Project Report - Beta

N. Cohen, T. Coroneo, V. Iyer, G. Kaminer, and L. Madison

Thursday 27th March, 2025

Abstract

We implement an explicit model checker for bilateral state-based modal logic (BSML), as described in [AAY24]. BSML has been used to account for Free-Choice (*FC*) and related inferences which arise from speakers' distaste for interpretations that verify a sentence by empty configuration (*neglect-zero tendency*).

Free-Choice (*FC*) inferences are instances of conjunctive meaning being unexpectedly derived from a disjunctive sentence. In the following example, a modalized disjunction yields a conjunction of modals.

- You may go to the beach or to the cinema $\rightsquigarrow$ You may go to the beach and you may go to the cinema
- $\Diamond(b \vee c) \rightsquigarrow \Diamond b \wedge \Diamond c$

Aloni (2022) explain that in interpreting a sentence, a speaker identifies which are the structures of reality that would reflect the sentence (the models in which the sentence would be assertable). In our disjunction, there are four associated worlds:

1. W_{bc} in which both b and c are true (you go to both the beach and the cinema)
2. W_b in which b is true (you go to the beach)
3. W_c in which c is true (you go to the cinema)
4. W_z in which neither b nor c is true (you don't go anywhere)

We can use BSML to model the information states (i.e. sets of worlds) in which the disjunction $b \vee c$ would be assertable. A proposition is assertable at a state s if it is assertable in all its substates. A disjunction is assertable in a state s if each of the disjuncts is supported in a substate of s.

1. In a state consisting of W_b and W_c, $b \vee c$ is assertable
2. In a state consisting of W_{bc} and W_b (or a state consisting of W_{bc} and W_c), $b \vee c$ is assertable
3. In a state consisting of W_b (or a state consisting of W_c), $b \vee c$ is assertable since each of the disjuncts is supported in a substate. b is supported in W_b, and c is supported in the empty state.

4. In a state consisting of W_z and W_b (and any other state which includes W_z), $b \vee c$ is *not* assertable because in W_z both b and c are false, so this state would leave open the possibility that neither b nor c is the case.

So among the four types of states, $b \vee c$ is assertable in all but the last. This is a problem, because if $b \vee c$ is assertable In a state consisting of only W_b (or a state consisting of only W_c) then we have that $\Diamond(b \vee c)$ is true while $\Diamond b \wedge \Diamond c$ is false, so the FC inference fails. The problematic state, then, is the zero-model: one of the states which it uses to satisfy the disjunction is the empty state.

How do we account for FC inferences then? Aloni argues pragmatically that a speaker would not consider the zero-model as one of the candidate states. Neglecting the zero model then, the FC inference would hold because the only states that would support $b \vee c$ would be (1) or (2). To model neglect-zero (to make sure that $b \vee c$ is not assertable in the zero-model), we require that to satisfy a disjunction, the state must be the union of two non-empty substates rather than just the union of two substates. So we enrich each formula with a $[]^+$ function which conjuncts the non-emptiness atom (NE) to each formula and all its subformulas.

After enrichment, the disjunction $b \vee c$ is no longer assertable in a state consisting of only W_b (or a state consisting of only W_c) since the non-emptiness atom (NE) would not be supported in all the substates. Finally then, if the only states in which the disjunction holds are (1) and (2), then the FC inference holds.

Contents

1 How to use this?

To generate the PDF, open `report.tex` in your favorite LaTeXeditor and compile. Alternatively, you can manually do `pdflatex report; bibtex report; pdflatex report; pdflatex report` in a terminal.

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with your code: `stack ghci`
- To run the executable from Section ??: `stack build && stack exec myprogram`
- To run the tests from Section 5: `stack clean && stack test --coverage`

2 Bilateral State-Based Modal Logic

This section describes the basic definitions for the explicit model checker for Bilateral State-Based Modal Logic (henceforth BSML). We begin by importing modules necessary for this. Unlike previous model checkers we have seen, which use (association) lists, we utilise maps and sets in our models. We do this to prepare for the eventuality of using IntSets and IntMaps - which are a much more efficient structure for storing and retrieving integers than lists.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}
module Defs where

-- Potential TODO: Change Set to IntSet (and Map to IntMap) for performance.
import Data.Set (Set)
import qualified Data.Set as Set

import Data.Map (Map)
import qualified Data.Map as Map

import Test.QuickCheck
import Text.Parsec
import Text.Parsec.String
import qualified Text.Parsec.Expr as Expr
import Data.Char
import Data.Functor.Identity (Identity)
import Language.Haskell.TH (parensE)
```

We establish the data type `Form`, which we use to describe BSML formulae. We later establish the data type `MForm` to describe modal formulae in the basic modal language.

```
type Proposition = Int
type World = Int

-- BSML formulas
data Form
  = Bot
  | NE
  | Prop Proposition
  | Neg Form
  | And Form Form
  | Or  Form Form
```

```
  | Dia Form
  deriving (Eq,Show)
```

BSML relies on team semantics, and a team is a set of worlds. A model in this logic is a standard Kripke model, consisting of a set of worlds, a relation on the set of worlds and a valuation function; which tells us which propositions are true at a given world. We store the relation as a Map from Worlds to sets of Worlds. This gives us easy access to the successors of any given world, with cheap lookup thanks to using sets.

```
type Team = Set World
type Rel = Map World (Set World)
type Val = Map World (Set Proposition)

data KrM = KrM {worlds :: Set World,
                rel    :: Rel,
                val    :: Val}
  deriving (Show)

data TeamPointedModel = TPM KrM Team
  deriving (Show)

data WorldPointedModel = WPM KrM World
  deriving (Show)
```

We define below `rel'` and `val'`; two functions allow us to treat the Map-valued fields `rel` and `val` as (partial) functions.

```
rel' :: KrM -> World -> Set World
rel' = (Map.!) . rel

val' :: KrM -> World -> Set Proposition
val' = (Map.!) . val
```

Finally, we describe a function that gives us the successors of all worlds in a given team.

```
teamRel :: KrM -> Team -> Set World
teamRel m s = Set.unions $ Set.map (rel m Map.!) s
```

We define now notions of support and antisupport for formulae with respect to a model and a team. Supportability's closest analogue in more familiar logics is $\models$, although the definition varies slightly since we now have a new operator (NE or non-empty) to contend with. Antisupportability is defined analogously to negation as will be evident below.

We also define classes `Supportable` and `Antisupportable`, and present two alternate definitions of the support (and dually, the antisupport) function. The minimal definition for the class only requires one of these to be provided; the other is the curried/uncurried equivalent.

```
class Supportable m s f where
  support :: m -> s -> f -> Bool
  support = curry (|=)

  (|=) :: (m, s) -> f -> Bool
  (|=) = uncurry support

class Antisupportable m s f where
  antisupport :: m -> s -> f -> Bool
  antisupport = curry (=|)

  (=|) :: (m, s) -> f -> Bool
  (=|) = uncurry antisupport
```

We define now the semantics for BSML. For more detail, the reader may refer to page 5 of [AAY24]. The support-relation should look familiar for any readed well-versed in state-based modal logics, with the addition of the NE-atom and using the dual antisupport-relation to model negation (i.e. refutation of a formula).

Defining the semantics of $\vee$ for `support` and $\wedge$ for `antisupport` required us to use a helper function - `teamParts`. This function computes set of all pairs of subsets whose union is a given team s.

```
teamParts :: Team -> Set (Team, Team)
teamParts s = Set.fromList $ do
    s' <- ps
    s'' <- Set.toList $ Set.powerSet s'
    let augmentedCompl = Set.difference s s''
    return (s' :: Team, augmentedCompl)
  where ps = Set.toList $ Set.powerSet s

instance Supportable KrM Team Form where
  (_,s) |= Bot     = null s
  (_,s) |= NE      = not (null s)
  (m,s) |= Prop n  = all (elem n) $ Set.map (val' m) s
  (m,s) |= Neg f   = (m,s) =| f
  (m,s) |= And f g = (m,s) |= f && (m,s) |= g
  (m,s) |= Or f g  = any (\(t,u) -> t <> u == s && (m,t) |= f && (m,u) |= g) $ teamParts s
  (m,s) |= Dia f   = all (any (\t -> not (null t) && (m,t) |= f) . Set.powerSet . rel' m) s

instance Antisupportable KrM Team Form where
  _     =| Bot     = True
  (_,s) =| NE      = null s
  (m,s) =| Prop n  = not . any (elem n) $ Set.map (val' m) s
  (m,s) =| Neg f   = (m,s) |= f
  (m,s) =| And f g = any (\(t,u) -> t <> u == s && (m,t) =| f && (m,u) =| g) $ teamParts s
  (m,s) =| Or f g  = (m,s) =| f && (m,s) =| g
  (m,s) =| Dia f   = all (\w -> (m, rel' m w) =| f) s
```

One may also easily extend the above semantics to lists of formulae, as shown below.

```
instance Supportable KrM Team [Form] where
  support = (all .) . support

instance Antisupportable KrM Team [Form] where
  antisupport = (all .) . antisupport
```

We write first a function that describes $\Box$ formulae - much like in several standard modal logics, $\Box\varphi \equiv \neg\Diamond\neg\varphi$ for any formula φ.

We note that $\bot$ as defined here is only false on all non-empty teams. It is therefore referred to in [AAY24] as the *weak contradiction*. The strong contradiction (referred to in [AAY24] as $\perp\!\!\!\perp$) is the formula $\bot \wedge \mathtt{NE}$, which is called `botbot` below. Dually, the formula `NE` serves as the *weak tautology* here; it is true on all non-empty teams. The formula $\top\!\!\!\top := \neg\bot$ is true everywhere, and is therefore called the *strong tautology*.

```
box :: Form -> Form
box = Neg . Dia . Neg

botbot :: Form
botbot = And Bot NE

top :: Form
top = NE

toptop :: Form
toptop = Neg Bot
```

We may use the above to interpret empty disjunctions and conjunctions as below:

```
bigor :: [Form] -> Form
bigor [] = Bot
bigor fs = foldr1 Or fs

bigand :: [Form] -> Form
bigand [] = toptop
bigand fs = foldr1 And fs
```

The following code block implements the Arbitrary typeclass for models (KrM), pointed models (TeamPointedModel or WorldPointedModel), and formulas (both BML formulas in MForm, and BSML formulas in Form).

We start by defining some parameters that will be used in the generators.

```
-- The proposition are picked in the range (1, numProps) for MForm and Form.
-- We do not use the size parameter because it would introduce a bias in the occurence of
-- Propositions, where e.g. more nested subformulas will not contain high propositions.
numProps :: Int
numProps = 32
```

The instance for a Kripke model KrM first generates an arbitrary set of worlds ws, from 0 to an arbitrary k; then, it generates an arbitrary Map world to subset of all worlds to represent the model relation. Finally, it generates the valuation Map by picking an arbitrary list of propositions from the range (0, numProps).

```
subsetOf :: Ord a => [a] -> Gen (Set a)
subsetOf = (Set.fromList <$>) . sublistOf

instance Arbitrary KrM where
  arbitrary = sized (\n -> do
    k <- choose (0, n)
    let ws = Set.fromList [0..k]
    r <- Map.fromList . zip [0..k] <$> vectorOf (k+1) (subsetOf [0..k])
    v <- Map.fromList . zip [0..k] <$> vectorOf (k+1) (subsetOf [1..numProps])
    return $ KrM ws r v)
```

In the instances for pointed models, we want to make sure that the team or world that we're focusing on is actually part of the model. We do that by picking a team or a world respectively as an arbitrary subset or element of the model's worlds.

```
instance Arbitrary TeamPointedModel where
  arbitrary = do
    m <- arbitrary
    s <- subsetOf $ Set.toList $ worlds m
    return $ TPM m s

instance Arbitrary WorldPointedModel where
  arbitrary = do
    m <- arbitrary
    w <- elements (Set.toList $ worlds m)
    return $ WPM m w
```

For the Arbitrary instance of the formulas, we check the size parameter: if it is 0, we choose an atom to terminate the recursion. Otherwise, we pick one of the other available operators that we can use in an arbitrary formula.

```
randomMProp :: Gen MForm
randomMProp = MProp <$> choose (1, numProps)

instance Arbitrary MForm where
  arbitrary = sized $ \case
    0 -> randomMProp
    _ -> oneof [
```

```
      randomMProp,
      MNeg <$> f,
      MAnd <$> f <*> f,
      MOr  <$> f <*> f,
      MDia <$> f]
    where f = scale (`div` 2) arbitrary

randomAtom :: Gen Form
randomAtom = oneof [Prop <$> choose (1, numProps), pure NE, pure Bot]

instance Arbitrary Form where
  arbitrary = sized $ \case
    0 -> randomAtom
    _ ->  oneof [
        randomAtom,
        Neg <$> f,
        And <$> f <*> f,
        Or  <$> f <*> f,
        Dia <$> f
      ]
    where f = scale (`div` 2) arbitrary
```

Some example models.

```
-- Aloni2024 - Figure 3.
w0, wp, wq, wpq :: Int
wp  = 0
wq  = 1
wpq = 2
w0  = 3

u3 :: Set World
u3 = Set.fromList [0..3]

r3a, r3b, r3c :: Map World (Set World)
r3a = Map.fromSet (const Set.empty) u3

r3b = Map.fromSet r u3 where
  r 2 = Set.fromList [wp, wpq]
  r 3 = Set.singleton wq
  r _ = Set.empty

r3c = Map.fromSet r u3 where
  r 2 = Set.fromList [wp, wq]
  r _ = Set.empty

v3 :: Map World (Set Proposition)
v3 = Map.fromList [
    (0, Set.singleton 1),
    (1, Set.singleton 2),
    (2, Set.fromList [1,2]),
    (3, Set.empty)
  ]

m3a, m3b, m3c :: KrM
m3a = KrM u3 r3a v3
m3b = KrM u3 r3b v3
m3c = KrM u3 r3c v3

s3a1, s3a2, s3b, s3c :: Team
s3a1 = Set.singleton wq
s3a2 = Set.fromList [wp, wq]
s3b  = Set.fromList [wpq, w0]
s3c  = Set.singleton wpq

-- Lara NarrowScope True
wNSa, wNSb, wNS :: Int
wNSa  = 0
wNSb  = 1
wNS = 2

uNS :: Set World
```

```
uNS = Set.fromList [0..2]

rNS :: Map World (Set World)
rNS = Map.fromSet r uNS where
  r 2 = Set.fromList [wNSa, wNSb]
  r _ = Set.empty

vNS :: Map World (Set Proposition)
vNS = Map.fromList [
    (0, Set.singleton 1),
    (1, Set.singleton 2),
    (2, Set.empty)
  ]

mNS :: KrM
mNS = KrM uNS rNS vNS

sNS :: Team
sNS  = Set.singleton wNS

-- Lara NarrowScope False
wNSF1, wNSF2 :: Int
wNSF1 = 2
wNSF2 = 3

uNSF :: Set World
uNSF = Set.fromList [0..3]

rNSF :: Map World (Set World)
rNSF = Map.fromSet r uNSF where
  r 2 = Set.singleton wNSa
  r 3 = Set.singleton wNSb
  r _ = Set.empty

vNSF :: Map World (Set Proposition)
vNSF = Map.fromList [
    (0, Set.singleton 1),
    (1, Set.singleton 2),
    (2, Set.empty),
    (3, Set.empty)
  ]

mNSF :: KrM
mNSF = KrM uNSF rNSF vNSF

sNSF :: Team
sNSF  = Set.fromList [wNSF1, wNSF2]

-- Lara TautologyBoxDEF
wBa, wBb, wBc, wBd, wBbc, wBe, wBcd :: Int
wBa = 0
wBb = 1
wBc = 2
wBd = 3
wBbc = 4
wBe = 5
wBcd = 6

uB :: Set World
uB = Set.fromList [0..6]

rB :: Map World (Set World)
rB = Map.fromSet r uB where
  r 0 = Set.singleton wBd
  r 1 = Set.fromList [wBe, wBd, wBc]
  r 2 = Set.singleton wBc
  r 4 = Set.singleton wBcd
  r _ = Set.empty

vB :: Map World (Set Proposition)
vB = Map.fromList [
    (0, Set.singleton 1),
    (1, Set.singleton 2),
```

```
    (2, Set.singleton 3),
    (3, Set.singleton 4),
    (4, Set.fromList [2, 3]),
    (5, Set.singleton 5),
    (6, Set.fromList [3, 4])
  ]


mB :: KrM
mB = KrM uB rB vB

sB :: Team
sB  = Set.fromList [wBa, wBb, wBc]
```

```
-- Basic Modal Logic formulas
data MForm
  = MProp Proposition
  | MNeg MForm
  | MAnd MForm MForm
  | MOr  MForm MForm
  | MDia MForm
  deriving (Eq,Show)

instance Supportable KrM World MForm where
  (m,w) |= MProp n  = n `elem` val' m w
  (m,w) |= MNeg f   = not $ (m,w) |= f
  (m,w) |= MAnd f g = (m,w) |= f && (m,w) |= g
  (m,w) |= MOr f g  = (m,w) |= f || (m,w) |= g
  (m,w) |= MDia f   = any (\v -> (m,v) |= f) $ rel' m w


-- Modal formulas are a subset of BSML-formulas
toBSML :: MForm -> Form
toBSML (MProp n)  = Prop n
toBSML (MNeg f)   = Neg (toBSML f)
toBSML (MAnd f g) = And (toBSML f) (toBSML g)
toBSML (MOr f g)  = Or (toBSML f) (toBSML g)
toBSML (MDia f)   = Dia (toBSML f)

-- In Aloni2024, this is indicated by []+
enrich :: MForm -> Form
enrich (MProp n)  = Prop n `And` NE
enrich (MNeg f)   = Neg (enrich f) `And` NE
enrich (MDia f)   = Dia (enrich f) `And` NE
enrich (MAnd f g) = (enrich f `And` enrich g) `And` NE
enrich (MOr f g)  = (enrich f `Or`  enrich g) `And` NE
```

Implementation of a parser with Parsec.

```
-- from: https://jakewheat.github.io/intro_to_parsing

formParser :: Parser Form
formParser = Expr.buildExpressionParser parseTable parseTerm

parseTable :: Expr.OperatorTable String () Identity Form
parseTable = [
  [Expr.Prefix (Neg <$ symbol "~"),
    Expr.Prefix (Dia <$ symbol "<>"),
    Expr.Prefix (Neg . Dia . Neg <$ symbol "[]")
  ],
  [
    Expr.Infix (And <$ symbol "&") Expr.AssocLeft,
    Expr.Infix (Or <$ symbol "V") Expr.AssocLeft
  ]]

parseTerm :: Parser Form
parseTerm = parens <|> parseProp <|> parseBottom <|> parseNE

parseProp :: Parser Form
parseProp = Prop <$> integer
```

```
parseBottom :: Parser Form
parseBottom = Bot <$ symbol "BOT"

parseNE :: Parser Form
parseNE = NE <$ symbol "NE"

symbol :: String -> Parser String
symbol s = string s <* spaces

integer :: Parser Int
integer = read <$> many1 digit <* spaces

parens :: Parser Form
parens = between (symbol "(") (symbol ")") formParser
```

3 Lexer

Below we describe a brief code for a lexer we built for BSML formulae. The lexer was built to utilise the alex lexing tool mentioned in class.

We note that this code is heavily based on the code of Katejan Dvoracek, who built a Natural Deduction Prover, and used within it a lexer running on Alex, and a parser running on Happy. You can find the PDF for the project here. Thanks a bunch Katejan!

```
{
module Lexer where
import Token
}


%wrapper "posn"
$dig = 0 -9 -- digits
tokens:-
-- ignore whitespace and comments :
$white + ;
" --" .* ;
-- keywords and punctuation :
"(" { \ p _ -> TokenOB p }
")" { \ p _ -> TokenCB p }
"_|_" { \ p _ -> TokenBot p }
"NE" {\ p _ -> TokenNE p}
"[]" { \ p _ -> TokenBox p }
"<>" { \ p _ -> TokenDmd p }
"~" { \ p _ -> TokenNot p }
"&" { \ p _ -> TokenCon p }
"v" { \ p _ -> TokenDis p }
-- Integers and Strings :
$dig + { \ p s -> TokenInt ( read s ) p }
[P - Z ] { \ p s -> TokenPrp ( ord ( head s ) - ord 'P') p }
[p - z ] { \ p s -> TokenPrp ( ord ( head s ) - ord 'p' + 11) p }
```

The basic overview of the file above is as follows: We instruct Alex to construct a lexer that goes through a given string and picks out "tokens" from it. These tokens are symbols to be kept in mind while the string is being assigned meaning - which is done using a parser built by Happy.

This particular file has a .x extension - once written, the user may run alex Lexer.x in their terminal. This command generates Lexer.hs, a fully functioning lexer for the above tokens built by Alex. The code is housed inside a module called Lexer, which we import within the parser file.

4 Parser

We now take a look at the code for the parser, which works using Happy. As with the lexer, the code is inspired by Katejan Dvoracek's work for a lexer and parser for Natural Deductions.

We first import the modules that we require to run the parser.

```
> {
> module Parser where
> import Data.Char
> import Token
> import Lexer
> import Defs
> }
```

Below, we describe the tokens that Happy needs to keep track of while reading the string. These tokens were described in the module Lexer.

```

> %name formula
> %tokentype { Token AlexPosn }
> %monad { ParseResult }
>
> %token
> BOT { TokenBot _ }
> NE {TokenNE _}
> AND { TokenCon _ }
> OR { TokenDis _ }
> NOT { TokenNot _ }
> BOX { TokenBox _ }
> DMD { TokenDmd _ }
> NUM { TokenInt $$ _ }
> '(' { TokenOB _ }
> ')' { TokenCB _}
```

We describe also the binding hierarchy for binary operators in our language. The order of precedence is described by listing operators from weakest to strongest, as evidenced below. Note that the binding for all unary operators is stronger than the binding for binary operators, and unary operators operate at the same binding strength. This behaviour keeps in line with the way we would like to parse formulae in our language.

```
> %left OR
> %left AND
>
> %%
```

We now detail two different types of formulae - bracketed formulae and non-bracketed formulae. The reasoning for the distinction is rather simple - the presence of parentheses around a formula requires that any operations within the parentheses need to be given priority over operations outside of them. This may break regular precedence rules, and hence need to be accounted for.

```
> Form :: { Form }
> Form : BrForm { $1 }
> | Form AND Form { And $1 $3 }
> | Form OR Form { Or $1 $3 }
> BrForm :: { Form }
> BrForm : NUM { Prop $1 }
> | BOT { Bot }
> | NE { NE }
> | NOT BrForm { Neg $2 }
> | BOX BrForm { Defs.box $2 }
> | DMD BrForm { Dia $2 }
> | '(' Form AND Form ')' { And $2 $4 }
> | '(' Form OR Form ')' { Or $2 $4 }
```

Next, we outline error messages for our parser. These error messages (very helpfully and wonderfully described by Katejan) describe where the error occurs exactly in the string, and why Happy failed to parse it.

```
> {
> data ParseError = ParseError { pe_str :: String
>                              ,pe_msg :: String
>                              ,pe_col :: Int}
>      deriving (Eq , Show)
>
> type ParseResult a = Either ParseError a
>
>
> happyError :: [ Token AlexPosn ] -> ParseResult a
> happyError [] = Left $
>      ParseError { pe_str = " " , pe_msg = " Unexpected end of input : " , pe_col = -1}
> happyError ( t : ts ) = Left $
>      ParseError { pe_str = " " , pe_msg = " Parse error : " , pe_col = col }
>      where ( AlexPn abs lin col ) = apn t
>
> myAlexScan :: String -> ParseResult [ Token AlexPosn ]
> myAlexScan str = go ( alexStartPos , '\n' ,[] , str )
>   where
>       go :: AlexInput -> ParseResult [ Token AlexPosn ]
>       go inp@( pos ,_ ,_ , str ) =
>           case alexScan inp 0 of
>               AlexEOF -> Right []
>               AlexError (( AlexPn _ _ column ) ,_ ,_ , _ ) -> Left $
>                   ParseError { pe_str = str , pe_msg = " Lexical error : " , pe_col =
    column -1}
>               AlexSkip inp' len -> go inp'
>               AlexToken inp' len act -> go inp'  >>=
>                   (\ x -> Right $ act pos ( take len str ) : x )
```

Finally, we describe the actual parsing function itself, called `parseFormula`. Upon running `happy Parser.ly`, we get a Haskell file `Parser.hs` which contains the parseFormula function. The output for parseFormula is of type `Either ParseError Form`, so we know that the user has an appropriate input when they receive an output prefixed by the constructor `Left`.

```
> parseFormula :: String -> ParseResult Form
> parseFormula str = go $ myAlexScan str  >>= formula
>   where
>       go ( Left err ) = Left $ err { pe_str = str }
>       go ( Right res ) = Right res
>
> }
```

5 Simple Tests

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Defs

import qualified Data.Set as Set

import Test.Hspec
import Test.Hspec.QuickCheck
import Test.QuickCheck

import qualified Data.Set as Set
```

The following uses the HSpec library to define different tests. We use a mix of QuickCheck and specific inputs, depending on what we are testing for.

The "Figure 3" section corresponds to the three examples labeled 3a, 3b, and 3c [AAY24]. The paper gives a couple formulas per example to illustrate the semantics of BSML. We test each of these formulas to confirm our implementation contains the expected semantics.

```
main :: IO ()
main = hspec $ do
  describe "Figure 3" $ do
    it "Figure 3a1, p v q" $
      (m3a, s3a1) |= (p `Or` q) `shouldBe` True
    it "Figure 3a1, (p ^ NE) v (q ^ NE)" $
      (m3a, s3a1) |= (And p NE `Or` And q NE) `shouldBe` False
    it "Figure 3a2, (p ^ NE) v (q ^ NE)" $
      (m3a, s3a2) |= (And p NE `Or` And q NE) `shouldBe` True
    it "Figure 3b, <>q" $
      (m3b, s3b) |= Dia q `shouldBe` True
    it "Figure 3b, <>p" $
      (m3b, s3b) |= Dia p `shouldBe` False
    it "Figure 3b, []q" $
      (m3b, s3b) |= box q `shouldBe` False
    it "Figure 3b, []p v []q" $
      (m3b, s3b) |= (box p `Or` box q) `shouldBe` True
    it "Figure 3b, <>p ^ <>q" $
      (m3b, s3b) |= (Dia p `And` Dia q) `shouldBe` False
    it "Figure 3b, [<>(p ^ q)]+" $
      (m3b, s3b) |= enrich (MDia (mp `MOr` mq)) `shouldBe` False
    it "Figure 3c, <>(p v q)" $
      (m3c, s3c) |= (Dia p `Or` Dia q) `shouldBe` True
    it "Figure 3c, [<>(p v q)]+" $
      (m3c, s3c) |= enrich (MDia (mp `MOr` mq)) `shouldBe` True
```

We will expand this section later, by adding more tautologies that should hold for BSML logic ensuring our implementation is correct. Here we use QuickCheck, but we need to limit the maximal size of the arbitrary models we generate. This is necessary because the evaluation of support in team semantics is inherently exponential in complexity (see e.g. the clause for support of disjunctions).

```
  describe "Tautologies" $ modifyMaxSize (`div` 10) $ do
    prop "box f <==> !<>!f" $
      \(TPM m s) f -> (m::KrM,s::Team) |= box (f::Form) == (m,s) |= Neg(Dia (Neg f))
    prop "Dual-Prohibition, !<>(a v b) |= !<>a ^ !<>b" $
      \(TPM m s) -> (m, s) |= Neg (Dia (p `Or` q)) == (m,s) |= (Neg(Dia p) `And` Neg(Dia q)
          )
    prop "strong tautology is always supported" $
      \(TPM m s) -> (m,s) |= toptop
    prop "strong contradiction is never supported" $
      \(TPM m s) -> not $ (m,s) |= botbot
    prop "p v ~p is never supported"  $
      \(TPM m s) -> (m,s) |= (p `Or` Neg p)
    prop "NE v ~NE does *can* be supported" $
      expectFailure $ \(TPM m s) -> (m,s) |= (top `Or` Neg top)
    prop "strong tautology !== top" $
      expectFailure $ \(TPM m s) -> (m,s) |= toptop == (m,s) |= top
```

The paper [AAY24] discusses various interesting properties that should hold for our implementation. Narrow-scope and wide-scope relate to the "pragmatic enrichment function." The flatness test confirms that our implementation of ML formulas are flat.

```
  describe "Properties from Paper" $ modifyMaxSize (const 10) $ do
    prop "NarrowScope, <>(a v b) =| (<>a ^ <>b)" $
      \(TPM m s) -> (m,s) |= enrich (MDia (mp `MOr` mq)) == (m,s) |= enrich (MDia mp `MAnd`
          MDia mq)
    prop "Wide Scope, <>a v <>b) =| <>a ^ <>b" $
      \(TPM m s) -> all (\w -> rel' m w == s) s <= ((m,s)  |= enrich (MDia mp `MOr` MDia mq
          ) <= (m,s) |= enrich (MDia mp `MAnd` MDia mq))
  describe "Flatness" $ modifyMaxSize (const 10) $ do
```

```
  prop "(M,s) |= f <==> M,{w} |= f forall w in s" $
    \(TPM m s) f -> (m,s) |= toBSML (f::MForm) == all (\w -> (m, Set.singleton w) |=
        toBSML f) s
  prop "M,{w} |= f <==> M,w |= f" $
    \(WPM m w) f -> (m, Set.singleton w) |= toBSML (f::MForm) == (m,w) |= f
  prop "Full BSML is *not* flat" $ expectFailure $
    \(TPM m s) f -> (m,s) |= (f::Form) == all (\w -> (m, Set.singleton w) |= f) s
where
  p = Prop 1
  q = Prop 2
  mp = MProp 1
  mq = MProp 2
```

6 Conclusion

References

[AAY24] Maria Aloni, Aleksi Anttila, and Fan Yang. State-based modal logics for free choice. *Notre Dame Journal of Formal Logic*, 65(4), November 2024.