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

Group 2

Friday 14th March, 2025

**Abstract**

Bilateral State-based Modal Logic (BSML), which proposed by**Aloni2024**, 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

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:

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

# 2 BSML Semantics

Bilateral State-based Modal Logic (BSML) is a modal logic that employs *team semantics* (also known as state-based semantics). It was introduced to account for *Free Choice* (FC) inferences in natural language, where conjunctive meanings are unexpectedly derived from disjunctive sentences. 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" BSML extends classical modal logic with a *nonemptiness atom* (NE), which is true in a state if and only if the state is nonempty. This extension allows BSML to formalize the *neglect-zero tendency*, a cognitive tendency to disregard structures that verify sentences by virtue of some empty configuration.

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 $p \in \mathrm{Prop}$. The classical modal logic (ML) is the NE-free fragment of BSML.

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,
- $V : \text{Prop} \to \mathcal{P}(W)$ 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:

$$
\begin{aligned}
&M, s \models p \quad \text{iff} \quad \forall w \in s, w \in V(p) \\
&M, s \dashv p \quad \text{iff} \quad \forall w \in s, w \notin V(p) \\
&M, s \models \bot \quad \text{iff} \quad s = \emptyset \\
&M, s \dashv \bot \quad \text{always} \\
&M, s \models \text{NE} \quad \text{iff} \quad s \neq \emptyset \\
&M, s \dashv \text{NE} \quad \text{iff} \quad s = \emptyset \\
&M, s \models \neg\varphi \quad \text{iff} \quad M, s \dashv \varphi \\
&M, s \dashv \neg\varphi \quad \text{iff} \quad M, s \models \varphi \\
&M, s \models \varphi \wedge \psi \quad \text{iff} \quad M, s \models \varphi \text{ and } M, s \models \psi \\
&M, s \dashv \varphi \wedge \psi \quad \text{iff} \quad \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \dashv \varphi \text{ and } M, u \dashv \psi \\
&M, s \models \varphi \vee \psi \quad \text{iff} \quad \exists t, u \subseteq s \text{ s.t. } s = t \cup u \text{ and } M, t \models \varphi \text{ and } M, u \models \psi \\
&M, s \dashv \varphi \vee \psi \quad \text{iff} \quad M, s \dashv \varphi \text{ and } M, s \dashv \psi \\
&M, s \models \varphi \veebar \psi \quad \text{iff} \quad M, s \models \varphi \text{ or } M, s \models \psi \\
&M, s \dashv \varphi \veebar \psi \quad \text{iff} \quad M, s \dashv \varphi \text{ and } M, s \dashv \psi \\
&M, s \models \Diamond\varphi \quad \text{iff} \quad \forall w \in s, \exists t \subseteq R[w] \text{ s.t. } t \neq \emptyset \text{ and } M, t \models \varphi \\
&M, s \dashv \Diamond\varphi \quad \text{iff} \quad \forall w \in s, M, R[w] \dashv \varphi
\end{aligned}
$$

The box modality  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:

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

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

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

# 3 The Syntax of BSML

Here, we describe the syntax of BSML with global disjunction:

```haskell
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)

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

Note that Dis is the "V" disjunction, while Gdis is the "\V" disjunction.

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

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

## 4 The Definiton of Model Checker Data Type

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

```haskell
-- {-# LANGUAGE InstanceSigs #-}
module Checker 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] ++ ")"
```

## 5 Wrapping it up in an exectuable

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

```haskell
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
  print somenumbers
  print (map funnyfunction somenumbers)
  myrandomnumbers <- randomnumbers
  print myrandomnumbers
  print (map funnyfunction myrandomnumbers)
  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
```

# 6 Conclusion

Finally.

# References

references

references.bib