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

Group 2

Sunday 23rd March, 2025

Abstract

Bilateral State-based Modal Logic (BSML), which proposed by Aloni et al. [2024], 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 Motivition

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:

- $\mathbf{BSML}^{\mathbb{V}}$: This extension adds the *global disjunction* $\mathbb{V}$, 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 $\mathbf{BSML}^{\mathbb{V}}$ 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 $p \in \mathrm{Prop}$.

$\mathbf{BSML}^{\mathbb{V}}$ extends **BSML** with global disjunction $\mathbb{V}$ by adding the clause $\varphi \mathbb{V} \varphi$.

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

Below is the implmentation of the syntax of $\mathbf{BSML}^{\mathbb{V}}$. For the sake of breviety, we call the type `BSMLForm` for the formulas of $\mathbf{BSML}^{\mathbb{V}}$. 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 "$\mathbin{\rotatebox[origin=c]{180}{$\wedge$}\!\!\vee}$" 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** $\rightarrow$ **BSML**is recursively defined as:

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

We implment pragmatic enrichment `prag` as a partial function `BSMLForm -> BSMLForm`, leaving it up to the user to only use NE-free inputs.

```
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         = undefined
```

4 Semantics

The semantics of BSML is based on *team semantics*, where formulas are interpreted with respect to sets of possible worlds (called *states*) 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 : Prop $\rightarrow \mathcal{P}(W)$ is a valuation function.

A *state* s is a subset of W. The support and anti-support conditions for BSML formulas are defined recursively as follows:

$$
\begin{array}{lll}
M, s \models p & \text{iff} & \forall w \in s, w \in V(p) \\
M, s \mathrel{=\mkern-3mu|} p & \text{iff} & \forall w \in s, w \notin V(p) \\
M, s \models \bot & \text{iff} & s = \emptyset \\
M, s \mathrel{=\mkern-3mu|} \bot & \text{always} & \\
M, s \models \mathrm{NE} & \text{iff} & s \neq \emptyset \\
M, s \mathrel{=\mkern-3mu|} \mathrm{NE} & \text{iff} & s = \emptyset \\
M, s \models \neg\varphi & \text{iff} & M, s \mathrel{=\mkern-3mu|} \varphi \\
M, s \mathrel{=\mkern-3mu|} \neg\varphi & \text{iff} & M, s \models \varphi \\
M, s \models \varphi \wedge \psi & \text{iff} & M, s \models \varphi \text{ and } M, s \models \psi \\
M, s \mathrel{=\mkern-3mu|} \varphi \wedge \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \mathrel{=\mkern-3mu|} \varphi \text{ and } M, u \mathrel{=\mkern-3mu|} \psi \\
M, s \models \varphi \vee \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \models \varphi \text{ and } M, u \models \psi \\
M, s \mathrel{=\mkern-3mu|} \varphi \vee \psi & \text{iff} & M, s \mathrel{=\mkern-3mu|} \varphi \text{ and } M, s \mathrel{=\mkern-3mu|} \psi \\
M, s \models \varphi \bigvee \psi & \text{iff} & M, s \models \varphi \text{ or } M, s \models \psi \\
M, s \mathrel{=\mkern-3mu|} \varphi \bigvee \psi & \text{iff} & M, s \mathrel{=\mkern-3mu|} \varphi \text{ and } M, s \mathrel{=\mkern-3mu|} \psi \\
M, s \models \Diamond\varphi & \text{iff} & \forall w \in s, \exists t \subseteq R[w] \text{ s.t. } t \neq \emptyset \text{ and } M, t \models \varphi \\
M, s \mathrel{=\mkern-3mu|} \Diamond\varphi & \text{iff} & \forall w \in s, M, R[w] \mathrel{=\mkern-3mu|} \varphi
\end{array}
$$

The box modality $\Box$ is defined as the dual of the $\Diamond$, meaning $\Box\varphi$ is equivalent to $\neg\Diamond\neg\varphi$. This leads to the following support and antisupport clauses:

$$
\begin{array}{lll}
M, s \models \Box\varphi & \text{iff} & \forall w \in s, M, R[w] \models \varphi \\
M, s \mathrel{=\mkern-3mu|} \Box\varphi & \text{iff} & \forall w \in s, \exists t \subseteq R[w] \text{ s.t. } t \neq \emptyset \text{ and } M, t \mathrel{=\mkern-3mu|} \varphi
\end{array}
$$

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

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



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

import Syntax


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

type ModelState = (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] ++ ")"
```

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

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

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

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


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

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

```
indisputable :: ModelState -> Bool
indisputable (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 $w \in s, R[w] = s$.

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

```
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 = (example1, [0,1,2])

example12 :: ModelState
example12 = (example2, [0,1])
```

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 ($=\mid$).

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

Maria Aloni, Aleksi Anttila, and Fan Yang. State-based modal logics for free choice. *Notre Dame Journal of Formal Logic*, 65(4):367–413, 2024. doi: 10.1215/00294527-2024-0027.