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

Yuan Ma   Simon Chiu   Siyuan Wu   Hangjian Chen   Hongqian Xia

Sunday 30th March, 2025

Abstract

Bilateral State-based Modal Logic (BSML), proposed by **?**, extends classical modal logic by adopting state-based semantics and introducing a non-emptiness atom to account for free choice inferences in natural language. This project focuses on developing a Haskell-based model checker for BSML. We begin with a breif introduction to the motivation and semantics of the framework, followed by a discussion of its implementation in Haskell. Additionally, we use QuickCheck to verify several key properties about BSML. Finally, we present a web interface that allows users to interact with the model checker and explore BSML.

Contents

1 Introduction

Haskell, as a functional programming language, emphasizes purity (no side effects) and immutability (no data modification). This makes Haskell particularly well-suited for handling logical systems, as it provides a clean and efficient way to model and manipulate abstract structures.

BSML is a non-classical modal logic system within formal semantics. This project implements a model checker in Haskell, using QuickCheck to verify the reliability of the implementation. Additionally, a webpage interface has been created to facilitate user interaction.

Chapter 2 introduces the linguistic background and motivation behind BSML, along with the system's key non-classical aspects. Chapter 3 delves into the syntax and semantics of BSML, as well as the Haskell implementation, which forms the core of the model checker. Chapter 4 introduces QuickCheck. Finally, Chapter 5 explains the web implementation and provides practical usage examples.

2 Linguistic Motivation and BSML Semantics

BSML (Bilateral State-Based Modal Logic) originates from a project led by Aloni, titled N∅thing is Logical (NihiL)[1]. This project concerns formal semantics. It is a field that uses formalized methods to analyze semantic phenomena and seeks to uncover the underlying rules about how meanings of complex expressions are built from their parts. Natural language, as used in daily communication, often does not conform to classical logic. Consequently, formal semanticists develop non-classical logical systems to better capture linguistic phenomena.

BSML was initially designed to address a well-known issue in linguistics: free choice (FC). Consider the following sentence:

> You may go to the beach or to the cinema.

In natural language, this typically allows the inference:

> You may go to the beach, and you may go to the cinema.

However, in classical modal logic, this inference does not hold. Specifically, from $\Diamond(a \vee b)$, we cannot derive $\Diamond a \wedge \Diamond b$.

A common strategy in semantics is to attribute certain inferences to pragmatics. That is, while semantics concerns literal meaning, pragmatics examines how meaning is shaped by context. For instance, if one says:

> Some students in this class are studying logic.

Listeners typically infer that not all students are studying logic——otherwise, the speaker would have simply stated, "All students are studying logic". This inference is not a semantic entailment but rather a pragmatic inference.

[1]For details, see `https://www.marialoni.org/Nihil`.

Similarly, in the case of free choice, Aloni's approach to free choice is based on the idea that speakers construct mental models of reality when interpreting sentences. Her central claim (see **?**), termed **Neglect-Zero**, is as follows

> **Neglect-Zero:** When interpreting a sentence, speakers construct mental models of reality.In doing so, they systematically neglect structures that satisfy the sentence through an empty configuration (zero-models).

Intuitively, this means that when interpreting a disjunction, speakers ignore the possibility of an empty disjunct.

Here, we introduce several key non-classical semantics in BSML. The full definitions will be provided in the next chapter.

First, BSML formalizes this tendency by introducing the *nonemptiness atom* (NE), which ensures that only nonempty states are considered in the interpretation of sentences. The NE operator is defined as follows:

Nonemptiness Atom (NE):

- **Support Condition:** $M, s \models \mathrm{NE}$ iff $s \neq \varnothing$.
- **Anti-Support Condition:** $M, s \dashv \mathrm{NE}$ iff $s = \varnothing$.

The operator NE is used to define a pragmatic enrichment function $[\quad]^+$, which is not inherently part of the logical system, but the introduction of NE enables us to formally capture and implement this pragmatic effect within the logic system.:

$$\begin{aligned}
[p]^+ &= p \wedge \mathrm{NE} \\
[\neg\alpha]^+ &= \neg[\alpha]^+ \wedge \mathrm{NE} \\
[\alpha \vee \beta]^+ &= ([\alpha]^+ \vee [\beta]^+) \wedge \mathrm{NE} \\
[\alpha \wedge \beta]^+ &= ([\alpha]^+ \wedge [\beta]^+) \wedge \mathrm{NE} \\
[\Diamond\alpha]^+ &= \Diamond[\alpha]^+ \wedge \mathrm{NE}
\end{aligned}$$

Second, BSML is built upon team semantics, meaning that formulas are evaluated with respect to sets of possible worlds (teams) rather than individual worlds. For example, in Figure**??**, the black small square represents a state in a universe. And (b) is the zero-model of $a \vee b$, because one of the disjunctors is empty.

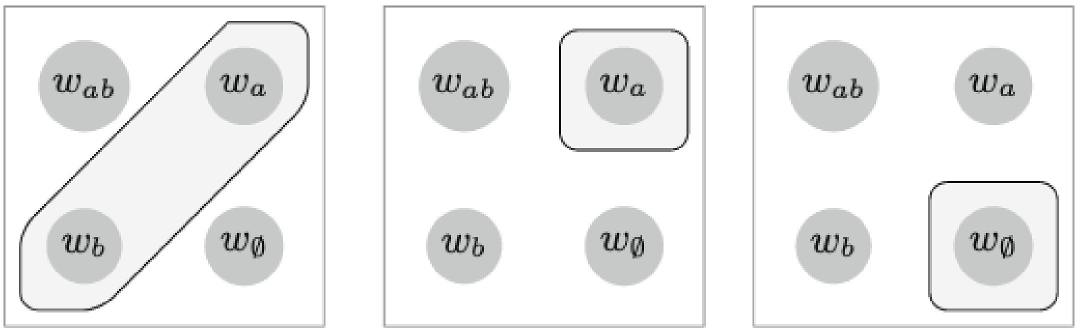
Additionallt, BSML relies on a bilateral semantics, where each formula has both **support-/assertion conditions** ($\models$) and **anti-support/rejection conditions** ($\dashv$).

$M, s \models \varphi$: φ is assertable in information state s, with $s \subseteq W$.

$M, s \dashv \varphi$: φ is rejectable in information state s, with $s \subseteq W$.

And logical consequence is defined as preservation of support.

$$\varphi \models \psi \quad \text{iff} \quad \forall M, s : M, s \models \varphi \Rightarrow M, s \models \psi$$



(a) Verifier (b) Zero-model (c) Falsifier

Figure 1: Models for $a \vee b$

Based on team-semantics, BSML adopts a split disjunction based on state-based semantics:

$M, s \models \varphi \vee \psi$: $\Leftrightarrow$ there exist t, u such that $s = t \cup u$ and $M, t \models \varphi$ and $M, u \models \psi$.
$M, s \Dashv \varphi \vee \psi$: $\Leftrightarrow M, s \Dashv \varphi$ and $M, s \Dashv \psi$.

BSML can be extended with global disjunction:

- **BSML**$^{\vee\!\!\vee}$: This extension adds the *global disjunction* $\vee\!\!\vee$, which allows for the expression of properties that are invariant under bounded bisimulation.

The global disjunction also called inquisitive disjunction. Inquisitive semantics commonly employs a global disjunction to capture questions.

With these approach, we establish a robust logical framework capable of predicting linguistic phenomena. By comparing these predictions with actual language usage in the real world, we can gain valuable insights into the underlying mechanisms of language.

It is easy to see that pragmatic enrichment has a non-trivial effect on disjunctions, and it has non-trivial effects only on disjunctions and only if they occur in a positive environment. In other words, the effect of Neglect-Zero is restricted to disjunction sentences, vanishing under negation but reappearing under double negation. The following are some linguistic properties, which are crucial for understanding the system's behavior and its implications for free choice inferences:

- **Narrow Scope FC:** $[\Diamond(\alpha \vee \beta)]^+ \models \Diamond\alpha \wedge \Diamond\beta$
- **Wide Scope FC:** $[\Diamond\alpha \vee \Diamond\beta]^+ \models \Diamond\alpha \wedge \Diamond\beta$ (if R is indisputable)
- **Dual Prohibition:** $[\neg\Diamond(\alpha \vee \beta)]^+ \models \neg\Diamond\alpha \wedge \neg\Diamond\beta$
- **Double Negation:** $[\neg\neg\Diamond(\alpha \vee \beta)]^+ \models \Diamond\alpha \wedge \Diamond\beta$

This project adopts **BSML**$^{\vee\!\!\vee}$, and the following is the specific Haskell implementation.

- LANGUAGE InstanceSigs -

3 Syntax

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 $p \in \mathrm{Prop}$.

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

```haskell
module Syntax where
import Test.QuickCheck


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 "$\mathbb{V}$" disjunction.

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

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

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

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

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

```haskell
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 (Dia (prag f)) NE
prag (Gdis f g) = Con (Gdis (prag f) (prag g)) NE
prag Bot        = Con Bot NE
prag NE         = NE
```

The following is a function prints out **BSML** formulas in a more readible manner:

```haskell
ppBSML :: BSMLForm -> String
ppBSML (P n) = "p" ++ show n
ppBSML (Neg f)    = "!" ++ ppBSML f
ppBSML (Con f g)  = "(" ++ ppBSML f ++ " & " ++ ppBSML g ++ ")"
ppBSML (Dis f g)  = "(" ++ ppBSML f ++ " | " ++ ppBSML g ++ ")"
ppBSML (Dia f)     = "<>" ++ ppBSML f
ppBSML (Gdis f g) = "(" ++ ppBSML f ++ " / " ++ ppBSML g ++ ")"
ppBSML Bot         = "bot"
ppBSML NE          = "ne"
```

Here we define an `Arbitrary` instance for `BSMLForm`:

```haskell
instance Arbitrary BSMLForm where
  arbitrary = sized randomForm where
    randomForm :: Int -> Gen BSMLForm
    randomForm 0 = oneof [P <$> elements [1..5], pure Bot, pure NE]
    randomForm n = oneof [ P <$> elements [1..5]
                         , Neg <$> randomForm (n `div` 2)
                         , Con <$> randomForm (n `div` 2) <*> randomForm (n `div` 2)
                         , Dis <$> randomForm (n `div` 2) <*> randomForm (n `div` 2)
                         , Gdis <$> randomForm (n `div` 2) <*> randomForm (n `div` 2)
                         , Dia <$> randomForm (n `div` 2) ]
```

3.1 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,
- $V : \text{Prop} \to \mathcal{P}(W)$ 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 $=\!|$ is not the same as $\not\models$. This extends to the clauses for falsum $\bot$. Negation is then defined in terms of anti-support.

$$\begin{array}{lll}
M, s \models \varphi \wedge \psi & \text{iff} & M, s \models \varphi \text{ and } M, s \models \psi \\
M, s =\!| \; \varphi \wedge \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t =\!| \; \varphi \text{ and } M, u =\!| \; \psi \\
M, s \models \varphi \vee \psi & \text{iff} & \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \models \varphi \text{ and } M, u \models \psi \\
M, s =\!| \; \varphi \vee \psi & \text{iff} & M, s =\!| \; \varphi \text{ and } M, s =\!| \; \psi
\end{array}$$

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.

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

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 $\Diamond$.

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

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 $[\varphi \vee \psi]^+$ 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.

$$\begin{array}{llll} M, s & \models \varphi \mathbin{\vee\!\!\!\!\vee} \psi & \text{iff} & M, s \models \varphi \text{ or } M, s \models \psi \\ M, s & \mathrel{=\!\!|} \varphi \mathbin{\vee\!\!\!\!\vee} \psi & \text{iff} & M, s \mathrel{=\!\!|} \varphi \text{ and } M, s \mathrel{=\!\!|} \psi \end{array}$$

Global disjunction, initial ruled out in favour of the split disjunction in **?** for the semantics of $\vee$, was reintroduced into **BSML** in **?** for better model-theoretic properties.

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

```haskell
-- Based on the Homework

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

import Test.QuickCheck
import Data.List

import Syntax
import Data.Maybe



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 n) = all (\w -> n `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 n) = all (\w -> n `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 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) (sublistOf [0..10]) -- fixed vocabulary
  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 $w, v \in s, R[w] = R[v]$.

```
isIndisputable :: ModelState -> Bool
isIndisputable (MS (KrM _ _ r) s) = all (\w -> all (\v -> sort (r ! w) == sort (r ! v )) s
    ) s
```

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

```
isStateBased :: ModelState -> Bool
isStateBased (MS (KrM _ _ r)  s) = all (\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 = MS example1 [0,1,2]

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

Here, we encode the example in Figure 2, ?. Here, `a = P 2, b = P 3`:

```
w0, wa, wb, wab :: Integer
(w0, wa, wb, wab) = (0,1,2,3)

val2a18 :: Valuation
val2a18 = fromJust . flip lookup [(w0,[]), (wa,[2]),(wb,[3]),(wab,[2,3])]

m2a18 :: KripkeModel
m2a18 = KrM [w0,wa,wb,wab] val2a18 []

ms2a18, ms2b18 :: ModelState
ms2a18 = MS m2a18 [wa, wb]
ms2b18 = MS m2a18 [wa]
```

Here is another example, from Figure 2(c), ?. Here `p = P 0, q = P 1`:

```
wp, wq, wpq :: Integer
(wp, wq, wpq) = (1,2,3) -- same w0 as above

val2c24 :: Valuation
val2c24 = fromJust . flip lookup [(w0,[]),(wp,[0]),(wq,[1]),(wpq,[0,1])]

rel2c24 :: Relation
rel2c24 = [(w0,wq), (wpq,wp), (wpq,wq)]

m2c24 :: KripkeModel
m2c24 = KrM [w0,wp,wq,wpq] val2c24 rel2c24

ms2c241, ms2c242 :: ModelState
ms2c241 = MS m2c24 [w0]
ms2c242 = MS m2c24 [wpq]
```

The following example is from Figure 3(a), ?

```
val3a22 :: Valuation
```

```
val3a22 = val2a18

rel3a22 :: Relation
rel3a22 = [(wa,w0), (wa,wab), (wb,w0), (wb,wab)]

m3a22 :: KripkeModel
m3a22 = KrM [w0,wa,wb,wab] val3a22 rel3a22

ms3a22 :: ModelState
ms3a22 = MS m3a22 [wa,wb]

counterexamplews1 :: ModelState
counterexamplews1 = MS (KrM [0,1,2,3,4] (fromJust . flip lookup [(0,[0,4,6,8,9,10])
    ,(1,[1,4,6,8]),(2,[0,3,5,8,9]),(3,[0,1,4,6,8]),(4,[1,2,3,7,8])]) [(0,0),(1,1),(1,4)
    ,(2,1),(2,4),(3,1),(3,2),(4,1)] ) [0,2,3]

counterexamplewmd :: ModelState
counterexamplewmd = MS (KrM [0,1,2] (fromJust . flip lookup [(0,[1,3,4,7,9])
    ,(1,[0,2,6,7,8,9,10]),(2,[1,6,8,9])]) [(1,0),(1,1),(2,1),(2,2)]) [0,1]

p :: BSMLForm
p = P 0
q :: BSMLForm
q = P 1
```

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

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

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

3.2 Simple Tests

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

```
{-# OPTIONS_GHC -Wno-name-shadowing #-}
module Main where

import Syntax
import Semantics
import Test.Hspec
import Test.QuickCheck
import Test.Hspec.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.

```
main :: IO ()
main = hspec $ do
    describe "BSML Properties" $ modifyMaxSize (const 5)$ modifyMaxDiscardRatio(const 50)$
        do
      it "Figure 2, Aloni2018 [wa,wb] |= a | b" $
        ms2a18 |= Dis a b `shouldBe` True
      it "Figure 2, Aloni2018 [wa,wb] not |= a / b" $
        ms2a18 |= Gdis a b `shouldBe` False
      it "Figure 2, Aloni2018 [wa] |= (a | b) & (a / b)" $
        ms2b18 |= Con (Dis a b) (Gdis a b) `shouldBe` True
      it "Figure 2(c), Aloni2024  [w0] not |= [<>(p | q)]+" $
        ms2c241 |= prag (Dia (Dis p q)) `shouldBe` False
      it "Figure 2(c), Aloni2024  [wpq] |= [<>(p | q)]+" $
        ms2c241 |= prag (Dia (Dis p q)) `shouldBe` False
      it "Figure 2(a), Aloni2022  [wa,wb] is indisputable" $
        isIndisputable ms3a22 `shouldBe` True
      it "Figure 2(a), Aloni2022  [wa,wb] is not state-based" $
        isStateBased ms3a22 `shouldBe` False
      it "State-basedness implies indisputability" $
        property $ \ms -> isStateBased ms ==> isIndisputable ms
      it "Narrow Scope FC" $
        property $ \ms -> ms |= prag (Dia (Dis p q)) ==> ms |= Con (Dia p) (Dia q)
      it "Wide Scope FC" $
        property $ \ms -> isIndisputable ms ==> ms |= prag (Dis (Dia p) (Dia q)) ==> ms |=
            Con (Dia p) (Dia q)
      it "Dual Prohibition" $
        property $ \ms -> ms |= prag ((Neg . Dia) (Dis p q)) ==> ms |= Con (Neg (Dia p)) (
            Neg (Dia q))
      it "Double Negation" $
        property $ \ms -> ms |= prag ((Neg . Neg . Dia) (Dis p q)) ==> ms |= Con (Dia p) (
            Dia q)
      it "Modal Disjunction" $
        property $ \ms -> isStateBased ms ==> ms |= prag (Dis p q) ==> ms |= Con (Dia p) (
            Dia q)
    where
        p = P 0
        q = P 1
        a = P 2
        b = P 3
```

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`
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`.

4 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.

The web server's source code is available on GitHub ?. We developed this project with assistance from **V0 AI** ?, and detailed prompt information can be found in the README.

4.1 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($|=$), false means not support ($=|$).

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

4.2 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** $\Diamond$

```haskell
module Parser where

import Syntax
import Text.Parsec

-- Based on the Parsec Homework
pForm :: Parsec String () BSMLForm
pForm = spaces >> pCnt <* (spaces >> eof) where
```

```
  pCnt =  chainl1 pDiaBox (spaces >> (pGdis <|> pDisj <|> pConj))

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

  -- Diamond operator has higher precedence than conjunction
  pDiaBox = spaces >> ( try (pDiaOp <|> pBoxOp) <|> pAtom)
  pDiaOp = char '<' >> char '>' >> Dia <$> pDiaBox
  pBoxOp = char '[' >> char ']' >> box <$> pDiaBox

  -- 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 <$> pDiaBox

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
```

4.3 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
            (MS kripkeModel state') = modelState
            KrM _ _ relation' = kripkeModel

            -- Parse and Check Formula
            result = do
              parsedFormula <- parseForm (formula input)
              let finalFormula = if isPrag input then prag parsedFormula else parsedFormula
              return $ if isSupport input
                         then modelState |= finalFormula
                         else modelState =| finalFormula

            -- 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
    post "/quickcheck" $ do
        inputQuickCheck <- jsonData :: ActionM InputQuickCheck

        -- Parse both formulas
        let pf1 = parseForm (formulaL inputQuickCheck)
            pf2 = parseForm (formulaR inputQuickCheck)

        case (pf1, pf2) of
          (Right f1, Right f2) -> do
            let finalF1 = if isPragL inputQuickCheck then prag f1 else f1
                finalF2 = if isPragR inputQuickCheck then prag f2 else f2

                prop m = prop_implicationHolds finalF1 finalF2 m

            result <- liftIO $ quickCheckWithResult stdArgs { maxSuccess = 30 } prop

            let jsonResult = case result of
                  Success {} -> object [
                      "status" .= ("passed" :: String),
                      "numTests" .= numTests result
                    ]
                  GaveUp {} -> object [
                      "status" .= ("gave up" :: String),
                      "reason" .= reason result
                    ]
                  Failure { output = out, usedSeed = _ } -> object [
                      "status" .= ("failed" :: String),
                      "reason" .= reason result,
                      "output" .= out
                    ]
                  NoExpectedFailure {} -> object [
                      "status" .= ("unexpected success" :: String)
                    ]

            json jsonResult

          _ -> json $ object [
              "status" .= ("parse error" :: String),
              "errorL" .= show pf1,
              "errorR" .= show pf2
            ]
```

4.4 Usage

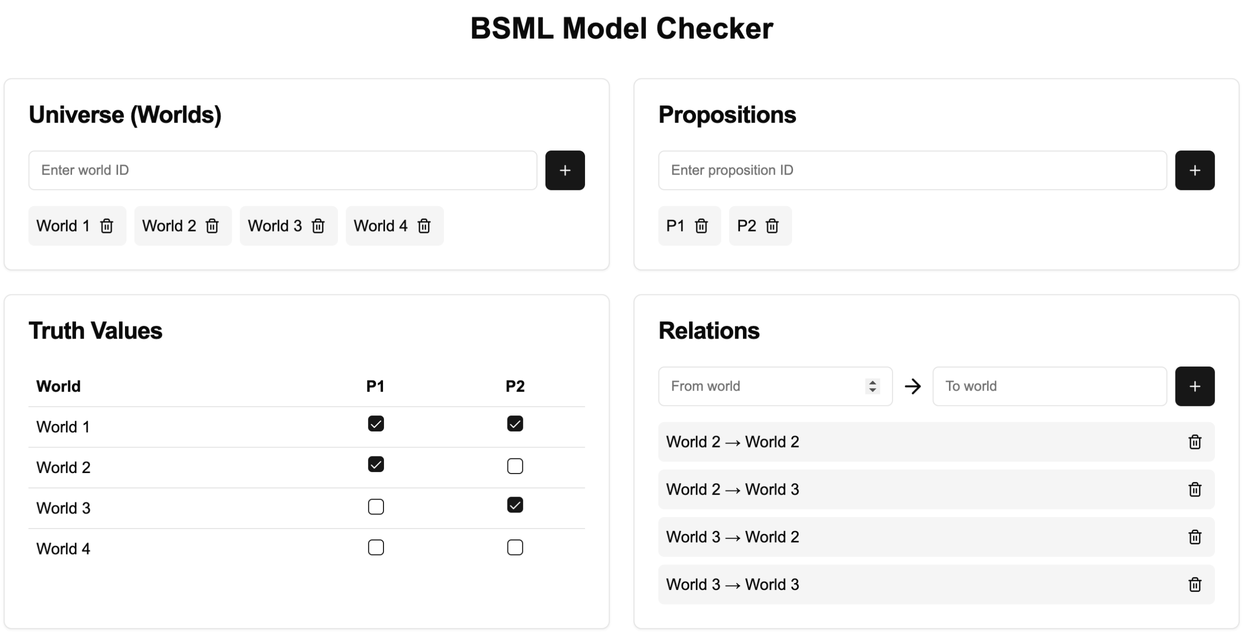
Our BSML model checker web application is available at `https://bsmlmc.seit.me`.

Here we give an example to illustrate how to use the web application.

We define a model with universe:{W1,W2,W3,W4}, state:{W2,W3}.In W1, p1 and p2 are true; in W2, p1 is true; in W3, p2 is true; W4 is empty. W2 and W3 are reflexive and mutually related. We test whether formula $\Diamond p1 \wedge \Diamond p2$ holds in this model. Figure **??** shows how to input worlds, valuations and relations in interface the. Figure **??** shows how to choose worlds in the state and input the formula. By clicking **Evaluate** bottow, web can return model-checking results. Figure **??** displays a graphical representation of the model and state.

5 Conclusion

In this project, we have implemented a Haskell-based model checker for BSML (Bilateral State-Based Modal Logic). We gave a concise introduction about the basic framework of BSML and


BSML Model Checker

Universe (Worlds)

Enter world ID +

World 1 · World 2 · World 3 · World 4

Propositions

Enter proposition ID +

P1 · P2

Truth Values

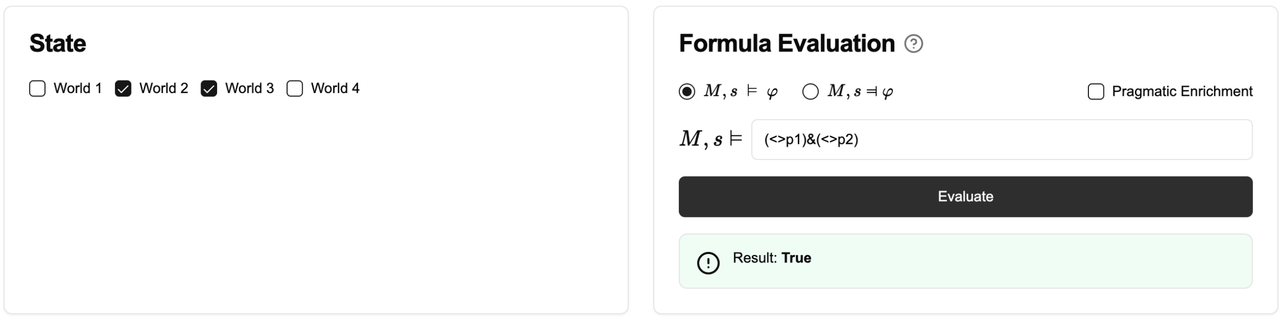
World	P1	P2
World 1	☑	☑
World 2	☑	☐
World 3	☐	☑
World 4	☐	☐

Relations

From world → To world +

World 2 → World 2
World 2 → World 3
World 3 → World 2
World 3 → World 3

Figure 2: interface1


State

☐ World 1 ☑ World 2 ☑ World 3 ☐ World 4

Formula Evaluation

● $M, s \vDash \varphi$ ○ $M, s \Dashv \varphi$ ☐ Pragmatic Enrichment

$M, s \vDash$ (<>p1)&(<>p2)

Evaluate

Result: **True**

Figure 3: interface2

its extension with global disjunction. We then implemented the models, syntax and semantics in Haskell. Lastly, we used QuickCheck to check several facts about BSML. BSML provides a powerful framework for handling free-choice phenomena in natural language, and by using Haskell, we have built an efficient and reliable tool to evaluate complex models and formulas.

Currently, the implementation of BSML is limited to natural deduction, and there is no way to handle assumptions in Haskell. Therefore, future work may involve providing a sequent calculus for the BSML system.

Additionally, BSML has many other extended versions, all of which could be implemented in Haskell in the future. For example:

QBSML (see **?**) extends BSML with quantification over possible worlds and states. Implementing this extension in Haskell would refine our model checker to handle richer linguistic sentences, thus enhancing the expressiveness of BSML.

Bilateral Update Semantics (BiUS) (See **?**) introduces a dynamic perspective on meaning change, incorporating updates. Implementing BiUS in the current model checker could enhance its ability to model information dynamics in discourse.

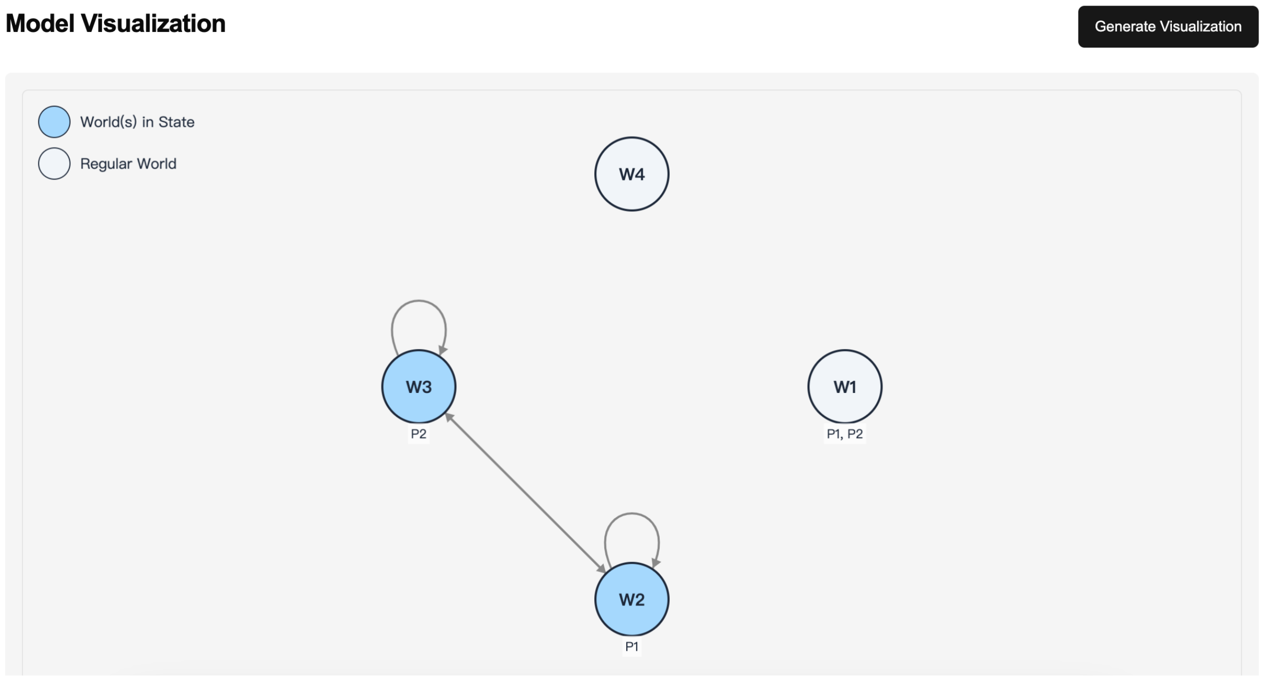


Figure 4: model graph

These extensions will improve the expressiveness of BSML and further demonstrate Haskell's suitability for formal semantic modeling.