CoLoSS: The Coalgebraic Logic Satisfiability Solver (System Description)

Georgel Calin[1]

Jacobs University Bremen, Germany

Rob Myers[2] Dirk Pattinson[3]

Imperial College London, UK

Lutz Schröder[4,5]

DFKI Lab Bremen and Universität Bremen, Germany

Abstract

CoLoSS, the Coalgebraic Logic Satisfiability Solver, decides satisfiability of modal formulas in a generic and compositional way. It implements a uniform polynomial space algorithm to decide satisfiability for modal logics that are amenable to coalgebraic semantics. This includes e.g. the logics K, KD, Pauly's coalition logic, graded modal logic, and probabilistic modal logic. Logics are easily integrated into CoLoSS by providing a complete axiomatisation of their coalgebraic semantics in a specific format. Moreover, CoLoSS is compositional: it synthesises decision procedures for modular combinations of logics that include the fusion of two modal logics as a special case. One thus automatically obtains reasoning support e.g. for logics interpreted over probabilistic automata that combine non-determinism and probabilities in different ways.

Keywords: Modal logic, automatic proving, coalgebra.

Introduction

While there are numerous automatic proof tools for modal logics with a standard Kripke semantics, developed in particular in tow of the recent rise of modal logic as a background formalism for description logics (e.g. Pellet [19], FaCT/FaCT++ [20],

[1] g.calin@iu-bremen.de

[2] rm606@doc.ic.ac.uk

[3] dirk@doc.ic.ac.uk

[4] Lutz.Schroeder@dfki.de

[5] Research supported by the DFG project HasCASL2 (KR 1191/7-2)

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and RACER [3]), almost no tool support exists (with the notable exception of the conditional logic prover CondLean [11]) for a growing number of newly designed non-normal modal logics whose semantics involves structures which are different from, and mostly more complex than, standard Kripke frames. Examples of such logics include probabilistic modal logic [9,5], majority logic [12], coalition logic [13], and Presburger modal logic [2].

Coalgebraic modal logic has recently been shown to provide a suitable generic semantic framework for modal logics which in particular allows the design of generic decision procedures that can be instantiated for specific modal logics with moderate effort. In particular, many logics axiomatised without nesting of modal operators can be decided in *PSPACE* using a generic satisfiability solver [15].

Here, we give a system description of the Coalgebraic Logic Satisfiability Solver (CoLoSS), an implementation framework in which instances of such generic algorithms are easily integrated. CoLoSS is implemented in Haskell and exploits the Haskell type class mechanism to facilitate easy creation of new logic instances; it comes with a (growing) number of pre-implemented instances including K, KD, graded and probabilistic modal logics, and coalition logic. Implementing a new logic instance essentially amounts to no more than defining its syntax and providing an interface function embodying its axiomatisation.

CoLoSS moreover supports the modular combination of logics and decision procedures described in [16]. This means e.g. that by providing instances for probabilistic modal logic and multi-agent K, one automatically obtains instances also for composite modal logics such as the modal logic of Segala systems (cf. Sect. 3) or the modal logic of alternating systems [16].

The material is organised as follows. We provide brief summaries of the theoretical background on the generic *PSPACE* algorithm for coalgebraic modal logic and logic combination in Sections 1 and 3. The implementation of the core algorithm for satisfiability (in fact, validity) checking is described in Sect. 2, while the realisation of the logic combination mechanism is discussed in Sect. 4. The source code of CoLoSS is available under `http://www.doc.ic.ac.uk/~dirk/COLOSS/coloss.tar.gz`

1 Generic Validity Checking for Coalgebraic Modal Logics

We briefly review coalgebraic modal logic as a generic semantic framework for modal logic, and the generic *PSPACE* decision procedure for satisfiability based on it [15], which we dualise to an algorithm for validity. The syntax of a modal logic is given by a *modal signature* Λ containing *modal operators* of given arities; the set $\mathcal{F}(\Lambda)$ of Λ-formulas is then determined by the grammar

$$\psi ::= \bot \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid L(\psi_1, \ldots, \psi_n),$$

where $L \in \Lambda$ is n-ary. We parametrize the semantics of such modal languages by the choice of a Λ-*structure* $\mathcal{M}$ (*over* T) consisting of a signature functor

$$T : \mathbf{Set} \to \mathbf{Set}$$

on the category **Set** of sets and maps, and an assignment of an *n-ary predicate lifting* $\mathcal{M}[\![L]\!]$ for T, i.e. a natural transformation

$$\mathcal{M}[\![L]\!] : \mathcal{Q}^n \to \mathcal{Q} \circ T^{op}$$

with $\mathcal{Q} : \mathbf{Set}^{op} \to \mathbf{Set}$ denoting contravariant powerset, to every n-ary modal operator $L \in \Lambda$. We omit the mention of $\mathcal{M}$ when this is unlikely to cause confusion. Recall that a *T-coalgebra* $C = (X, \xi)$ consists of a set X of *states* and a *transition map* $\xi : X \to TX$. The structure $\mathcal{M}$ determines satisfaction relations $\models_C$ between *states* $x \in X$ of T-coalgebras $C = (X, \xi)$ and $\mathcal{F}(\Lambda)$-formulas over V. The relation $\models_C$ is defined inductively, with the usual clauses for boolean operators. The clause for an n-ary modal operator L is

$$x \models_C L(\phi_1, \ldots, \phi_n) \;\Leftrightarrow\; \xi(x) \in [\![L]\!]([\![\phi_1]\!]_C, \ldots, [\![\phi_n]\!]_C)$$

where $[\![\phi]\!]_C = \{x \in X \mid x \models_C \phi\}$. We drop the subscripts C when these are clear from the context. Our main interest is in the *validity problem* over $\mathcal{M}$ (and its dual, the *satisfiability problem*):

Definition 1.1 *An $\mathcal{F}(\Lambda)$-formula ϕ is* valid *if $x \models_C \phi$ for every state x in every T-coalgebra C.*

Examples of structures will be given below. The decision procedure for satisfiability described in [15] relies on a complete axiomatisation w.r.t. the above semantics in the shape of *one-step rules* ϕ/ψ, where ϕ is a propositional combination of variables $a \in V$ and ψ is a clause over atoms of the form $L(a_1, \ldots, a_n)$, $L \in \Lambda$, $a_i \in V$. This axiomatisation is moreover required to be closed under *rule resolution*, where a resolvent of two rules is formed by resolving their conclusions and conjoining their premises, and under *contraction*, i.e. removal of duplicate literals in rule conclusions arising after identification of two variables.

Given such an axiomatization by a set $\mathcal{R}$ of rules, denote by $\mathcal{R}_C$ the extension of $\mathcal{R}$ by replacement of equivalents. One then has the following algorithm on an alternating Turing machine:

Algorithm 1 *(Decide validity of $\phi \in \mathcal{F}(\Lambda)$)*

(i) *(Universal) Choose a clause θ from a conjunctive normal form (CNF) of ϕ.*

(ii) *(Existential) Guess a non-empty subclause ρ of θ.*

(iii) *(Existential) Guess $\chi/\psi \in \mathcal{R}_C$ such that $\rho \equiv \psi\sigma$ for some substitution σ.*

(iv) *(Universal) Choose a clause γ from a CNF of χ.*

(v) *Recursively check that $\gamma\sigma$ is valid.*

The algorithm succeeds if all possible choices at steps marked *universal* lead to successful termination, and for all steps marked *existential*, there exists a choice leading to successful termination.

Observe that in Step (iii), it suffices to take rules modulo propositional equivalence of premises. To ensure polynomial runtime of the algorithm, one needs to require that rules are coded in such a way that one needs to consider only rule codes of polynomially bounded size. Rule sets satisfying this condition and a few minor

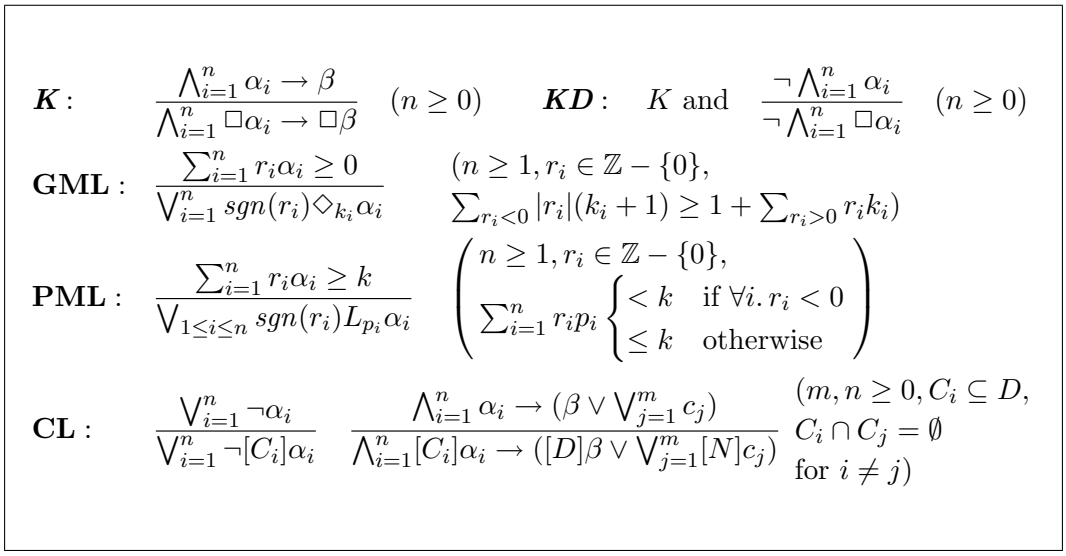


Fig. 1. Proof rules for the logics of Example 1.3

sanity requirements are called *PSPACE*-tractable. One obtains

Theorem 1.2 *If $\mathcal{M}$ is completely axiomatised by a set $\mathcal{R}$ of one-step rules which is PSPACE-tractable and closed under resolution and contraction, then Algorithm 1 decides validity of Λ-formulas over $\mathcal{M}$ in alternating polynomial time, i.e. in polynomial space.*

By virtue of completeness, validity coincides with provability. The hypothesis of the above theorem are in particular satisfied by the following logics:

Example 1.3 We give informal descriptions of some coalgebraic modal structures discussed more formally in [15]:

(i) The modal logic K is interpreted over the covariant powerset functor $\mathcal{P}$ by $[\![\Box]\!]_X A = \{B \in \mathcal{P}(X) \mid B \subseteq A\}$.

(ii) Similarly, the logic KD is interpreted over the non-empty powerset functor.

(iii) Graded modal logic (GML) with modal operators $\Diamond_k$ 'more than k successors satisfy ...' is interpreted over the finite multiset functor, whose coalgebras are multigraphs, i.e. graphs whose edges carry multiplicities.

(iv) Probabilistic modal logic (PML), with operators L_p 'with probability at least p, the successor satisfies ...', is interpreted over the finite distribution functor, whose coalgebras are finitely branching Markov chains.

(v) Coalition logic (CL) over a fixed finite set N of agents, with operators $[C]$ 'coalition $C \subseteq N$ can force ... in the next move', is interpreted over a class-valued functor whose coalgebras are game frames in the sense of [13].

Figure 1 shows rule sets for these logics satisfying the assumptions of Theorem 1.2. The arithmetic expressions in the rule premises for GML and PML refer to the (propositionally expressible) arithmetic of characteristic functions, and $sgn(r)\phi$ is ϕ if $r > 0$, and $\neg\phi$ otherwise.

2 Implementation of the Generic PSPACE Algorithm

CoLoSS is implemented in Haskell [14] and makes use of polymorphic types and type classes to separate the generic aspects of CoLoSS from the details of a particular logic. From this perspective, a particular logic instance is identified by a type which comprises a set of (unary) modal operators describing the syntax of the logic; the class interface to be implemented for each instance essentially consists of a matching function embodying a set of one-step rules that governs the deductive process.

We begin by describing how CoLoSS represents the syntax of different modal logics in a generic way. The syntax of each particular modal logic is represented using an algebraic data type that lists the modal operators provided by the logic. For K (and similarly for KD) we have

```
data K = K deriving (Eq,Show)
```

whereas the syntax of Pauly's coalition logic is given by

```
data C = C [Int] deriving (Eq, Show)
```

so that every modal operator of coalition logic is specified by a list of agents (the total number of agents is presently hardwired as a constant).

The syntax of modal logics as such then becomes a polymorphic type, where the type variable represents the modal operators of the logic, viz:

```
data L a =
   F | T | Atom Int | Neg (L a) | And (L a) (L a) | Or (L a) (L a) |
   M a (L a) deriving (Eq,Show)
```

The last line of the above definition is responsible for integrating the modalities of a specific language in that it allows constructing a formula by means of a modal operator (an inhabitant of the type variable `a`) and a formula of the language that is being defined.

The formulas of, say, coalition logic are then the inhabitants of the type `L C`, in which e.g. the formula $[0,2,4]p_0 \rightarrow [0]p_0$ (which expresses that if agent 0 can force p_0 with the help of agents 2 and 4, then she can force p_0 already herself) is represented as

```
(cbox [0, 2, 4] (p 0)) --> (cbox [0] (p 0))
```

with the help of an infix operator `-->`, the abbreviation `p` for `Atom`, and the modality

```
cbox :: [Int] -> L C -> L C; c ag phi = M (C ag) phi
```

that are added as syntactic sugar. Applying the CoLoSS functions `provable` and `satisfiable` confirms that the above formula is (of course) satisfiable but not provable, while the formula

```
(cbox [1 .. 3] (cbox [3 .. 5] (p 1) /\ cbox [1,2] (p 2))
    /\ cbox [4] (cbox [3] (p 2) /\ cbox [1,2] (p 1))) -->
         cbox [1 .. 5] (cbox [1 .. 5] ((p 1) /\ (p 2)))
```

is provable.

The logic-specific part of the generic provability (validity) checker is encapsulated in Steps (iii) and (iv) of Algorithm 1, which pass from the conclusions to

premises of one-step rules. It is this *matching* process that forms the core of the interface of the generic logic class in CoLoSS. In other words, every instance of the logic class in CoLoSS (embodied by the data type that encodes the modal operators) needs to provide a function that returns the set of rule matchings of a particular clause. Thus, the type class `Logic` is defined as follows:

```
class (Eq a) => Logic a where
  match :: Clause a -> [[L a]]
```

where inhabitants of the type `Clause a` consist of two lists that collect positive and negative literals, respectively, over atoms of the form $L(\phi_1, \ldots, \phi_n)$, where L is a modal operator. The result of `match c` is a list of rule premises such that `c` is provable iff one of the premises in the list is provable; each premise is given as a list of formulas representing the clauses of its CNF, with the substitution according to the matching with `c` already performed. (In a parallel implementation, the two logic specific steps of Algorithm 1 have been implemented as separate interface functions, and the substitution step has been handled generically; the relative merits of the two approaches are under investigation.)

Since the resolution closed rule set for K is

$$\frac{\bigwedge_{i=1}^{n} a_i \to b}{\bigwedge_{i=1}^{k} \Box a_i \to \Box b},$$

we have the following instance declaration for this logic:

```
instance Logic K where
  match (Clause (pl,nl)) =
    let (nls,pls) = (map neg (strip nl), strip pl)
    in map (\x -> [disj (x:nls)]) pls
```

where `strip` removes the outermost $\Box$ from every atom $\Box\phi$ in a list and `disj` turns a list of formulas into a disjunction.

That is to say that the set of matchings of a given clause $\chi = \bigvee_{i\in I} \neg\Box\phi_i \vee \bigvee_{j\in J} \Box\psi_j$ consists of all formulas $\bigwedge_{i\in I} \phi_i \to \psi_j$, $j \in J$. Note that – in this particular case – conjunctions over a subset $I' \subseteq I$ can be ignored as provability of the formula $\bigwedge_{i\in I'} \phi_i \to \psi_i$ implies that of $\bigwedge_{i\in I} \phi_i \to \psi_j$ as returned by the matching algorithm. Similarly, permutations of the literals in the conclusion can be ignored, as such permutations induce propositionally equivalent premises. More generally, permutations can be ignored when they lead to premises that arise also by matching with the original conclusion.

The axiomatisations of the other logics that are currently implemented in CoLoSS are represented similarly. We comment on two particular cases: In the rule set

$$\frac{\bigwedge_{i=1}^{n} a_i \to b \vee \bigvee_{j=1}^{m} c_j}{\bigwedge_{i=1}^{n} [C_i]a_i \to [D]b \vee \bigvee_{j=1}^{m} [N]c_j}$$

of coalition logic, we have to pay heed to the side condition that the C_i are pairwise disjoint subsets of D (recall that N represents the set of all agents). We can match against this condition by looking for cliques in the graph whose nodes are the C_i with edges from C_i to C_j if C_i and C_j are disjoint. (Note that the number of agents

is fixed, so that this does not add another layer of NP-hardness.)

For graded modal logic and probabilistic modal logic, we have to compute an upper bound on the coefficients that need to be taken into consideration to cover all rule matchings. A glance at the general format of the rule premise for these logics shows that it suffices to consider solutions which are pointwise maximal. It is conceivable that integer linear programming methods could be fruitfully applied to speed up the search for such solutions.

Given a function that computes rule matchings for a particular logic, the generic part of the algorithm that decides provability computes the conjunctive normal form of an input formula and recursively checks that every component of the conjunctive normal form has at least one provable matching, so that the heart of the algorithm takes the following form:

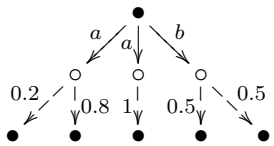
```
provable :: (Logic a) => L a -> Bool
provable phi = all (\c -> any (all provable) (match c)) (cnf phi)
```

A comment is in order as to whether the above function really uses polynomial space. On the surface it seems that overly large objects, such as the CNF of `phi` or the list of matchings for a given clause, are being passed around. However, Haskell is a non-strict language where compilers employ lazy evaluation; as a consequence, entries in large lists are computed on a by-need basis, and one can rely on built-in memory management for efficiency of space usage.

3 Modular Satisfiability Checking for Composite Logics

It has variously been observed that coalgebraic modal logic lends itself to modularisation in the sense that logics and structures can be composed in unison, preserving properties such as soundness and completeness [1]. An improved approach to logic composition in this sense that allows also for the modular treatment of algorithmic results such as the ones presented in Sect. 1 has been suggested in [16]. The crucial novelty is to interpret composite logics, whose formulas are naturally multi-sorted, over multi-sorted coalgebras. For purposes of the CoLoSS implementation, it suffices to understand the syntactic aspects of the logic combination mechanism, which we recall below.

As a motivating example, consider the modal logic for Segala systems [8]. Recall that in a (simple) Segala system [17,18], each system state can non-deterministically perform actions that lead to a probability distribution over its successor states:



The logic of [8] distinguishes non-deterministic formulas and probabilistic formulas, which are inductively given by

$$\mathcal{L}_n \ni \phi ::= \top \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \Box_a\psi \quad \text{(nondeterministic formulas; } \psi \in \mathcal{L}_p,\ a \in A\text{)}$$
$$\mathcal{L}_p \ni \psi ::= \top \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid L_q\phi \quad \text{(probabilistic formulas; } \phi \in \mathcal{L}_n,\ q \in [0,1] \cap \mathbb{Q}\text{)}.$$

One thus has a layered two-sorted logic, which one can regard as generated by a modal signature comprising sorts n and p (nondeterminism and probability) and two families of modal operators

$$\Diamond_a : p \to n \quad (a \in A) \qquad \text{and} \qquad L_q : n \to p \quad (q \in [0,1] \cap \mathbb{Q}).$$

This logic arises as a combination of two *features*, nondeterminism and uncertainty, in a way that can formalised as follows (see [16] for details):

An *n-ary feature* F comprises a multi-sorted modal signature, where modal operators have profiles $L : i_1, \dots i_k \to *$ made up of formal input sorts $1 \leq i_1, \dots, i_k \leq n$ and a formal target sort $*$. Features are additionally equipped with a coalgebraic semantics, consisting of an n-ary set functor $[\![\mathsf{F}]\!]$ and interpretations of the modal operators as suitable predicate liftings, and sets of one-step rules, which now take the form

$$\frac{\phi_1 : 1; \dots; \phi_n : n}{\psi},$$

where for $i = 1, \dots, n$, ϕ_i is a purely propositional formula over a set V_i of propositional variables and ψ is a clause over atoms of the form $L(a_1, \dots, a_k)$ with $L : i_1, \dots, i_k \to *$ in Λ and $a_j \in V_{i_j}$, $j = 1, \dots, k$. Premises ϕ_i omitted in the notation of concrete rules are, by default, equal to $\top$.

Example 3.1 The logics of Example 1.3 can be regarded as unary features K, KD, GML, PML, CL. Several typical constructions in logic combination can be cast as binary features as follows, with associated rule sets shown in Figure 2.

Choice: The binary feature Ch, interpreted over the disjoint binary sum functor, has a single modal operator $+ : 1, 2 \to *$, read as a case statement.

Fusion: The binary feature Fu, interpreted over the binary product functor, has two modal operators $[\pi_i] : i \to *$, $i = 1, 2$, making statements about the two components of a pair. The fusion of two logics with signatures Λ_1, Λ_2 is coded by translating $L\phi$ to $[\pi_i]L\phi$ for $L \in \Lambda_i$.

Conditionality: The binary feature Cnd, interpreted over the binary functor $\lambda X, Y.\, \mathcal{Q}(X) \to Y$ (with $\mathcal{Q}$ denoting contravariant powerset and $\to$ denoting function space) has a binary modal operator $\overset{\bullet}{\Rightarrow}: 1, 2 \to *$, read as a non-monotonic generalised conditional.

The construction of logics by combination of features is formalised by the notion of gluing. A gluing G is an $\mathcal{S}$-indexed family of terms over a set $\mathcal{S}$ of basic sort symbols formed using features as function symbols. Such a family of terms induces in the obvious way an endofunctor $[\![\mathsf{G}]\!]$ on $\mathbf{Set}^{\mathcal{S}}$, whose coalgebras serve as the semantic structures for the induced modal logic. The latter is itself multi-sorted, with one syntactic sort for each proper subexpression occurring in G (the full expression at a sort s itself being identified with the base sort s) and formulas generated by a rather obvious typing discipline, similarly for deduction. E.g. the logic in the motivating example above is fully described as the gluing

$$\mathsf{Seg} = (n \to \mathsf{HML}(\mathsf{PML}(n))),$$

$$\mathsf{Ch}: \quad \frac{(\bigwedge_{j=1}^{m} \alpha_j \to \bigvee_{k=1}^{n} \beta_k) : 1 \quad (\bigwedge_{j=1}^{m} \gamma_j \to \bigvee_{k=1}^{n} \delta_k) : 2}{\bigwedge_{j=1}^{m} (\alpha_j + \gamma_j) \to \bigvee_{k=1}^{n} (\beta_k + \delta_k)} \; (m, n \geq 0)$$

$$\mathsf{Fu}: \quad \frac{(\bigwedge_{j=1}^{m} \alpha_j \to \bigvee_{k=1}^{n} \beta_k) : i}{\bigwedge_{j=1}^{m} [\pi_i]\alpha_j \to \bigvee_{k=1}^{n} [\pi_i]\beta_k} \quad (i = 1, 2; m, n \geq 0)$$

$$\mathsf{Cnd}: \quad \frac{(\bigwedge_{j=1}^{m} \alpha_j \to \bigvee_{k=1}^{n} \beta_k) : 2}{\bigwedge_{j=1}^{m} (\gamma \stackrel{\bullet}{\Rightarrow} \alpha_j) \to \bigvee_{k=1}^{n} (\gamma \stackrel{\bullet}{\Rightarrow} \beta_k)} \quad (m, n \geq 0)$$

Fig. 2. Rule sets for the features of Example 3.1

where HML is the feature associated to Hennessy-Milner logic, i.e. the multi-agent version of K.

The main result of [16] states that gluings can be *flattened*, preserving satisfiability of formulas. Here, a gluing is called *flat* if each of its feature expressions contains exactly one feature. The flattening construction introduces a new sort for each proper subexpression occurring in a given gluing. E.g. the flattening of Seg is

$$(n \to \mathsf{HML}(p), p \to \mathsf{PML}(n)),$$

where p is a new sort representing the subexpression $\mathsf{PML}(n)$.

The advantage of using flat gluings is that the results of Sect. 1 generalise straightforwardly to flat gluings. Validity of formulas in a flat gluing (and hence, by the equivalence of a gluing with its flattening, in any gluing) is thus decidable by a modularised variant of Algorithm 1 which has a validity-checking routine for each sort; these routines call each other recursively (as in Step (v) of the algorithm) according to the typing discipline of the gluing that defines the logic.

4 Provability Checking for Combined Logics

To check provability for combined logics, CoLoSS synthesises a Haskell program that implements the particular combination of logical features at hand, based on existing implementations of the individual features. The main difficulty which requires resorting to this seemingly roundabout strategy is to generate mutually recursive datatypes of well-sorted formulas from a given gluing, a challenge which seems to overtax the expressive means of existing polytypic libraries and extensions of Haskell (cf. e.g. [10]). The only alternative to code synthesis that comes to mind is to use a deep encoding of the typing discipline (while the present approach just uses Haskell's type checking mechanisms). This is left for future versions of the tool that would incorporate a full logic definition language.

In CoLoSS, combined logics are described as gluings as discussed in the previous section. The flattening of a gluing is then constructed automatically (presently only for single-sorted gluings). At present, CoLoSS supports the features listed in Fig. 3. The syntax given in the figure is partly based on a syntactic sugaring of a datatype of single-sorted gluings whose constructors are the unique sort variable

Unary features

K, KD	The logics K and KD
CL	Coalition logic
GML	Graded modal logic
PML	Probabilistic modal logic
HML	Hennessy-Milner logic

Binary features

<+>	Binary Choice
<*>	Fusion

Auxiliary symbols

S	Formal sort variable
<.>	Feature application

Fig. 3. Features presently integrated in CoLoSS

S and unary and binary feature application: the infix operator <.> abbreviates application of unary features, and application of the two binary features presently implemented is sugared in the shape of the infix operators <+> and <*>, respectively, with precedence ordering <.>,<*>,<+> and <.> associating to the right. Thus, one has e.g. the following gluings

```
Seg = HML <.> PML <.> S          Alt = HML <.> S <+> PML <.> S
KKD = KD <.> S <*> K <.> S       PC  = CL <.> PML <.> S
```

with Seg corresponding to simple Segala systems as in the previous section and Alt representing the logic for *alternating automata* [16] where non-deterministic and probabilistic transitions can be interleaved arbitrarily [4]. The gluing KKD is the fusion of K and KD, and PC represents a logic of probabilistic coalitions that allows reasoning about coalitional strategies with a probabilistic outcome; we shall return to this logic at the end of this section. Flat gluings can be generated using the function flatten; e.g. flatten Seg yields the flat gluing

```
[FlatUnary HML 1, FlatUnary PML 0]
```

where non-negative integers figure as sort names. We read the above expression as the gluing

$$(s_0 \to \mathsf{HML}(s_1), s_1 \to \mathsf{PML}(s_0))$$

of the previous section.

Given a flat gluing G, the function call gen G then produces the provability checker for the logic associated with G in the file MySat.hs. This file defines in particular the syntax of the combined logic and contains copies of the matching routines of the component logics, rewired by adjusting their typing as appropriate to the structure of the gluing. This process is governed by multi-parameter type classes that embody the fact that the matching functions now connect potentially different logics. In general, one needs one such type class per arity; as the features appearing in typical applications tend to be at most binary, CoLoSS presently implements only two classes for unary and binary features, respectively, with the following interface declarations:

```
class (Logic f, Logic g) =>
  UnaryMatch f g | f -> g where
    matchu :: Clause f -> [[g]]
```

```
class (Logic f, Logic g, Logic h) =>
  BinaryMatch f g h | f -> g, f -> h where
    matchb :: Clause f -> [([g],[h])]
```

Here, we use one of the Glasgow Haskell extensions, namely *functional dependencies* [7] `f -> g` etc., which indicate that there is at most one instance for a given source logic `f`. This allows in particular the use of class constraints such as `UnaryMatch f g` in polymorphic functions whose types do not mention `g`. We use this feature to implement polymorphic provability functions according to the arity of the feature at hand, e.g. for binary features

```
provable2 :: BinaryMatch f g h => f -> Bool
provable2 phi =
  let both (as,bs) = (all provable as) && (all provable bs)
  in all (\c -> any (\pair -> both pair) (matchb c))(cnf phi)
```

These arity-specific implementations of the provability function are associated with a single provability function `provable` in the `Logic` class interface, on which recursive calls are made as illustrated above without having to keep further record of arities. Note the exploitation of Haskell type inference to ensure the correct wiring of the respective provability functions.

As logical features can be used more than once in a flat gluing, CoLoSS uses running indices to disambiguate potential duplicate ocurrences of the same feature. For the case of Segala systems, CoLoSS generates the following language definition:

```
data L0 = (propositional connectives) | HML0 Char L1
data L1 = (propositional connectives) |  PML1 Rational L0
```

(omitting the `deriving` clauses). Here, the modal operator $\Box_a$ of Hennessy-Milner logic is represented as `HML0 'a'` and the operator L_p of probabilistic modal logic is written `PML1 p`, with the numbers (0 and 1) being the result of the disambiguation process.

As laid out in Sect. 3, a formula is provable iff every component of its conjunctive normal form allows for at least one matching that is itself provable. In the case of binary features, this necessitates that matching produces a pair of formulas, both of which then have to be checked for provability as shown above. This is reflected e.g. in the matching function for the (binary) choice feature in the gluing `Alt` for alternating systems above:

```
instance BinaryMatch L0 L1 L2 where
  matchb (Clause (pls,nls)) =
    let f1 = (disj $ striplst pls) \/ (ndisj $ striplst nls)
        f2 = (disj $ striplst pls) \/ (ndisj $ striplst nls)
    in [([f1],[f2])]
```

where `ndisj` produces the disjunction of the negations of all formulas contained in a given list. Note that the type of `matchb` above is:

```
matchb :: Clause L0 -> [([L1],[L2])]
```

In the logic of simple Segala systems defined by the gluing `Seg`, the formula

$$\Box_i(L_{0.8}\Diamond_i p_0 \to L_{0.5}\Diamond_i p_0)$$

can now be written as

```
box 'i' ((l (8%10) (dia 'i' (p1 0))) --> (l (5%10) (dia 'i' (p1 0))))
```

(with syntactic sugar `dia c phi = neg (HML0 c (neg phi))` and `box c phi = HML0 c phi` for HML box and diamond, `l = PML1` for the probabilistic L_p operator, and `p1` for propositional variables). CoLoSS correctly finds that this (admittedly simple) formula is provable. As a second example, we consider the logic of probabilistic coalitions where coalitions of agents can only force probability distributions over facts. This is expressed by the gluing `PC` (for proabilistic coalitions) above. We obtain a two-sorted language similar to the logic of Segala systems

$$\begin{aligned}
&\mathcal{L}_c \ni \phi ::= \top \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid [C]\psi \quad \text{(coalition formulas; } \psi \in \mathcal{L}_p,\ C \subseteq N\text{)}\\
&\mathcal{L}_p \ni \psi ::= \top \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid L_q\phi \quad \text{(probabilistic formulas; } \phi \in \mathcal{L}_c,\ q \in [0,1] \cap \mathbb{Q}\text{)}
\end{aligned}$$

where N is the set of all agents as in Section 1. Using syntactic sugar for the modal operators and propositional varaiables, we can e.g. check the provability or satisfiability of the formula

```
phi = cbox [0, 2, 4] (l 0.5 p0) /\ cbox [1, 3, 5] (l 0.5 p1)
      --> cbox [0, 1, 2, 3, 4, 5] (l 0.25 (p0 /\ p1))
```

that asserts that two disjoint coalitions can join forces to yield the combination of their individual returns with the product of the respective probabilities. However, as (the denotations of) the propositional variables p_0 and p_1 are in general not independent events, this formula is not provable (although of course satisfiable). However, if we specify that, under the given lower bounds for the respective probabilities of p_0 and p_1, the conjunction of p_0 and p_1 holds with probability at least 0.4 in every subsequent state of the strategic game by means of the formula

```
psi = cbox [] ((l 0.5 p0) /\ (l 0.5 p1) --> (l 0.4 (p0 /\ p1)))
```

then the formula `phi --> psi` is indeed provable.

5 Conclusion

We have laid out the design and implementation of the Coalgebraic Logic Satisfiability Solver CoLoSS, which provides a generic and extensible framework for satisfiability/validity checking in a wide variety of modal logics. Crucial implementation details include the use of the Haskell class mechanism to support genericity over individual logics, and a code generation mechanism that facilitates the modular combination of logics. The main purpose of CoLoSS at its present stage of development is to provide a proof of concept. The state of the tool is largely experimental, and alternative options are being explored in terms of the overall design, in particular concerning the handling of logic combination and the class interface for modal logics. There has been some parallel development, and the two branches are in the process of being fully merged; in particular, a generic modal logic parser

is already available and will be integrated with the tool shortly, thus supplanting the present style of entering formulas directly as Haskell code. (Note however that representing formulas directly as code is a time-honored practice; e.g. the default textual interface of RACER [3] expects formulas as LISP code.)

A major topic for future development is to increase the efficiency both of the generic parts of the tool and of the specific logic instances. Generic optimizations will mainly concern propositional aspects; one may either move to proper propositional tableaux and employ heuristic optimization strategies such as the ones described in [6], or represent propositional layers in formulas efficiently as binary decision diagrams. Logic-specific heuristics to be explored include the use of (integer) linear programming tools to reduce the search space for graded and probabilistic modal logics. Once such optimization strategies are in place, benchmarking will become an issue. Predefined benchmark suites that would allow evaluating the performance of the tool in its main intended application area, namely non-Kripke modal logics, seem to be hard to come by. The design of such benchmarks is therefore a further important topic of future research.

References

[1] Cîrstea, C. and D. Pattinson, *Modular construction of modal logics*, Theoret. Comput. Sci. To appear. Earlier version in P. Gardner, N. Yoshida, editors, *Concurrency Theory, CONCUR 04*, vol. 3170 of *Lect. Notes Comput. Sci.*, pp. 258–275, Springer, 2004.

[2] Demri, S. and D. Lugiez, *Presburger modal logic is only PSPACE-complete*, in: U. Furbach and N. Shankar, editors, *Automated Reasoning, IJCAR 06*, Lect. Notes Artificial Intelligence **4130** (2006), pp. 541–556.

[3] Haarslev, V. and R. Mller, *RACER system description*, in: R. Goré, A. Leitsch and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR 2001*, Lect. Notes Comput. Sci. **2083** (2001), pp. 701–705.

[4] Hansson, H. and B. Jonsson, *A calculus for communicating systems with time and probabilities*, in: *Real-Time Systems, RTSS 90* (1990), pp. 278–287.

[5] Heifetz, A. and P. Mongin, *Probabilistic logic for type spaces*, Games and Economic Behavior **35** (2001), pp. 31–53.

[6] Horrocks, I. and P. F. Patel-Schneider, *Optimising description logic subsumption*, J. Logic Comput. **9** (1999), pp. 267–293.

[7] Jones, M. P., *Type classes with functional dependencies*, in: G. Smolka, editor, *European Symposium on Programming, ESOP 2000*, Lect. Notes Comput. Sci. **1782** (2000), pp. 230–244.

[8] Jonsson, B., W. Yi and K. G. Larsen, *Probabilistic extensions of process algebras*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001 pp. 685–710.

[9] Larsen, K. and A. Skou, *Bisimulation through probabilistic testing*, Inform. Comput. **94** (1991), pp. 1–28.

[10] Norell, U. and P. Jansson, *Polytypic programming in haskell*, in: P. W. Trinder, G. Michaelson and R. Pena, editors, *Implementation of Functional Languages, IFL 2003*, Lect. Notes Comput. Sci. **3145** (2003), pp. 168–184.

[11] Olivetti, N. and G. L. Pozzato, *CondLean: A theorem prover for conditional logics*, in: M. C. Mayer and F. Pirri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2003*, Lect. Notes Comput. Sci. **2796** (2003), pp. 264–270.

[12] Pacuit, E. and S. Salame, *Majority logic*, in: D. Dubois, C. Welty and M.-A. Williams, editors, *Principles of Knowledge Representation and Reasoning, KR 04* (2004), pp. 598–605.

[13] Pauly, M., *A modal logic for coalitional power in games*, J. Logic Comput. **12** (2002), pp. 149–166.

[14] Peyton-Jones, S., editor, "Haskell 98 Language and Libraries — The Revised Report," Cambridge, 2003, also: J. Funct. Programming **13** (2003).

[15] Schröder, L. and D. Pattinson, *PSPACE reasoning for rank-1 modal logics*, in: R. Alur, editor, *Logic in Computer Science, LICS 06*, IEEE, 2006, pp. 231–240, extended version available on CoRR, arxiv.org/abs/0706.4044.

[16] Schröder, L. and D. Pattinson, *Modular algorithms for heterogeneous modal logics*, in: L. Arge, A. Tarlecki and C. Cachin, editors, *Automata, Languages and Programming, ICALP 07*, Lect. Notes Comput. Sci. **4596** (2007), pp. 459–471.

[17] Segala, R., "Modelling and Verification of Randomized Distributed Real-Time Systems," Ph.D. thesis, Massachusetts Institute of Technology (1995).

[18] Segala, R. and N. Lynch, *Probabilistic simulations for probabilistic processes*, Nordic Journal of Computing **2** (1995), pp. 250–273.

[19] Sirin, E., B. Parsia, B. C. Grau, A. Kalyanpur and Y. Katz, *Pellet: A practical OWL-DL reasoner*, J. Web Semantics. To appear.

[20] Tsarkov, D. and I. Horrocks, *FaCT++ description logic reasoner: System description*, in: U. Furbach and N. Shankar, editors, *Int. Joint Conf. on Automated Reasoning, IJCAR 2006*, Lecture Notes in Artificial Intelligence **4130** (2006), pp. 292–297.