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

Group 2

Friday 14^{th} March, 2025

Abstract

Bilateral State-based Modal Logic (BSML) 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 de- velop a model checker for BSML, enabling automated reasoning over its logical properties. The tool will implement a decision procedure for model checking BSML formulas, ensuring computational efficiency and correctness. Applica- tions include verifying linguistic and logical constraints in team semantics.

Contents

1 The Syntax of BSML

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

```
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** $\rightarrow$ **BSML** is describe recursively as follows:

```
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
```

2 The Definiton of Model Checker Data Type

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

```
-- {-# 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] ++ ")"
```

3 Wrapping it up in an exectuable

We will now use the library form Section 2 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
```

4 Conclusion

Finally, we can see that [LW13] is a nice paper.

References

[LW13] Fenrong Liu and Yanjing Wang. Reasoning about agent types and the hardest logic puzzle ever. *Minds and Machines*, 23(1):123–161, 2013.