases and knowledge representation formalisms, have been recognized to be variants of modal logics.

The work presented here has arisen out of an effort to verify graph transformations, which are themselves important for reasoning about processes that modify graph-like structures, for example pointer-manipulating imperative programs. We will discuss this application area and resulting proof problems in Section 2. The essential feature of the formulas to be verified is that they give rise to models that are genuine graph and not tree structures, which is a considerable complication. The logic used in this paper will be defined in Section 3.

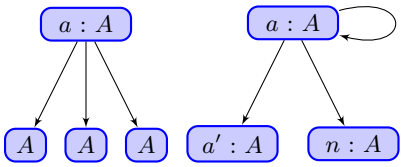
We here restrict our attention to propositional multi-modal logics. Syntactically, they are made up of propositional formulas to which modal operators indexed by binary relations, traditionally $\Box_r$ and $\Diamond_r$, can be applied. Semantically, these formulas are interpreted in Kripke structures, *i.e.* sets of possible worlds linked by binary accessibility relations. In each of these worlds, different combinations of elementary predicates may hold.

When it comes to proofs methods, several approaches are possible:

– Many modal logics can easily be embedded into first order logic, so that one can use first-order provers for attempting a proof. This approach is rather straightforward and easily adaptable to different variants of modal logic, but it has a severe drawback: many modal logics have pleasant meta-theoretic properties, such as the finite model property, and many of them are decidable. But there are no guarantees that a standard first-order prover will come to a halt when started on the translation of a modal logic formula. Indeed, it can be expected that the difficulties to be discussed in this article, like containing the number of worlds created during model exploration, have an analogy in first-order proof search, like preventing the generation of useless instances of universally quantified formulas.
– Tableaux are a standard method for trying to construct a model of a formula. They proceed by decomposing connectors until only elementary propositions are left and it is evident whether a model exists or not. Modal operators complicate the picture, because they lead to the creation of new worlds, or to copies of formulas from one world into another, and it is not evident that this procedure stops, in particular in the presence of graph structures. In Section 4, we will sketch tableau methods with the purpose of highlighting their difficulties, and for preparing the ground for the state space generation techniques to be introduced subsequently.
– SAT solvers have become impressively efficient for finding models of propositional proof problems. The difficulty consists in deriving a propositional formula corresponding to a modal formula, and this is the main topic of this paper. Our approach works in two phases: we traverse the modal formula a first time. By exploring the structure of modal operators in the formula, we derive an over-approximation of the graph that will yield the Kripke structure. Using this graph structure, we can traverse the modal formula a second time, in order to generate a propositional formula that can then be submitted to the SAT solver and determine the propositions true in each world.

Related work: There have been previous efforts to use SAT solvers for deciding modal [6] and description logics [10], and SMT solvers to come to terms with number restrictions in description logics [7]. As compared to the work considered here, the models constructed only have tree form, which is makes state space generation (*cf.* Section 5.2) substantially simpler. In [1], state space generation and model checking are interleaved, whereas we generate the potential state space in one run and then search for a possible model. There is a growing interest in quantifier instantiation [9] in conjunction with SAT/SMT solvers; the modal operators considered here pose the problem of quantifier instantiation (and the number of instances to be considered) in a specific context. There exist enumerative techniques, using bounded model checking, that are for example used in the Alloy analyzer [8] but that, contrary to the work presented here, are not complete.

```
vars a, n;   concepts A;   roles R;
pre: (a : A) && (a : (>= 3 R A));
select n with (a R n) && (n : A);
delete(a R n);
delete(a : A);
post: (a : !A) && (a : (>= 2 R A));
```



(a) Program (b) Input graph (c) Counterexample

Fig. 1: An example transformation

2 Intended Application

Our intention is to verify programs for graph transformations, automatically and in a sound and complete fashion. The idea is best illustrated by an example program, as in Figure 1a. The program consists of executable code (in black) and pre- and post-conditions (in blue, marked with `pre:` and `post:`). The pre-conditions describe the shape an input graph is assumed to have; the post-conditions describe the shape after the transformation.

The pre-condition states that the graph has a node `a` belonging to concept `A` (for this terminology, see Section 3; roughly, concepts are like types or classes) and that `a` has at least 3 successors with relation `R` which also belong to concept `A`. This latter requirement is expressed by `a: (>= 3 R A)`. A typical input graph is displayed in Figure 1b, where the successors of `a` are not named, but only their concept membership is shown. The program selects non-deterministically a node `n` linked to `a` with relation `R` and that is also of concept `A`. It then deletes the arc between `a` and `n` and removes `a` from concept `A`. The post-condition to be satisfied in the end states that `a` does not belong to concept `A` and `a` has at least 2 `R`-successors of concept `A`, which is correct for the input graph Figure 1b, However, when running our verifier, it comes up with the counterexample of Figure 1c, which is a graph also satisfying the pre-condition but violating the post-condition after completing the transformation.

We do not spell out the details of the program verification methodology here, see for example [5]. For our purposes, it suffices to say that correctness statements of the program (and in part also selection conditions as in the `select` statement) are formulas of a Description Logic, an extension of $\mathcal{ALCQ}$ whose main ingredients are individual variables (such as `a`), relations (`R`) and concepts (simple ones such as `A` and complex ones such as `(>= 3 R A)`, called number restrictions). The logic to be presented in Section 3 does not include number restrictions, see there. The example also highlights the fact that for our verification purposes, dealing with genuine graph structures is essential.

3 Logic

The logic considered in this paper can be understood as an extension of description logic $\mathcal{ALC}$ [2] or, alternatively, as an extension of a multi-modal logic. It

consists of a hierarchy of three syntactic categories: Concepts C (see Figure 2a) are Boolean combinations of set expressions built up from elementary concepts. Facts $fact$ (see Figure 2b) correspond to set membership statements of the form $i : C$ or being an instance of a role $i\ r\ i$. Differently said, we can reason with unary and binary relations (concepts resp. roles). On the last level of the hierarchy, there are formulas fm, which are Boolean combinations of facts.

The syntax is not minimal. Role complement $i\ (\neg r)\ i$ has been introduced for stating clash conditions in the tableau calculus, see Section 4. We often require formulas to be in negation normal form, obtained by recursively pushing negations inside, thereby swapping Boolean connectives, such as $\neg(C \sqcap D) = \neg C \sqcup \neg D$ and modal operators, such as $\neg(\Diamond_r\ C) = (\Box_r\ \neg C)$, or by feeding them into the next level of the syntactic hierarchy, such as $\neg(x : C) = (x : \neg C)$.

$C ::= \top$		(universal concept)
	$\bot$	(empty concept)
	c	(atomic concept)
	$\neg\ C$	(negation)
	$C\ \sqcap\ C$	(conjunction)
	$C\ \sqcup\ C$	(disjunction)
	$(\Diamond_r\ C)$	(existential quantifier)
	$(\Box_r\ C)$	(universal quantifier)

(a) Concepts

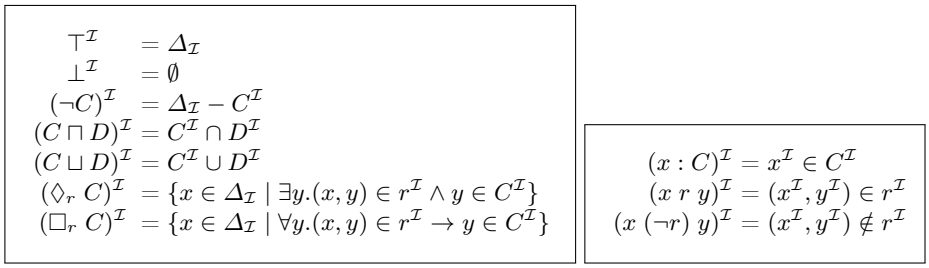
$fact ::= i : C$		(instance of concept)
	$i\ r\ i$	(instance of role)
	$i\ (\neg r)\ i$	(. . . role complement)
$fm ::= fact$		
	$\neg fm$	
	$fm \wedge fm$	
	$fm \vee fm$	

(b) Facts and formulas

Fig. 2: Syntax of the logic

For defining the semantics, we assume an interpretation to be a quadruple consisting of (1) a domain of elements $\Delta_{\mathcal{I}}$; (2) an interpretation function mapping elementary concepts c to sets $c^{\mathcal{I}}$ of elements; (3) an interpretation function mapping roles r to sets of pairs $r^{\mathcal{I}}$ of elements; and (4) an interpretation function mapping individual variables x to elements $x^{\mathcal{I}}$. We use this notion of interpretation both for quantifier-free first-order formulas with unary and binary predicate symbols, as introduced in Section 5.3, and for the modal logic of this section. Interpretations are extended to concepts as defined in Figure 3a: The concept constructors $\neg, \sqcap, \sqcup$ are translated by their set-theoretic counterparts; $\Diamond_r\ C$ is interpreted as the set of elements having an r-successor with property C, and $\Box_r\ C$ as the set of elements all of whose r-successors have property C. Facts are interpreted to produce a truth value (see Figure 3b); the extension to formulas (not shown) is then straightforward.

As usual, a *model* is an interpretation satisfying a formula. An alternative view on models is as graphs such as the one in Figure 1c, where nodes are tagged uniquely by variable names and decorated by the elementary concepts which are true for the corresponding variable, and directed arcs decorated by relation names.



(a) Semantics of concepts

(b) Semantics of facts

Fig. 3: Semantics

4 Tableau methods

We give an overview of the tableau method, with a particular emphasis on the way the modal operators are handled. We will see shortly that $\Diamond$ permits to "generate new worlds", and that there is a subtle interplay between $\Diamond$ and $\Box$ that the SAT method will have to simulate. We will first describe the calculus in Section 4.1 and then state essential properties in Section 4.2.

4.1 Calculus

The procedure manipulates a tableau, which is a set $\mathcal{A}$ of formulas. The formulas in $\mathcal{A}$ are decomposed depending on their shape, according to the rules in Figure 4, giving rise to a new tableau $\mathcal{A}'$. This process is formally modelled by a transition relation $\mathcal{A} \overset{f}{\hookrightarrow} \mathcal{A}'$. This relation is non-deterministic, as witnessed by the rules DISJC and DISJF. All formulas in $\mathcal{A}$ are supposed to be in negation normal form, an invariant maintained by the procedure; for this reason, there are no explicit rules for negated formulas.

We briefly comment on the rules: The rules for decomposing binary connectives (CONJC/DISJC for concepts and CONJF/DISJF for formulas) are standard for tableau procedures and directly reflect the semantics. For example, CONJC states that if x is member of the intersection of concepts $C_1 \sqcap C_2$, then it is member of each of C_1 and C_2. The side conditions of the rule ensure the termination of the calculus.

The modal rules are best explained with the graph view of models: the statement $x : (\Diamond_r C)$ expresses that node x in the graph has an r-successor marked C. Provided such a node does not yet exist, we create a new node z, an arc $(x\ r\ z)$ and mark node z with C. In this case, we also record the variable that has been created in the transition relation: $\mathcal{A} \overset{(z,f)}{\longrightarrow} \mathcal{A} \cup \{z : C, x\ r\ z\}$. The statement $x : (\Box_r C)$ postulates that all r-successors of x are marked with C, *i.e.* for all existing arcs $(x\ r\ y)$, the nodes y are marked with C. The applicability condition ensures that not all r-successors of x are already marked, because then we would not make progress.

$$\textsc{conj}\mathrm{C}\ \frac{(x : (C_1 \sqcap C_2)) \in \mathcal{A} \quad \neg((x : C_1) \in \mathcal{A} \ \wedge (x : C_2) \in \mathcal{A})}{\mathcal{A} \xhookrightarrow{(x:(C_1 \sqcap C_2))} \mathcal{A} \cup \{x : C_1, x : C_2\}}$$

$$\textsc{disj}\mathrm{C}\ \frac{\begin{array}{c}(x : (C_1 \sqcup C_2)) \in \mathcal{A} \quad (x : C_1) \notin \mathcal{A} \quad (x : C_2) \notin \mathcal{A} \\ \mathcal{A}' = \mathcal{A} \cup \{x : C_1\} \ \vee \mathcal{A}' = \mathcal{A} \cup \{x : C_2\}\end{array}}{\mathcal{A} \xhookrightarrow{(x:(C_1 \sqcup C_2))} \mathcal{A}'}$$

$$\Diamond\mathrm{C}\ \frac{(x : (\Diamond_r C)) \in \mathcal{A} \quad \forall y.\neg((x\ r\ y) \in \mathcal{A} \ \wedge (y : C) \in \mathcal{A}) \quad z \notin fv(\mathcal{A})}{\mathcal{A} \xhookrightarrow{(z,(x:(\Diamond_r C)))} \mathcal{A} \cup \{z : C, x\ r\ z\}}$$

$$\Box\mathrm{C}\ \frac{(x : (\Box_r C)) \in \mathcal{A} \qquad \exists y.(x\ r\ y) \in \mathcal{A} \wedge (y : C) \notin \mathcal{A}}{\mathcal{A} \xhookrightarrow{(x:(\Box_r C))} \mathcal{A} \cup \{g.(\exists y.(g = (y : C)) \wedge (x\ r\ y) \in \mathcal{A})\}}$$

$$\textsc{conj}\mathrm{F}\ \frac{f_1 \wedge f_2 \in \mathcal{A} \quad \neg(f_1 \in \mathcal{A} \ \wedge \ f_2 \in \mathcal{A})}{\mathcal{A} \xhookrightarrow{f_1 \wedge f_2} (\mathcal{A} - \{f_1 \wedge f_2\}) \cup \{f_1, f_2\}}$$

$$\textsc{disj}\mathrm{F}\ \frac{\begin{array}{c}f_1 \vee f_2 \in \mathcal{A} \quad f_1 \notin \mathcal{A} \quad f_2 \notin \mathcal{A} \\ \mathcal{A}' = (\mathcal{A} - \{f_1 \vee f_2\}) \cup \{f_1\} \ \vee \mathcal{A}' = (\mathcal{A} - \{f_1 \vee f_2\}) \cup \{f_2\}\end{array}}{\mathcal{A} \xhookrightarrow{f_1 \vee f_2} \mathcal{A}'}$$

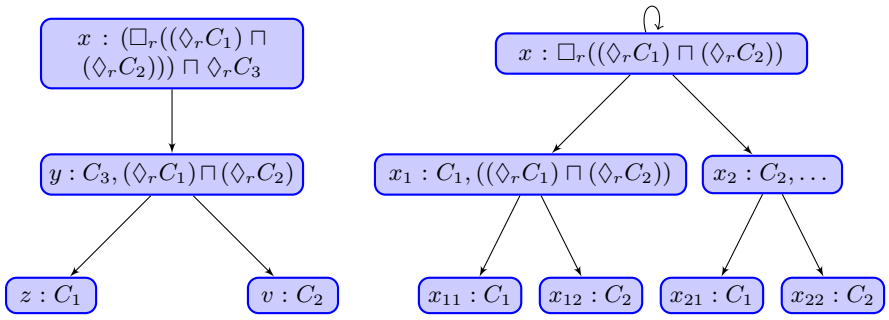
Fig. 4: Tableau rules

The aim of the procedure is to derive a contradiction, called a *clash*: A tableau $\mathcal{A}$ contains a clash if, for C a concept, r a role and x, y individual variables, $(x : \bot) \in \mathcal{A}$ or $\{x : C, x : \neg C\} \subseteq \mathcal{A}$ or $\{(x\ r\ y), (x\ (\neg r)\ y)\} \subseteq \mathcal{A}$. For determining whether an initial tableau $\mathcal{A}$ is satisfiable, the tableau procedure explores all complete tableaux reachable via the rule relation (a tableau is *complete* if no further rule is applicable). If all complete reachable tableaux contain a clash, then the initial tableau is unsatisfiable; otherwise, there is a complete clash-free tableau from which a model of the original tableau can be derived.

We note that most of the rules can only be applied at most once to a particular formula, which blocks the rule for further application. The $\Box$C rule is the only exception – and the essential difficulty of the calculus: application of other rules in the tableau can lead to the generation of new arcs $(x\ r\ z)$ that may trigger rule $\Box$C again. We consider two examples to illustrate the situation.

Example 1. The first example is of the kind typically considered in the literature for Description Logics. The aim is to ascertain whether a concept C is consistent, which can be done by verifying that the tableau $\{x : C\}$ is satisfiable. An instance of this situation is the following: $\mathcal{A}_0 = \{x : (\Box_r((\Diamond_r C_1) \sqcap (\Diamond_r C_2))) \sqcap \Diamond_r C_3\}$.

Decomposing the conjunction yields $\mathcal{A}_1 = \mathcal{A}_0 \cup \{x : \Box_r((\Diamond_r C_1) \sqcap (\Diamond_r C_2)), x : \Diamond_r C_3\}$. Applying the $\Diamond$C rule introduces a new variable y, such that $\mathcal{A}_2 =$



(a) see Example 1 (b) see Example 2

$\mathcal{A}_1 \cup \{(x\ r\ y), y : C_3\}$. With the $\Box$C rule, we get $\mathcal{A}_3 = \mathcal{A}_2 \cup \{y : (\Diamond_r C_1) \sqcap (\Diamond_r C_2)\}$. Another round of decompositions and $\Diamond$C rule applications gives the complete, satisfiable tableau $\mathcal{A}_4 = \mathcal{A}_3 \cup \{(y\ r\ z), z : C_1, (y\ r\ v), v : C_2\}$.

It should be noted that the resulting graph is a tree (see Figure 5a, here annotated with atomic and composite concepts), and an easy induction shows that this is indeed so for all models derived from an initial tableau of the form $\{x : C\}$. Furthermore, from the syntax tree of the original concept, one can read off the structure of the resulting model: Each direct $\Diamond_r$ subtree in the syntax (direct = not separated by a modal operator) gives potentially rise to a new child node in the model, and these are the only children. It is therefore possible to index the generated nodes by positions in the original formula. This is the approach taken in [6] to identify all nodes in the model to be generated.

Example 2. The initial tableau $\mathcal{A} = \{(x\ r\ x),\ x : \Box_r((\Diamond_r C_1) \sqcap (\Diamond_r C_2))\}$ already starts out with a partial model (the node x with a self-loop) that is not tree-shaped. Without the arc $(x\ r\ x)$, the tableau would be complete; with it, we apply the $\Box$C rule and add $x : (\Diamond_r C_1) \sqcap (\Diamond_r C_2)$ to the tableau. Decomposition and rule $\Diamond$C yield two nodes $x_1 : C_1$ and $x_2 : C_2$ and arcs $(x\ r\ x_1)$ and $(x\ r\ x_2)$. This triggers the $\Box$C rule twice, and we obtain $x_1 : (\Diamond_r C_1) \sqcap (\Diamond_r C_2)$ and similarly for x_2. After a renewed decomposition, we obtain still another set of nodes $x_{11}, x_{21} : C_1$ and $x_{12}, x_{22} : C_2$.

There are two main differences compared to Example 1: Indeed, the resulting model is not a tree (see Figure 5b) but, in a sense, tree-like: all the newly generated nodes only have one parent; and the new nodes cannot be indexed by the syntactic structure of the formula in a straightforward way. In the node generating algorithm of Section 5.2, we have therefore avoided to do so.

4.2 Properties

We sketch soundness and completeness arguments for the tableau calculus (which are relatively standard) and present more in detail a novel termination result that will also be instrumental for the discussion in Section 5.2.

Let us designate by $\mathcal{A} \overset{*}{\hookrightarrow} \mathcal{A}'$ that tableau $\mathcal{A}'$ is reachable by a sequence of applications of tableau rules from tableau $\mathcal{A}$. Let $\overset{+}{\hookrightarrow}$ be the strict part of $\overset{*}{\hookrightarrow}$.

Theorem 1 (Soundness). *The tableau calculus is sound: if* $\mathcal{A} \overset{*}{\hookrightarrow} \mathcal{A}'$ *and* $\mathcal{A}'$ *is satisfiable, then so is* $\mathcal{A}$.

Proof. The proof is by induction on the derivation, showing that each rule application does not create new models. □

Theorem 2 (Completeness). *The tableau calculus is complete: if* $\mathcal{A}'$ *is unsatisfiable for all* $\mathcal{A}'$ *with* $\mathcal{A} \overset{*}{\hookrightarrow} \mathcal{A}'$, *then so is* $\mathcal{A}$.

Proof. The proof is by induction on the derivation, showing that the model property is preserved by at least one of the alternatives of each derivation. □

As far as termination of the calculus is concerned, the situation is considerably more complex for graph-like structures than for tree structures, as motivated in the above examples. When starting with a proof problem $x : C$, a tableau derivation generates a tree with new nodes of the form $x' : C'$, where C' is a strict subconcept of C. The well-foundedness of the subconcept order is the essential ingredient for the termination argument in the case of tree structures.

However, in a graph structure with nodes $x_1 : C_1$ and $x_2 : C_2$ and relation $(x_1\ r\ x_2)$, the node x_2 may be decorated with more complex concepts C than C_2 in the course of the proof, for example if C_1 is $\Box_r C$, with C more complex than C_2.

The termination argument developed here uses the notion of *bound*, which is a set of formulas with which a variable can potentially be decorated. In contrast, the *annotation* of a variable x in a tableau $\mathcal{A}$ is the set $\{C \,.\, (x : C) \in \mathcal{A}\}$ with which the node is actually annotated. To take a simple example, the node decorated with $x : (A \sqcup B) \sqcap C$ will be bounded by the set $\{(A \sqcup B) \sqcap C, A \sqcup B, A, B, C\}$. After a decomposition with the CONJC rule, additional annotations will be $x : A \sqcup B$ and $x : C$, all of which remain within the bounds.

We will define an order on pairs $(\mathit{bound}, \mathit{annotation})$ as follows: $(b_1, a_1) < (b_2, a_2) := (\mathit{finite}(b_2) \wedge b_1 \subseteq b_2 \wedge b_2 \supseteq a_1 \wedge a_1 \supset a_2)$. In a tableau proof, when deriving a new node from an existing node, the bound may become smaller $(b_1 \subseteq b_2)$ whereas the actual annotation should increase $(a_1 \supset a_2)$ and come closer to the former bound without exceeding it $(b_2 \supseteq a_1)$. It can be shown that this order is well-founded. With these ingredients, we can now state and prove that the tableau rules do not permit infinite derivations:

Theorem 3 (Termination). *Relation* $\overset{+}{\hookrightarrow}$ *is well-founded.*

Proof. We begin by defining a variant of the calculus that, apart from formulas, also manages bounds. In the initial tableau, all the variables have the same bound, *viz.* the set of all subconcepts occurring in the tableau. In general, nodes keep their bounds as the result of a rule application; the only exception is the $\Diamond$C rule, which is the only rule that creates new nodes. If x is the node that the $\Diamond$C rule is applied to and it has a bound containing modal operators $\Box_{r_1} B_1, \ldots, \Box_{r_m} B_m, \Diamond_{s_1} D_1, \ldots, \Diamond_{s_n} D_n$, then the new node z that is generated (*cf.* Figure 4) will have a bound that is the subconcept-closure of $\{B_1, \ldots, B_m, D_1, \ldots, D_n\}$.

We now define the *potential of a node* x as the pair

– $(bound, annotation)$ of node x with the order defined above;
– number of facts $x : \Diamond_r C$ in the tableau to which the rule $\Diamond$C is still applicable;

endowed with a lexicographic order. The *potential of a formula* is the tuple

– size of the formula, defined as the number of formula constructors of the syntax trees, where facts have size 0;
– node potential of the node x for facts of the form $x : C$;

endowed with a lexicographic order. The potential of a tableau is then the multiset of the potentials of the formulas it contains.

Obviously, the rules CONJF and DISJF decrease the potential of a tableau, by decreasing the potential of a formula. The rules CONJC and DISJC decrease the potential of a node, by increasing the *annotation* component while keeping the bound constant. The $\Diamond$C rule replaces the potential of node x by two smaller potentials (the node x with the number of $\Diamond$C decreased, and the node z with lower bounds). The $\Box$C rule, when applied to a node x linked to a node y (*cf.* Figure 4), will increase the annotation of node y. It may at the same increase the number of possible $\Diamond$C applications of y, but the net effect is to decrease the potential of y. □

5 Translation to Propositional Logic

5.1 Principles

As has been seen in Section 4, a tableau interleaves decomposition of formulas and generation of new variables corresponding to nodes of the model. The disjunctive rules are non-deterministic. In practice, different strategies exist for exploring these alternatives, among them depth-first search with backtracking or breadth-first search. In both cases, there is a risk of duplicate work, *i.e.* of testing over and over again which combination among a set of mutually incompatible choices (such as $x : C$ or $x : \neg C$) leads to a satisfiable formula.

The principal hypothesis of our approach is that it is more efficient to lay out the whole space of possible worlds in a first step. A central ingredient for this is the notion of *node*, which is the representative of a variable x in a tableau and which collects all the concept membership information $x : C_1, \ldots x : C_n$ available in a tableau. For this, we generate in parallel all the nodes a tableau procedure might create, and then carry out the combinatorial exploration by a procedure optimized for that purpose, namely a SAT solver.

The node generation phase (Section 5.2) mimics the tableau algorithm, as far as the modal operators are concerned: for $\Diamond$, this means to create new successor nodes, and for $\Box$, enrich nodes with new concepts, and this is repeated until reaching a fixpoint. In this process, we have to memorize which instances have already been created, and various other information. In all of the following discussion, we assume formulas, facts and concepts in negation normal form.

Terminology: Before describing the structure of nodes, we fix some terminology. The set of *subconcepts* of a concept C is defined as consisting of C and recursively of all immediately constituent subconcepts of C. The set of *Boolean subconcepts* is the set of all subconcepts except for those below modal operators. Thus, the set of Boolean concepts of $\Box_r C_1 \sqcap (C_2 \sqcup \Diamond_r (C_3 \sqcap C_4))$ is $\{\Box_r C_1,\ C_2,\ \Diamond_r (C_3 \sqcap C_4)\}$.

We denote y as a *successor instance* of a fact $x : \Box_r C'$ if y is an r-successor of x and $y : C'$. We now describe the record structure *node* that contains this information. It consists of the fields

- *name*: a unique identifier for the node, whose precise structure is immaterial.
- *new*: a list of newly added subconcepts that still have to be processed.
- *old*: keeps track of subconcepts that already have been processed; necessary for ensuring termination.
- *somec*: in principle, a list of Boolean $\Diamond$-subconcepts of the initial concept C of this node, *i.e.* the Boolean subconcepts of C that have the form $\Diamond_r C'$. Instead of storing a list of concepts $\Diamond_r C'$, we record a list of tuples (r, C').
- *allc*: in principle, a list of Boolean $\Box$-subconcepts of the initial concept C of this node, *i.e.* the Boolean subconcepts of C that have the form $\Box_r C'$. As for $\Diamond$, instead of a list of concepts $\Box_r C'$, we record a list of tuples (r, C').

A note on syntax: The description of the algorithms presented in the following has been derived from an implementation in Ocaml, from which we have borrowed the syntax for lists: [] for the empty list and :: for consing. Duplicate-free concatenation is written $\cup$. We enumerate record components within banana brackets, and write record updates as $r(\!|c := v|\!)$ (update of component c of record r with value v). Selection of component c of record r is written as $r.c$. Adding an element e to a component c in a record r is written as $r(\!|c += e|\!)$, shorthand for $r(\!|c := \{e\} \cup r.c|\!)$.

5.2 Generating the set of nodes

Nodes of a concept We now present function $\mathcal{N}_c()$ which computes the set of nodes for a concept. The function manipulates a work list of nodes (record structure *node*) that still have to be processed; once a node is finished, it is added to the list of nodes whose processing is complete. Manipulating the work list amounts to manipulating in turn the nodes that will make up the Kripke structure. Apart from recursing over the list of nodes, we also consider the *new* subconcepts of the current node, which corresponds to decomposing the concepts of the current node. We also keep track of the relations between these nodes, with a relation represented as a list of triples (r, x, y): relation name r, source x and target y node identifier. Thus, altogether, function $\mathcal{N}_c()$ takes as arguments a list of completed nodes *nds*, a list of relations *rels* between nodes, and the worklist *wns*. The function returns the list of completed nodes and the relations.

The function can be written straightforwardly as a tail-recursive function, but in order to better distinguish the different patterns, we present the function in a rule format in Figure 6. Our actual implementation differs in one detail:

$$\textsc{End}\ \frac{(nds', wns') = partition\_allc\_instances(nds, rels) \qquad wns' = []}{\mathcal{N}_c(nds, rels, []) = (nds', rels)}$$

$$\textsc{Rel}\ \frac{(nds', wns') = partition\_allc\_instances(nds, rels) \qquad wns' \neq []}{\mathcal{N}_c(nds, rels, []) = \mathcal{N}_c(nds', rels, wns')}$$

$$\textsc{PopN}\ \frac{wn.new = []}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(wn :: nds, rels, wns)}$$

$$\textsc{Sk}\Diamond\ \frac{wn.new = \Diamond_r C :: cs \quad (r, C) \in wn.somec}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(nds, rels, (wn(\!|new := cs;\ old+ = \Diamond_r C|\!) :: wns))}$$

$$\textsc{Dec}\Diamond\ \frac{\begin{array}{c} wn.new = \Diamond_r C :: cs \quad (r, C) \notin wn.somec \\ (nnd, nrel) = create\_somec\_successor(wn.name, r, C) \\ wn' = wn(\!|new = cs;\ old+ = \Diamond_r C;\ somec+ = (r, C)|\!) \end{array}}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(nds, nrel :: rels, nnd :: wn' :: wns)}$$

$$\textsc{Dec}\Box\ \frac{wn.new = \Box_r C :: cs \qquad wn' = wn(\!|new = cs;\ old+ = \Box_r C;\ allc+ = (r, C)|\!)}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(nds, rels, wn' :: wns)}$$

$$\textsc{SkB}\ \frac{wn.new = c :: cs \qquad c = (C_1 \sqcap C_2) \vee c = (C_1 \sqcup C_2) \qquad (C_1\, \epsilon\, wn \wedge C_2\, \epsilon\, wn)}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(nds, rels, (wn(\!|new := cs;\ old+ = c|\!)) :: wns)}$$

$$\textsc{DecB}\ \frac{\begin{array}{c} wn.new = c :: cs \qquad c = (C_1 \sqcap C_2) \vee c = (C_1 \sqcup C_2) \\ \neg(C_1\, \epsilon\, wn \wedge C_2\, \epsilon\, wn) \qquad wn' = (wn(\!|new := \{C_1, C_2\} \cup cs;\ old+ = c|\!)) \end{array}}{\mathcal{N}_c(nds, rels, wn :: wns) = \mathcal{N}_c(nds, rels, wn' :: wns)}$$

Fig. 6: Computing the node set

the implemented function takes an additional counter for keeping track of node names. To avoid clutter, we assume here that generation of fresh names happens behind the scenes.

Rules: Let us now comment on the rules.

– End and Rel are the cases when the worklist is empty. For deciding what to do, we split the existing node set nds into a set of definitely finished nodes nds' and nodes wns' that have to be reprocessed. If wns' is empty, we are

done (End) and return the nodes and relations accumulated so far. Otherwise, we relaunch (Rel) the function with the new worklist. The auxiliary function *partition_allc_instances* will be described further below.

- PopN transfers a worklist node with an empty *new* component to the list of completed nodes, because it contains no more concepts to be processed.
- The remaining rules all assume that the *new* list of the current node wn is not empty, and manipulate its first element. If this first element is $\Diamond_r c$, there are two cases:
  - Sk$\Diamond$: there already exists an r-successor node containing C attached to the current node ($(r, C) \in wn.somec$). This corresponds to the situation when the tableau rule $\Diamond$C is not applicable; in this case, we skip $\Diamond_r C$ and continue with the rest of the *new* list.
  - Dec$\Diamond$: no such successor exists. This is the case when the rule $\Diamond$C is applicable, and we decompose the operator. So assume that the current node has name x, then function *create_somec_successor* generates a node *nnd* with a fresh name, say z, and a relation $nrel = (r, x, z)$. We memorize that we now have an appropriate r-successor of concept C for node x, so that we do not create one again (the case handled by Sk$\Diamond$). The new node and the modified current node are added to the worklist.
- Dec$\Box$: decomposition of the box is applied if the first element is $\Box_r C$. Nothing interesting happens at this point: we record in the *allc* component that the current node has a $\Box$ modal operator. It is during partitioning with function *partition_allc_instances* that this information will be propagated.
- SkB and DecB for handling Boolean connectors. Let us look at conjunction $C_1 \sqcap C_2$ first. The skip rule SkB corresponds to the case when the ConjC tableau rule is not applicable (we write $C \,\epsilon\, wn$ for $C \in wn.new \cup wn.old$), and the concept $c = C_1 \sqcap C_2$ is simply marked as old. Otherwise, the decompose rule DecB adds C_1 and C_2 to the new nodes, in analogy to the ConjC rule. Perhaps surprizingly, disjunction is handled the same way, corresponding to following simultaneously two different evolutions of the tableau as of rule DisjC. The fact the set of concepts accumulated in a node now possibly becomes inconsistent is not relevant at this stage and will be taken care of by formula translation in Section 5.3.

Partition instances Without giving a full definition, we now describe the idea of function *partition_allc_instances*. To motivate the special treatment reserved to the $\Box$ operator, let us remark that all the rules of the tableau calculus remain inhibited after one application to a particular instance; only the $\Box$C rule is an exception, where an application to instance $x : \Box_r C$ can be reactivated whenever a new link to a node y is created. So *partition_allc_instances*$(nds, rels)$ does the following: For each node x of *nds* with $x : \Box_r C$ as recorded in the *allc* field, and for each relation $(r\ x\ y)$ in *rels*, it checks whether $y : C$ (*i.e.* concept C is in the *old* field of node y). If this is so, node y remains inactive. Otherwise, C is added to the *new* component of y, and node y is put back into the worklist.

Nodes of a formula Function $\mathcal{N}_c()$ is by far the most complex function. The analogous function on formulas, $\mathcal{N}_f(nds, rels, fm)$, traverses formula fm recursively, gathering all the nodes and relations it finds. For a fact $x : C$, it either creates a node x and adds concept C to its *new* list, or simply adds C to *new* if node x already exists. We will write $\mathcal{N}_f(fm)$ instead of $\mathcal{N}_f([], [], fm)$.

5.3 Translating concepts and formulas

Given a formula, once we have computed an over-approximation of the nodes and relations of the Kripke structure of this formula, we can translate it to propositional logic. Checking the satisfiability of this translated formula either demonstrates its unsatisfiability (in which case the original formula is unsatisfiable as well) or yields a model.

We now define this translation, first with function $\mathcal{T}_c()$ for concepts which takes as additional argument the set of nodes *nds* which are a (not necessarily strict) superset of the nodes of the model to be constructed. The translation rules are displayed in Figure 7.

$$
\begin{aligned}
\mathcal{T}_c(nds, x, \top) &= true \\
\mathcal{T}_c(nds, x, \bot) &= false \\
\mathcal{T}_c(nds, x, c) &= c(x) \\
\mathcal{T}_c(nds, x, \neg\, c) &= \neg c(x) \\
\mathcal{T}_c(nds, x, C_1 \sqcap C_2) &= \mathcal{T}_c(nds, x, C_1) \wedge \mathcal{T}_c(nds, x, C_2) \\
\mathcal{T}_c(nds, x, C_1 \sqcup C_2) &= \mathcal{T}_c(nds, x, C_1) \vee \mathcal{T}_c(nds, x, C_2) \\
\mathcal{T}_c(nds, x, \Diamond_r\ C') &= \bigvee_{y \in Y} r(x, y) \wedge \mathcal{T}_c(nds, y, C') \\
\mathcal{T}_c(nds, x, \Box_r\ C') &= \bigwedge_{y \in Y} (r(x, y) \longrightarrow \mathcal{T}_c(nds, y, C')) \\
& \quad where\ Y = \{y.\ \ y = nd.name \wedge nd \in nds\}
\end{aligned}
$$

Fig. 7: Translating concepts to propositional logic

The rules for $\top, \bot$ and elementary (positive or negative) concepts c are straightforward. The rules for conjunction and disjunction reflect the semantics of the respective connective. As to the translation of the modal operators $\Diamond$ and $\Box$, remember that the set of nodes *nds* represents an over-approximation of the domain of the model to be constructed. For both operators, we project out the names of the nodes, to obtain the set Y of all node names. We then quantify existentially resp. universally over this set. This reflects the semantics of the operators (*cf.* Figure 3a) with $(x \in (\Diamond_r\ C)^{\mathcal{I}}) = \exists y \in \Delta_{\mathcal{I}}.\ (x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}$ and $(x \in (\Box_r\ C)^{\mathcal{I}}) = \forall y \in \Delta_{\mathcal{I}}.\ (x, y) \in r^{\mathcal{I}} \longrightarrow y \in C^{\mathcal{I}}$. The translation of formulas, $\mathcal{T}_f(nds, fm)$, is then a recursive function traversing formula fm, such

that $\mathcal{T}_f(nds, (x : C)) = \mathcal{T}_c(nds, x, C)$. Both $\mathcal{T}_c()$ and $\mathcal{T}_f()$ are defined by simple structural recursion, so their termination is evident.

6 Soundness and Completeness

We summarize the essential steps of our verification framework and then demonstrate that it effectively provides a decision procedure for the formulas defined in Section 3. Given a formula fm:

- Generate the set of nodes and use these to translate the formula to propositional logic: $\mathcal{T}_f(\mathcal{N}_f(fm), fm)$.
- Submit the resulting formula to a SAT solver which will either state its unsatisfiability or provide a model, yielding a model of the original formula.

Lemma 1 (Termination of node generation). *$\mathcal{N}_f()$ and $\mathcal{N}_c()$ terminate.*

Proof. $\mathcal{N}_f()$ is a simple structurally recursive function whose termination is immediate. For $\mathcal{N}_c()$, we can essentially use the concept of *potential* introduced for the proof of Theorem 3, and show that recursive calls of the function lead to a decrease in the multiset of potentials of nds. □

We first show that the translation function $\mathcal{T}_c()$ preserves models, under certain circumstances. To get an intuition, take the formula $x : (\Diamond_r c) \wedge x : (\Diamond_r (\neg c))$. It stipulates that x has a successor y_1 where c holds, and a successor y_2 where $\neg c$ holds. The formula, translated to predicate logic, is $(\bigvee_{y \in Y} .r(x, y) \wedge c(y)) \wedge (\bigvee_{y \in Y} .r(x, y) \wedge \neg c(y))$. This formula is only satisfiable for a set Y consisting of at least two nodes, thus if y_1 and y_2 are not forced to be the same. We conclude that $\mathcal{T}_c()$ preserves models if the node set nds that is a parameter of $\mathcal{T}_c()$ is "sufficiently large". In a sense, the main difficulty of the soundness and completeness result resides in showing that $\mathcal{N}_c()$ expands to a universe with enough elements.

Lemma 2 (Model preservation of translation).

1. *If $\mathcal{I}$ is a model of $\mathcal{T}_c(nds, x, C)$, then also of $x : C$.*
2. *Let $\mathcal{I}$ be a model of $x : C$, and let there be an injective mapping from $\Delta_\mathcal{I}$ into nds. Then there exists a model $\mathcal{I}'$ of $\mathcal{T}_c(nds, x, C)$.*

Analogous results hold for the translation function $\mathcal{T}_f()$.

Proof.

1. A model of $\mathcal{T}_c(nds, x, C)$ immediately gives a model of $x : C$ (*cf.* semantics in Figure 3a).
2. We construct inductively a model $\mathcal{I}'$ whose domain $\Delta_{\mathcal{I}'}$ is a superset of $\Delta_\mathcal{I}$. The cases where the concept C is a constant or an elementary concept are immediate. The case of disjunction is easy. For the case of a conjunction $C_1 \sqcap C_2$, the construction inductively yields two models $\mathcal{I}'_1$ and $\mathcal{I}'_2$ of C_1 and C_2 respectively which might be mutually incompatible (as seen in the above

example). By a remapping of variables, we can construct two models that coincide on the interpretation of x and otherwise map variables to disjoint elements of $\Delta_{\mathcal{I}}$. By joining these two models into a single interpretation $\mathcal{I}'$, we obtain a model of the translation of $C_1 \sqcap C_2$.
As to the modal operators: Let $\mathcal{I}$ be a model of $x : \Diamond_r C'$, so $x^{\mathcal{I}} \in \Delta_{\mathcal{I}}$ and there exists $y^{\mathcal{I}} \in \Delta_{\mathcal{I}}$ such that $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in r^{\mathcal{I}} \wedge y^{\mathcal{I}} \in C^{\mathcal{I}}$. Let m be the injective mapping from $\Delta_{\mathcal{I}}$ into nds. The interpretation of the translated formula $\bigvee_{y \in Y} r(x, y) \wedge \mathcal{T}_c(nds, y, C')$ is possibly over a larger domain (the set Y may contain more elements than $\Delta_{\mathcal{I}}$), so we decide to interpret any $\mathcal{I}'(y)$ for y in the image of m as $\mathcal{I}(m^{-1}(y))$, and any other y arbitrarily. Similarly, relation $r(x, y)$ for y in the image of m is interpreted as for $\mathcal{I}$ and as false otherwise. This interpretation satisfies the translated formula. The argument for the $\Box$ operator is analogous. □

Theorem 4 (Soundness). *The proof method, when applied to a formula fm, is sound: A model found by the solver for $\mathcal{T}_f(fm)$ is also a model of fm.*

Proof. We assume that the SAT solver is sound, so a model that the solver claims to be one for $\mathcal{T}_f(fm)$ is indeed one. With Lemma 2(1), we thus obtain a model of fm. Remark that this is independent of the correctness of $\mathcal{N}_c()$ and $\mathcal{N}_f()$. □

Theorem 5 (Completeness). *The proof method, when applied to a formula fm, is complete: Whenever there exists a model for fm, then the solver will find one for $\mathcal{T}_f(\mathcal{N}_f(fm), fm)$.*

Proof. Suppose fm has a model. According to Theorem 2 and 3, starting from tableau $\mathcal{A} = \{fm\}$, there exists a terminating run $\mathcal{A} \overset{*}{\hookrightarrow} \mathcal{A}'$ such that $\mathcal{A}'$ permits to construct a model $\mathcal{I}$ of fm with domain $\Delta_{\mathcal{I}} = vars(\mathcal{A}')$. It is easy to show that $\mathcal{N}_c()$ returns a set of nodes nds such that $vars(\mathcal{A}') \subseteq nds$. According to Lemma 2(2), there exists a model $\mathcal{I}'$ of $\mathcal{T}_f(nds, fm)$, and since the solver is assumed to be complete, it will find a model of $\mathcal{T}_f(nds, fm)$. □

7 Conclusions

We have presented a decision procedure for an extension of a modal logic that is particularly appropriate for reasoning about graphs and their transformations, and thus constitutes an essential increase of expressivity *w.r.t.* logics restricted to tree models.

The approach described here has been implemented in a prototype in Ocaml, using alternatively CVC4 [3] or veriT [4] as SAT solvers. The prototype shows a good response time for formulas from the application scenario it is intended for, but it has not been exercised on performance benchmarks. The models obtained by this method are sometimes surprizing, yielding (often very compact) genuine graphs where a tree model would also exist. Sometimes, however, the model contains a great number of nodes that are spurious in the sense that already a subgraph would be a model. This indicates a potential for optimizations. Further work to be considered are more expressive logics, for example including number restrictions (as in Section 2) or particular properties of relations, like transitivity.

Acknowledgements. Most of the work described here has been carried out while the author was at Inria Nancy during a research stay financially supported by Inria and hosted by the VeriDis team. I am grateful to Stephan Merz for inviting me, and to him and to Pascal Fontaine for discussions about and suggestions for improvement of this paper.

References

1. Areces, C., Fontaine, P., Merz, S.: Modal satisfiability via SMT solving. In: Hennicker, R., de Nicola, R. (eds.) Software, Services, and Systems. Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, LNCS, vol. 8950, pp. 30–45. Springer (2015)
2. Baader, F., Lutz, C.: Description logic. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) The Handbook of Modal Logic, pp. 757–820. Elsevier (2006)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 171–177. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)
4. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, Springer-Verlag (2009)
5. Brenas, J.H., Echahed, R., Strecker, M.: A Hoare-like calculus using the SROIQ$^\sigma$ logic on transformations of graphs. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) Theoretical Computer Science, Lecture Notes in Computer Science, vol. 8705, pp. 164–178. Springer Berlin Heidelberg (2014)
6. Giunchiglia, E., Tacchella, A., Giunchiglia, F.: SAT-based decision procedures for classical modal logics. Journal of Automated Reasoning **28**(2), 143–171 (2002). https://doi.org/10.1023/A:1015071400913
7. Haarslev, V., Sebastiani, R., Vescovi, M.: Automated reasoning in $ALCQ$ via SMT. In: Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, 2011. Proceedings. pp. 283–298 (2011)
8. Jackson, D.: Software Abstractions. MIT Press (2011)
9. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS, Thessaloniki, Greece. Lecture Notes in Computer Science, vol. 10806, pp. 112–131. Springer (2018)
10. Sebastiani, R., Vescovi, M.: Automated reasoning in modal and description logics via SAT encoding: the case study of K (m)/ALC-satisfiability. Journal of Artificial Intelligence Research **35**(1), 343 (2009)