Model Checking on Succinct Epistemic Models

May Lee
Dimitrios Koutsoulis

August 26, 2018

Abstract

This paper documents our Haskell implementation for succinct epistemic models, as envisaged by Charrier and Schwartzenruber [2] and an algorithm for model checking given these models. The advantages of this representation is expected to be a more concise description of epistemic states and lower computational complexity during model checking.

Contents

1 Introduction

In the framework of Dynamic Epistemic Logic, the beliefs of agents under a given situation are encoded by Epistemic Models. The beliefs pertaining to events taking place are encoded by Event Models. The Product Update defined based on these two represents what the agents believe and how they would update their believes after an event has taken place.

Model Checking is the process of verifying whether some proposition ϕ of the language of propositional epistemic logic $\mathcal{L}_{EL}$ holds true for a given Epistemic Model M, universally $M \vDash \phi$ or locally $M, w \vDash \phi$. When the Epistemic Model is a Kripke Model, this is done in accordance to the Kripke semantics. An Epistemic Kripke Model consists of a set of agents I, a set of worlds W, an accessibility relation R_i for each of the agents $i \in I$ and a valuation function $V : W \to \mathcal{L}_{PL}$ where $\mathcal{L}_{PL}$ is the language of propositional logic.

2 Motivation

Epistemic Models are usually expressed as Kripke Models. Alternative representations have been proposed that are purportedly more concise.

One of them is the Knowledge Structure by J. V. Benthem, M. Gattinger et al. [3] [4] where worlds are encoded as subsets of some vocabulary V that satisfy some state law θ. The relation of each agent i is represented by a subset of V called the observable by i variables O_i. An exponential increase in conciseness is achieved this way. It is not known whether model checking on Knowledge Structures can be done in PSPACE as is the case for Kripke Models.

We know this to be the case for another concise representation, namely the succinct one by T. Charrier and F. Schwarzentruber [2]. This is similar to Knowledge Structures but instead of subsets of observable variables we have a mental program, i.e. a recursively defined relation between subsets of V, π_i in the notation of [2], that encodes the accessibility relation for each agent i. This calls for a Haskell implementation of the aforementioned Succinct Epistemic Model and its model checking algorithm for the static part.

3 Implementation

We implemented Kripke and Succinct Epistemic models. We also implemented translation functions between the two. Note that the translation from Kripke to Succinct models introduces nominal propositions whose sole purpose is to identify a world each. Given that we represent agents, worlds and propositions of Kripke models as positive integers, we let the nominal propositions be the negations of the worlds they represent. We presuppose that the worlds are identified by positive integers so that their nominal propositions do not conflict with the actual propositions. As such translating back and forth from a Kripke model gets us one whose universe of propositions is a strict superset of the original. Still, if we restrict it to the original universe, it is isomorphic to the original Kripke model. Finally we implemented Model Checking for both the Kripke and the Succinct models.

4 Code

4.1 Propositional Logic

Propositional logic related definitions adapted from the slides.

```
type Proposition = Integer
data Form = P Proposition | Neg Form | Conj Form Form | Disj Form Form | Top |
    Bottom | Knows Agent Form deriving (Eq,Ord)

satisfies :: Assignment -> Form -> Bool
satisfies v (P k)       = k `elem` v
satisfies v (Neg f)     = not (satisfies v f)
satisfies v (Conj f g)  = satisfies v f && satisfies v g
satisfies v (Disj f g)  = satisfies v f || satisfies v g
satisfies _ Top         = True
satisfies _ Bottom      = False
satisfies _ (Knows _ _) = False
```

Function `makeBeta` that given AP_M, AP_W and the valuations of all worlds, computes β_M which we use to discard valuations that do not correspond to any possible world.

```
makeBeta :: [Proposition] -> [Proposition] -> [(World, Valuation)] -> Form
makeBeta ap_M ap_W = makeBeta2 ap_M ap_W ap_W

makeBeta2 :: [Proposition] -> [Proposition] -> [Proposition] -> [(World, Valuation
    )] -> Form
makeBeta2 _ _  ap_W [] = uExists ap_W
makeBeta2 _ [] ap_W _  = uExists ap_W
makeBeta2 ap_M (prop_w:restProps) ap_W (val:valuations) = Conj (Disj (Neg (P
    prop_w)) (desc ap_M (snd val))) (makeBeta2 ap_M restProps ap_W valuations)
```

- `uExists` constructs the formula that corresponds to the exclusive existential $!\exists$. ap is the list of all propositions.
- `possibleWorld` is the conjunction of implications from nominal propositions to the descriptions of the world that they denote. So that if some p_w is true for an interpretation I, then if I is to satisfy the conjunct, it must also satisfy $V(w)$

```
uExists :: [Proposition] -> Form
uExists ap = uExists2 [(x, remove x ap) | x <- ap]

uExists2 :: [(Proposition, [Proposition])] -> Form
uExists2 [] = Bottom
uExists2 ((active_prop, inactive_props):rest) =
    Disj (possibleWorld active_prop inactive_props) (uExists2 rest)

possibleWorld :: Proposition -> [Proposition] -> Form
possibleWorld active_prop
  = foldr (Conj . Neg . P) (P active_prop)
```

The description `desc` of a valuation V is the conjunction of the propositions true in V along with the negations of those not true in V.

```
desc :: [Proposition] -> [Proposition] -> Form
desc ap props = desc2 (nub ap \\ props) props

desc2 :: [Proposition] -> [Proposition] -> Form
desc2 out_props (in_prop:in_props) = Conj (P in_prop) (desc2 out_props in_props)
desc2 (out_prop:out_props) []      = Conj (Neg (P out_prop)) (desc2 out_props [])
desc2 [] []                        = Top
```

4.2 Our implementation of the Mental Program.

Note that the `UniversalProgram` can reach any world from any world while the `IdleProgram` cannot reach anything. The universal program is meant to replace the `set` program as it appears on the paper. This optimization makes sense since the universal program when restricted to assignments over ap equals the set program.

```
data Program = AssignTo Proposition Form
             | Question Form
             | Semicolon Program Program
             | Cap Program Program
             | Cup Program Program
             | IdleProgram
             | UniversalProgram
             deriving (Eq)
```

The valuation function `runProgram` for programs follows below. The implementation follows closely the definitions in the paper. We were forced to optimize the `Semicolon` program in the case where either of its subprograms is a `Question`. Ignoring unequal arguments (assignments) to the `Question` program, which by its very definitions are automatically discarded, shaves off a considerable chunk of runtime. Note that in the general case only assignments that satisfy β are considered as intermediate assignments.

```
runProgram :: Form -> [Proposition] -> Program -> Assignment -> Assignment -> Bool
runProgram _ _ (AssignTo prop formula) a_1 a_2  | satisfies a_1 formula =
    sort ( nub $ prop:a_1) == sort ( nub a_2)
                                                | otherwise             =
    sort ( nub (remove prop a_1)) == sort ( nub a_2)
runProgram _ _ (Question formula) a_1 a_2         =
    (sort ( nub a_1) == sort ( nub a_2)) && satisfies a_1 formula
runProgram beta ap (Semicolon (Question form) pi_2) a_1 a_2 =
    runProgram beta ap (Question form) a_1 a_1 && runProgram beta ap pi_2 a_1 a_2
runProgram beta ap (Semicolon pi_1 (Question form)) a_1 a_2 =
    runProgram beta ap pi_1 a_1 a_2 && runProgram beta ap (Question form) a_2 a_2
runProgram beta ap (Semicolon pi_1 pi_2) a_1 a_2     =
    not (all (\x -> not (runProgram beta ap pi_1 a_1 x && runProgram beta ap pi_2
        x a_2)) [ assignment | assignment <- allAssignmentsFor ap, satisfies
        assignment beta] )
runProgram beta ap (Cap pi_1 pi_2) a_1 a_2           =
    runProgram beta ap pi_1 a_1 a_2 && runProgram beta ap pi_2 a_1 a_2
runProgram beta ap (Cup pi_1 pi_2) a_1 a_2           =
    runProgram beta ap pi_1 a_1 a_2 || runProgram beta ap pi_2 a_1 a_2
runProgram _ _ IdleProgram _ _                     = False
runProgram _ _ UniversalProgram _ _                = True
```

4.3 Succinct Epistemic Model

`SuccEpistM` is our representation of a Succinct Epistemic Model and consists of a list of propositions AP_M, β_M and a program for each agent π_a.

```
data SuccEpistM = Mo
       [Proposition]
       Form
       [(Agent, Program)]
       deriving (Eq)
```

Our implementation of model checking Succinct Epistemic Models, `modelCheck`. For the case of the knowledge operator we consider all assignments that correspond to a valid world of the original Kripke model (satisfy β). In the case of the knowledge operator, if there is no program for some agent then that agent vacuously knows

everything. That's why when we `lookup` that agent in the program list and the lookup returns `Nothing`, we return `True`.

```
modelCheck :: SuccEpistM -> Assignment -> Form -> Bool
modelCheck _ world (P k)            = k `elem` world
modelCheck model world (Neg f)          =
    not (modelCheck model world f)
modelCheck model world (Conj f g)       =
    modelCheck model world f && modelCheck model world g
modelCheck model world (Disj f g)       =
    modelCheck model world f || modelCheck model world g
modelCheck (Mo ap beta programs) world (Knows agent f) =
    case lookup agent programs of
         Just program_a -> all (\x -> not (runProgram beta ap program_a world  x)
                  || modelCheck (Mo ap beta programs) x f ) [ assignment |
                      assignment <- allAssignmentsFor ap, satisfies assignment
                      beta]
         Nothing -> True
modelCheck _ _ Top                      = True
modelCheck _ _ Bottom                   = False
```

A different interface `modelCheck2` takes as its second argument the actual world number instead of its corresponding assignment, like `modelCheck` does. This is meant to be more user friendly.

```
modelCheck2 :: SuccEpistM -> World -> Form -> Bool
modelCheck2 (Mo props beta programs) world =
    modelCheck (Mo props beta programs) assignment
    where assignment = head $ filter (\x -> -world `elem` x) [ assignment2 |
        assignment2 <- allAssignmentsFor props, satisfies assignment2 beta]
```

4.4 Kripke Model

The Kripke Epistemic Model is represented by `KripkeM`. The internal representation of relations consists of a list of tuples of which the first element is a world and the second a list of its neighbors. A wrapping function `relationWrap`, that translates the more common 2-tuple list, relations to this representation is also provided.

```
data KripkeM = KMo
       [(Agent, Relation)]
       [(World, Valuation)]
       deriving (Eq,Show)

type World = Integer
type Valuation = [Proposition]
type Relation = [(World, [World])]
```

We will be using the `dropWorlds` function to restrict attention to submodels. Its input is some Kripke Epistemic Model and a list of worlds to be disregarded.

```
dropWorlds :: KripkeM -> [World] -> KripkeM
dropWorlds (KMo agentRelations worldValuations) worldsToDrop =
    KMo
    [(agent, [(a,[b | b <- neighbors, b `notElem` worldsToDrop]) | (a,neighbors)
        <- relation, a `notElem` worldsToDrop]) | (agent, relation) <-
        agentRelations]
    [(world, valuation) | (world, valuation) <- worldValuations, world `notElem`
        worldsToDrop]
```

The Kripke Model Checking function `modelCheckKrpk`.

```
modelCheckKrpk :: KripkeM -> World -> Form -> Bool
modelCheckKrpk (KMo _ valuations) world (P k) =
    case lookup world valuations of
      Just valuation -> k `elem` valuation
      Nothing -> False
modelCheckKrpk model world (Neg f)          =
    not (modelCheckKrpk model world f)
modelCheckKrpk model world (Conj f g)       =
    modelCheckKrpk model world f && modelCheckKrpk model world g
modelCheckKrpk model world (Disj f g)       =
    modelCheckKrpk model world f || modelCheckKrpk model world g
modelCheckKrpk (KMo relations valuations) world (Knows agent f) =
    case lookup agent relations of
         Just relation_a -> case lookup world relation_a of
            Just neighbors -> all (\x -> modelCheckKrpk (KMo relations valuations)
                 x f ) neighbors
            Nothing -> True
         Nothing -> True
modelCheckKrpk _ _ Top                      = True
modelCheckKrpk _ _ Bottom                   = False
```

`kripkeToSuccinct` translates Kripke Epistemic Models to Succinct ones. Note that for each world the nominal proposition is given by negating the number representing the original world.

```
kripkeToSuccinct :: KripkeM -> SuccEpistM
kripkeToSuccinct (KMo relations valuations) =
    Mo ap beta programs
     where ap = sort (ap_M ++ ap_W)
           beta = makeBeta ap_M ap_W valuations
           programs = [(agent, relationToProgram1 relation) | (agent, relation) <-
                relations]
           ap_M = nub $ concat [snd x | x <- valuations]
           ap_W = [-(fst x) | x <- valuations]
```

4.5 Model Translation Algorithms

`succinctToKripke` translates Succinct Epistemic models to Kripke ones.

```
succinctToKripke :: SuccEpistM -> KripkeM
succinctToKripke (Mo ap beta programs) =
    KMo relations worldsAndAssignments
    where relations = [(agent, programToRelation program ap beta
        worldsAndAssignments) | (agent, program) <- programs]
          worldsAndAssignments = zip [1..(toInteger $ length assignments)]
              assignments
          assignments = [ assignment | assignment <- allAssignmentsFor ap,
              satisfies assignment beta]
```

`succinctToKripke2` is similar to the above with the added caveat that each assignment is interpreted as the world it represented in the original Kripke Model. This works under the guarantee that the Succinct model to be translated was constructed using `kripkeToSuccinct` and, as such, we have access to nominal propositions which are negatives of the original worlds. Therefore successive use of `KripkeToSuccinct` and `succinctToKripke2` on a Kripke Model M nets the model M itself (plus the nominal propositions).

```
succinctToKripke2 :: SuccEpistM -> KripkeM
succinctToKripke2 (Mo ap beta programs) =
    KMo relations worldsAndAssignments
    where relations = [(agent, programToRelation program ap beta
        worldsAndAssignments) | (agent, program) <- programs]
          worldsAndAssignments = [(negate $ minimum a, a) | a <- assignments]
          assignments = [ assignment | assignment <- allAssignmentsFor ap,
              satisfies assignment beta]
```

As the name suggets, `programToRelation` translates mental programs to relations of the neighbor representation. It is a recursive helper function of `succinctToKripke`.

```
programToRelation :: Program -> [Proposition] -> Form -> [(World, Assignment)] ->
    Relation
programToRelation program ap beta worldsAndAssignments = programToRelation1
    program ap beta worldsAndAssignments worldsAndAssignments

programToRelation1 :: Program -> [Proposition] -> Form -> [(World, Assignment)] ->
     [(World, Assignment)] -> Relation
programToRelation1 _ _ _ [] _ = []
programToRelation1 program ap beta ((world,assignment):rest) worldsAndAssignments
    =
        (world, [vorld | (vorld, ass1) <- worldsAndAssignments, runProgram beta ap
             program assignment ass1]): programToRelation1 program ap beta rest
            worldsAndAssignments
```

`relationToProgram1` is the inverse of `programToRelation`.

```
relationToProgram1 :: Relation -> Program
relationToProgram1 [] = IdleProgram
relationToProgram1 ((world, neighbors):rest) =
   Cup (relationToProgram2 world neighbors) (relationToProgram1 rest)

relationToProgram2 :: World -> [World] -> Program
relationToProgram2 _ [] = IdleProgram
relationToProgram2 w_1 (w_2:rest) =
    Cup ( Semicolon (Semicolon (Question (P (-w_1)) ) UniversalProgram) (Question
        (P (-w_2))) ) (relationToProgram2 w_1 rest)
```

As we mentioned earlier, `relationWrap` can be used to translate relations of the usual list of pairs representation to the neighbor representation used internally by our Kripke Models.

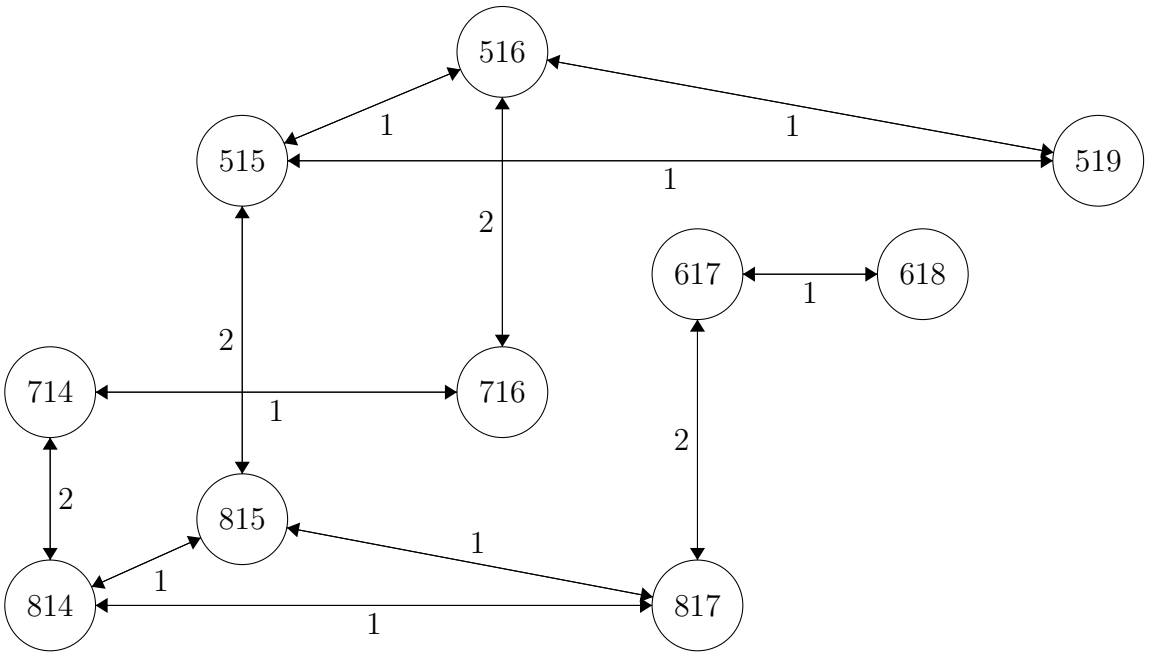
```
relationWrap :: [(World, World)] -> [(World,[World])]
relationWrap relation = map (\x -> (x,nub [b |(a,b) <- relation, a == x])) worlds
                        where worlds = nub [a | (a,_) <- relation]
```

5 Model Checking Examples

In this section we provide examples of Model Checking on Succinct Epistemic Models of puzzles taken from the Stanford Encyclopedia of Philosophy [1].

5.1 Cheryl's Birthday

Let's have a look at Cheryl's Birthday Puzzle. We encode the initial arrangement given by Cheryl to Albert (1) and Bernard (2) as the following Kripke Model.



The reflexive edges on each world are hidden towards a clearer picture.

```
birthdayModel :: KripkeM
birthdayModel = KMo
                [(1, relationWrap $ nub $ reflexize $ symmetrize [(515, 516),
                    (515,519), (516,519), (714,716), (617,618), (814,815),
                    (815,817), (814,817)]),
                 (2, relationWrap $ nub $ reflexize $ symmetrize [(714,814),
                     (515,815), (516,716), (617,817), (519,519), (618,618)])]
                (zip worlds [[a] | a <- worlds])
                where worlds = [515,516,519,714,716,617,618,814,815,817]
```

Each world corresponds to a date with the first digit being the month and the following two the day of the month. The valuation of each world contains only one nominal proposition. Worlds corresponding to dates that some agent is not able to discern, e.g. two dates in May for Albert are paired in that agent's relation.

The function `knowsDate` takes as input an agent and generates a proposition that states that the given agent knows the date.

```
knowsDate :: Agent -> Form
knowsDate agent = foldr (\x y -> Disj y $ Knows agent $ P x) Bottom
    [515,516,519,714,716,617,618,814,815,817]
```

As the puzzle prescribes, Albert starts by publicly announcing that he knows that Bernard does not know the date of Cheryl's birthday. This is evaluated as true in worlds 714, 716, 814, 815, 817.

```
all (\world -> modelCheck2 (kripkeToSuccinct birthdayModel)  world $ Knows 1 $ Neg
     $ knowsDate 2 ) [714,716,814,815,817]
True
```

and false in the others

```
not $ all (\world -> modelCheck2 (kripkeToSuccinct birthdayModel)  world $ Neg $
    Knows 1 $ Neg $ knowsDate 2 ) [515,516,519,617,618]
False
```

Note that the above computations take considerable time to execute. In the following steps where we deal with a submodel of `birthdayModel` after dropping some worlds, execution times drop dramatically.
Bernard taking into account the public announcement just now, restricts his attention to the worlds for which the above statement holds. He then publicly announces that he now knows the date. Turns out this can be the case only in 716, 815 and 817.

```
all (\world -> modelCheck2 (kripkeToSuccinct $ dropWorlds birthdayModel
    [515,516,519,617,618])  world $ knowsDate 2 ) [716,815,817]
True
```

While it fails on 714 and 814.

```
not $ all (\world -> modelCheck2 (kripkeToSuccinct $ dropWorlds birthdayModel
    [515,516,519,617,618])  world $ Neg $ knowsDate 2 ) [714,814]
False
```

Finally Albert states that he does know the date too. This holds true only on 716.

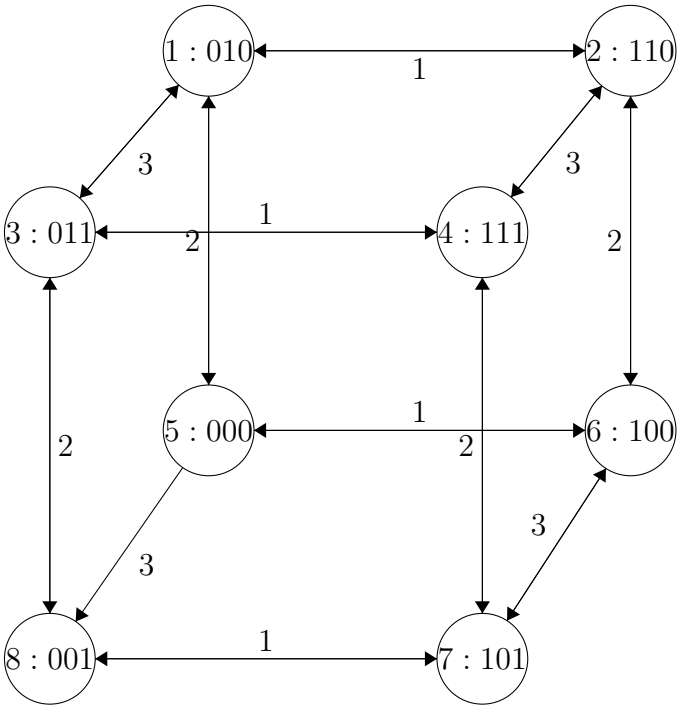
```
all (\world -> modelCheck2 (kripkeToSuccinct $ dropWorlds birthdayModel
    [515,516,519,617,618,714,814])  world $ knowsDate 1 ) [716]
True

not $ all (\world -> modelCheck2 (kripkeToSuccinct $ dropWorlds birthdayModel
    [515,516,519,617,618,714,814])  world $ Neg $ knowsDate 1 ) [815,817]
False
```

Therefore Cheryl's birthday is on the 16th of July.

5.2 Muddy Children

Below is the initial arrangement of 3 muddy children with all possible combinations of muddy (1) and not muddy (0).



The reflexive edges on each world are hidden towards a clearer picture.

```
mdModel :: KripkeM
mdModel = KMo
            [(1, relationWrap $ nub $ reflexize $ symmetrize [(1,2), (3,4), (5,6),
                 (8,7)]),
             (2, relationWrap $ nub $ reflexize $ symmetrize [(1,5), (2,6), (3,8),
                  (4,7)]),
             (3, relationWrap $ nub $ reflexize $ symmetrize [(1,3), (2,4), (5,8),
                  (6,7)])]
            [(1,[2]), (2,[1,2]), (3,[2,3]), (4,[1,2,3]), (5, []), (6,[1]), (7,
                [1,3]), (8,[3])]
```

At first the father announces that there is at least one muddy child. This means that world 5 is to be dropped. The next public announcement amounts to no child knowing whether their forehead is muddy. This is true on worlds 2, 3, 4 and 7.

```
all (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5] )   (fst x) $
    Neg (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [2,3,4,7], y <- [1,2,3]]
True

all (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5] )   (fst x) $
    Neg (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [1], y <- [1,2,3]]
False

all (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5] )   (fst x) $
    Neg (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [6], y <- [1,2,3]]
False

all (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5] )   (fst x) $
    Neg (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [8], y <- [1,2,3]]
False
```

We can then drop worlds 1, 6 and 8. The next announcement conveys that there is a child that now knows their forehead to be dirty.

```
any (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5,1,6,8] )   (fst x
    ) (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [2], y <- [1,2,3]]
True

any (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5,1,6,8] )   (fst x
    ) (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [3], y <- [1,2,3]]
True

any (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5,1,6,8] )   (fst x
    ) (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [7], y <- [1,2,3]]
True

any (\x -> modelCheck2 (kripkeToSuccinct $ dropWorlds mdModel [5,1,6,8] )   (fst x
    ) (Knows (snd x)  (P (snd x))) ) [(x,y) | x <- [4], y <- [1,2,3]]
False
```

Looking at the valuations of worlds 2, 3 and 7 it becomes clear that there are exactly two muddy children in any case.

6 Translating to and back from a Succinct Model

Using `kripkeToSuccinct` to translate some Kripke model M to a Succinct model N and then `succinctToKripke2` on N gives us back M. Note that the final result contains the nominal propositions which were not present in the original M. Below are example executions on the Muddy children and Cheryl's birthday Kripke models.

```
succinctToKripke2 $ kripkeToSuccinct  mdModel
KMo [(1,[(8,[8,7]),(7,[8,7]),(6,[6,5]),(5,[6,5]),(4,[4,3]),(3,[4,3]),(2,[2,1])
    ,(1,[2,1])]),(2,[(8,[8,3]),(7,[7,4]),(6,[6,2]),(5,[5,1]),(4,[7,4]),(3,[8,3])
    ,(2,[6,2]),(1,[5,1])]),(3,[(8,[8,5]),(7,[7,6]),(6,[7,6]),(5,[8,5]),(4,[4,2])
    ,(3,[3,1]),(2,[4,2]),(1,[3,1])])] [(8,[-8,3]),(7,[-7,1,3]),(6,[-6,1]),(5,[-5])
    ,(4,[-4,1,2,3]),(3,[-3,2,3]),(2,[-2,1,2]),(1,[-1,2])]

succinctToKripke2 $ kripkeToSuccinct birthdayModel
KMo [(1,[(817,[817,815,814]),(815,[817,815,814]),(814,[817,815,814])
    ,(716,[716,714]),(714,[716,714]),(618,[618,617]),(617,[618,617])
    ,(519,[519,516,515]),(516,[519,516,515]),(515,[519,516,515])])
    ,(2,[(817,[817,617]),(815,[815,515]),(814,[814,714]),(716,[716,516])
    ,(714,[814,714]),(618,[618]),(617,[817,617]),(519,[519]),(516,[716,516])
    ,(515,[815,515])])] [(817,[-817,817]),(815,[-815,815]),(814,[-814,814])
    ,(716,[-716,716]),(714,[-714,714]),(618,[-618,618]),(617,[-617,617])
    ,(519,[-519,519]),(516,[-516,516]),(515,[-515,515])]
```

If we disregard the nominal propositions, these are identical to the original models.

7 Conclusion

The model checking on the examples of the previous sections gives the expected results. Therefore, our implementation of the translation from Kripke to Succinct and our model checking algorithm for succinct models is likely to be correct.

Translating a Kripke to Succinct and then back to Kripke gives us the original model plus the nominal propositions so our claim that we get an isomorphic model holds.

8 Further work

A natural extension would be the implementation of the Succinct Event Model, the Product Update and the model checking algorithm thereof. To that end, the current design with the nominal propositions being negations of the world numbers should be swapped for a more robust one, which does not depend on the sign of the integers. This is because the current approach would not generalize well if we had to apply successive product updates.

References

[1] Alexandru Baltag and Bryan Renne. "Dynamic Epistemic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016.

[2] Tristan Charrier and François Schwarzentruber. *A Succinct Language for Dynamic Epistemic Logic (long version)*. Research Report. Irisa ; Ens Rennes, Mar. 2017. URL: `https://hal.archives-ouvertes.fr/hal-01487001`.

[3] Malvin Gattinger. "New Directions in Model Checking Dynamic Epistemic Logic". PhD thesis. University of Amsterdam, 2018. ISBN: 978-94-028-1025-7. URL: `https://malv.in/phdthesis`.

[4] Johan Van Benthem et al. "Symbolic model checking for dynamic epistemic logic". In: *International Workshop on Logic, Rationality and Interaction*. Springer. 2015, pp. 366–378.