# My Report

Me

Monday 17th 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.

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

Such an inference is invalid in classical modal logic, since when $b$ is false and $c$ is true $\Diamond(b \vee c)$ is true but $\Diamond b \wedge \Diamond c$ is false. Yet the inference in FC makes perfect sense in natural language. How to account for the discrepancy then? Aloni (2022) explain FC inference as a consequence of the *neglect-zero tendency*, the cognitive preference for representations of reality that verify a sentence by concrete occurrences rather than an empty configuration. For example, though the sentence "I have no red hats" is verified both by a closet of blue hats and by a closet with no hats, "blue hats" is less cognitively taxing than "no hats", the zero-model.

# 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 3: `stack build && stack exec myprogram`
- To run the tests from Section 4: `stack clean && stack test --coverage`

# 2 Bilateral State-based Modal Logic

This section describes the basic definitions for the explicit model checker.

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

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

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

import Test.QuickCheck

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)

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)

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

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

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

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

teamParts :: Team -> Set (Team, Team)
teamParts s = cartesianProduct (Set.powerSet s) (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

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

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

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

botbot :: Form
botbot = And Bot NE

top :: Form
top = NE

toptop :: Form
toptop = Neg Bot

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

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

subsetOf :: Ord a => Set a -> Gen (Set a)
subsetOf s = Set.fromList <$> sublistOf (Set.toList s)

genFunctionToSubset :: Ord a => CoArbitrary a => Set a -> Gen (Int -> Set a)
genFunctionToSubset ws = do
  outputs <- vectorOf (length ws) (subsetOf ws)
  fmap (\f x -> f x ! outputs) arbitrary

(!) :: Int -> [Set a] -> Set a
```

```
(!) _ [] = Set.empty
(!) i xs = xs !! (i `mod` length xs)

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

instance {-# OVERLAPPING #-} Arbitrary (KrM, Team) where
  arbitrary = do
    m <- arbitrary
    s <- subsetOf (worlds m)
    return (m, s)
```

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
```

```
-- 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
```

# 3 Wrapping it up in an exectuable

We will now use the library form Section **??** in a program.

```
module Main where

main :: IO ()
main = do
  putStrLn "Hello!"
  putStrLn "GoodBye"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

The output of the program is something like this:

```
Hello!
[1,2,3,4,5,6,7,8,9,10]
[100,100,300,300,500,500,700,700,900,900]
[1,3,0,1,1,2,8,0,6,4]
[100,300,42,100,100,100,700,42,500,300]
GoodBye
```

# 4 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 Test.Hspec
```

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

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

```
{--
Properties to test for:
Narrow-scope FC:
\Diamond(\alpha\vee\beta)\vDash\Diamond\alpha\wedge\Diamond\beta

Dual-prohibition:
\neg\Diamond(\alpha\vee\beta)\vDash\neg\Diamond\alpha\wedge\neg\Diamond\beta

Universal FC:
\forall\Diamond(\alpha\vee\beta)\vDash\forall\Diamond\alpha\wedge\Diamond\beta

Wide-scope FC:
\Diamond\alpha\vee\beta\vDash\Diamond\alpha\wedge\Diamond\beta

--}

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
  describe "Abbreviations" $ do
    prop "strong tautology is always supported" $
      \(m,s) -> (m::KrM, s::Team) |= toptop
    prop "strong contradiction is never supported" $
      \(m,s) -> not $ (m::KrM, s::Team) |= botbot
    modifyMaxSize (const 10) $ prop "p v ~p is never supported"  $
      \(m,s) -> (m::KrM, s::Team) |= (p `Or` Neg p)
    prop "NE v ~NE does *can* be supported" $
      expectFailure $ \(m,s) -> (m::KrM, s::Team) |= (top `Or` Neg top)
    prop "strong tautology !== top" $
      expectFailure $ \(m,s) -> (m::KrM, s::Team) |= toptop == (m,s) |= top
  describe "Flatness" $ do
    xprop "M,s |= f <==> M,{w} |= f forall w in s (needs Arbitrary MForm)" (undefined ::
        Property)
--      \m s f -> (m::KrM, s::Team) |= toBSML (f::MForm) ==
--        all (\w -> (m, Set.singleton w) |= toBSML f) s
    xprop "M,{w} |= f <==> M,w |= f (needs Arbitrary MForm)" (undefined :: Property)
--      \m w f -> (m::KrM, Set.singleton w) |= toBSML (f::MForm) == (m,w) |= f
  where
    p = Prop 1
    q = Prop 2
    mp = MProp 1
    mq = MProp 2
```

To run the tests, use `stack test`.

To also find out which part of your program is actually used for these tests, run `stack clean && stack test --coverage`. Then look for "The coverage report for ... is available at ... .html" and open this file in your browser. See also: `https://wiki.haskell.org/Haskell_program_coverage`.

# 5 Conclusion

Finally, we can see that [LW13] is a nice paper.

# 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.

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.