A Model Checker for Bilateral State-based Modal Logic (BSML)

Group 2

Wednesday 26th March, 2025

Abstract

Bilateral State-based Modal Logic (BSML), which proposed by [Aloni(2022)], extends classical modal logic by adopting state-based semantics and introducing a non-emptiness atom to account for free choice inferences in natural language. Despite its expressive power, no automated verification tool exists for BSML. This project aims to develop a model checker for BSML, enabling automated reasoning over its logical properties.

Contents

1 Introduction

2 Motivation

BSML was developed to account for *Free Choice* (FC) inferences, where disjunctive sentences give rise to conjunctive interpretations. For example, the sentence "You may go to the beach or to the cinema" typically implies that "You may go to the beach *and* you may go to the cinema" This inference is unexpected from a classical logical perspective, as disjunction does not typically imply conjunction.

The key idea in BSML is the *neglect-zero tendency*, which posits that humans tend to disregard models that verify sentences by virtue of some empty configuration. BSML formalizes this tendency by introducing the *nonemptiness atom* (NE), which ensures that only nonempty states are considered in the interpretation of sentences. This leads to the prediction of both narrow-scope and wide-scope FC inferences, as well as their cancellation under negation.

BSML has been extended in two ways:

- $\mathbf{BSML}^{\mathbin{\vee\!\!\!\vee}}$: This extension adds the *global disjunction* $\mathbin{\vee\!\!\!\vee}$, which allows for the expression of properties that are invariant under bounded bisimulation.
- $\mathbf{BSML}^{\oslash}$: This extension adds the *emptiness operator* $\oslash$, which can be used to cancel out the effects of the nonemptiness atom (NE).

These extensions are expressively complete for certain classes of state properties, and natural deduction axiomatizations have been developed for each of these logics.

3 The Syntax of BSML

The syntax of **BSML** is defined over a set of propositional variables Prop. The formulas of BSML are generated by the following grammar:

$$\varphi ::= p \mid \bot \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid \Diamond\varphi \mid \mathrm{NE}$$

where $p \in \mathrm{Prop}$.

$\mathbf{BSML}^{\mathbin{\vee\!\!\!\vee}}$ extends **BSML** with global disjunction $\mathbin{\vee\!\!\!\vee}$ by adding the clause $\varphi \mathbin{\vee\!\!\!\vee} \varphi$.

The classical modal logic **ML** is the NE-free fragment of BSML.

Below is the implementation of the syntax of $\mathbf{BSML}^{\mathbin{\vee\!\!\!\vee}}$. For the sake of brevity, we use the type `BSMLForm` for the formulas of $\mathbf{BSML}^{\mathbin{\vee\!\!\!\vee}}$. We use `Int` to index the set of propositional variables.

```
module Syntax where


type Prop = Int

data BSMLForm = P Prop | Bot | Neg BSMLForm | Con BSMLForm BSMLForm | Dis BSMLForm BSMLForm
     | Dia BSMLForm | NE | Gdis BSMLForm BSMLForm
  deriving (Eq,Ord,Show)
```

Note that `Dis` is the "$\vee$" disjunction, while `Gdis` is the "$\mathbin{\mathpalette\make@circled\vee}$" disjunction.

The box modality $\Box$ is defined as the dual of the $\Diamond$: $\Box\varphi := \neg\Diamond\neg\varphi$.

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

The pragmatic enrichment function $[\quad]^+$: **ML** $\to$ **BSML**is recursively defined as:

$$
\begin{aligned}
[p]^+ &:= p \wedge \mathrm{NE} \\
[\bigcirc a]^+ &:= \bigcirc([a]^+) \wedge \mathrm{NE} \qquad \text{for } \bigcirc \in \{\neg, \Diamond, \Box\} \\
[\alpha \bigtriangleup \beta]^+ &:= ([\alpha]^+ \bigtriangleup [\beta]^+) \wedge \mathrm{NE} \qquad \text{for } \bigtriangleup \in \{\wedge, \vee\}
\end{aligned}
$$

We implment pragmatic enrichment `prag` as a partial function `BSMLForm -> BSMLForm`, as the **ML** formulas are not used anywhere else in this project.

```
prag :: BSMLForm -> BSMLForm
prag (P n)      = Con (P n) NE
prag (Neg f)    = Con (Neg $ prag f) NE
prag (Con f g)  = Con (Con (prag f) (prag g)) NE
prag (Dis f g)  = Con (Dis (prag f) (prag g)) NE
prag (Dia f)    = Con (prag (Dia f)) NE
prag (Gdis f g) = Con (Gdis (prag f) (prag g)) NE
prag Bot        = Con Bot NE
prag NE         = error "Cannot pragmatically enrich formulas containing NE"
```

4 Semantics

The semantics of BSML is based on *team semantics*, where formulas are interpreted with respect to sets of possible worlds, which called *states* or *teams* rather than single worlds. A *model* M is a triple (W, R, V), where:

- W is a nonempty set of possible worlds,
- $R \subseteq W \times W$ is an accessibility relation,
- $V : \mathrm{Prop} \to \mathcal{P}(W)$ is a valuation function.

A *state* s is a subset of W, interpreted as an information state. The semantics of **BSML** is bilateral, with separate conditions for support ($\models$), meaning assertion, and anti-support ($=\!\!|$), meaning rejection. The conditions are defined recursively as follows:

$$
\begin{array}{lll}
M, s \models p & \text{iff} & \forall w \in s, w \in V(p) \\
M, s =\!\!| \; p & \text{iff} & \forall w \in s, w \notin V(p) \\
\\
M, s \models \bot & \text{iff} & s = \emptyset \\
M, s =\!\!| \; \bot & & \text{always} \\
\\
M, s \models \neg\varphi & \text{iff} & M, s =\!\!| \; \varphi \\
M, s =\!\!| \; \neg\varphi & \text{iff} & M, s \models \varphi
\end{array}
$$

An atomic proposition is supported at a state if it holds at every world of that state, and is anti-supported if every world falsifies it. From this, we can already see that $\dashv$ is not the same as $\not\models$. This extends to the clauses for falsum $\bot$. Negation is then defined in terms of anti-support.

$$\begin{array}{lll} M, s \models \varphi \land \psi & \text{iff} & M, s \models \varphi \text{ and } M, s \models \psi \\ M, s \dashv \varphi \land \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \dashv \varphi \text{ and } M, u \dashv \psi \\ M, s \models \varphi \lor \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \models \varphi \text{ and } M, u \models \psi \\ M, s \dashv \varphi \lor \psi & \text{iff} & M, s \dashv \varphi \text{ and } M, s \dashv \psi \end{array}$$

A state supports a disjunction if it is a union of two substates, each supporting one of the disjuncts. The intuition is that a disjunction is supported when the information state contains evidence for both of the disjuncts. This is also known as *split disjunction*, since it requires splitting the state into substates. Dually, a conjunction is anti-suported when there is evidence falsifying both of the conjuncts.

$$\begin{array}{lll} M, s \models \Diamond\varphi & \text{iff} & \forall w \in s, \exists t \subseteq R[w] \text{ s.t. } t \neq \emptyset \text{ and } M, t \models \varphi \\ M, s \dashv \Diamond\varphi & \text{iff} & \forall w \in s,\ M, R[w] \dashv \varphi \\ \\ M, s \models \Box\varphi & \text{iff} & \forall w \in s,\ M, R[w] \models \varphi \\ M, s \dashv \Box\varphi & \text{iff} & \forall w \in s, \exists t \subseteq R[w] \text{ s.t. } t \neq \emptyset \text{ and } M, t \dashv \varphi \end{array}$$

A state supports a diamond formula $\Diamond\varphi$ if each word sees a (non-empty) state supporting φ; A state anti-supports $\Diamond\varphi$ if all worlds accessible from the state together anti-supports φ. The clauses for $\Box$ are derived from the clauses for $\neg$ and $\Diamond$.

$$\begin{array}{lll} M, s \models \text{NE} & \text{iff} & s \neq \emptyset \\ M, s \dashv \text{NE} & \text{iff} & s = \emptyset \end{array}$$

As the name suggests, the nonemptiness atom is supported when the state is empty, and is rejected otherwise. When combined with disjunction in the form of pragmatic enrichment, a state supports an enriched disjunction $[\varphi \lor \psi]^+$ when it is a union of two *non-empty*substates, each supporting one of the disjuncts. This is what enables **BSML** to derive FC inferences.

$$\begin{array}{lll} M, s & \models \varphi \vee\!\!\!\vee \psi \quad \text{iff} & M, s \models \varphi \text{ or } M, s \models \psi \\ M, s & \dashv \varphi \vee\!\!\!\vee \psi \quad \text{iff} & M, s \dashv \varphi \text{ and } M, s \dashv \psi \end{array}$$

more to add...

The following is the definition of our Data Type for Model Checker.

```haskell
-- Based on the Homework

-- {-# LANGUAGE InstanceSigs #-}
module Semantics where



import Control.Monad
import System.Random
import Test.QuickCheck
import Data.List

import Syntax
import Control.Lens (below)


type World = Integer
type Universe = [World]
type Proposition = Int
type State = [World]

type Valuation = World -> [Proposition]
type Relation = [(World,World)]

data KripkeModel = KrM Universe Valuation Relation

data ModelState = MS KripkeModel State

instance Show KripkeModel where
  show (KrM u v r) = "KrM " ++ show u ++ " " ++ vstr ++ " " ++ show r where
    vstr = "(fromJust . flip lookup " ++ show [(w, v w) | w <- u] ++ ")"

instance Show ModelState where
  show (MS k s) = "MS " ++ show k ++ " " ++ show s
```

The following helper function defines the set of all successors of a world:

```haskell
(!) :: Relation -> World -> [World]
(!) r w = map snd $ filter ((==) w . fst) r
```

Here we define the semantics of **BSML** ...

```haskell

-- helper function to find all pairs of worlds t and u that the union of t and u is the
    input s
allPairs :: [World] -> [([World], [World])]
allPairs []      = [([],[])]
allPairs (x:xs) =
  [ (x:ts, x:us) | (ts,us) <- allPairs xs ] ++
  [ (x:ts, us)   | (ts,us) <- allPairs xs ] ++
  [ (ts, x:us)   | (ts,us) <- allPairs xs ]

-- helper function to find all non-empty subsets of a list
subsetsNonEmpty :: [World] -> [[World]]
subsetsNonEmpty [] = []
subsetsNonEmpty (x:xs) =
  let rest = subsetsNonEmpty xs
  in [[x]] ++ rest ++ map (x:) rest

(|=) :: ModelState -> BSMLForm -> Bool
(MS (KrM _ v _) s) |= (P p) = all (\w -> p `elem` v w) s
(MS _ s) |= Bot = null s
(MS _ s) |= NE = not $ null s
(MS (KrM u v r) s) |= (Neg f) = (MS (KrM u v r) s) =| f
m |= (Con f g) = m |= f && m |= g
(MS k s) |= (Dis f g) = any (\(ts,us) -> (MS k ts) |= f && (MS k us) |= g) (allPairs s)
m |= (Gdis f g) = m |= f || m |= g
(MS (KrM u v r) s) |= (Dia f) = all (\w -> any (\l -> (MS (KrM u v r) l) |= f ) (
    subsetsNonEmpty (r ! w)))  s



(=|) :: ModelState -> BSMLForm -> Bool
(MS (KrM _ v _) s) =| (P p) = all (\w -> p `notElem` v w) s
(MS _ _) =| Bot = True
(MS _ s) =| NE = null s
(MS (KrM u v r) s) =| (Neg f) = (MS (KrM u v r) s) |= f
(MS k s) =| (Con f g) = any (\(ts,us) -> (MS k ts) =| f && (MS k us) =| g) (allPairs s)
m =| (Dis f g) = m =| f && m =| g
m =| (Gdis f g) = m =| f && m =| g
(MS (KrM u v r) s)  =| (Dia f) = all (\w -> (MS (KrM u v r) (r ! w)) =| f)  s
```

The following provide QuickCheck properties for the ModelState.

```haskell
-- Based on homework

instance Arbitrary ModelState where
  arbitrary = sized modelStateGen

modelStateGen :: Int -> Gen ModelState
modelStateGen n = do
  model@(KrM u _ _) <- modelGen n
  state <- sublistOf u  -- choose a set from universe as state
  return (MS model state)

modelGen :: Int -> Gen KripkeModel
modelGen 0 = do
  let u = [0]  -- at least one world
  v <- arbitraryValuation u
  r <- arbitraryRelation u
  return $ KrM u v r
modelGen n = do
  size <- choose (1, n)
  let u = [0 .. fromIntegral size - 1]
  v <- arbitraryValuation u
  r <- arbitraryRelation u
  return $ KrM u v r

arbitraryValuation :: Universe -> Gen Valuation
arbitraryValuation u = do
  props <- vectorOf (length u) (listOf arbitrary)
  let val w = props !! fromIntegral w -- function
  return val

arbitraryRelation :: Universe -> Gen Relation
arbitraryRelation u = do
  pairs <- sublistOf [(x, y) | x <- u, y <- u]
  return (nub pairs)
```

A model state pair (M, s) is indisputable if for all $w, v \in s, R[w] = R[v]$.

```haskell
indisputable :: ModelState -> Bool
indisputable (MS (KrM _ _ r) s) = any (\w -> any (\v -> sort (r ! w) == sort (r ! v )) s )
    s
```

A model state pair is state-based if for all $w \in s, R[w] = s$.

```haskell
stateBased :: ModelState -> Bool
stateBased (MS (KrM _ _ r)  s) = any (\w -> sort (r ! w) == sort s) s
```

```haskell
example1 :: KripkeModel
example1 = KrM [0,1,2] myVal [(0,1), (1,2), (2,1)] where
  myVal 0 = [0]
  myVal _ = [4]

example2 :: KripkeModel
example2 = KrM [0,1] myVal [(0,1), (1,1)] where
  myVal 0 = [0]
  myVal _ = [0, 4]

example11 :: ModelState
example11 = MS example1 [0,1,2]

example12 :: ModelState
example12 = MS example2 [0,1]
```

The following is a QuickCheck example, change this to a better tautology test.

```haskell
badTautology :: BSMLForm
badTautology =  Neg(Bot `Con` NE)

prop_tautologyHolds :: ModelState -> Bool
prop_tautologyHolds m = m |= badTautology
```

5 Web frontend for the model checker

To enhance the usability of the BSML model checker, we have developed a web-based interface using Haskell for the backend and various modern web technologies for the frontend. The web application allows users to input modal logic formulas, visualize Kripke models, and dynamically view verification results. The process works as follows:

We implemented the frontend using Next.js, KaTeX, and HTML5 Canvas. Users can enter models, states, and formulas through input fields, and the web application submits model-checking queries via HTTP requests to the backend.

On the backend, we use Scotty, a lightweight Haskell web framework, to handle requests from the frontend and run the model checker. Once the computation is complete, the backend returns the verification result (True/False) to the frontend.

Additionally, the frontend generates a graph representation of the Kripke model and states, providing users with a visual understanding of the verification process.

This web server makes the BSML model checker more accessible and user-friendly, allowing users to verify modal logic formulas without writing any Haskell code.

5.1 Web-Based User Interface

We developed a Next.js frontend that provides an intuitive user interface for building and evaluating logical models. To handle mathematical formulas, we integrated KaTeX, a JavaScript library, which dynamically renders user-entered LaTeX formulas into HTML for clear and precise display. For visualizing Kripke models, we utilized HTML5 Canvas to dynamically draw the worlds (nodes) and relationships (edges) of the logical model. The nodes are color-coded to represent different states, and the graph updates in real-time based on user input, providing an interactive and responsive experience.

To facilitate communication with the Haskell backend, we defined a structured interface, ModelEvaluationRequest, which encapsulates the essential elements of Kripke models and logical formulas. This interface includes:

- universe: A list of world identifiers.
- valuation: A mapping of worlds to the propositions that hold true in them.
- relation: A list of relationships (edges) between worlds.
- state: The selected states (worlds) for evaluation.
- formula: The logical formula to be evaluated.
- isSupport: A boolean value, true means support($\models$), false means not support ($\dashv$).

The frontend sends this data as a POST request to the backend, enabling seamless evaluation and retrieval of results.

5.2 Formula Evaluation

The Parser module is responsible for parsing logical formulas into the internal BSMLForm representation. It supporrs:

- **Atomic propositions**: e.g., p_1, p_2
- **Negation**: ! (not)
- **Conjunction**: &
- **Disjunction**: |
- **Global disjunction**: /
- **Diamond** $\Diamond$

```
pForm :: Parsec String () BSMLForm
pForm = spaces >> pCnt <* (spaces >> eof) where
  pCnt =  chainl1 pDia (spaces >> (pGdis <|> pDisj <|> pConj))

  pConj = char '&' >> return Con
  pDisj = char '|' >> return Dis
  pGdis = char '/' >> return Gdis

  -- Diamond operator has higher precedence than conjunction
  pDia = try pDiaOp <|> pAtom
  pDiaOp = spaces >> char '<' >> char '>' >> Dia <$> pAtom

  -- An atom is a variable, negation, or a parenthesized formula
  pAtom = spaces >> (pBot <|> pNE <|> pVar <|> pNeg <|> (spaces >> char '(' *> pCnt <* char
       ')' <* spaces))

  -- A variable is 'p' followed by digits
  pVar = char 'p' >> P . read <$> many1 digit <* spaces

  pBot = string "bot" >> return Bot
  pNE = string "ne" >> return NE

  -- A negation is '!' followed by an atom
  pNeg = char '!' >> Neg <$> pAtom

parseForm :: String -> Either ParseError BSMLForm
parseForm = parse pForm "input"

parseForm' :: String -> BSMLForm
parseForm' s = case parseForm s of
  Left e -> error $ show e
  Right f -> f
```

5.3 Web server configuration

The server is built using Scotty and listens for POST requests at /input:

```
main :: IO ()
main = scotty 3001 $ do
    middleware allowCors

    post "/input" $ do
        input <- jsonData :: ActionM Input
        let modelState = inputToModelState input
            (kripkeModel, state') = modelState
            KrM universe' valuation' relation' = kripkeModel
            -- parser and examle formula
            result = do
              parsedFormula <- parseForm (formula input)
              return $ if isSupport input
                         then modelState |= parsedFormula  -- support
                         else modelState =| parsedFormula  -- not support

            -- generate response
            finalResult = case result of
              Left err -> object [
                  "error" .= show err
                , "formula" .= formula input
                , "state" .= state'
                ]
              Right checkResult -> object [
                  "result" .= checkResult
                , "formula" .= formula input
                , "state" .= state'
                , "relation" .= show relation'
                , "relation_type" .= (if isSupport input then "support |=" else "reject =|"
                    :: String)
                ]

        json finalResult
```

5.4 Usage

To run the backend server, execute:

```
stack exec/Main.lhs
```

For the frontend, navigate to the Next.js project directory and run:

```
pnpm install
pnpm dev
```

This will start the frontend at `http://localhost:3001`, where users can interact with the system.

6 Conclusion

References

[Aloni(2022)] Maria Aloni. Logic and conversation: The case of free choice. *Semantics & Pragmatics*, 15:1–60, 2022. doi: 10.3765/sp.15.5. URL `https://doi.org/10.3765/sp.15.5`.