Project Report - Beta

Group 3 - N. Cohen, T. Coroneo, V. Iyer, G. Kaminer, and L. Madison

Tuesday 18th 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 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 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 lists), we utilise sets in our models. We do this to prepare for the eventuality of using IntSets - which are a much more efficient structure for storing and retrieving integers than lists.

```
{-# 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
```

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 (much like one in modal logics like K or S4) consists 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 function from the set of Worlds to the powerset of Worlds - it is effectively a successor function. This gives us easy access to the successors of any given world, without needing to perform lookup operations for the same.

```
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)
```

We define below `rel'` and `val'`; two functions that help us get the successors of a particular world in a model, and the propositions true at a given world in the model respectively.

```
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 supportability and antisupportability 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], but the gist of it is that most of the definitions would be familiar to any reader well-versed in modal logics such as K. From below, it should become clear why we mentioned earlier that `antisupport` acts like negation.

Defining the semantics of $\vee$ for `support` and $\wedge$ for `antisupport` required us to use a helper function - `teamParts`. This function provides the user with the set of all pairs of subsets of a given team 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

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

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

```haskell
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.

```haskell
-- 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
```

```haskell
-- 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 --cove`
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.