SMCDEL – An Implementation of Symbolic Model Checking for Dynamic Epistemic Logic with Binary Decision Diagrams

Malvin Gattinger
ILLC, University of Amsterdam
malvin@w4eg.eu

Last Update: Wednesday 25th May, 2016

Abstract

We present *SMCDEL*, a symbolic model checker for Dynamic Epistemic Logic (DEL) implemented in Haskell. At its core is a translation of epistemic and dynamic formulas to boolean formulas which are represented as Binary Decision Diagrams (BDDs). Ideas underlying this implementation have been developed as joint work with Johan van Benthem, Jan van Eijck and Kaile Su [2].

The report is structured as follows.

In the Section 1 we recapitulate the syntax and intended meaning of DEL and define a data type for formulas. Section 2 describes the well-known semantics for DEL on Kripke models. We give a minimal implementation of explicit model checking as a reference.

Section 3 introduces the idea of knowledge structures and contains the main functions of our symbolic model checker. In Section 4 we give methods to go back and forth between the two semantics, both for models and actions. This shows in which sense and why the semantics are equivalent and why knowledge structures can be used to do symbolic model checking for `S5` DEL, also with its original semantics. To check that the implementations are correct we provide methods for automated randomized testing in Section 5 using QuickCheck.

In Section 6 we show how SMCDEL can be used. We go through various examples that are common in the literature both on DEL and model checking: Muddy Children, Drinking Logicians, Dining Cryptographers, Russian Cards and Sum and Product. These examples also suggest themselves as benchmarks which we will do in Section 7 to compare the different versions of our model checker to the existing tools DEMO-S5 and MCMAS.

In Section 8 we provide a standalone executable which reads simple text files with knowledge structures and formulas to be checked. This program makes the basic functionality of our model checker usable without any knowledge of Haskell. Additionally, Section 9 implements a web interface.

The last section discusses future work, both on concrete improvements for SMCDEL and on theoretical aspects of our framework.

The appendix has some installation guidelines, a helper functions module and an implementation of a number triangle analysis of the Muddy Children problem [22].

The report is given in literate Haskell style, including all source code and the results of example programs directly in the text.

SMCDEL is released as free software under the GPL.

See `https://github.com/jrclogic/SMCDEL` for the latest version.

Contents

1 The Language of Dynamic Epistemic Logic

This module defines the language of DEL. Keeping the syntax definition separate from the semantics allows us to use the same language throughout the whole report, both for the explicit and the symbolic model checkers.

```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}

module SMCDEL.Language where
import Data.List (nub,intercalate,(\\))
import Data.Maybe (fromJust)
import Test.QuickCheck
import SMCDEL.Internal.TexDisplay
```

Propositions are represented as integers in Haskell. Agents are strings.

```
data Prp = P Int  deriving (Eq,Ord,Show)
instance Enum Prp where
  toEnum = P
  fromEnum (P n) = n

instance Arbitrary Prp where
  arbitrary = P <$> choose (0,4)

freshp :: [Prp] -> Prp
freshp [] = P 1
freshp prps = P (maximum (map fromEnum prps) + 1)

type Agent = String

alice,bob,carol :: Agent
alice   = "Alice"
bob     = "Bob"
carol   = "Carol"

data AgAgent = Ag Agent deriving (Eq,Ord,Show)

instance Arbitrary AgAgent where
  arbitrary = oneof $ map (pure.Ag.(show)) $ [1..(5::Integer)]

instance Arbitrary [AgAgent] where
  arbitrary = sublistOf $ map (Ag.(show)) [1..(5::Integer)]
```

Definition 1. *The language $\mathcal{L}(V)$ for a set of propositions V and a finite set of agents I is given by*

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \bigwedge \Phi \mid \bigvee \Phi \mid \bigoplus \Phi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi \mid \forall P \varphi \mid \exists P \varphi \mid K_i \varphi \mid C_\Delta \varphi \mid [!\varphi]\varphi \mid [\Delta!\varphi]\varphi$$

where $p \in V$, $P \subseteq V$, $|P| < \omega$, $\Phi \subseteq \mathcal{L}_{\mathsf{DEL}}$, $|\Phi| < \omega$, $i \in I$ and $\Delta \subset I$. We also write $\varphi \land \psi$ for $\bigwedge\{\varphi,\psi\}$ and $\varphi \lor \psi$ for $\bigvee\{\varphi,\psi\}$. The boolean *formulas are those without K_i, C_Δ, $[!\varphi]$ and $[_\Delta!\varphi]$.*

Hence, a formula can be (in this order): The constant top or bottom, an atomic proposition, a negation, a conjunction, a disjunction, an exclusive or, an implication, a bi-implication, a universal or existential quantification over a set of propositions, or a statement about knowledge, common-knowledge, a public announcement or an announcement to a group.

Some of these connectives are inter-definable, for example $\varphi \leftrightarrow \psi$ and $\bigwedge\{\psi \to \varphi, \varphi \to \psi\}$ are equivalent according to all semantics which we will use here. Another example are $C_{\{i\}}\varphi$ and $K_i\varphi$. Hence we could shorten Definition 1 and treat some connectives as abbreviations. This would lead to brevity and clarity in the formal definitions, but also to a decrease in performance of our model checking implementations. To continue with the first example: If we have Binary Decision Diagrams (BDDs) for φ and ψ, computing the BDD for $\varphi \leftrightarrow \psi$ in one operation by calling the appropriate method of a BDD package will be faster than rewriting it to a conjunction of two implications and then making three calls to these corresponding functions of the BDD package.

Definition 2 (Whether-Formulas). *We extend our language with abbreviations for "knowing whether" and "announcing whether":*

$$K_i^? \varphi := \bigvee \{K_i \varphi, K_i(\neg\varphi)\}$$

$$[?!\varphi]\psi := \bigwedge \{\varphi \to [!\varphi]\psi, \neg\varphi \to [!\neg\varphi]\psi\}$$

$$[I?!\varphi]\psi := \bigwedge \{\varphi \to [I!\varphi]\psi, \neg\varphi \to [I!\neg\varphi]\psi\}$$

In Haskell we represent formulas using the following data type. Note that – also for performance reasons – also the three "whether" operators occur as primitives and not as abbreviations.

```
data Form
  = Top                         -- ^ True Constant
  | Bot                         -- ^ False Constant
  | PrpF Prp                    -- ^ Atomic Proposition
  | Neg Form                    -- ^ Negation
  | Conj [Form]                 -- ^ Conjunction
  | Disj [Form]                 -- ^ Disjunction
  | Xor [Form]                  -- ^ n-ary X-OR
  | Impl Form Form              -- ^ Implication
  | Equi Form Form              -- ^ Bi-Implication
  | Forall [Prp] Form           -- ^ Boolean Universal Quantification
  | Exists [Prp] Form           -- ^ Boolean Existential Quantification
  | K Agent Form                -- ^ Knowing that
  | Ck [Agent] Form             -- ^ Common knowing that
  | Kw Agent Form               -- ^ Knowing whether
  | Ckw [Agent] Form            -- ^ Common knowing whether
  | PubAnnounce Form Form       -- ^ Public announcement that
  | PubAnnounceW Form Form      -- ^ Public announcement whether
  | Announce [Agent] Form Form  -- ^ (Semi-)Private announcement that
  | AnnounceW [Agent] Form Form -- ^ (Semi-)Private announcement whether
  deriving (Eq,Ord,Show)

showSet :: Show a => [a] -> String
showSet xs = intercalate "," (map show xs)

-- | Pretty print a formula
ppForm :: Form -> String
ppForm Top = "T"
ppForm Bot = "F"
ppForm (PrpF (P n))  = show n
ppForm (Neg f)       = "~" ++ ppForm f
ppForm (Conj fs)     = "(" ++ intercalate " & " (map ppForm fs) ++ ")"
ppForm (Disj fs)     = "(" ++ intercalate " | " (map ppForm fs) ++ ")"
ppForm (Xor fs)      = "XOR{" ++ intercalate "," (map ppForm fs) ++ "}"
ppForm (Impl f g)    = "(" ++ ppForm f ++ "->" ++ ppForm g ++ ")"
ppForm (Equi f g)    = ppForm f ++ "=" ++ ppForm g
ppForm (Forall ps f) = "Forall {" ++ showSet ps ++ "}: " ++ ppForm f
ppForm (Exists ps f) = "Exists {" ++ showSet ps ++ "}: " ++ ppForm f
ppForm (K i f)     = "K " ++ i ++ " " ++ ppForm f
ppForm (Ck is f)   = "Ck " ++ intercalate "," is ++ " " ++ ppForm f
ppForm (Kw i f)    = "Kw " ++ i ++ " " ++ ppForm f
ppForm (Ckw is f) = "Ckw " ++ intercalate "," is ++ " " ++ ppForm f
ppForm (PubAnnounce f g)  = "[! " ++ ppForm f ++ "] " ++ ppForm g
ppForm (PubAnnounceW f g) = "[?! " ++ ppForm f ++ "] " ++ ppForm g
ppForm (Announce is f g)  = "[" ++ intercalate ", " is ++ " ! " ++ ppForm f ++ "]" ++
    ppForm g
ppForm (AnnounceW is f g) = "[" ++ intercalate ", " is ++ " ?! " ++ ppForm f ++ "]" ++
    ppForm g
```

We often want to check the result of multiple announcements after each other. Hence we define an abbreviation for such sequences of announcements using `foldr`.

```
pubAnnounceStack :: [Form] -> Form -> Form
pubAnnounceStack = flip $ foldr PubAnnounce

pubAnnounceWhetherStack :: [Form] -> Form -> Form
pubAnnounceWhetherStack = flip $ foldr PubAnnounceW
```

The following abbreviates that exactly a given subset of a set of propositions is true.

```
booloutofForm :: [Prp] -> [Prp] -> Form
booloutofForm ps qs = Conj $ [ PrpF p | p <- ps ] ++ [ Neg $ PrpF r | r <- qs \\ ps ]
```

The function `substit` below substitutes a formula for a proposition. As a safety measure this method will fail whenever the proposition to be replaced occurs in a quantifier. All other cases are done by recursion. The function `substitSet` applies multiple substitutions after each other. Note that this is *not* the same as simultaneous substitution.

```
substit :: Prp -> Form -> Form -> Form
substit _ _   Top            = Top
substit _ _   Bot            = Bot
substit q psi (PrpF p)       = if p==q then psi else PrpF p
substit q psi (Neg form)     = Neg (substit q psi form)
substit q psi (Conj forms)   = Conj (map (substit q psi) forms)
substit q psi (Disj forms)   = Disj (map (substit q psi) forms)
substit q psi (Xor  forms)   = Xor  (map (substit q psi) forms)
substit q psi (Impl f g)     = Impl (substit q psi f) (substit q psi g)
substit q psi (Equi f g)     = Equi (substit q psi f) (substit q psi g)
substit q psi (Forall ps f) = if q `elem` ps
  then error ("substit failed: Substituens "++ show q ++ " in 'Forall " ++ show ps)
  else Forall ps (substit q psi f)
substit q psi (Exists ps f) = if q `elem` ps
  then error ("substit failed: Substituens " ++ show q ++ " in 'Exists " ++ show ps)
  else Exists ps (substit q psi f)
substit q psi (K  i f)      = K  i (substit q psi f)
substit q psi (Kw i f)      = Kw i (substit q psi f)
substit q psi (Ck ags f)    = Ck ags (substit q psi f)
substit q psi (Ckw ags f)   = Ckw ags (substit q psi f)
substit q psi (PubAnnounce f g)   = PubAnnounce (substit q psi f) (substit q psi g)
substit q psi (PubAnnounceW f g)  = PubAnnounceW (substit q psi f) (substit q psi g)
substit q psi (Announce ags f g)  = Announce ags (substit q psi f) (substit q psi g)
substit q psi (AnnounceW ags f g) = AnnounceW ags (substit q psi f) (substit q psi g)

substitSet :: [(Prp,Form)] -> Form -> Form
substitSet [] f = f
substitSet ((q,psi):rest) f = substitSet rest (substit q psi f)
```

Another helper function allows us to replace propositions in a formula. In contrast to the previous substitution function this one *is* simultaneous.

```
replPsInF :: [(Prp,Prp)] -> Form -> Form
replPsInF _       Top       = Top
replPsInF _       Bot       = Bot
replPsInF repl (PrpF p)     | p `elem` map fst repl = PrpF (fromJust $ lookup p repl)
                            | otherwise = PrpF p
replPsInF repl (Neg f)      = Neg $ replPsInF repl f
replPsInF repl (Conj fs)    = Conj $ map (replPsInF repl) fs
replPsInF repl (Disj fs)    = Disj $ map (replPsInF repl) fs
replPsInF repl (Xor  fs)    = Xor  $ map (replPsInF repl) fs
replPsInF repl (Impl f g)   = Impl (replPsInF repl f) (replPsInF repl g)
replPsInF repl (Equi f g)   = Equi (replPsInF repl f) (replPsInF repl g)
replPsInF repl (Forall ps f) = Forall (map (fromJust . flip lookup repl) ps) (replPsInF
    repl f)
replPsInF repl (Exists ps f) = Exists (map (fromJust . flip lookup repl) ps) (replPsInF
    repl f)
replPsInF repl (K i f)      = K i (replPsInF repl f)
replPsInF repl (Kw i f)     = Kw i (replPsInF repl f)
replPsInF repl (Ck ags f)   = Ck ags (replPsInF repl f)
replPsInF repl (Ckw ags f) = Ckw ags (replPsInF repl f)
replPsInF repl (PubAnnounce f g)   = PubAnnounce   (replPsInF repl f) (replPsInF repl g)
replPsInF repl (PubAnnounceW f g)  = PubAnnounceW  (replPsInF repl f) (replPsInF repl g)
replPsInF repl (Announce ags f g)  = Announce  ags (replPsInF repl f) (replPsInF repl g)
replPsInF repl (AnnounceW ags f g) = AnnounceW ags (replPsInF repl f) (replPsInF repl g)
```

The following helper function gets all propositions occurring in a formula.

```
propsInForm :: Form -> [Prp]
propsInForm Top                 = []
propsInForm Bot                 = []
propsInForm (PrpF p)            = [p]
propsInForm (Neg f)             = propsInForm f
propsInForm (Conj fs)           = nub $ concatMap propsInForm fs
propsInForm (Disj fs)           = nub $ concatMap propsInForm fs
propsInForm (Xor  fs)           = nub $ concatMap propsInForm fs
propsInForm (Impl f g)          = nub $ concatMap propsInForm [f,g]
propsInForm (Equi f g)          = nub $ concatMap propsInForm [f,g]
propsInForm (Forall ps f)       = nub $ ps ++ propsInForm f
propsInForm (Exists ps f)       = nub $ ps ++ propsInForm f
propsInForm (K _ f)             = propsInForm f
propsInForm (Kw _ f)            = propsInForm f
propsInForm (Ck _ f)            = propsInForm f
propsInForm (Ckw _ f)           = propsInForm f
propsInForm (Announce _ f g)    = nub $ propsInForm f ++ propsInForm g
propsInForm (AnnounceW _ f g)   = nub $ propsInForm f ++ propsInForm g
propsInForm (PubAnnounce f g)   = nub $ propsInForm f ++ propsInForm g
propsInForm (PubAnnounceW f g) = nub $ propsInForm f ++ propsInForm g

propsInForms :: [Form] -> [Prp]
propsInForms fs = nub $ concatMap propsInForm fs

instance TexAble Prp where
  tex (P 0) = " p "
  tex (P n) = " p_{" ++ show n ++ "} "

instance TexAble [Prp] where
  tex [] = " \\varnothing "
  tex ps = "\\{" ++ intercalate "," (map tex ps) ++ "\\}"
```

The following algorithm simplifies a formula using boolean equivalences. For example it removes double negations and "bubbles up" $\bot$ and $\top$ in conjunctions and disjunctions respectively.

```
simplify :: Form -> Form
simplify f = if simStep f == f then f else simplify (simStep f)

simStep :: Form -> Form
simStep Top              = Top
simStep Bot              = Bot
simStep (PrpF p)         = PrpF p
simStep (Neg Top)        = Bot
simStep (Neg Bot)        = Top
simStep (Neg (Neg f)) = simStep f
simStep (Neg f)          = Neg $ simStep f
simStep (Conj [])        = Top
simStep (Conj [f])       = simStep f
simStep (Conj fs)        | Bot `elem` fs = Bot
                         | or [ Neg f `elem` fs | f <- fs ] = Bot
                         | otherwise      = Conj (nub $ map simStep (filter (Top /=) fs))
simStep (Disj [])        = Bot
simStep (Disj [f])       = simStep f
simStep (Disj fs)        | Top `elem` fs = Top
                         | or [ Neg f `elem` fs | f <- fs ] = Top
                         | otherwise      = Disj (nub $ map simStep (filter (Bot /=) fs))
simStep (Xor  [])        = Bot
simStep (Xor  [f])       = Neg $ simStep f
simStep (Xor  fs)        = Xor (map simStep fs)
simStep (Impl Bot _)     = Top
simStep (Impl _ Top)     = Top
simStep (Impl Top f)     = simStep f
simStep (Impl f Bot)     = Neg (simStep f)
simStep (Impl f g)       | f==g        = Top
                         | otherwise = Impl (simStep f) (simStep g)
simStep (Equi Top f)     = simStep f
simStep (Equi Bot f)     = Neg (simStep f)
simStep (Equi f Top)     = simStep f
simStep (Equi f Bot)     = Neg (simStep f)
simStep (Equi f g)       | f==g        = Top
                         | otherwise = Equi (simStep f) (simStep g)
```

```
simStep (Forall ps f) = Forall ps (simStep f)
simStep (Exists ps f) = Exists ps (simStep f)
simStep (K a f)       = K a (simStep f)
simStep (Kw a f)      = Kw a (simStep f)
simStep (Ck ags f)    = Ck ags (simStep f)
simStep (Ckw ags f)   = Ckw ags (simStep f)
simStep (PubAnnounce Top f) = simStep f
simStep (PubAnnounce Bot _) = Top
simStep (PubAnnounce  f g)  = PubAnnounce  (simStep f) (simStep g)
simStep (PubAnnounceW f g)  = PubAnnounceW (simStep f) (simStep g)
simStep (Announce  ags f g) = Announce  ags (simStep f) (simStep g)
simStep (AnnounceW ags f g) = AnnounceW ags (simStep f) (simStep g)
```

We end this module with a function that generates LaTeX code for a formula.

```
texForm :: Form -> String
texForm Top            = "\\top "
texForm Bot            = "\\bot "
texForm (PrpF p)       = tex p
texForm (Neg (PubAnnounce f (Neg g))) = "\\langle !" ++ texForm f ++ " \\rangle " ++
    texForm g
texForm (Neg f)        = "\\lnot " ++ texForm f
texForm (Conj [])      = "\\top "
texForm (Conj [f])     = texForm f
texForm (Conj [f,g])   = " ( " ++ texForm f ++ " \\land " ++ texForm g ++" ) "
texForm (Conj fs)      = "\\bigwedge \\{\n" ++ intercalate "," (map texForm fs) ++" \\} "
texForm (Disj [])      = "\\bot "
texForm (Disj [f])     = texForm f
texForm (Disj [f,g])   = " ( " ++ texForm f ++ " \\lor "++ texForm g ++ " ) "
texForm (Disj fs)      = "\\bigvee \\{\n " ++ intercalate "," (map texForm fs) ++ " \\} "
texForm (Xor [])       = "\\bot "
texForm (Xor [f])      = texForm f
texForm (Xor [f,g])    = " ( " ++ texForm f  ++ " \\oplus " ++ texForm g ++ " ) "
texForm (Xor fs)       = "\\bigoplus \\{\n" ++ intercalate "," (map texForm fs) ++ " \\} "
texForm (Equi f g)     = " ( "++ texForm f ++" \\leftrightarrow "++ texForm g ++" ) "
texForm (Impl f g)     = " ( "++ texForm f ++" \\rightarrow "++ texForm g ++" ) "
texForm (Forall ps f) = " \\forall " ++ tex ps ++ " " ++ texForm f
texForm (Exists ps f) = " \\exists " ++ tex ps ++ " " ++ texForm f
texForm (K i f)        = "K_{\\text{" ++ i ++ "}} " ++ texForm f
texForm (Kw i f)       = "K^?_{\\text{" ++ i ++ "}} " ++ texForm f
texForm (Ck ags f)     = "Ck_{\\{\n" ++ intercalate "," ags ++ "\n\\}} " ++ texForm f
texForm (Ckw ags f)    = "Ck^?_{\\{\n" ++ intercalate "," ags ++ "\n\\}} " ++ texForm f
texForm (PubAnnounce f g)   = "[!" ++ texForm f ++ "] " ++ texForm g
texForm (PubAnnounceW f g)  = "[?!" ++ texForm f ++ "] " ++ texForm g
texForm (Announce ags f g)  = "[" ++ intercalate "," ags ++ "!" ++ texForm f ++ "] " ++
    texForm g
texForm (AnnounceW ags f g) = "[" ++ intercalate "," ags ++ "?!" ++ texForm f ++ "] " ++
    texForm g

instance TexAble Form where
  tex = texForm
```

For example, consider this rather unnatural formula:

```
testForm :: Form
testForm = Forall [P 3] $ Equi (Disj [Bot,PrpF $ P 3,Bot]) (Conj [Top,Xor [Top,Kw alice (
    PrpF (P 4))], AnnounceW [alice,bob] (PrpF (P 5)) (Kw bob $ PrpF (P 5))  ])
```

$$\forall\{p_3\}(\bigvee\{\bot, p_3, \bot\} \leftrightarrow \bigwedge\{\top, (\top \oplus K^?_{\text{Alice}} p_4), [Alice, Bob?!p_5] K^?_{\text{Bob}} p_5\})$$

And this simplification:

$$\forall\{p_3\}(p_3 \leftrightarrow ((\top \oplus K^?_{\text{Alice}} p_4) \land [Alice, Bob?!p_5] K^?_{\text{Bob}} p_5))$$

The following will allow us to run QuickCheck on functions that take DEL formulas as input. We first provide an instance for the Boolean fragment, wrapped with the BF constructor.

```haskell
data BF = BF Form deriving (Eq,Ord,Show)

instance Arbitrary BF where
  arbitrary = sized randomboolform

randomboolform ::  Int -> Gen BF
randomboolform sz = BF <$> bf' sz' where
  maximumvar = 1000
  sz' = min maximumvar sz
  bf' 0 = (PrpF . P) <$> choose (0, sz')
  bf' n = oneof [ pure SMCDEL.Language.Top
                , pure SMCDEL.Language.Bot
                , (PrpF . P) <$> choose (0, sz')
                , Neg <$> st
                , (\x y -> Conj [x,y]) <$> st <*> st
                , (\x y -> Disj [x,y]) <$> st <*> st
                , Impl <$> st <*> st
                , Equi <$> st <*> st
                , (\x y -> Xor [x,y]) <$> st <*> st
                , (\m f -> Exists [P m] f) <$> choose (0, maximumvar) <*> st
                , (\m f -> Forall [P m] f) <$> randomvar <*> st
                ]
    where
      st = bf' (n `div` 2)
      randomvar = elements [0..maximumvar]
```

We can now run things like `quickCheckResult ( (BF f) -> f == f)`.

The following is a general Arbitrary instance for formulas. It is used in Section 5 below. Quantifiers and common knowledge operators are disabled for performance reasons.

```haskell
instance Arbitrary Form where
  arbitrary = sized form where
    form 0 = oneof [ pure Top
                   , pure Bot
                   , PrpF <$> arbitrary ]
    form n = oneof [ pure SMCDEL.Language.Top
                   , pure SMCDEL.Language.Bot
                   , PrpF <$> arbitrary
                   , Neg <$> form n'
                   , Conj <$> listOf (form n')
                   , Disj <$> listOf (form n')
                   , Xor  <$> listOf (form n')
                   , Impl <$> form n' <*> form n'
                   , Equi <$> form n' <*> form n'
                   -- , Exists <$> arbitrary <*> form n'
                   -- , Forall <$> arbitrary <*> form n'
                   , K   <$> arbitraryAg <*> form n'
                   -- , Ck  <$> arbitraryAgs <*> form n'
                   , Kw  <$> arbitraryAg <*> form n'
                   -- , Ckw <$> arbitraryAgs <*> form n'
                   , PubAnnounce  <$> form n' <*> form n'
                   , PubAnnounceW <$> form n' <*> form n'
                   , Announce  <$> arbitraryAgs <*> form n' <*> form n'
                   , AnnounceW <$> arbitraryAgs <*> form n' <*> form n'
                   ]
      where
        n' = n `div` 4
        arbitraryAg = (\(Ag i) -> i) <$> arbitrary
        arbitraryAgs = sublistOf $ map show [1..(5::Integer)]
```

2 DEL Semantics on Kripke Models

We start with a quick summary of the standard semantics for DEL on Kripke models. The module of this section provides a very simple explicit state model checker. It is mainly provided as a basis for the translation methods in Section 4 and not meant to be used in practice otherwise. A more advanced and user-friendly explicit state model checker for DEL is DEMO from [15] which we will also use later on.

```
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, FlexibleContexts #-}

module SMCDEL.Explicit.Simple where
import Control.Arrow (second,(&&&))
import Data.List
import SMCDEL.Language
import SMCDEL.Internal.TexDisplay
import SMCDEL.Internal.Help (alleq,fusion,apply)
```

2.1 Kripke Models

Definition 3. *A* Kripke model *for a set of agents $I = \{1, \dots, n\}$ is a tuple $\mathcal{M} = (W, \pi, \mathcal{K}_1, \dots, \mathcal{K}_n)$, where W is a set of* worlds, *π associates with each world a truth assignment to the primitive propositions, so that $\pi(w)(p) \in \{\top, \bot\}$ for each world w and primitive proposition p, and $\mathcal{K}_1, \dots, \mathcal{K}_n$ are binary accessibility relations on W. By convention, $W^{\mathcal{M}}$, $\mathcal{K}_i^{\mathcal{M}}$ and $\pi^{\mathcal{M}}$ are used to refer to the components of $\mathcal{M}$. We omit the superscript $\mathcal{M}$ if it is clear from context. Finally, let $\mathcal{C}_\Delta^{\mathcal{M}}$ be the transitive closure of $\bigcup_{i \in \Delta} \mathcal{K}_i^{\mathcal{M}}$.*

A pointed Kripke model *is a pair $(\mathcal{M}, w)$ consisting of a Kripke model and a world $w \in W^{\mathcal{M}}$. A model $\mathcal{M}$ is called an* S5 Kripke model *iff, for every i, $\mathcal{K}_i^{\mathcal{M}}$ is an equivalence relation. A model $\mathcal{M}$ is called* finite *iff $W^{\mathcal{M}}$ is finite.*

The following data types capture Definition 3 in Haskell. Possible worlds (aka states) are represented by integers. Equivalence relations are modeled as partitions, i.e. lists of lists of states.

```
type State = Int

type Partition = [[State]]
type Assignment = [(Prp,Bool)]

data KripkeModel = KrM [State] [(Agent,Partition)] [(State,Assignment)] deriving (Eq,Ord,
    Show)
type PointedModel = (KripkeModel,State)
```

Definition 4. *Semantics for $\mathcal{L}(V)$* on pointed Kripke models *are given inductively as follows.*

1. $(\mathcal{M}, w) \vDash p$ *iff* $\pi^M(w)(p) = \top$.
2. $(\mathcal{M}, w) \vDash \neg\varphi$ *iff not* $(\mathcal{M}, w) \vDash \varphi$
3. $(\mathcal{M}, w) \vDash \varphi \land \psi$ *iff* $(\mathcal{M}, w) \vDash \varphi$ *and* $(\mathcal{M}, w) \vDash \psi$
4. $(\mathcal{M}, w) \vDash K_i\varphi$ *iff for all* $w' \in W$, *if* $w\mathcal{K}_i^M w'$, *then* $(\mathcal{M}, w') \vDash \varphi$.
5. $(\mathcal{M}, w) \vDash C_\Delta\varphi$ *iff for all* $w' \in W$, *if* $w\mathcal{C}_\Delta^{\mathcal{M}} w'$, *then* $(\mathcal{M}, w') \vDash \varphi$.
6. $(\mathcal{M}, w) \vDash [\psi]\varphi$ *iff* $(\mathcal{M}, w) \vDash \psi$ *implies* $(\mathcal{M}^\psi, w) \vDash \varphi$ *where* $\mathcal{M}^\psi$ *is a new Kripke model defined by the set* $W^{\mathcal{M}^\psi} := \{w \in W^{\mathcal{M}} \mid (\mathcal{M}, w) \vDash \psi\}$, *the relations* $\mathcal{K}_i^{\mathcal{M}^\psi} := \mathcal{K}_i^M \cap (W^{\mathcal{M}^\psi})^2$ *and the valuation* $\pi^{\mathcal{M}^\psi}(w) := \pi^{\mathcal{M}}(w)$.
7. $(\mathcal{M}, w) \vDash [\psi]_\Delta\varphi$ *iff* $(\mathcal{M}, w) \vDash \psi$ *implies that* $(\mathcal{M}_\psi^\Delta, w) \vDash \varphi$ *where* $(\mathcal{M}_\psi^\Delta, w)$ *is a new Kripke model defined by the same set of worlds* $W^{\mathcal{M}_\psi^\Delta} := W^{\mathcal{M}}$, *modified relations such that*

- *if $i \in \Delta$, let $w\mathcal{K}_i^{\mathcal{M}_\psi^\Delta} w'$ iff (i) $w\mathcal{K}_i^{\mathcal{M}} w'$ and (ii) $(\mathcal{M}, w) \vDash \psi$ iff $(\mathcal{M}, w') \vDash \psi$*
- *otherwise, let $w\mathcal{K}_i^{\mathcal{M}_\psi^\Delta} w'$ iff $w\mathcal{K}_i^{\mathcal{M}} w'$*

and the same valuation $\pi^{\mathcal{M}_\psi^\Delta}(w) := \pi^{\mathcal{M}}(w)$.

These semantics can be translated to a model checking function `eval` in Haskell at follows. Note the typical recursion: All cases besides constants and atomic propositions call `eval` again.

```
eval :: PointedModel -> Form -> Bool
eval _ Top = True
eval _ Bot = False
eval (KrM _ _ val, cur) (PrpF p) = apply (apply val cur) p
eval pm (Neg form)     = not $ eval pm form
eval pm (Conj forms)   = all (eval pm) forms
eval pm (Disj forms)   = any (eval pm) forms
eval pm (Xor  forms)   = odd $ length (filter id $ map (eval pm) forms)
eval pm (Impl f g)     = not (eval pm f) || eval pm g
eval pm (Equi f g)     = eval pm f == eval pm g
eval pm (Forall ps f) = eval pm (foldl singleForall f ps) where
  singleForall g p = Conj [ substit p Top g, substit p Bot g ]
eval pm (Exists ps f) = eval pm (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
eval (m@(KrM _ rel _),w) (K ag form) = all (\w' -> eval (m,w') form) vs where
  vs = concat $ filter (elem w) (apply rel ag)
eval (m@(KrM _ rel _),w) (Kw ag form) = alleq (\w' -> eval (m,w') form) vs where
  vs = concat $ filter (elem w) (apply rel ag)
eval (m@(KrM _ rel _),w) (Ck ags form) = all (\w' -> eval (m,w') form) vs where
  vs    = concat $ filter (elem w) ckrel
  ckrel = fusion $ concat [ apply rel i | i <- ags ]
eval (m@(KrM _ rel _),w) (Ckw ags form) = alleq (\w' -> eval (m,w') form) vs where
  vs    = concat $ filter (elem w) ckrel
  ckrel = fusion $ concat [ apply rel i | i <- ags ]
eval pm (PubAnnounce form1 form2) =
  not (eval pm form1) || eval (pubAnnounce pm form1) form2
eval pm (PubAnnounceW form1 form2) =
  if eval pm form1
    then eval (pubAnnounce pm form1) form2
    else eval (pubAnnounce pm (Neg form1)) form2
eval pm (Announce ags form1 form2) =
  not (eval pm form1) || eval (announce pm ags form1) form2
eval pm (AnnounceW ags form1 form2) =
  if eval pm form1
    then eval (announce pm ags form1) form2
    else eval (announce pm ags (Neg form1)) form2
```

Public and group announcements are functions which take a pointed model and give us a new one. Because `eval` already checks whether an announcement is truthful before executing it we let the following two functions raise an error in case the announcement is false on the given model.

```
pubAnnounce :: PointedModel -> Form -> PointedModel
pubAnnounce pm@(m@(KrM sts rel val), cur) form =
  if eval pm form then (KrM newsts newrel newval, cur)
                  else error "pubAnnounce failed: Liar!"
  where
    newsts = filter (\s -> eval (m,s) form) sts
    newrel = map (second relfil) rel
    relfil = filter ([]/=) . map (filter (`elem` newsts))
    newval = filter (\p -> fst p `elem` newsts) val

announce :: PointedModel -> [Agent] -> Form -> PointedModel
announce pm@(m@(KrM sts rel val), cur) ags form =
  if eval pm form then (KrM sts newrel val, cur)
                  else error "announce failed: Liar!"
  where
    split ws = map sort.(\(x,y) -> [x,y]) $ partition ( \s -> eval (m,s) form) ws
    newrel = map nrel rel
    nrel (i,ri) | i `elem` ags = (i,filter ([]/=) (concatMap split ri))
                | otherwise    = (i,ri)
```

With a few lines we can also visualize our models using the module SMCDEL.Internal.TexDisplay. For example output, see Sections 6.1 and 6.2.

```
instance KripkeLike PointedModel where
  directed = const False
  getNodes (KrM ws _ val, _) = map (show &&& labelOf) ws where
    labelOf w = tex $ apply val w
  getEdges (KrM _ rel _, _) =
    nub $ concat $ concat $ concat [ [ [ [(a,show x,show y) | x<y] | x <- part, y <- part ]
         | part <- apply rel a ] | a <- map fst rel ]
  getActuals (KrM {}, cur) = [show cur]

instance TexAble PointedModel where tex = tex.ViaDot
```

2.2 Action Models

To model epistemic change in general we implement action models [1]. For now we only consider S5 action models without factual change.

Definition 5. *An* action model *for a given vocabulary V and set of agents $I = \{1, \dots, n\}$ is a tuple $\mathcal{A} = (A, \mathsf{pre}, R_1, \dots, R_n)$ where A is a set of so-called* action points*, $\mathsf{pre} : A \to \mathcal{L}(V)$ assigns to each action point a formula called its* precondition *and $R_1, \dots, R_n$ are binary relations on A. If all the relations are equivalence relations we call $\mathcal{A}$ an S5 action model.*

Given a Kripke model and an action model we define their product update *as $\mathcal{M} \times \mathcal{A} := (W', \pi', \mathcal{K}_1, \dots, \mathcal{K}_n)$ where $W' := \{(w, \alpha) \in W \times A \mid \mathcal{M}, w \vDash \mathsf{pre}(\alpha)\}$, $\pi'((w, \alpha)) := \pi(w)$ and $(v, \alpha)\mathcal{K}_i'(w, \beta)$ iff $v\mathcal{K}_i w$ and $\alpha R_i \beta$.*

For any $\alpha \in A$ we call $(\mathcal{A}, \alpha)$ a pointed *(S5) action model.*

```
data ActionModel = ActM [State] [(State,Form)] [(Agent,Partition)]
  deriving (Eq,Ord,Show)
type PointedActionModel = (ActionModel,State)

instance KripkeLike PointedActionModel where
  directed = const False
  getNodes (ActM as actval _, _) = map (show &&& labelOf) as where
    labelOf w = ppForm $ apply actval w
  getEdges (ActM _ _ rel, _) =
    nub $ concat $ concat $ concat [ [ [ [(a,show x,show y) | x<y] | x <- part, y <- part ]
         | part <- apply rel a ] | a <- map fst rel ]
  getActuals (ActM {}, cur) = [show cur]

instance TexAble PointedActionModel where tex = tex.ViaDot

productUpdate :: PointedModel -> PointedActionModel -> PointedModel
productUpdate pm@(m@(KrM oldstates oldrel oldval), oldcur) (ActM actions precon actrel,
    faction) =
  let
    startcount          = maximum oldstates + 1
    copiesOf (s,a)      = [ (s, a, a * startcount + s) | eval (m, s) (apply precon a) ]
    newstatesTriples    = concat [ copiesOf (s,a) | s <- oldstates, a <- actions ]
    newstates           = map (\(_,_,x) -> x) newstatesTriples
    newval              = map (\(s,_,t) -> (t, apply oldval s)) newstatesTriples
    listFor ag          = cartProd (apply oldrel ag) (apply actrel ag)
    newPartsFor ag      = [ cartProd as bs | (as,bs) <- listFor ag ]
    translSingle pair = filter (`elem` newstates) $ map (\(_,_,x) -> x) $ copiesOf pair
    transEqClass        = concatMap translSingle
    nTransPartsFor ag = filter (\x-> x/=[]) $ map transEqClass (newPartsFor ag)
    newrel              = [ (a, nTransPartsFor a)  | a <- map fst oldrel ]
    ((_,_,newcur):_)    = copiesOf (oldcur,faction)
    factTest            = apply precon faction
    cartProd xs ys      = [ (x,y) | x <- xs, y <- ys ]
  in case ( map fst oldrel == map fst actrel, eval pm factTest ) of
    (False, _) -> error "productUpdate failed: Agents of KrM and ActM are not the same!"
    (_, False) -> error "productUpdate failed: Actual precondition is false!"
    _          -> (KrM newstates newrel newval, newcur)
```

3 DEL Semantics on Knowledge Structures

In this section we implement an alternative semantics for $\mathcal{L}(V)$ and show how it allows a symbolic model checking algorithm. Our model checker can be used with two different BDD packages. Both are written in other languages than Haskell and therefore have to be used via bindings:

i) *CacBDD* [27], a modern BDD package with dynamic cache management implemented in C++. We use it via the library *HasCacBDD* [21] which provides Haskell-to-C-to-C++ bindings.

ii) *CUDD* [28], probably the best-known BDD library which is used many in other model checkers, including MCMAS [25], MCK [20] and NuSMV [6]. It is implemented in C and we use it via a binding library from `https://github.com/davidcock/cudd`.

The corresponding Haskell modules are `SMCDEL.Symbolic.HasCacBDD` and `SMCDEL.Symbolic.CUDD`. Here we list the *CacBDD* variant.

```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances, MultiParamTypeClasses,
    FlexibleContexts #-}

module SMCDEL.Symbolic.HasCacBDD where
import Data.HasCacBDD hiding (Top,Bot)
import Data.HasCacBDD.Visuals
import Data.List (sort,intercalate,(\\))
import System.IO (hPutStr, hGetContents, hClose)
import System.IO.Unsafe (unsafePerformIO)
import System.Process (runInteractiveCommand)
import SMCDEL.Internal.Help (alleq,apply,rtc,seteq)
import SMCDEL.Language
import SMCDEL.Internal.TexDisplay
```

We first link the boolean part of our language definition to functions of the BDD package. The following translates boolean formulas to BDDs and evaluates them with respect to a given set of true atomic propositions. The function will raise an error if it is given an epistemic or dynamic formula.

```
boolBddOf :: Form -> Bdd
boolBddOf Top           = top
boolBddOf Bot           = bot
boolBddOf (PrpF (P n))  = var n
boolBddOf (Neg form)    = neg$ boolBddOf form
boolBddOf (Conj forms)  = conSet $ map boolBddOf forms
boolBddOf (Disj forms)  = disSet $ map boolBddOf forms
boolBddOf (Xor forms)   = xorSet $ map boolBddOf forms
boolBddOf (Impl f g)    = imp (boolBddOf f) (boolBddOf g)
boolBddOf (Equi f g)    = equ (boolBddOf f) (boolBddOf g)
boolBddOf (Forall ps f) = forallSet (map fromEnum ps) (boolBddOf f)
boolBddOf (Exists ps f) = existsSet (map fromEnum ps) (boolBddOf f)
boolBddOf _             = error "boolBddOf failed: Not a boolean formula."

boolEval :: [Prp] -> Form -> Bool
boolEval truths form = result where
  values = map (\(P n) -> (n, P n `elem` truths)) (propsInForm form)
  bdd    = restrictSet (boolBddOf form) values
  result | bdd==top  = True
         | bdd==bot  = False
         | otherwise = error "boolEval failed: BDD leftover."
```

3.1 Knowledge Structures

Knowledge structures are a compact representation of S5 Kripke models. Their set of states is defined by a boolean formula and instead of epistemic relations we use observational variables. More explanations and proofs that they are indeed equivalent to S5 Kripke models can be found in [2].

Definition 6. *Suppose we have n agents. A* knowledge structure *is a tuple $\mathcal{F} = (V, \theta, O_1, \dots, O_n)$ where V is a finite set of propositional variables, θ is a boolean formula over V and for each agent i, $O_i \subseteq V$.*

Set V is the vocabulary *of $\mathcal{F}$. Formula θ is the* state law *of $\mathcal{F}$. It determines the set of states of $\mathcal{F}$ and may only contain boolean operators. The variables in O_i are called agent i's* observable variables. *An assignment over V that satisfies θ is called a* state *of $\mathcal{F}$. Any knowledge structure only has finitely many states. Given a state s of $\mathcal{F}$, we say that $(\mathcal{F}, s)$ is a* scene *and define the* local state *of an agent i at s as $s \cap O_i$.*

To interpret common knowledge we use the following definitions. Given a knowledge structure $(V, \theta, O_1, \dots, O_n)$ and a set of agents Δ, let $\mathcal{E}_\Delta$ be the relation on states of $\mathcal{F}$ defined by $(s,t) \in \mathcal{E}_\Delta$ iff there exists an $i \in \Delta$ with $s \cap O_i = t \cap O_i$. and let $\mathcal{E}_\mathcal{V}^*$ to denote the transitive closure of $\mathcal{E}_\mathcal{V}$.

In our data type for knowledge structures we represent the state law θ not as a formula but as a Binary Decision Diagram.

```
data KnowStruct = KnS [Prp] Bdd [(Agent,[Prp])] deriving (Eq,Show)
type KnState = [Prp]
type Scenario = (KnowStruct,KnState)

statesOf :: KnowStruct -> [KnState]
statesOf (KnS props lawbdd _) = map (sort.translate) resultlists where
  resultlists = map (map convToProp) $ allSatsWith (map (\(P n) -> n) props) lawbdd :: [[(
      Prp, Bool)]]
  convToProp (n,bool) = (P n,bool)
  translate l = map fst (filter snd l)

numberOfStates :: KnowStruct -> Int
numberOfStates (KnS ps lawbdd _) = satCountWith (map (\(P n) -> n) ps) lawbdd

restrictState :: KnState -> [Prp] -> KnState
restrictState s props = filter (`elem` props) s

shareknow :: KnowStruct -> [[Prp]] -> [(KnState,KnState)]
shareknow kns sets = filter rel [ (s,t) | s <- statesOf kns, t <- statesOf kns ] where
  rel (x,y) = or [ seteq (restrictState x set) (restrictState y set) | set <- sets ]

comknow :: KnowStruct -> [Agent] -> [(KnState,KnState)]
comknow kns@(KnS _ _ obs) ags = rtc $ shareknow kns (map (apply obs) ags)
```

Definition 7. *Semantics for $\mathcal{L}(V)$* on scenes *are defined inductively as follows.*

1. $(\mathcal{F}, s) \vDash p$ *iff* $s \vDash p$.
2. $(\mathcal{F}, s) \vDash \neg\varphi$ *iff not* $(\mathcal{F}, s) \vDash \varphi$
3. $(\mathcal{F}, s) \vDash \varphi \land \psi$ *iff* $(\mathcal{F}, s) \vDash \varphi$ *and* $(\mathcal{F}, s) \vDash \psi$
4. $(\mathcal{F}, s) \vDash K_i\varphi$ *iff for all t of $\mathcal{F}$, if $s \cap O_i = t \cap O_i$, then $(\mathcal{F}, t) \vDash \varphi$.*
5. $(\mathcal{F}, s) \vDash C_\Delta\varphi$ *iff for all t of $\mathcal{F}$, if $(s,t) \in \mathcal{E}_\Delta^*$, then $(\mathcal{F}, t) \vDash \varphi$.*
6. $(\mathcal{F}, s) \vDash [\psi]\varphi$ *iff* $(\mathcal{F}, s) \vDash \psi$ *implies* $(\mathcal{F}^\psi, s) \vDash \varphi$ *where $\|\psi\|_\mathcal{F}$ is given by Definition 8 and*
$$\mathcal{F}^\psi := (V, \theta \land \|\psi\|_\mathcal{F}, O_1, \dots, O_n)$$
7. $(\mathcal{F}, s) \vDash [\psi]_\Delta\varphi$ *iff* $(\mathcal{F}, s) \vDash \psi$ *implies* $(\mathcal{F}_\psi^\Delta, s \cup \{p_\psi\}) \vDash \varphi$ *where p_ψ is a new propositional variable, $\|\psi\|_\mathcal{F}$ is a boolean formula given by Definition 8 and*
$$\mathcal{F}_\psi^\Delta := (V \cup \{p_\psi\}, \theta \land (p_\psi \leftrightarrow \|\psi\|_\mathcal{F}), O_1^*, \dots, O_n^*)$$
where $O_i^* := \begin{cases} O_i \cup \{p_\psi\} & \textit{if } i \in \Delta \\ O_i & \textit{otherwise} \end{cases}$

If we have $(\mathcal{F}, s) \vDash \varphi$ for all states s of $\mathcal{F}$, then we say that φ is valid on $\mathcal{F}$ and write $\mathcal{F} \vDash \varphi$.

The following function `eval` implements these semantics. An important warning: This function is not a symbolic algorithm! It is a direct translation of Definition 7. In particular it calls `statesOf` which means that the set of stats is explicitly generated. The symbolic counterpart of `eval` is `evalViaBdd`, see below.

```
eval :: Scenario -> Form -> Bool
eval _       Top            = True
eval _       Bot            = False
eval (_,s)   (PrpF p)       = p `elem` s
eval (kns,s) (Neg form)     = not $ eval (kns,s) form
eval (kns,s) (Conj forms)   = all (eval (kns,s)) forms
eval (kns,s) (Disj forms)   = any (eval (kns,s)) forms
eval (kns,s) (Xor  forms)   = odd $ length (filter id $ map (eval (kns,s)) forms)
eval scn     (Impl f g)     = not (eval scn f) || eval scn g
eval scn     (Equi f g)     = eval scn f == eval scn g
eval scn     (Forall ps f) = eval scn (foldl singleForall f ps) where
  singleForall g p = Conj [ substit p Top g, substit p Bot g ]
eval scn     (Exists ps f) = eval scn (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
eval (kns@(KnS _ _ obs),s) (K i form) = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns@(KnS _ _ obs),s) (Kw i form) = alleq (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns,s) (Ck ags form)  = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> (sort s, sort s') `elem` comknow kns ags) (statesOf kns)
eval (kns,s) (Ckw ags form)  = alleq (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> (sort s, sort s') `elem` comknow kns ags) (statesOf kns)
eval scn (PubAnnounce form1 form2) =
  not (eval scn form1) || eval (pubAnnounceOnScn scn form1) form2
eval (kns,s) (PubAnnounceW form1 form2) =
  if eval (kns, s) form1
    then eval (pubAnnounce kns form1, s) form2
    else eval (pubAnnounce kns (Neg form1), s) form2
eval scn (Announce ags form1 form2) =
  not (eval scn form1) || eval (announceOnScn scn ags form1) form2
eval scn (AnnounceW ags form1 form2) =
  if eval scn form1
    then eval (announceOnScn scn ags form1      ) form2
    else eval (announceOnScn scn ags (Neg form1)) form2
```

We also have to define how knowledge structures are changed by public and group announcements. The following functions correspond to the last two points of Definition 7.

```
pubAnnounce :: KnowStruct -> Form -> KnowStruct
pubAnnounce kns@(KnS props lawbdd obs) psi = KnS props newlawbdd obs where
  newlawbdd = con lawbdd (bddOf kns psi)

pubAnnounceOnScn :: Scenario -> Form -> Scenario
pubAnnounceOnScn (kns,s) psi
  | eval (kns,s) psi = (pubAnnounce kns psi,s)
  | otherwise        = error "Liar!"

announce :: KnowStruct -> [Agent] -> Form -> KnowStruct
announce kns@(KnS props lawbdd obs) ags psi = KnS newprops newlawbdd newobs where
  proppsi@(P k) = freshp props
  newprops  = proppsi:props
  newlawbdd = con lawbdd (equ (var k) (bddOf kns psi))
  newobs    = [(i, apply obs i ++ [proppsi | i `elem` ags]) | i <- map fst obs]

announceOnScn :: Scenario -> [Agent] -> Form -> Scenario
announceOnScn (kns@(KnS props _ _),s) ags psi
  | eval (kns,s) psi = (announce kns ags psi, freshp props : s)
  | otherwise        = error "Liar!"
```

The following definition and its implementation `bddOf` is the key idea for symbolic model checking

DEL: Given a knowledge structure $\mathcal{F}$ and a formula φ, it generates a BDD which represents a boolean formula that on $\mathcal{F}$ is equivalent to φ. In particular, this function does not generate longer and longer formulas. It only makes calls to itself, the announcement functions and the boolean operations provided by the BDD package.

Definition 8. *For any knowledge structure $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$ and any formula φ we define its* local boolean translation $\|\varphi\|_{\mathcal{F}}$ *as follows.*

1. *For any primitive formula, let $\|p\|_{\mathcal{F}} := p$.*
2. *For negation, let $\|\neg\psi\|_{\mathcal{F}} := \neg\|\psi\|_{\mathcal{F}}$.*
3. *For conjunction, let $\|\psi_1 \wedge \psi_2\|_{\mathcal{F}} := \|\psi_1\|_{\mathcal{F}} \wedge \|\psi_2\|_{\mathcal{F}}$.*
4. *For knowledge, let $\|K_i\psi\|_{\mathcal{F}} := \forall(V \setminus O_i)(\theta \rightarrow \|\psi\|_{\mathcal{F}})$.*
5. *For common knowledge, let $\|C_\Delta\psi\|_{\mathcal{F}} :=$ **gfp**Λ where Λ is the following operator on boolean formulas and **gfp**Λ denotes its greatest fixed point:*

$$\Lambda(\alpha) := \|\psi\|_{\mathcal{F}} \wedge \bigwedge_{i \in \Delta} \forall(V \setminus O_i)(\theta \rightarrow \alpha)$$

6. *For public announcements, let $\|[\psi]\xi\|_{\mathcal{F}} := \|\psi\|_{\mathcal{F}} \rightarrow \|\xi\|_{\mathcal{F}^\psi}$.*
7. *For group announcements, let $\|[\psi]_\Delta\xi\|_{\mathcal{F}} := \|\psi\|_{\mathcal{F}} \rightarrow (\|\xi\|_{\mathcal{F}^\Delta_\psi})(\frac{p_\psi}{\top})$.*

where $\mathcal{F}^\psi$ and $\mathcal{F}^\Delta_\psi$ are as given by Definition 7.

```
bddOf :: KnowStruct -> Form -> Bdd
bddOf _   Top             = top
bddOf _   Bot             = bot
bddOf _   (PrpF (P n))    = var n
bddOf kns (Neg form)      = neg $ bddOf kns form
bddOf kns (Conj forms)    = conSet $ map (bddOf kns) forms
bddOf kns (Disj forms)    = disSet $ map (bddOf kns) forms
bddOf kns (Xor  forms)    = xorSet $ map (bddOf kns) forms
bddOf kns (Impl f g)      = imp (bddOf kns f) (bddOf kns g)
bddOf kns (Equi f g)      = equ (bddOf kns f) (bddOf kns g)
bddOf kns (Forall ps f) = forallSet (map fromEnum ps) (bddOf kns f)
bddOf kns (Exists ps f) = existsSet (map fromEnum ps) (bddOf kns f)
bddOf kns@(KnS allprops lawbdd obs) (K i form) =
  forallSet otherps (imp lawbdd (bddOf kns form)) where
    otherps = map (\(P n) -> n) $ allprops \\ apply obs i
bddOf kns@(KnS allprops lawbdd obs) (Kw i form) =
  disSet [ forallSet otherps (imp lawbdd (bddOf kns f)) | f <- [form, Neg form] ] where
    otherps = map (\(P n) -> n) $ allprops \\ apply obs i
bddOf kns@(KnS allprops lawbdd obs) (Ck ags form) = gfp lambda where
  lambda z = conSet $ bddOf kns form : [ forallSet (otherps i) (imp lawbdd z) | i <- ags ]
  otherps i = map (\(P n) -> n) $ allprops \\ apply obs i
bddOf kns (Ckw ags form) = dis (bddOf kns (Ck ags form)) (bddOf kns (Ck ags (Neg form)))
bddOf kns@(KnS props _ _) (Announce ags form1 form2) =
  imp (bddOf kns form1) (restrict bdd2 (k,True)) where
    bdd2  = bddOf (announce kns ags form1) form2
    (P k) = freshp props
bddOf kns@(KnS props _ _) (AnnounceW ags form1 form2) =
  ifthenelse (bddOf kns form1) bdd2a bdd2b where
    bdd2a = restrict (bddOf (announce kns ags form1) form2) (k,True)
    bdd2b = restrict (bddOf (announce kns ags form1) form2) (k,False)
    (P k) = freshp props
bddOf kns (PubAnnounce form1 form2) =
  imp (bddOf kns form1) (bddOf (pubAnnounce kns form1) form2)
bddOf kns (PubAnnounceW form1 form2) =
  ifthenelse (bddOf kns form1) newform2a newform2b where
    newform2a = bddOf (pubAnnounce kns form1) form2
    newform2b = bddOf (pubAnnounce kns (Neg form1)) form2
```

Theorem 9. *Definition 8 preserves and reflects truth. That is, for any formula φ and any scene $(\mathcal{F}, s)$ we have that $(\mathcal{F}, s) \vDash \varphi$ iff $s \vDash \|\varphi\|_{\mathcal{F}}$.*

Knowing that the translation is correct we can now define the symbolic evaluation function `evalViaBdd`. Note that it has exactly the same type and thus takes the same input as `eval`.

```
evalViaBdd :: Scenario -> Form -> Bool
evalViaBdd (kns@(KnS allprops _ _),s) f = bool where
  bool | b==top = True
       | b==bot = False
       | otherwise = error ("evalViaBdd failed: BDD leftover:\n" ++ show b)
  b    = restrictSet (bddOf kns f) list
  list = [ (n, P n `elem` s) | (P n) <- allprops ]
```

Moreover, we have the following theorem which allows us to check the validity of a formula on a knowledge structure simply by checking if its boolean equivalent is implied by the state law.

Theorem 10. *Definition 8 preserves and reflects validity. That is, for any formula φ and any knowledge structure $\mathcal{F}$ with the state law θ we have that $\mathcal{F} \vDash \varphi$ iff $\theta \to \|\varphi\|_{\mathcal{F}}$ is a boolean tautology.*

```
validViaBdd :: KnowStruct -> Form -> Bool
validViaBdd kns@(KnS _ lawbdd _) f = top == lawbdd `imp` bddOf kns f
```

```
whereViaBdd :: KnowStruct -> Form -> [KnState]
whereViaBdd kns f = statesOf (pubAnnounce kns f)
```

3.2 Knowledge Transformers

For now our language is restricted to two kinds of events – public and group announcements. However, the symbolic model checking method can be extended to cover other epistemic events. What action models (see Definition 5) are to Kripke models, the following knowledge transformers are to knowledge structures. The analog of product update is knowledge transformation.

Definition 11. *A* knowledge transformer *for a given vocabulary V and set of agents $I = \{1, \ldots, n\}$ is a tuple $\mathcal{X} = (V^+, \theta^+, O_1, \ldots, O_n)$ where V^+ is a set of atomic propositions such that $V \cap V^+ = \varnothing$, θ^+ is a possibly epistemic formula from $\mathcal{L}(V \cup V^+)$ and $O_i \subseteq V^+$ for all agents i. An* event *is a knowledge transformer together with a subset $x \subseteq V^+$, written as $(\mathcal{X}, x)$.*

The knowledge transformation *of a knowledge structure $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$ with a knowledge transformer $\mathcal{X} = (V^+, \theta^+, O_1^+, \ldots, O_n^+)$ for V is defined by:*

$$\mathcal{F} \times \mathcal{X} := (V \cup V^+, \theta \wedge ||\theta^+||_{\mathcal{F}}, O_1 \cup O_1^+, \ldots, O_n \cup O_n^+)$$

Given a scene $(\mathcal{F}, s)$ and an event $(\mathcal{X}, x)$ we define $(\mathcal{F}, s) \times (\mathcal{X}, x) := (\mathcal{F} \times \mathcal{X}, s \cup x)$.

The two kinds of events discussed above fit well into this general definition: The public announcement of φ is the event $((\varnothing, \varphi, \varnothing, \ldots, \varnothing), \varnothing)$. The semi-private announcement of φ to a group of agents Δ is given by $((\{p_\varphi\}, p_\varphi \leftrightarrow \varphi, O_1^+, \ldots, O_n^+), \{p_\varphi\})$ where $O_i^+ = \{p_\varphi\}$ if $i \in \Delta$ and $O_i^+ = \varnothing$ otherwise.

In the implementation we can see that the elements of `addprops` are shifted to a large enough index so that they become disjoint with `props`.

```
data KnowTransf = KnT [Prp] Form [(Agent,[Prp])] deriving (Eq,Show)
type Event = (KnowTransf,KnState)

knowTransform :: Scenario -> Event -> Scenario
knowTransform (kns@(KnS props lawbdd obs),s) (KnT addprops addlaw eventobs, eventfacts) =
  (KnS (props ++ map snd shiftrel) newlawbdd newobs, s++shifteventfacts) where
    shiftrel = zip addprops [(freshp props)..]
```

```
    newobs = [ (i , sort $ apply obs i ++ map (apply shiftrel) (apply eventobs i)) | i <-
        map fst obs ]
    shiftaddlaw = replPsInF shiftrel addlaw
    newlawbdd = con lawbdd (bddOf kns shiftaddlaw)
    shifteventfacts = map (apply shiftrel) eventfacts
```

We end this module with helper functions to generate LaTeX code that shows a knowledge structure, including a BDD of the state law. See Section 6 for examples of what the output looks like.

```
texBDD :: Bdd -> String
texBDD b = unsafePerformIO $ do
  (i,o,_,_) <- runInteractiveCommand "dot2tex --figpreamble=\"\\huge\" --figonly -traw"
  hPutStr i (genGraph b ++ "\n")
  hClose i
  hGetContents o

instance TexAble Scenario where
  tex (KnS props lawbdd obs, state) = concat
    [ " \\left( \n"
    , tex props ++ ", "
    , " \\begin{array}{l} \\scalebox{0.4}{"
    , texBDD lawbdd
    , "} \\end{array}\n "
    , ", \\begin{array}{l}\n"
    , intercalate " \\\\\n " (map (\(_,os) -> (tex os)) obs)
    , "\\end{array}\n"
    , " \\right) , "
    , tex state ]
```

4 Connecting the two Semantics

In this module we define and implement translation methods to connect the semantics from the two previous sections. This essentially allows us to switch back and forth between explicit and symbolic model checking methods.

```
module SMCDEL.Translations where
import Control.Arrow (second)
import Data.List (groupBy,sort,(\\),elemIndex,intersect)
import Data.Maybe (fromJust)
import SMCDEL.Language
import SMCDEL.Symbolic.HasCacBDD
import SMCDEL.Explicit.Simple
import SMCDEL.Internal.Help (anydiff,alldiff,alleq,apply,powerset)

import Data.HasCacBDD hiding (Top,Bot)
```

Lemma 12. *Suppose we have a knowledge structure $\mathcal{F} = (V, \theta, O_1, \dots, O_n)$ and a finite S5 Kripke model $M = (W, \pi, \mathcal{K}_1, \dots, \mathcal{K}_n)$ with a set of primitive propositions $U \subseteq V$. Furthermore, suppose we have a function $g : W \to \mathcal{P}(V)$ such that*

C1 For all $w_1, w_2 \in W$, and all i such that $1 \leq i \leq n$, we have that $g(w_1) \cap O_i = g(w_2) \cap O_i$ iff $w_1 \mathcal{K}_i w_2$.

C2 For all $w \in W$ and $p \in U$, we have that $p \in g(w)$ iff $\pi(w)(p) = \top$.

C3 For every $s \subseteq V$, s is a state of $\mathcal{F}$ iff $s = g(w)$ for some $w \in W$.

Then, for every $\mathcal{L}(U)$-formula φ we have $(\mathcal{F}, g(w)) \vDash \varphi$ iff $(\mathcal{M}, w) \vDash \varphi$.

4.1 From Knowledge Structures to Kripke Models

Definition 13. *For any $\mathcal{F} = (V, \theta, O_1, \dots, O_n)$, we define the Kripke model $\mathcal{M}(\mathcal{F}) := (W, \pi, \mathcal{K}_1, \dots, \mathcal{K}_n)$ as follows*

1. *W is the set of all states of $\mathcal{F}$,*

2. *for each $w \in W$, let the assignment $\pi(w)$ be w itself and*

3. *for each agent i and all $v, w \in W$, let $v \mathcal{K}_i w$ iff $v \cap O_i = w \cap O_i$.*

Theorem 14. *For any knowledge structure $\mathcal{F}$, any state s of $\mathcal{F}$, and any φ we have $(\mathcal{F}, s) \vDash \varphi$ iff $(M(\mathcal{F}), s) \vDash \varphi$.*

```
knsToKripke :: Scenario -> PointedModel
knsToKripke (kns@(KnS ps _ obs),curs) =
  if curs `elem` statesOf kns
     then (KrM worlds rel val, cur)
     else error "knsToKripke failed: Invalid state."
  where
    lav    = zip (statesOf kns) [0..(length (statesOf kns)-1)]
    val    = map ( \(s,n) -> (n,state2kripkeass s) ) lav where
      state2kripkeass s = map (\p -> (p, p `elem` s)) ps
    rel    = [(i,rfor i) | i <- map fst obs]
    rfor i = map (map snd) (groupBy ( \ (x,_) (y,_) -> x==y ) (sort pairs)) where
      pairs = map (\s -> (restrictState s (apply obs i), apply lav s)) (statesOf kns)
    worlds = map fst val
    cur    = apply lav curs
```

4.2 From Kripke Models to Knowledge Structures

Definition 15. *For any S5 model $\mathcal{M} = (W, \pi, \mathcal{K}_1, \dots, \mathcal{K}_n)$ with some set of primitive propositions U we define a knowledge structure $\mathcal{F}(\mathcal{M})$ as follows. For each agent i, write $\gamma_{i,1}, \dots, \gamma_{i,k_i}$ for the equivalence classes given by $\mathcal{K}_i$ and let $l_i := \mathsf{ceiling}(\log_2 k_i)$. Let O_i be a set of l_i many fresh propositions. This yields the sets of observational variables $O_1, \dots, O_n$, all disjoint to each other. If agent i has a total relation, i.e. only one equivalence class, then we have $O_i = \varnothing$. Enumerate k_i many subsets of O_i as $O_{\gamma_{i,1}}, \dots, O_{\gamma_{i,k_i}}$ and define $g_i : W \to \mathcal{P}(O_i)$ by $g_i(w) := O_{\gamma_i(w)}$ where $\gamma_i(w)$ is the $\mathcal{K}_i$-equivalence class of w. Let $V := U \cup \bigcup_{0<i\leq n} O_i$ and define $g : W \to \mathcal{P}(V)$ by*

$$g(w) := \{v \in U \mid \pi(w)(v) = \top\} \cup \bigcup_{0<i\leq n} g_i(w)$$

Finally, let $\mathcal{F}(\mathcal{M}) := (V, \theta_M, O_1, \dots, O_n)$ using

$$\theta_M := \bigvee \{g(w) \sqsubseteq V \mid w \in W\}$$

where $\sqsubseteq$ abbreviates a formula saying that out of the propositions in the second set exactly those in the first are true: $A \sqsubseteq B := \bigwedge A \wedge \bigwedge \{\neg p \mid p \in B \setminus A\}$.

Theorem 16. *For any finite pointed S5 Kripke model $(\mathcal{M}, w)$ and every formula φ, we have that $(\mathcal{M}, w) \vDash \varphi$ iff $(\mathcal{F}(\mathcal{M}), g(w)) \vDash \varphi$.*

```
kripkeToKns :: PointedModel -> Scenario
kripkeToKns (KrM worlds rel val, cur) = (KnS ps law obs, curs) where
  v          = map fst $ apply val cur
  ags        = map fst rel
  newpstart  = fromEnum $ freshp v -- start counting new propositions here
  amount i   = ceiling (logBase 2 (fromIntegral $ length (apply rel i)) :: Float) -- |O_i|
  newpstep   = maximum [ amount i | i <- ags ]
  numberof i = fromJust $ elemIndex i (map fst rel)
  newps i    = map (\k -> P (newpstart + (newpstep * numberof i) +k)) [0..(amount i - 1)] --
       O_i
  copyrel i  = zip (apply rel i) (powerset (newps i)) -- label equiv.classes with P(O_i)
  gag i w    = snd $ head $ filter (\(ws,_) -> elem w ws) (copyrel i)
  g w        = filter (apply (apply val w)) v ++ concat [ gag i w | i <- ags ]
  ps         = v ++ concat [ newps i | i <- ags ]
  law        = disSet [ booloutof (g w) ps | w <- worlds ]
  obs        = [ (i,newps i) | i<- ags ]
  curs       = sort $ g cur

booloutof :: [Prp] -> [Prp] -> Bdd
booloutof ps qs = conSet $
  [ var n | (P n) <- ps ] ++
  [ neg $ var n | (P n) <- qs \\ ps ]
```

An alternative approach, trying to add fewer propositions:

```
uniqueVals :: KripkeModel -> Bool
uniqueVals (KrM _ _ val) = alldiff (map snd val)

voc :: KripkeModel -> [Prp]
voc (KrM _ _ val) = map fst . snd . head $ val

-- | Get lists of variables which agent i does (not) observe
-- in model m. This does *not* preserve all information, i.e.
-- does not characterize every possible S5 relation!
obsnobs :: KripkeModel -> Agent -> ([Prp],[Prp])
obsnobs m@(KrM _ rel val) i = (obs,nobs) where
  propsets = map (map (map fst . filter snd . apply val)) (apply rel i)
  obs = filter (\p -> all (alleq (elem p)) propsets) (voc m)
  nobs = filter (\p -> any (anydiff (elem p)) propsets) (voc m)

-- | Test if all relations can be described using observariables.
descableRels :: KripkeModel -> Bool
```

```haskell
descableRels m@(KrM ws rel val) = all descable (map fst rel) where
  wpairs = [ (v,w) | v <- ws, w <- ws ]
  descable i = cover && correct where
    (obs,nobs) = obsnobs m i
    cover = sort (voc m) == sort (obs ++ nobs) -- implies disjointness
    correct = all (\pair -> oldrel pair == newrel pair) wpairs
    oldrel (v,w) = v `elem` head (filter (elem w) (apply rel i))
    newrel (v,w) = (factsAt v `intersect` obs) == (factsAt w `intersect` obs)
    factsAt w = map fst $ filter snd $ apply val w

-- | Try to find an equivalent knowledge structure without
-- additional propositions. Will succeed iff valuations are
-- unique and relations can be described using observariables.
smartKripkeToKns :: PointedModel -> Maybe Scenario
smartKripkeToKns (m, cur) =
  if uniqueVals m && descableRels m
    then Just (smartKripkeToKnsWithoutChecks (m, cur))
    else Nothing

smartKripkeToKnsWithoutChecks :: PointedModel -> Scenario
smartKripkeToKnsWithoutChecks (m@(KrM worlds rel val), cur) =
  (KnS ps law obs, curs) where
    ps = voc m
    g w = filter (apply (apply val w)) ps
    law = disSet [ booloutof (g w) ps | w <- worlds ]
    obs = map (\(i,_) -> (i,obsOf i) ) rel
    obsOf = fst.obsnobs m
    curs = map fst $ filter snd $ apply val cur
```

4.3 From Action Models to Knowledge Transformers

For any S5 action model there is an equivalent knowledge transformer and vice versa. The translations are similar to Definitions 13 and 15 and their soundness also follows from Lemma 12. The implementation below works on pointed models, to simplify tracking the actual world and action.

Definition 17. *The function* Trf *maps an S5 action model* $\mathcal{A} = (A, (R_i)_{i\in I}, \mathsf{pre})$ *to a transformer as follows. Let* P *be a finite set of fresh propositions such that there is an injective labeling function* $g : A \to \mathcal{P}(P)$ *and let*

$$\Phi := \bigwedge \{(g(a) \sqsubseteq P) \to \mathsf{pre}(a) \,|\, a \in A\}$$

where $\sqsubseteq$ *is the "out of" abbreviation from Definition 15. Now, for each* i*: Write* A/R_i *for the set of equivalence classes induced by* R_i*. Let* O_i^+ *be a finite set of fresh propositions such that there is an injective* $g_i : A/R_i \to \mathcal{P}(O_i^+)$ *and let*

$$\Phi_i := \bigwedge \left\{ (g_i(\alpha) \sqsubseteq O_i) \to \left(\bigvee_{a\in\alpha} (g(a) \sqsubseteq P) \right) \middle| \, \alpha \in A/R_i \right\}$$

Finally, define $\mathsf{Trf}(\mathcal{A}) := (V^+, \theta^+, O_1^+, \ldots, O_n^+)$ *where* $V^+ := P \cup \bigcup_{i\in I} P_i$ *and* $\theta^+ := \Phi \wedge \bigwedge_{i\in I} \Phi_i$*.*

Theorem 18. *For any pointed S5 Kripke model* $(\mathcal{M}, w)$*, any pointed S5 action model* $(\mathcal{A}, \alpha)$ *and any formula* φ *over the vocabulary of* $\mathcal{M}$ *we have:*

$$\mathcal{M} \times \mathcal{A}, (w, \alpha) \vDash \varphi \iff \mathcal{F}(\mathcal{M}) \times \mathsf{Trf}(\mathcal{A}), (g_\mathcal{M}(w) \cup g_\mathcal{A}(\alpha)) \vDash \varphi$$

where $g_\mathcal{M}$ *is from the construction of* $\mathcal{F}(\mathcal{M})$ *in Definition 13 and* $g_\mathcal{A}$ *is from the construction of* $\mathsf{Trf}(\mathcal{A})$ *in Definition 17.*

```
actionToEvent :: PointedActionModel -> Event
actionToEvent (ActM actions precon actrel, faction) = (KnT eprops elaw eobs, efacts) where
  ags          = map fst actrel
  eprops       = actionprops ++ actrelprops
  (P fstnewp)  = freshp $ propsInForms (map snd precon)
  actionprops  = [P fstnewp..P maxactprop] -- new props to distinguish all actions
  maxactprop   = fstnewp + ceiling (logBase 2 (fromIntegral $ length actions) :: Float) -1
  copyactprops = zip actions (powerset actionprops)
  actforms     = [ Impl (booloutofForm (apply copyactprops a) actionprops) (apply precon a)
       | a <- actions ] -- connect the new propositions to the preconditions
  actrelprops  = concat [ newps i | i <- ags ] -- new props to distinguish actions for i
  actrelpstart = maxactprop + 1
  numberof i = fromJust $ elemIndex i (map fst actrel)
  newps i      = map (\k -> P (actrelpstart + (newpstep * numberof i) +k)) [0..(amount i -
      1)]
  amount i     = ceiling (logBase 2 (fromIntegral $ length (apply actrel i)) :: Float)
  newpstep     = maximum [ amount i | i <- ags ]
  copyactrel i = zip (apply actrel i) (powerset (newps i)) -- actrelprops <-> actionprops
  actrelfs i   = [ Impl (booloutofForm (apply (copyactrel i) as) (newps i)) (Disj [adesc a|
      a<-as]) | as <- apply actrel i ] where adesc a = booloutofForm (apply copyactprops a)
       actionprops
  actrelforms  = concatMap actrelfs ags
  factsFor i   = snd $ head $ filter (\(as,_) -> elem faction as) (copyactrel i)
  efacts       = apply copyactprops faction ++ concatMap factsFor ags
  elaw         = simplify $ Conj (actforms ++ actrelforms)
  eobs         = [ (i,newps i) | i<- ags ]
```

4.4 From Knowledge Transformers to Action Models

Definition 19. *For any Knowledge Transformer $\mathcal{X} = (V^+, \theta^+, O_1^+, \dots, O_n^+)$ we define an S5 action model* $\mathsf{Act}(\mathcal{X})$ *as follows. First, let the set of actions be $A := \mathcal{P}(V^+)$. Second, for any two actions $\alpha, \beta \in A$, let $\alpha R_i \beta$ iff $\alpha \cap O_i^+ = \beta \cap O_i^+$. Third, for any α, let $\mathsf{pre}(\alpha) := \theta^+ \left(\frac{\alpha}{\top}\right) \left(\frac{V^+ \setminus \alpha}{\bot}\right)$. Finally, let* $\mathsf{Act}(\mathcal{X}) := (A, (R_i)_{i \in I}, \mathsf{pre})$.

Theorem 20. *For any scene $(\mathcal{F}, s)$, any event $(\mathcal{X}, x)$ and any formula φ over the vocabulary of $\mathcal{F}$ we have:*

$$(\mathcal{F}, s) \times (\mathcal{X}, x) \vDash \varphi \iff (\mathcal{M}(\mathcal{F}) \times \mathsf{Act}(\mathcal{X})), (s, x) \vDash \varphi$$

Note that this definition of Act can yield action models with contradictions as preconditions. In the implementation below we remove all actions where $\mathsf{pre}(\alpha) = \bot$.

```
eventToAction' :: Event -> PointedActionModel
eventToAction' (KnT eprops eform eobs, efacts) = (ActM actions precon actrel, faction)
    where
  actions    = [0..(2 ^ length eprops - 1)]
  actlist    = zip (powerset eprops) actions
  precon     = [ (a, simplify $ preFor ps) | (ps,a) <- actlist ] where
    preFor ps = substitSet (zip ps (repeat Top) ++ zip (eprops\\ps) (repeat Bot)) eform
  actrel     = [(i,rFor i) | i <- map fst eobs] where
    rFor i   = map (map snd) (groupBy ( \ (x,_) (y,_) -> x==y ) (pairs i))
    pairs i  = sort $ map (\(set,a) -> (restrictState set $ apply eobs i,a)) actlist
  faction    = apply actlist efacts

eventToAction :: Event -> PointedActionModel
eventToAction (KnT eprops eform eobs, efacts) = (ActM actions precon actrel, faction) where
  (ActM _ precon' actrel', faction) = eventToAction' (KnT eprops eform eobs, efacts)
  precon   = filter (\(_,f) -> f/=Bot) precon' -- remove actions w/ contradictory precon
  actions  = map fst precon
  actrel   = map (second fltr) actrel'
  fltr r   = filter ([]/=) $ map (filter (`elem` actions)) r
```

5 Automated Testing

In this chapter we test our implementations for correctness, using QuickCheck for automation and randomization. We generate random formulas and then evaluate them on Kripke models and knowledge structures of which we already know that they are equivalent. The test algorithm then checks whether the different methods we implemented agree on the result.

```
module Main where
import Control.Monad
import System.Exit
import Test.QuickCheck
import Test.QuickCheck.Test
import Text.Printf
import SMCDEL.Language
import SMCDEL.Symbolic.HasCacBDD
import SMCDEL.Explicit.Simple
import SMCDEL.Translations
import SMCDEL.Examples
```

5.1 Testing equivalence of the two semantics

The following creates a Kripke model and a knowledge structure which are equivalent to each other by Lemma 12. In this model/structure Alice knows everything and the other agents do not know anything. The function `test` checks for a given number of random formulas whether the implementations of the different semantics and translation methods agree on whether the formula holds on the model and the structure.

```
mymodel :: PointedModel
mymodel = (KrM ws rel (zip ws table), 0) where
  ws    = [0..31]
  rel   = ("0", map (:[]) ws) : [ (show i,[ws]) | i <- [1..5::Int] ]
  table = foldl buildTable [[]] [P k | k<- [0..4::Int]]
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr<-partrows ]

myscn :: Scenario
myscn = (KnS ps (boolBddOf Top) (("0",ps):[(show i,[]) | i<-[1..5::Int]]) , ps)
  where ps = [P 0 .. P 4]

semanticEquivTest :: Form -> Bool
semanticEquivTest f = and [a==b,b==c,c==d,d==e] where
  a = SMCDEL.Explicit.Simple.eval mymodel f                 -- evaluate directly on
      Kripke
  b = SMCDEL.Symbolic.HasCacBDD.eval myscn f                -- evaluate directly on KNS
  c = SMCDEL.Symbolic.HasCacBDD.evalViaBdd myscn f          -- evaluate boolean
      equivalent on KNS
  d = SMCDEL.Explicit.Simple.eval (knsToKripke myscn) f     -- evaluate on corresponding
      Kripke
  e = SMCDEL.Symbolic.HasCacBDD.eval (kripkeToKns mymodel) f  -- evaluate on corresponding
      KNS
```

5.2 Public Announcements

We can do public announcements in various ways. The following test checks that the result of all three methods is the same.

```
pubAnnounceTest :: Prp -> Form -> Bool
pubAnnounceTest prp g = (a==b) && (b==c) && (c==d) where
  f = PrpF prp
  a = SMCDEL.Explicit.Simple.eval mymodel (PubAnnounce f g)
  b = SMCDEL.Symbolic.HasCacBDD.eval (kripkeToKns mymodel) (PubAnnounce f g)
  c = SMCDEL.Symbolic.HasCacBDD.evalViaBdd (kripkeToKns mymodel) (PubAnnounce f g)
  d = SMCDEL.Symbolic.HasCacBDD.eval (knowTransform (kripkeToKns mymodel) (actionToEvent (
      pubAnnounceAction (map show [1..(5::Int)]) f))) g
```

5.3 Group Announcements

```
announceTest :: Form -> [AgAgent] -> Form -> Bool
announceTest f agags g = (a==b) && (b==c) && (c==d) where
  ags = map (\(Ag i) -> i) agags
  a = SMCDEL.Explicit.Simple.eval mymodel (Announce ags f g)
  b = SMCDEL.Symbolic.HasCacBDD.eval (kripkeToKns mymodel) (Announce ags f g)
  c = SMCDEL.Symbolic.HasCacBDD.evalViaBdd (kripkeToKns mymodel) (Announce ags f g)
  d = not (SMCDEL.Symbolic.HasCacBDD.eval (kripkeToKns mymodel) f)
      || SMCDEL.Symbolic.HasCacBDD.eval (knowTransform (kripkeToKns mymodel) (actionToEvent
           (groupAnnounceAction (map show [1..(5::Int)]) ags f))) g
```

5.4 Random Action Models

This generates a random action model with four actions. To ensure that it is compatible with all models the actual action token has $\top$ as precondition. The other three action tokens have random formulas as preconditions. Similar to the model above the first agent can tell the actions apart and everyone else confuses them.

```
instance Arbitrary ActionModel where
  arbitrary = do
    f <- arbitrary
    g <- arbitrary
    h <- arbitrary
    return $
      ActM [0..3] [(0,Top),(1,f),(2,g),(3,h)] ( ("0",[[0],[1],[2],[3]]):[(show k,[[0..3::
          Int]]) | k<-[1..5::Int] ])

singleActionTest :: ActionModel -> Form -> Bool
singleActionTest myact f = a==b where
  a = SMCDEL.Explicit.Simple.eval (productUpdate mymodel (myact,0)) f
  b = SMCDEL.Symbolic.HasCacBDD.evalViaBdd (knowTransform (kripkeToKns mymodel) (
      actionToEvent (myact,0))) f
```

5.5 Running QuickCheck

All tests should pass:

```
main :: IO ()
main  = do
  results <- mapM (\(s,a) -> printf "%-25s: " s >> a)
    [ ("semanticEquivTest", quickCheckResult semanticEquivTest)
    , ("pubAnnounceTest"  , quickCheckResult pubAnnounceTest  )
    , ("announceTest"     , quickCheckResult announceTest     )
    , ("singleActionTest" , quickCheckResult singleActionTest ) ]
  unless (all isSuccess results) exitFailure
```

6 Examples

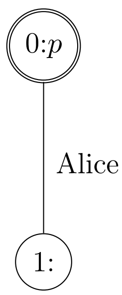
This section shows how to use our model checker on concrete cases. We start with some toy examples and then deal with famous puzzles and protocols from the literature.

```
module SMCDEL.Examples where
import Data.List (delete,intersect,(\\),elemIndex)
import Data.Maybe (fromJust)
import SMCDEL.Language
import SMCDEL.Internal.Help (powerset)
import SMCDEL.Symbolic.HasCacBDD
import SMCDEL.Explicit.Simple
import SMCDEL.Translations
```

6.1 Knowledge and Meta-Knowledge

In the following Kripke model, Bob knows that p is true and Alice does not. Still, Alice knows that Bob knows whether p. This is because in all worlds that Alice confuses with the actual world Bob either knows that p or he knows that not p.

```
modelA :: PointedModel
modelA = (KrM [0,1] [(alice,[[0,1]]),(bob,[[0],[1]])] [ (0,[(P 0,True)]), (1,[(P 0,False)])
    ], 0)
```




```
>>> map (SMCDEL.Explicit.Simple.eval modelA) [K bob (PrpF (P 0)), K alice (PrpF (P 0))]
```

```
[True,False]
```

2.04 seconds

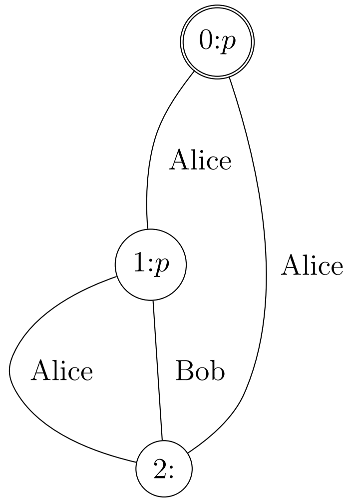
```
>>> SMCDEL.Explicit.Simple.eval modelA (K alice (Kw bob (PrpF (P 0))))
```

```
True
```

1.88 seconds

In a slightly different model with three states, again Bob knows that p is true and Alice does not. And additionally here Alice does not even know whether Bob knows whether p.

```
modelB :: PointedModel
modelB = (KrM [0,1,2] [(alice,[[0,1,2]]),(bob,[[0],[1,2]])] [ (0,[(P 0,True)]), (1,[(P 0,
    True)]), (2,[(P 0,False)]) ], 0)
```

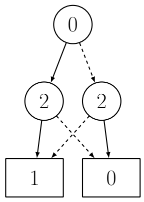


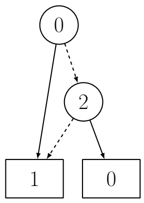

```
>>> SMCDEL.Explicit.Simple.eval modelB (K bob (PrpF (P 0)))
True
1.94 seconds
```

```
>>> SMCDEL.Explicit.Simple.eval modelB (Kw alice (Kw bob (PrpF (P 0))))
False
1.99 seconds
```

Let us see how such meta-knowledge (or in this case: meta-ignorance) is reflected in knowledge structures. Both knowledge structures contain one additional observational variable:

```
knsA, knsB :: Scenario
knsA = kripkeToKns modelA
knsB = kripkeToKns modelB
```

$\Big(\{p, p_2\},$  $, \begin{array}{c}\varnothing \\ \{p_2\}\end{array}\Big), \{p, p_2\}$

$\Big(\{p, p_2\},$  $, \begin{array}{c}\varnothing \\ \{p_2\}\end{array}\Big), \{p, p_2\}$

The only difference is in the state law of the knowledge structures. Remember that this component determines which assignments are states of this knowledge structure. In our implementation this is not a formula but a BDD, hence we show its graph here. The BDD in `knsA` demands that the propositions p and p_2 have the same value. Hence knsA has just two states while `knsB` has three:

```
>>> let (structA,foo) = knsA in statesOf structA
[[P 0,P 2],[]]
2.10 seconds
```

```
>>> let (structB,foo) = knsB in statesOf structB
```

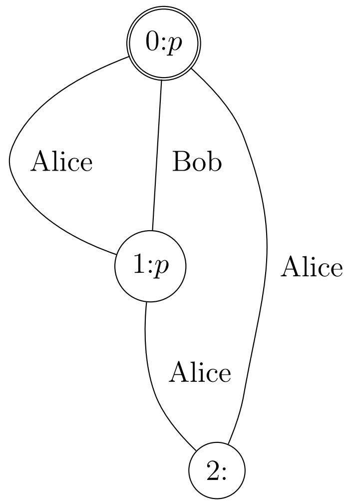
```
[[P 0],[P 0,P 2],[]]
```

2.13 seconds

6.2 Minimization via Translation

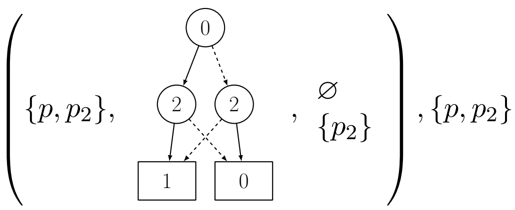
Consider the following Kripke model where **0** and **1** are bisimilar – it is redundant.

```
redundantModel :: PointedModel
redundantModel = (KrM [0,1,2] [(alice,[[0,1,2]]),(bob,[[0,1],[2]])] [ (0,[(P 0,True)]),
    (1,[(P 0,True)]), (2,[(P 0,False)]) ], 0)
```



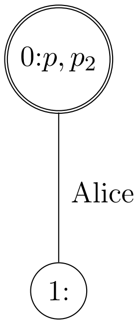
If we transform this model to a knowledge structure, we get the following:

```
myKNS :: Scenario
myKNS = kripkeToKns redundantModel
```



Moreover, if we transform this knowledge structure back to a Kripke Model, we get a model which is bisimilar to the first one but has only two states – the redundancy is gone. This shows how knowledge structures can be used to find smaller bisimilar models.

```
minimizedModel :: PointedModel
minimizedModel = knsToKripke myKNS
```



6.3 Different Announcements

We can represent a public announcement as an action model and then get the corresponding knowledge transformer.

```
pubAnnounceAction :: [Agent] -> Form -> PointedActionModel
pubAnnounceAction ags f = (ActM [0] [(0,f)] [ (i,[[0]]) | i <- ags ], 0)

examplePaAction :: PointedActionModel
examplePaAction = pubAnnounceAction [alice,bob] (PrpF (P 0))
```

```
>>> examplePaAction

(ActM [0] [(0,PrpF (P 0))] [("Alice",[[0]]),("Bob",[[0]])],0)

2.02 seconds
```

```
>>> actionToEvent examplePaAction

(KnT [] (PrpF (P 0)) [("Alice",[]),("Bob",[])],[])

1.95 seconds
```

Similarly a group announcement can be defined as an action model with two states. The automatically generated equivalent knowledge transformer uses two atomic propositions which at first sight seems different from how we defined group announcements on knowledge structures.

```
groupAnnounceAction :: [Agent] -> [Agent] -> Form -> PointedActionModel
groupAnnounceAction everyone listeners f = (ActM [0,1] [(0,f),(1,Top)] actrel, 0)
  where actrel = [ (i,[[0],[1]]) | i <- listeners ]
              ++ [ (i,[[0 , 1]]) | i <- everyone \\ listeners ]

exampleGroupAnnounceAction :: PointedActionModel
exampleGroupAnnounceAction = groupAnnounceAction [alice, bob] [alice] (PrpF (P 0))
```

```
>>> exampleGroupAnnounceAction

(ActM [0,1] [(0,PrpF (P 0)),(1,Top)] [("Alice",[[0],[1]]),("Bob",[[0,1]])],0)

1.92 seconds
```

```
>>> actionToEvent exampleGroupAnnounceAction

(KnT [P 1,P 2] (Conj [Impl (PrpF (P 1)) (PrpF (P 0)),Impl (PrpF (P 2)) (PrpF (P 1)),
    Impl (Neg (PrpF (P 2))) (Neg (PrpF (P 1)))]) [("Alice",[P 2]),("Bob",[])],[P 1,P
    2])

2.05 seconds
```

But it is not hard to check that this is equivalent to the definition. Consider the θ^+ formula of this transformer, namely $\bigwedge\{p_1 \to p_1, p_2 \to p_1, \neg p_2 \to \neg p_1, p_1 \vee \neg p_1\}$. This is equivalent to $p_1 \leftrightarrow p_2$ and the actual event is given by both p_1 and p_2 being added to the current state, equivalent to the normal announcement. There is no canonical way to avoid such redundancy as long as we use the general two-step process in Definition 17 to translate action models to knowledge transformers.

We can also turn this knowledge transformer back to an action model. The result is the same as the action model we started with up to a renaming of action 1 to 3.

```
>>> eventToAction (actionToEvent exampleGroupAnnounceAction)

(ActM [0,3] [(0,PrpF (P 0)),(3,Top)] [("Alice",[[3],[0]]),("Bob",[[0,3]])],0)

1.96 seconds
```

6.4 Equivalent Action Models

The following are two action models which have bisimilar (in fact identical!) effects on any Kripke model.

```
actionOne :: PointedActionModel
actionOne = (ActM [0,1] [(0,p),(1, Disj [q, Neg q])] [("Alice",[[0],[1]]), ("Bob",[[0,1]])
    ], 0) where (p,q) = (PrpF $ P 0, PrpF $ P 1)

actionTwo :: PointedActionModel
actionTwo = (ActM [0,1,2] [(0,p),(1,q),(2,Neg q)] [("Alice",[[0],[1,2]]), ("Bob",[[0,1,2]])
     ], 0) where (p,q) = (PrpF $ P 0, PrpF $ P 1)
```

```
>>> actionToEvent actionOne

(KnT [P 2,P 3] (Conj [Impl (PrpF (P 2)) (PrpF (P 0)),Impl (PrpF (P 3)) (PrpF (P 2)),
    Impl (Neg (PrpF (P 3))) (Neg (PrpF (P 2)))]) [("Alice",[P 3]),("Bob",[])],[P 2,P
    3])

1.96 seconds
```

```
>>> actionToEvent actionTwo

(KnT [P 2,P 3,P 4] (Conj [Impl (Conj [PrpF (P 2),PrpF (P 3)]) (PrpF (P 0)),Impl (Conj
    [PrpF (P 2),Neg (PrpF (P 3))]) (PrpF (P 1)),Impl (Conj [PrpF (P 3),Neg (PrpF (P 2))
    ]) (Neg (PrpF (P 1))),Impl (PrpF (P 4)) (Conj [PrpF (P 2),PrpF (P 3)]),Impl (Neg (
    PrpF (P 4))) (Disj [Conj [PrpF (P 2),Neg (PrpF (P 3))],Conj [PrpF (P 3),Neg (PrpF (
    P 2))]]),Disj [Conj [PrpF (P 2),PrpF (P 3)],Conj [PrpF (P 2),Neg (PrpF (P 3))],Conj
     [PrpF (P 3),Neg (PrpF (P 2))]]]) [("Alice",[P 4]),("Bob",[])],[P 2,P 3,P 4])

1.91 seconds
```

6.5 Muddy Children

We now model the story of the muddy children which is known in many versions. See for example [24], [18, p. 24-30] or [10, p. 93-96]. Our implementation treats the general case for n children out of which m are muddy, but we focus on the case of three children who are all muddy. As usual, all children can observe whether the others are muddy but do not see their own face. This is represented by the observational variables: Agent 1 observes p_2 and p_3, agent 2 observes p_1 and p_3 and agent 3 observes p_1 and p_2.

```
mudScnInit :: Int -> Int -> Scenario
mudScnInit n m = (KnS vocab law obs, actual) where
  vocab  = [P 1 .. P n]
  law    = boolBddOf Top
```

```
  obs    = [ (show i, delete (P i) vocab) | i <- [1..n] ]
  actual = [P 1 .. P m]

myMudScnInit :: Scenario
myMudScnInit = mudScnInit 3 3
```

$$\left(\{p_1,p_2,p_3\},\ \boxed{1}\ ,\ \begin{matrix}\{p_2,p_3\}\\ \{p_1,p_3\}\\ \{p_1,p_2\}\end{matrix}\right),\{p_1,p_2,p_3\}$$

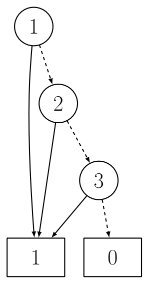
The following parameterized formulas say that child number `i` knows whether it is muddy and that none out of n children knows its own state, respectively:

```
knows :: Int -> Form
knows i = Kw (show i) (PrpF $ P i)

nobodyknows :: Int -> Form
nobodyknows n = Conj [ Neg $ knows i | i <- [1..n] ]
```

Now, let the father announce that someone is muddy and check that still nobody knows their own state of muddiness.

```
father :: Int -> Form
father n = Disj (map PrpF [P 1 .. P n])
mudScn0 :: Scenario
mudScn0 = pubAnnounceOnScn myMudScnInit (father 3)
```

$\Bigg($ $\{p_1,p_2,p_3\},$



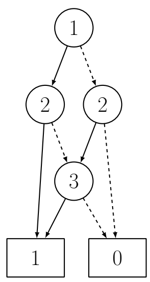
$,\ \begin{matrix}\{p_2,p_3\}\\ \{p_1,p_3\}\\ \{p_1,p_2\}\end{matrix}\Bigg),\{p_1,p_2,p_3\}$

```
>>> evalViaBdd mudScn0 (nobodyknows 3)

True

2.17 seconds
```

If we update once with the fact that nobody knows their own state, it is still true:

```
mudScn1 :: Scenario
mudScn1 = pubAnnounceOnScn mudScn0 (nobodyknows 3)
```

$\Bigg($ $\{p_1,p_2,p_3\},$



$,\ \begin{matrix}\{p_2,p_3\}\\ \{p_1,p_3\}\\ \{p_1,p_2\}\end{matrix}\Bigg),\{p_1,p_2,p_3\}$

```
>>> evalViaBdd mudScn1 (nobodyknows 3)
```

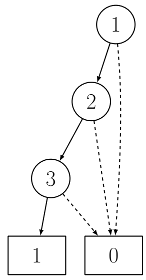
```
True
```

2.01 seconds

However, one more round is enough to make everyone know that they are muddy. We get a knowledge structure with only one state, marking the end of the story.

```
mudScn2 :: Scenario
mudKns2 :: KnowStruct
mudScn2@(mudKns2,_) = pubAnnounceOnScn mudScn1 (nobodyknows 3)
```

$\Bigg(\{p_1, p_2, p_3\},$



$, \begin{matrix}\{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\}\end{matrix}\Bigg), \{p_1, p_2, p_3\}$

```
>>> evalViaBdd mudScn2 (Conj [knows i | i <- [1..3]])
```

```
True
```

2.20 seconds

```
>>> SMCDEL.Symbolic.HasCacBDD.statesOf mudKns2
```

```
[[P 1,P 2,P 3]]
```

2.16 seconds

We also make heavy use of the muddy children example in the benchmarks in section 7.

6.6 Building Muddy Children using Knowledge Transformers

```
empty :: Int -> Scenario
empty n = (KnS [] (boolBddOf Top) obs,[]) where
  obs    = [ (show i, []) | i <- [1..n] ]

buildMC :: Int -> Int -> Event
buildMC n m = (KnT vocab Top obs, map P [1..m]) where
  obs = [ (show i, delete (P i) vocab) | i <- [1..n] ]
  vocab = map P [1..n]
```

6.7 Drinking Logicians

Three logicians – all very thirsty – walk into a bar and get asked “Does everyone want a beer?”. The first two reply ”I don’t know”. After this the third person says ”yes”.

This story is somewhat dual to the muddy children: In the initial state here the agents only know their own piece of information and nothing about the others. The important reasoning here is that an announcement of “I don’t know whether everyone wants a beer.” implies that the person making the announcement wants beer. Because if not, then she would know that not everyone wants beer.

We formalize the situation – generalized to n logicians in a knowledge structure as follows. Let P_i represent that logician i wants a beer.

```haskell
thirstyScene :: Int -> Scenario
thirstyScene n = (KnS [P 1..P n] (boolBddOf Top) [ (show i,[P i]) | i <- [1..n] ], [P 1..P
    n])

myThirstyScene :: Scenario
myThirstyScene = thirstyScene 3
```

$$\left(\{p_1, p_2, p_3\}, \boxed{\;1\;}, \begin{matrix} \{p_1\} \\ \{p_2\} \\ \{p_3\} \end{matrix} \right), \{p_1, p_2, p_3\}$$

We check that nobody knows whether everyone wants beer, but after all but one agent have announced that they do not know, the agent n knows that everyone wants beer. As a formula:

$$\bigwedge_i \neg \left(K_i^? \bigwedge_k P_k \right) \quad \wedge \quad [!\neg K_1^? \bigwedge_k P_k)] \ldots [!\neg K_{n-1}^? \bigwedge_k P_k)] \left(K_n \bigwedge_k P_k \right)$$

```haskell
thirstyF :: Int -> Form
thirstyF n = Conj [ Conj [ doesNotKnow k | k <- [1..n] ]
                  , pubAnnounceStack [ doesNotKnow i | i<-[1..(n-1)] ] $ K (show n)
                    allWantBeer ]
  where
    allWantBeer    = Conj [ PrpF $ P k | k <- [1..n] ]
    doesNotKnow i = Neg $ Kw (show i) allWantBeer

thirstyCheck :: Int -> Bool
thirstyCheck n = evalViaBdd (thirstyScene n) (thirstyF n)
```

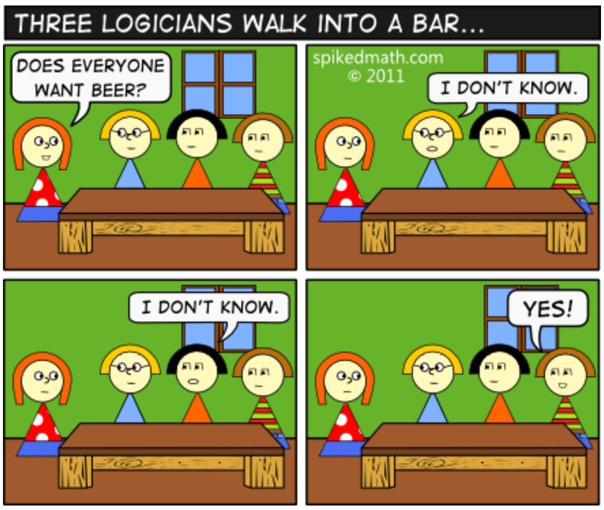
```
>>> thirstyCheck 3

True

2.15 seconds
```

```
>>> thirstyCheck 10

True

2.16 seconds
```

```
>>> thirstyCheck 100

True

2.32 seconds
```

```
>>> thirstyCheck 200

True

3.03 seconds
```

```
>>> thirstyCheck 400

True

6.80 seconds
```



http://spikedmath.com/445.html

6.8 Dining Cryptographers

We model the scenario described in [5]: Three cryptographers went out to have diner. After a lot of delicious and expensive food the waiter tells them that their bill has already been paid. The cryptographers are sure that either it was one of them or the NSA. They want to find what is the case but if one of them paid they do not want that person to be revealed. To accomplish this, they use the following protocol: For every pair of cryptographers a coin is flipped in such a way that only those two see the result. Then they announce whether the two coins they saw were different or the same. But, there is an exception: If one of them paid, then this person says the opposite. After these announcements are made, the cryptographers can infer that the NSA paid iff the number of people saying that they saw the same result on both coins is even.

The following function generates a knowledge structure to model this story. Given an index 0, 1, 2, or 3 for who paid and three boolean values for the random coins we get the corresponding scenario.

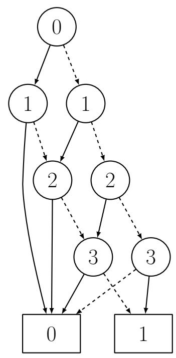
```
dcScnInit :: Int -> (Bool,Bool,Bool) -> Scenario
dcScnInit payer (b1,b2,b3) = ( KnS props law obs , truths ) where
  props = [ P 0    -- The NSA paid
          , P 1    -- Alice paid
          , P 2    -- Bob paid
          , P 3    -- Charlie paid
          , P 4    -- shared bit of Alice and Bob
          , P 5    -- shared bit of Alice and Charlie
          , P 6 ] -- shared bit of Bob and Charlie
  law    = boolBddOf $ Conj [ someonepaid, notwopaid ]
  obs    = [ (show (1::Int),[P 1, P 4, P 5])
           , (show (2::Int),[P 2, P 4, P 6])
           , (show (3::Int),[P 3, P 5, P 6]) ]
  truths = [ P payer ] ++ [ P 4 | b1 ] ++ [ P 5 | b2 ] ++ [ P 6 | b3 ]

dcScn1 :: Scenario
dcScn1 = dcScnInit 1 (True,True,False)
```

The set of possibilities is limited by two conditions: Someone must have paid but no two people (including the NSA) have paid:

```
someonepaid, notwopaid :: Form
someonepaid = Disj (map (PrpF . P) [0..3])
notwopaid = Conj [ Neg $ Conj [ PrpF $ P x, PrpF $ P y ] | x<-[0..3], y<-[(x+1)..3] ]
```

In this scenario Alice paid and the random coins are 1, 1 and 0:

$\Bigg(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$



$, \begin{matrix}\{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\}\end{matrix} \Bigg), \{p_1, p_4, p_5\}$

Every agent computes the `Xor` of all three variables he knows:

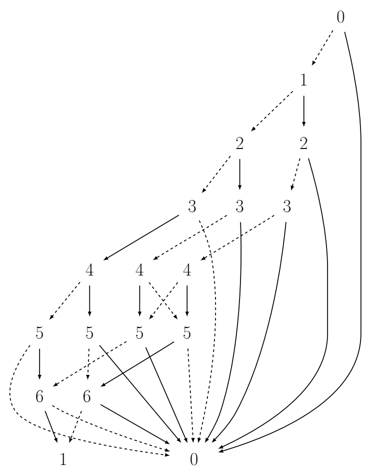
```
reveal :: Int -> Form
reveal 1 = Xor (map PrpF [P 1, P 4, P 5])
reveal 2 = Xor (map PrpF [P 2, P 4, P 6])
reveal _ = Xor (map PrpF [P 3, P 5, P 6])
```

```
>>> map (evalViaBdd dcScn1) [reveal 1, reveal 2, reveal 3]

[True,True,True]

1.99 seconds
```

Now these three facts are announced:

```
dcScn2 :: Scenario
dcScn2 = pubAnnounceOnScn dcScn1 (Conj [reveal 1, reveal 2, reveal 3])
```

$\Bigg(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$



$, \begin{matrix} \{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\} \end{matrix} , \{p_1, p_4, p_5\} \Bigg)$

And now everyone knows whether the NSA paid for the dinner or not:

```
everyoneKnowsWhetherNSApaid :: Form
everyoneKnowsWhetherNSApaid = Conj [ Kw (show i) (PrpF $ P 0) | i <- [1..3]::[Int] ]
```

```
>>> evalViaBdd dcScn2 everyoneKnowsWhetherNSApaid

True

2.10 seconds
```

Further more, it is only known to Alice that she paid:

```
>>> evalViaBdd dcScn2 (K (show 1) (PrpF (P 1)))

True

2.12 seconds
```

```
>>> evalViaBdd dcScn2 (K (show 2) (PrpF (P 1)))

False

2.16 seconds
```

```
>>> evalViaBdd dcScn2 (K (show 3) (PrpF (P 1)))

False

2.12 seconds
```

To check all of this in one formula we use the “announce whether” operator. Furthermore we parameterize the last check on who actually paid, i.e. if one of the three agents paid, then the other two do not know this.

```haskell
nobodyknowsWhoPaid :: Form
nobodyknowsWhoPaid = Conj
  [ Impl (PrpF (P 1)) (Conj [Neg $ K "2" (PrpF $ P 1), Neg $ K "3" (PrpF $ P 1) ])
  , Impl (PrpF (P 2)) (Conj [Neg $ K "1" (PrpF $ P 2), Neg $ K "3" (PrpF $ P 2) ])
  , Impl (PrpF (P 3)) (Conj [Neg $ K "1" (PrpF $ P 3), Neg $ K "2" (PrpF $ P 3) ]) ]

dcCheckForm :: Form
dcCheckForm = PubAnnounceW (reveal 1) $ PubAnnounceW (reveal 2) $ PubAnnounceW (reveal 3) $
    Conj [ everyoneKnowsWhetherNSApaid, nobodyknowsWhoPaid ]
```

```
>>> evalViaBdd dcScn1 dcCheckForm

True

2.22 seconds
```

We can also check that formula is valid on the whole knowledge structure. This means the protocol is secure not just for the particular instance where Alice paid and the random bits (i.e. flipped coins) are as stated above but for all possible combinations of payers and bits/coins.

```haskell
dcValid :: Bool
dcValid = validViaBdd dcStruct dcCheckForm where (dcStruct,_) = dcScn1
```

The whole check runs within a fraction of a second:

```
>>> dcValid

True

2.24 seconds
```

A generalized version of the protocol for more than 3 agents uses exclusive or instead of odd/even. The following implements this general case for n dining cryptographers and we will it for a benchmark in Section 7.2. Note that we need $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ many shared bits. This distinguishes the Dining Cryptographers from the Muddy Children and the Drinking Logicians example where the number of propositions needed to model the situation was just the number of agents.

```haskell
genSomeonepaid :: Int -> Form
genSomeonepaid n = Disj (map (PrpF . P) [0..n])

genNotwopaid :: Int -> Form
genNotwopaid n = Conj [ Neg $ Conj [ PrpF $ P x, PrpF $ P y ] | x<-[0..n], y<-[(x+1)..n] ]

genDcKnsInit :: Int -> KnowStruct
genDcKnsInit n = KnS props law obs where
  props = [ P 0 ] -- The NSA paid
    ++ [ (P 1) .. (P n) ] -- agent i paid
    ++ sharedbits
  law = boolBddOf $ Conj [genSomeonepaid n, genNotwopaid n]
  obs = [ (show i, obsfor i) | i<-[1..n] ]
  sharedbitLabels = [ [k,l] | k <- [1..n], l <- [1..n], k<l ] -- n(n-1)/2 shared bits
  sharedbitRel = zip sharedbitLabels [ (P $ n+1) .. ]
  sharedbits =  map snd sharedbitRel
  obsfor i =  P i : map snd (filter (\(label,_) -> i `elem` label) sharedbitRel)

genEveryoneKnowsWhetherNSApaid :: Int -> Form
genEveryoneKnowsWhetherNSApaid n = Conj [ Kw (show i) (PrpF $ P 0) | i <- [1..n] ]

genDcReveal :: Int -> Int -> Form
genDcReveal n i = Xor (map PrpF (fromJust $ lookup (show i) obs)) where (KnS _ _ obs) =
    genDcKnsInit n

genNobodyknowsWhoPaid :: Int -> Form
genNobodyknowsWhoPaid n =
```

```
  Conj [ Impl (PrpF (P i)) (Conj [Neg $ K (show k) (PrpF $ P i) | k <- delete i [1..n] ]) |
       i <- [1..n] ]

genDcCheckForm :: Int -> Form
genDcCheckForm n =
  pubAnnounceWhetherStack [ genDcReveal n i | i<-[1..n] ] $
    Conj [ genEveryoneKnowsWhetherNSApaid n, genNobodyknowsWhoPaid n ]

genDcValid :: Int -> Bool
genDcValid n = validViaBdd (genDcKnsInit n) (genDcCheckForm n)
```

For example, we can check the protocol for 4 dining cryptographers.

```
>>> genDcValid 4
```

```
True
```

2.29 seconds

6.9 Russian Cards

As a second case study we analyze the Russian Cards problem. One of its first logical treatments is [9] and the problem has since gained notable attention as an intuitive example of information-theoretically (in contrast to computationally) secure cryptography [8, 13].

The basic version of the problem is this: Seven cards, enumerated from 0 to 6, are distributed between Alice, Bob and Carol such that Alice and Bob both receive three cards and Carol one card. It is common knowledge which cards exist and how many cards each agent has. Everyone knows their own but not the others' cards. The goal of Alice and Bob now is to learn each others cards without Carol learning their cards. They are only allowed to communicate via public announcements.

We begin implementing this situation by defining the set of players and the set of cards. To describe a card deal with boolean variables, we let P_k encode that agent k modulo 3 has card $\mathsf{floor}(\frac{k}{3})$. For example, P_{17} means that agent 2, namely Carol, has card 5 because $17 = (3 * 5) + 2$. The function `hasCard` in infix notation allows us to write more natural statements. We also use aliases `alice`, `bob` and `carol` for the agents.

```
rcPlayers :: [Agent]
rcPlayers = [alice,bob,carol]

rcNumOf :: Agent -> Int
rcNumOf "Alice" = 0
rcNumOf "Bob"   = 1
rcNumOf "Carol" = 2
rcNumOf _ = error "Unknown Agent"

rcCards :: [Int]
rcCards   = [0..6]

rcProps :: [Prp]
rcProps   = [ P k | k <-[0..((length rcPlayers * length rcCards)-1)] ]

hasCard :: Agent -> Int -> Form
hasCard i n = PrpF (P (3 * n + rcNumOf i))
```

```
>>> carol `hasCard` 5
```

```
PrpF (P 17)
```

2.04 seconds

We now describe which deals of cards are allowed. For a start, all cards have to be given to at least one agent but no card can be given to two agents.

```
allCardsGiven, allCardsUnique :: Form
allCardsGiven  = Conj [ Disj [ i `hasCard` n | i <- rcPlayers ] | n <- rcCards ]
allCardsUnique = Conj [ Neg $ isDouble n | n <- rcCards ] where
  isDouble n = Disj [ Conj [ x `hasCard` n, y `hasCard` n ] | x <- rcPlayers, y <-
      rcPlayers, x/=y, x<=y ]
```

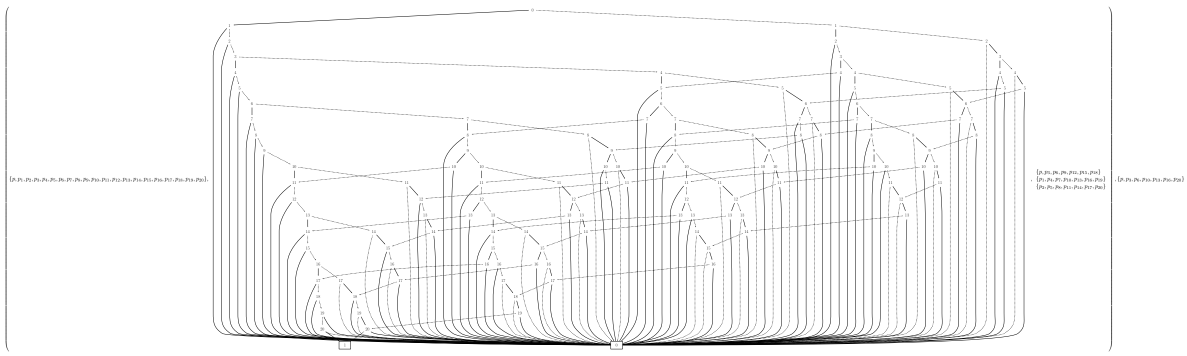
Moreover, Alice, Bob and Carol should get at least three, three and one card, respectively. As there are only seven cards in total this already implies that they can not have more.

```
distribute331 :: Form
distribute331 = Conj [ aliceAtLeastThree, bobAtLeastThree, carolAtLeastOne ] where
  aliceAtLeastThree = Disj [ Conj (map (alice `hasCard`) [x, y, z]) | x<-rcCards, y<-
      rcCards, z<-rcCards, x/=y, x/=z, y/=z  ]
  bobAtLeastThree = Disj [ Conj (map (bob `hasCard`) [x, y, z]) | x<-rcCards, y<-rcCards, z
      <-rcCards, x/=y, x/=z, y/=z  ]
  carolAtLeastOne = Disj [ carol `hasCard` k | k<-[0..6] ]
```

We can now define the initial knowledge structure. The state law describes all possible distributions using the three conditions we just defined. As a default deal we give the cards $\{0,1,2\}$ to Alice, $\{3,4,5\}$ to Bob and $\{6\}$ to Carol.

```
rusSCN :: Scenario
rusKNS :: KnowStruct
rusSCN@(rusKNS,_) = (KnS rcProps law [ (i, obs i) | i <- rcPlayers ], defaultDeal) where
  law = boolBddOf $ Conj [ allCardsGiven, allCardsUnique, distribute331 ]
  obs i = [ P (3 * k + rcNumOf i) | k<-[0..6] ]
  defaultDeal = [P 0,P 3,P 6,P 10,P 13,P 16,P 20]
```

The initial knowledge structure for Russian Cards looks as follows. The BDD describing the state law is generated within less than a second but drawing it is more complicated and the result quite huge:



Many different solutions for Russian Cards exist. Here we will focus on so-called five-hands protocols (and their extensions with six or seven hands) which are also used in [11]: First Alice makes an announcement of the form "My hand is one of these: ...". If her hand is 012 she could for example take the set $\{012, 034, 056, 135, 146, 236\}$. It can be checked that this announcement does not tell Carol anything, independent of which card it has. In contrast, Bob will be able to rule out all but one of the hands in the list because of his own hand. Hence the second and last step of the protocol is that Bob says which card Carol has. For example, if Bob's hand is 345 he would finish the protocol with "Carol has card 6.".

To verify this protocol with our model checker we first define the two formulas for Alice saying "My hand is one of 012, 034, 056, 135 and 246." and Bob saying "Carol holds card 6". Note we prefix the statements with knowledge operators. This reflects that Alice and Bob make the announcements and thus the real announcement is "Alice knows that one of her cards is 012, 034, 056, 135 and 246." and "Bob knows that Carol holds card 6.".

```
aAnnounce :: Form
aAnnounce = K alice $ Disj [ Conj (map (alice `hasCard`) hand) |
  hand <- [ [0,1,2], [0,3,4], [0,5,6], [1,3,5], [2,4,6] ] ]
```

```
bAnnounce :: Form
bAnnounce = K bob (carol `hasCard` 6)
```

To describe the goals of the protocol we need formulas about the knowledge of the three agents: Alice should know Bob's cards, Bob should know Alice's cards, and Carol should be ignorant, i.e. not know for any card that Alice or Bob has it. Note that Carol will still know for one card that neither Alice and Bob have them, namely his own. This is why we use $K^?$ (which is `Kw` in Haskell) for the first two but only the plain K for the last condition.

```
aKnowsBs, bKnowsAs, cIgnorant :: Form
aKnowsBs = Conj [ alice `Kw` (bob `hasCard` k) | k<-rcCards ]
bKnowsAs = Conj [ bob `Kw` (alice `hasCard` k) | k<-rcCards ]
cIgnorant = Conj $ concat [ [ Neg $ K carol $ alice `hasCard` i
                            , Neg $ K carol $ bob   `hasCard` i ] | i<-rcCards ]
```

We can now check how the knowledge of the agents changes during the communication, i.e. after the first and the second announcement. First we check that Alice says the truth.

```
rcCheck :: Int -> Form
rcCheck 0 = aAnnounce
```

After Alice announces five hands, Bob knows Alice's card and this is common knowledge among them.

```
rcCheck 1 = PubAnnounce aAnnounce bKnowsAs
rcCheck 2 = PubAnnounce aAnnounce (Ck [alice,bob] bKnowsAs)
```

And Bob knows Carol's card. This is entailed by the fact that Bob knows Alice's cards.

```
rcCheck 3 = PubAnnounce aAnnounce (K bob (PrpF (P 20)))
```

Carol remains ignorant of Alice's and Bob's cards, and this is common knowledge.

```
rcCheck 4 = PubAnnounce aAnnounce (Ck [alice,bob,carol] cIgnorant)
```

After Bob announces Carol's card, it is common knowledge among Alice and Bob that they know each others cards and Carol remains ignorant.

```
rcCheck 5 = PubAnnounce aAnnounce (PubAnnounce bAnnounce (Ck [alice,bob] aKnowsBs))
rcCheck 6 = PubAnnounce aAnnounce (PubAnnounce bAnnounce (Ck [alice,bob] bKnowsAs))
rcCheck _ = PubAnnounce aAnnounce (PubAnnounce bAnnounce (Ck rcPlayers cIgnorant))

rcAllChecks :: Bool
rcAllChecks = evalViaBdd rusSCN (Conj (map rcCheck [0..7]))
```

Verifying this protocol for the fixed deal 012|345|6 with our symbolic model checker takes about one second. Moreover, checking multiple protocols in a row does not take much longer because the BDD package caches results. Compared to that, the DEMO implementation from [11] needs 4 seconds to check one protocol.

```
>>> SMCDEL.Examples.rcAllChecks

True

2.22 seconds
```

We can not just verify but also *find* all protocols based on a set of five, six or seven hands, using the following combination of manual reasoning and brute-force. The following function `checkSet` takes a set of cards and returns whether it can safely be used by Alice.

```
checkSet :: [[Int]] -> Bool
checkSet set = all (evalViaBdd rusSCN) fs where
  aliceSays = K alice (Disj [ Conj $ map (alice ‘hasCard‘) h | h <- set ])
  bobSays = K bob (carol ‘hasCard‘ 6)
  fs = [ aliceSays
       , PubAnnounce aliceSays bKnowsAs
       , PubAnnounce aliceSays (Ck [alice,bob] bKnowsAs)
       , PubAnnounce aliceSays (Ck [alice,bob,carol] cIgnorant)
       , PubAnnounce aliceSays (PubAnnounce bobSays (Ck [alice,bob] $ Conj [aKnowsBs,
         bKnowsAs]))
       , PubAnnounce aliceSays (PubAnnounce bobSays (Ck rcPlayers cIgnorant)) ]

possibleHands :: [[Int]]
possibleHands = [ [x,y,z] | x <- rcCards, y <- rcCards, z <-rcCards, x < y, y < z ]

pickHands :: [ [Int] ] -> Int -> [ [ [Int] ] ]
pickHands _ 0 = [ [ [ ] ] ]
pickHands unused 1 = [ [h] | h <- unused ]
pickHands unused n = concat [ [ h:hs | hs <- pickHands (myfilter h unused) (n-1) ]  | h <-
    unused ] where
  myfilter h = filter (\xs -> length (h ‘intersect‘ xs) < 2 && h < xs)
```

The last line includes two important restrictions to the set of possible lists of hands that we will consider. First, Proposition 32 in [9] tells us that safe announcements from Alice never contain “crossing” hands, i.e. two hands which have more than one card in common. Second, without loss of generality we can assume that the hands in her announcement are lexicographically ordered. This leaves us with 1290 possible lists of five, six or seven hands of three cards.

```
allHandLists :: [ [ [Int] ] ]
allHandLists = concatMap (pickHands possibleHands) [5,6,7]
```

```
>>> length allHandLists
```

```
1290
```

2.14 seconds

Which of these are actually safe announcements that can be used by Alice? We can find them by checking 1290 instances of `checkSet` above. Our model checker can filter out the 102 safe announcements within seconds, generating and verifying the same list as in [9, Figure 3] where it was manually generated.

```
*EXAMPLES> mapM_ print (sort (filter checkSet allHandLists))
[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6]]
[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]]
[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,4,5]]
[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[2,3,6],[2,4,5]]
...
[[0,1,2],[0,5,6],[1,3,6],[1,4,5],[2,3,5],[2,4,6]]
[[0,1,2],[0,5,6],[1,3,6],[2,4,6],[3,4,5]]
[[0,1,2],[0,5,6],[1,4,5],[2,3,5],[3,4,6]]
[[0,1,2],[0,5,6],[1,4,6],[2,3,6],[3,4,5]]
(3.39 secs, 825215584 bytes)
```

```
>>> length (filter checkSet allHandLists)
```

```
102
```

3.90 seconds

Protocol synthesis . We now adopt an even more general perspective which is considered in [17]. Fix that Alice has $\{0, 1, 2\}$ and that she will announce 5 hands, including this one. Hence she has to pick 4 other hands of three cards each, i.e. she has to choose among 46376 possible actions.

```
pickHandsNaive :: [ [Int] ] -> Int -> [ [ [Int] ] ]
pickHandsNaive _ 0 = [ [ [ ] ] ]
pickHandsNaive unused 1 = [ [h] | h <- unused ]
pickHandsNaive unused n = concat [ [ h:hs | hs <- pickHandsNaive (myfilter h unused) (n-1)
    ]  | h <- unused ] where
  myfilter h = filter (\xs -> h < xs)

alicesActions :: [[[Int]]]
alicesActions = pickHandsNaive (delete [0,1,2] possibleHands) 4
```

```
>>> choose ((choose 7 3)-1) 4 == length alicesActions

True

2.10 seconds
```

```
alicesForms :: [Form]
alicesForms = map translate alicesActions

translate :: [[Int]] -> Form
translate set = Disj [ Conj $ map (alice `hasCard`) h | h <- [0,1,2]:set ]

bobsForms :: [Form]
bobsForms = [carol `hasCard` n | n <- reverse [0..6]] -- FIXME relax!

allPlans :: [(Form,Form)]
allPlans = [ (a,b) | a <- alicesForms, b <- bobsForms ]
```

For example: $\bigvee\{\bigwedge\{p, p_3, p_6\}, \bigwedge\{p, p_3, p_9\}, \bigwedge\{p, p_3, p_{12}\}, \bigwedge\{p, p_3, p_{15}\}, \bigwedge\{p, p_3, p_{18}\}\}$

```
testPlan :: (Form,Form) -> Bool
testPlan (aSays,bSays) = all (evalViaBdd rusSCN) fs where
  fs = [ aSays
       , PubAnnounce aSays bKnowsAs
       , PubAnnounce aSays (Ck [alice,bob] bKnowsAs)
       , PubAnnounce aSays (Ck [alice,bob,carol] cIgnorant)
       , PubAnnounce aSays bSays
       , PubAnnounce aSays (PubAnnounce bSays (Ck [alice,bob] $ Conj [aKnowsBs, bKnowsAs]))
       , PubAnnounce aSays (PubAnnounce bSays (Ck [alice,bob,carol] cIgnorant)) ]

rcSolutions :: [(Form, Form)]
rcSolutions = filter testPlan allPlans
```

It now takes 160.89 seconds to generate all the working plans when we fix `bobsForms = carol `hasCard` 6`. Given the definition above it takes 1125.06 seconds. In both cases we find the same 60 solutions.

6.10 Sum and Product

Our model checker can also be used to solve the Sum & Product puzzle from [19]. To represent numbers we use binary encodings for x, y, $x + y$ and $x * y$.

First we check on which states the DEL formula characterizing a solution holds. Finally, we verify that the state (4,13) is the only solution.

```
-- possible pairs 1<x<y, x+y<=100
pairs :: [(Int, Int)]
pairs = [(x,y) | x<-[2..100], y<-[2..100], x<y, x+y<=100]

-- 7 propositions are enough to label [2..100]
xProps, yProps, sProps, pProps :: [Prp]
xProps = [(P  1)..(P  7)]
yProps = [(P  8)..(P 14)]
sProps = [(P 15)..(P 21)]
pProps = [(P 22)..(P (21+amount))] where amount = ceiling (logBase 2 (50*50) :: Double)
```

```haskell
sapAllProps :: [Prp]
sapAllProps = xProps ++ yProps ++ sProps ++ pProps

xIs, yIs, sIs, pIs :: Int -> Form
xIs n = booloutofForm (powerset xProps !! n) xProps
yIs n = booloutofForm (powerset yProps !! n) yProps
sIs n = booloutofForm (powerset sProps !! n) sProps
pIs n = booloutofForm (powerset pProps !! n) pProps

xyAre :: (Int,Int) -> Form
xyAre (n,m) = Conj [ xIs n, yIs m ]
```

For example: $\bigwedge\{p_1, p_2, p_3, p_4, p_6, \neg p_5, \neg p_7\}$

```haskell
sapKnStruct :: KnowStruct
sapKnStruct = KnS sapAllProps law obs where
  law = boolBddOf $ Disj [ Conj [ xyAre (x,y), sIs (x+y), pIs (x*y) ] | (x,y) <- pairs ]
  obs = [ (alice, sProps), (bob, pProps) ]

sapKnows :: Agent -> Form
sapKnows i = Conj [ xyAre p `Impl` K i (xyAre p) | p <- pairs ]

sapForm1, sapForm2, sapForm3 :: Form
sapForm1 = K alice $ Neg (sapKnows bob) -- Sum: I knew that you didn't know the numbers.
sapForm2 = sapKnows bob -- Product: Now I know the two numbers
sapForm3 = sapKnows alice -- Sum: Now I know the two numbers too

sapProtocol :: Form
sapProtocol = Conj [ sapForm1
                   , PubAnnounce sapForm1 sapForm2
                   , PubAnnounce sapForm1 (PubAnnounce sapForm2 sapForm3) ]
```

The solutions to the puzzle are those states where this conjunction holds, i.e. the states which survive a public announcement of it.

```haskell
sapSolutions :: [[Prp]]
sapSolutions = statesOf (SMCDEL.Symbolic.HasCacBDD.pubAnnounce sapKnStruct sapProtocol)
```

```
>>> sapSolutions

[[P 1,P 2,P 3,P 4,P 6,P 7,P 8,P 9,P 10,P 13,P 15,P 16,P 18,P 19,P 20,P 22,P 23,P 24,P
    25,P 26,P 27,P 30,P 32,P 33]]

7.11 seconds
```

The following helper function tells us what this set of propositions means:

```haskell
sapExplainState :: [Prp] -> String
sapExplainState truths = concat [ "x = ", nmbr xProps, ", y = ", nmbr yProps, ", ",
  "x+y = ", nmbr sProps, " and x*y = ", nmbr pProps ] where
    nmbr set = show.fromJust $ elemIndex (set `intersect` truths) (powerset set)
```

```
>>> map sapExplainState sapSolutions

["x = 4, y = 13, x+y = 17 and x*y = 52"]

7.35 seconds
```

We can also verify that it is a solution, and that it is the unique solution.
If x==4 and y==13, then the announcements are truthful.

```
>>> validViaBdd sapKnStruct (Impl (Conj [xIs 4, yIs 13]) sapProtocol)

True

7.28 seconds
```

And if the announcements are truthful, then x==4 and y==13.

```
>>> validViaBdd sapKnStruct (Impl sapProtocol (Conj [xIs 4, yIs 13]))

True

7.13 seconds
```

Our implementation is faster than the one in [26].

6.11 What Sum

The following puzzle is from [29] where it was implemented using DEMO.

```
wsBound :: Int
wsBound = 50

wsTriples :: [ (Int,Int,Int) ]
wsTriples = filter
  ( \(x,y,z) -> x+y==z || x+z==y || y+z==x )
  [ (x,y,z) | x <- [1..wsBound], y <- [1..wsBound], z <- [1..wsBound] ]

aProps,bProps,cProps :: [Prp]
(aProps,bProps,cProps) = ([(P 0)..(P k)],[(P $ k+1)..(P l)],[(P $ l+1)..(P m)]) where
  [k,l,m] = map (wsAmount*) [1,2,3]
  wsAmount = ceiling (logBase 2 (fromIntegral wsBound) :: Double)

aIs, bIs, cIs :: Int -> Form
aIs n = booloutofForm (powerset aProps !! n) aProps
bIs n = booloutofForm (powerset bProps !! n) bProps
cIs n = booloutofForm (powerset cProps !! n) cProps

wsKnStruct :: KnowStruct
wsKnStruct = KnS wsAllProps law obs where
  wsAllProps = aProps++bProps++cProps
  law = boolBddOf $ Disj [ Conj [ aIs x, bIs y, cIs z ] | (x,y,z) <- wsTriples ]
  obs = [ (alice, bProps++cProps), (bob, aProps++cProps), (carol, aProps++bProps) ]

wsKnowSelfA,wsKnowSelfB,wsKnowSelfC :: Form
wsKnowSelfA = Disj [ K alice $ aIs x | x <- [1..wsBound] ]
wsKnowSelfB = Disj [ K bob   $ bIs x | x <- [1..wsBound] ]
wsKnowSelfC = Disj [ K carol $ cIs x | x <- [1..wsBound] ]

wsProtocol :: Form
wsProtocol = Conj
  [ Neg wsKnowSelfA
  , PubAnnounce (Neg wsKnowSelfA) (Neg wsKnowSelfB)
  , PubAnnounce (Neg wsKnowSelfA) (PubAnnounce (Neg wsKnowSelfB) (Neg wsKnowSelfC)) ]

wsSolutions :: [[Prp]]
wsSolutions = statesOf (SMCDEL.Symbolic.HasCacBDD.pubAnnounce wsKnStruct wsProtocol)

wsExplainState :: [Prp] -> String
wsExplainState truths = concat
  [ "a = ", nmbr aProps, ", b = ", nmbr bProps, ", ", "c = ", nmbr cProps ] where
    nmbr set = show.fromJust $ elemIndex (set `intersect` truths) (powerset set)
```

Use `fmap length (mapM (putStrLn.wsExplainState) wsSolutions)` to list and count solutions.

`wsBound`	Runtime DEMO [29]	Runtime SMCDEL	# Solutions
10	1.59	0.22	2
20	30.31	0.27	36
30	193.20	0.23	100
40	?	0.6-	198

7 Benchmarks

We now provide two different benchmarks for SMCDEL. All measurements were done under 64-bit Debian GNU/Linux 8 with kernel 3.16.0-4 running on an Intel Core i3-2120 3.30GHz processor and 4GB of memory. Code was compiled with GHC 7.10.3 and g++ 4.9.2.

7.1 Muddy Children

In this section we compare the performance of different model checking approaches to the muddy children example from Section 6.5.

- SMCDEL with two different BDD packages: CacBDD and CUDD.
- DEMO-S5, a version of the epistemic model checker DEMO optimized for S5 [14, 15].
- MCTRIANGLE, an ad-hoc implementation of [22], see Appendix 1 on page 60.

Note that to run this program all libraries, in particular the BDD packages have to be installed and get found by the dynamic linker.

```
module Main where
import Criterion.Main
import Data.Function
import Data.List
import Data.Maybe (fromJust)
import Data.Ord (comparing)
import SMCDEL.Language
import SMCDEL.Examples
import SMCDEL.Internal.Help (apply,seteq)
import qualified SMCDEL.Explicit.DEMO_S5 as DEMO_S5
import qualified SMCDEL.Explicit.Simple
import qualified SMCDEL.Symbolic.HasCacBDD
import qualified SMCDEL.Symbolic.CUDD
import qualified SMCDEL.Translations
import qualified SMCDEL.Other.MCTRIANGLE
import qualified SMCDEL.Other.NonS5
import Data.Map.Strict (fromList)
```

This benchmark compares how long it takes to answer the following question: "For n children, when m of them are muddy, how many announcements of »Nobody knows their own state.« are needed to let at least one child know their own state?". For this purpose we recursively define the formula to be checked and a general loop function which uses a given model checker to find the answer.

```
checkForm :: Int -> Int -> Form
checkForm n 0 = nobodyknows n
checkForm n k = PubAnnounce (nobodyknows n) (checkForm n (k-1))

findNumberWith :: (Int -> Int -> a, a -> Form -> Bool) -> Int -> Int -> Int
findNumberWith (start,evalfunction) n m = k where
  k | loop 0 == (m-1) = m-1
    | otherwise       = error $ "wrong Muddy Children result: " ++ show (loop 0)
  loop count = if evalfunction (start n m) (PubAnnounce (father n) (checkForm n count))
    then loop (count+1)
    else count

mudPs :: Int -> [Prp]
mudPs n = [P 1 .. P n]
```

We now instantiate this function with the `evalViaBdd` function from our four different versions of SMCDEL, linked to the different BDD packages.

```
findNumberCacBDD :: Int -> Int -> Int
findNumberCacBDD = findNumberWith (cacMudScnInit,SMCDEL.Symbolic.HasCacBDD.evalViaBdd)
    where
  cacMudScnInit n m = ( SMCDEL.Symbolic.HasCacBDD.KnS (mudPs n) (SMCDEL.Symbolic.HasCacBDD.
      boolBddOf Top) [ (show i,delete (P i) (mudPs n)) | i <- [1..n] ], mudPs m )

findNumberCUDD :: Int -> Int -> Int
findNumberCUDD = findNumberWith (cuddMudScnInit,SMCDEL.Symbolic.CUDD.evalViaBdd) where
  cuddMudScnInit n m = ( SMCDEL.Symbolic.CUDD.KnS (mudPs n) (SMCDEL.Symbolic.CUDD.boolBddOf
      Top) [ (show i,delete (P i) (mudPs n)) | i <- [1..n] ], mudPs m )

findNumberTrans :: Int -> Int -> Int
findNumberTrans = findNumberWith (start,SMCDEL.Symbolic.HasCacBDD.evalViaBdd) where
  start n m = SMCDEL.Translations.kripkeToKns $ mudKrpInit n m

mudKrpInit :: Int -> Int -> SMCDEL.Explicit.Simple.PointedModel
mudKrpInit n m = (SMCDEL.Explicit.Simple.KrM ws rel val, cur) where
  ws    = [0..(2^n-1)]
  rel   = [ (show i, erelFor i) | i <- [1..n] ] where
    erelFor i = sort $ map sort $
      groupBy ((==) `on` setForAt i) $
      sortBy (comparing (setForAt i)) ws
    setForAt i s = delete (P i) $ setAt s
    setAt s = map fst $ filter snd (apply val s)
  val         = zip ws table
  ((cur,_):_) = filter (\(_,ass)-> sort (map fst $ filter snd ass) == [P 1..P m]) val
  table = foldl buildTable [[]] [P k | k<- [1..n]]
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr<-partrows ]

findNumberNonS5 :: Int -> Int -> Int
findNumberNonS5 = findNumberWith
  (SMCDEL.Other.NonS5.mudGenScnInit,SMCDEL.Other.NonS5.evalViaBdd)

findNumberNonS5Trans :: Int -> Int -> Int
findNumberNonS5Trans = findNumberWith (start,SMCDEL.Other.NonS5.evalViaBdd) where
  start n m = SMCDEL.Other.NonS5.genKrp2Kns $ mudGenKrpInit n m

mudGenKrpInit :: Int -> Int -> SMCDEL.Other.NonS5.GeneralPointedModel Int
mudGenKrpInit n m = (fromList wlist, cur) where
  wlist = [ (w, (fromList (vals !! w), fromList $ relFor w)) | w <- ws ]
  ws    = [0..(2^n-1)]
  vals  = map sort (foldl buildTable [[]] [P k | k<- [1..n]])
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr <- partrows ]
  relFor w = [ (show i, seesFrom i w) | i <- [1..n] ]
  seesFrom i w = filter (\v -> samefor i (vals !! w) (vals !! v)) ws
  samefor i ps qs = seteq (delete (P i) (map fst $ filter snd ps)) (delete (P i) (map fst $
      filter snd qs))
  cur = fromJust (elemIndex curVal vals)
  curVal = sort $ [(p,True) | p <- [P 1 .. P m]] ++ [(p,True) | p <- [P (m+1) .. P n]]
```

However, for an explicit state model checker like DEMO-S5 we can not use the same loop function because we want to hand on the current model to the next step instead of computing it again and again.

```
mudDemoKrpInit :: Int -> Int -> DEMO_S5.EpistM [Bool]
mudDemoKrpInit n m = DEMO_S5.Mo states agents [] rels points where
  states = DEMO_S5.bTables n
  agents = map DEMO_S5.Ag [1..n]
  rels = [(DEMO_S5.Ag i, [[tab1++[True]++tab2,tab1++[False]++tab2] |
                 tab1 <- DEMO_S5.bTables (i-1),
                 tab2 <- DEMO_S5.bTables (n-i) ]) | i <- [1..n] ]
  points = [replicate (n-m) False ++ replicate m True]

findNumberDemoS5 :: Int -> Int -> Int
findNumberDemoS5 n m = findNumberDemoLoop n m 0 start where
  start = DEMO_S5.updPa (mudDemoKrpInit n m) (DEMO_S5.fatherN n)

findNumberDemoLoop :: Int -> Int -> Int -> DEMO_S5.EpistM [Bool] -> Int
findNumberDemoLoop n m count curMod =
  if DEMO_S5.isTrue curMod (DEMO_S5.dont n)
    then findNumberDemoLoop n m (count+1) (DEMO_S5.updPa curMod (DEMO_S5.dont n))
    else count
```

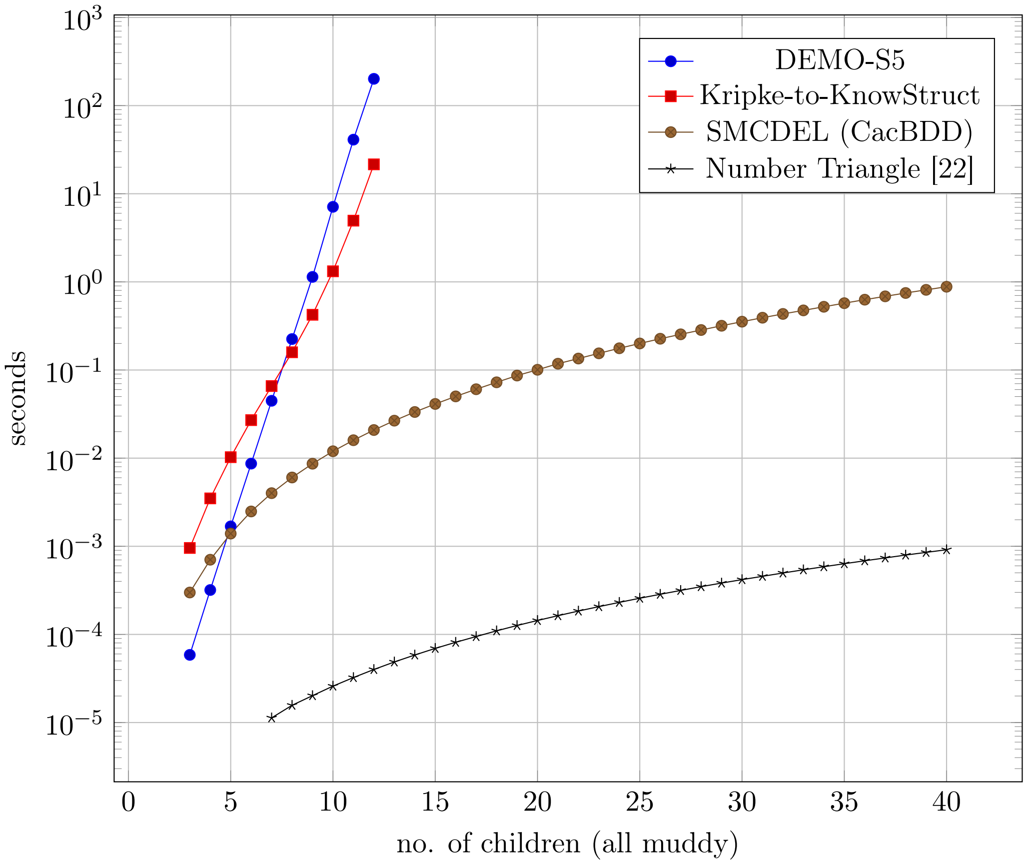


Figure 1: Benchmark Results on a logarithmic scale.

Also the number triangle approach to the Muddy Children puzzle has to be treated separately. See [22] and Appendix 1 on page 60 for the details. Here the formula `nobodyknows` does not depend on the number of agents and therefore the loop function does not have to pass on any variables.

```
findNumberTriangle :: Int -> Int -> Int
findNumberTriangle n m = findNumberTriangleLoop 0 start where
  start = SMCDEL.Other.MCTRIANGLE.update (SMCDEL.Other.MCTRIANGLE.mcModel (n-m,m)) (SMCDEL.
      Other.MCTRIANGLE.Qf SMCDEL.Other.MCTRIANGLE.some)

findNumberTriangleLoop :: Int -> SMCDEL.Other.MCTRIANGLE.McModel -> Int
findNumberTriangleLoop count curMod =
  if SMCDEL.Other.MCTRIANGLE.eval curMod SMCDEL.Other.MCTRIANGLE.nobodyknows
    then findNumberTriangleLoop (count+1) (SMCDEL.Other.MCTRIANGLE.update curMod SMCDEL.
        Other.MCTRIANGLE.nobodyknows)
    else count
```

The following function uses the library *Criterion* to benchmark all the solution methods we defined.

```
main :: IO ()
main = defaultMain (map mybench
  [ ("Triangle"  , findNumberTriangle  , [7..40] )
  , ("CacBDD"    , findNumberCacBDD    , [3..40] )
  , ("CUDD"      , findNumberCUDD      , [3..40] )
  , ("NonS5"     , findNumberNonS5     , [3..12] )
  , ("DEMOS5"    , findNumberDemoS5    , [3..12] )
  , ("Trans"     , findNumberTrans     , [3..12] )
  , ("NonS5Trans", findNumberNonS5Trans , [3..11] ) ])
  where
    mybench (name,f,range) = bgroup name $ map (run f) range
    run f k = bench (show k) $ whnf (\n -> f n n) k
```

As expected we can see in Figure 1 that *SMCDEL* is faster than the explicit model checker DEMO.

Finally, the number triangle approach from [22] is way faster than all others, especially for large numbers of agents. This is not surprising, though: Both the model and the formula which are checked here are smaller and the semantics was specifically adapted to the muddy children example. Concretely, the size of the model is linear in the number of agents and the length of the formula is constant. It will be subject to future work if the idea underlying this approach – the identification of agents in the same informational state – can be generalized to other protocols or ideally the full DEL language.

7.2 Dining Cryptographers

Muddy Children has also been used to benchmark MCMAS [25] but the formula checked there concerns the correctness of behavior and not how many rounds are needed. Moreover, the interpreted system semantics of model checkers like MCMAS are very different from DEL. Still, connections between DEL and temporal logics have been studied and translations are available [3, 12].

A protocol which fits nicely into both frameworks are the Dining Cryptographers [5] which we implemented in Section 6.8. We will now use it to measure the performance of *SMCDEL* in a way that is more similar to [25].

```
module Main (main) where
import Control.Monad (when)
import Data.Time (diffUTCTime,getCurrentTime,NominalDiffTime)
import System.Environment (getArgs)
import System.IO (hSetBuffering,BufferMode(NoBuffering),stdout)
import SMCDEL.Language
import SMCDEL.Symbolic.HasCacBDD
import SMCDEL.Examples (genDcKnsInit,genDcReveal)
```

The following statement was also checked with MCMAS in [25].

> "If cryptographer 1 did not pay the bill, then after the announcements are made, he knows that no cryptographers paid, or that someone paid, but in this case he does not know who did."

Following ideas and conventions from [3, 12] we can formalize it in DEL as

$$\neg p_1 \to [!\psi] \left(K_1(\bigwedge_{i=1}^{n} \neg p_i) \vee \left(K_1(\bigvee_{i=2}^{n} p_i) \wedge \bigwedge_{i=2}^{n} (\neg K_1 p_i) \right) \right)$$

where p_i says that agent i paid and $!\psi$ is the announcement whether the number of agents which announced a 1 is odd, i.e. $\psi := \bigoplus_i \bigoplus \{p \mid \text{Agent } i \text{ can observe } p\}$.

```
genDcCheckForm :: Int -> Form
genDcCheckForm n = Impl (Neg (PrpF $ P 1)) $
  PubAnnounceW (Xor [genDcReveal n i | i<-[1..n] ]) $
    Disj [ K "1" (Conj [Neg $ PrpF $ P k | k <- [1..n]  ])
         , Conj [ K "1" (Disj [ PrpF $ P k | k <- [2..n] ])
                , Conj [ Neg $ K "1" (PrpF $ P k) | k <- [2..n] ] ] ]
```

```
genDcValid :: Int -> Bool
genDcValid n = validViaBdd (genDcKnsInit n) (genDcCheckForm n)

dcTimeThis :: Int -> IO NominalDiffTime
dcTimeThis n = do
  start <- getCurrentTime
  let mykns@(KnS props _ _) = genDcKnsInit n
  putStr $ show (length props) ++ "\t"
  putStr $ show (length $ show mykns) ++ "\t"
  putStr $ show (length $ show $ genDcCheckForm n) ++ "\t"
  if genDcValid n then do
    end <- getCurrentTime
```

```
    return (end `diffUTCTime` start)
  else
    error "Wrong result."

mainLoop :: [Int] -> Int -> IO ()
mainLoop [] _ = putStrLn ""
mainLoop (n:ns) limit = do
  putStr $ show n ++ "\t"
  result <- dcTimeThis n
  print result
  when (result <= fromIntegral limit) $ mainLoop ns limit

main :: IO ()
main = do
  args <- getArgs
  hSetBuffering stdout NoBuffering
  limit <- case args of
    [aInteger] | [(n,_)] <- reads aInteger -> return n
    _ -> do
      putStrLn "No maximum runtime given, defaulting to one second."
      return 1
  putStrLn $ "n" ++ "\tn(prps)"++ "\tsz(KNS)"++ "\tsz(frm)" ++ "\ttime"
  mainLoop (3:(5 : map (10*) [1..])) limit
```

The program outputs the following table which shows (i) the number of cryptographers, (ii) the number of propositions used, (iii) the length of the knowledge structure, (iv) the length of the formula and (v) the time in seconds needed by SMCDEL to check it.

These results are satisfactory: While MCMAS already needs more than 10 seconds to check the interpreted system for 50 or more dining cryptographers (see [25, Table 4]), *SMCDEL* can deal with the case of up to 160 agents in less time.

7.3 Sum and Product

We compare the performance of SMCDEL and DEMO-S5 on the Sum & Product problem.

```
module Main
where
import Data.List (groupBy,sortBy)
import Data.Time (getCurrentTime, diffUTCTime)
import SMCDEL.Explicit.DEMO_S5
import SMCDEL.Examples (sapExplainState,sapSolutions)
```

We use the implementation in the module `SMCDEL.Examples`, see Section 6.10.

```
runSMCDEL :: IO ()
runSMCDEL = do
  putStrLn "The solution is:"
  mapM_ (putStrLn . sapExplainState) sapSolutions
```

The following is based on the DEMO version from `http://www.cs.otago.ac.nz/staffpriv/hans/sumpro/`.

```
--possible pairs 1<x<y, x+y<=100
allpairs :: [(Int,Int)]
allpairs  = [(x,y)|x<-[2..100], y<-[2..100], x<y, x+y<=100]

alice, bob :: Agent
(alice,bob) = (Ag 0,Ag 1)

--initial pointed epistemic model
msnp :: EpistM (Int,Int)
msnp = Mo allpairs [alice,bob] [] rels allpairs where
  rels  = [ (alice,partWith (+)) , (bob,partWith (*)) ]
  partWith op = groupBy (\(x,y) (x',y') -> op x y == op x' y') $
    sortBy (\(x,y) (x',y') -> compare (op x y) (op x' y')) allpairs

fmrs1e, fmrp2e, fmrs3e :: Form (Int,Int)
```

```
--Sum says: I knew that you didn't know the two numbers.
fmrs1e = Kn alice (Conj [Disj[Ng (Info p),
                         Ng (Kn bob (Info p))]| p<-allpairs])

--Product says: Now I know the two numbers
fmrp2e = Conj [ Disj[Ng (Info p),
                     Kn bob (Info p) ] |p<-allpairs]

--Sum says: Now I know the two numbers too
fmrs3e = Conj [ Disj[Ng (Info p),
                     Kn alice (Info p) ] |p<-allpairs]

runDemoS5 :: IO ()
runDemoS5 = print $ updsPa msnp [fmrs1e, fmrp2e, fmrs3e]
```

```
main :: IO ()
main = do
  putStrLn "*** Running DEMO_S5 ***"
  start <- getCurrentTime
  runDemoS5
  end <- getCurrentTime
  putStrLn $ "This took " ++ show (end `diffUTCTime` start) ++ " seconds.\n"

  putStrLn "*** Running SMCDEL ***"
  start2 <- getCurrentTime
  runSMCDEL
  end2 <- getCurrentTime
  putStrLn $ "This took " ++ show (end2 `diffUTCTime` start2) ++ " seconds.\n"
```

8 The Standalone Executable

To simplify the usage of our model checker, we also provide a standalone executable. This means we only have to compile the model checker once and then can run it on different structures and formulas. Our input format are simple text files, like this:

```
-- Three Muddy Children in SMCDEL
VARS 1,2,3
LAW  Top
OBS  alice: 2,3
     bob: 1,3
     carol: 1,2

VALID?
  ( ~ (alice knows whether 1)
  & ~ (bob   knows whether 2)
  & ~ (carol knows whether 3) )

WHERE?
  ~ (1|2|3)

WHERE?
  < ! (1|2|3) >
  ( (alice knows whether 1)
  | (bob   knows whether 2)
  | (carol knows whether 3) )

VALID?
  [ ! (1|2|3) ]
  [ ! ( (~ (alice knows whether 1))
      & (~ (bob   knows whether 2))
      & (~ (carol knows whether 3)) ) ]
  [ ! ( (~ (alice knows whether 1))
      & (~ (bob   knows whether 2))
      & (~ (carol knows whether 3)) ) ]
  (1 & 2 & 3)
```

If we run SMCDEL on this file we get the following output:

```
Is Conj [Conj [Neg (Kw "alice" (PrpF (P 1))),Neg (Kw "bob" (PrpF (P 2)))],Neg (Kw "carol" (
    PrpF (P 3)))] valid on the given structure?
True

At which states is Neg (Disj [Disj [PrpF (P 1),PrpF (P 2)],PrpF (P 3)]) true?
[]

At which states is Neg (PubAnnounce (Disj [Disj [PrpF (P 1),PrpF (P 2)],PrpF (P 3)]) (Neg (
    Neg (Conj [Conj [Neg (Kw "alice" (PrpF (P 1))),Neg (Kw "bob" (PrpF (P 2)))],Neg (Kw "
    carol" (PrpF (P 3)))])))) true?
[1]
[2]
[3]

Is PubAnnounce (Disj [Disj [PrpF (P 1),PrpF (P 2)],PrpF (P 3)]) (PubAnnounce (Conj [Conj [
    Neg (Kw "alice" (PrpF (P 1))),Neg (Kw "bob" (PrpF (P 2)))],Neg (Kw "carol" (PrpF (P 3))
    )]) (PubAnnounce (Conj [Conj [Neg (Kw "alice" (PrpF (P 1))),Neg (Kw "bob" (PrpF (P 2)))
    ],Neg (Kw "carol" (PrpF (P 3)))]) (Conj [Conj [PrpF (P 1),PrpF (P 2)],PrpF (P 3)])))
    valid on the given structure?
True
```

Alternatively, we can get the following LaTeX output by running SMCDEL with the `-tex` flag.

Given Knowledge Structure

$$\left(\{p_1, p_2, p_3\}, \boxed{\;1\;}, \begin{matrix} \{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\} \end{matrix} \right), \varnothing$$

Results

$$((\neg K^?_{\text{alice}} p_1 \wedge \neg K^?_{\text{bob}} p_2) \wedge \neg K^?_{\text{carol}} p_3)$$

Is this valid on the given structure? True

$$\neg(p_1 \vee (p_2 \vee p_3))$$

At which states is this true? $\varnothing$

For more examples, see the `Examples` folder.

8.1 The Main routine

```
module Main where

import Control.Arrow (second)
import Control.Monad (when,unless)
import Data.List (intercalate)
import System.Console.ANSI
import System.Directory (getTemporaryDirectory)
import System.Environment (getArgs,getProgName)
import System.Exit (exitFailure)
import System.Process (system)
import System.FilePath.Posix (takeBaseName)
import System.IO (Handle,hClose,hPutStrLn,stderr,stdout,openTempFile)
import SMCDEL.Internal.Lex
import SMCDEL.Internal.Parse
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Symbolic.HasCacBDD

main :: IO ()
main = do
  (input,options) <- getInputAndSettings
  print options
  let showMode = "-show" `elem` options
  let texMode = "-tex" `elem` options || showMode
  tmpdir <- getTemporaryDirectory
  (texFilePath,texFileHandle) <- openTempFile tmpdir "smcdel.tex"
  let outHandle = if showMode then texFileHandle else stdout
  unless texMode $ vividPutStrLn infoline
  when texMode $ hPutStrLn outHandle texPrelude
  let (CheckInput vocabInts lawform obs jobs) = parse $ alexScanTokens input
  let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
  when texMode $
    hPutStrLn outHandle $ unlines
      [ "\\section{Given Knowledge Structure}", "\\[ (\\mathcal{F},s) = (" ++ tex ((mykns
        ,[])::Scenario) ++ ") \\]", "\\section{Results}" ]
  mapM_ (doJob outHandle texMode mykns) jobs
  when texMode $ hPutStrLn outHandle texEnd
  when showMode $ do
    hClose outHandle
    let command = "cd /tmp && pdflatex -interaction=nonstopmode " ++ takeBaseName
        texFilePath ++ ".tex > " ++ takeBaseName texFilePath ++ ".pdflatex.log && xdg-open
        "++ takeBaseName texFilePath ++ ".pdf"
    putStrLn $ "Now running: " ++ command
    _ <- system command
    return ()
  putStrLn "\nDoei!"

doJob :: Handle -> Bool -> KnowStruct -> Either Form Form -> IO ()
doJob outHandle True mykns (Left f) = do
  hPutStrLn outHandle $ "Is $" ++ texForm (simplify f) ++ "$ valid on $\\mathcal{F}$?"
  hPutStrLn outHandle (show (validViaBdd mykns f) ++ "\n\n")
doJob outHandle False mykns (Left f) = do
  hPutStrLn outHandle $ "Is " ++ ppForm f ++ " valid on the given structure?"
  vividPutStrLn (show (validViaBdd mykns f) ++ "\n")
doJob outHandle True mykns (Right f) = do
  hPutStrLn outHandle $ "At which states is $" ++ texForm (simplify f) ++ "$ true? $"
  let states = map tex (whereViaBdd mykns f)
```

```
  hPutStrLn outHandle $ intercalate "," states
  hPutStrLn outHandle "$\n"
doJob outHandle False mykns (Right f) = do
  hPutStrLn outHandle $ "At which states is " ++ ppForm f ++ " true?"
  mapM_ (vividPutStrLn.show.map(\(P n) -> n)) (whereViaBdd mykns f)
  putStr "\n"

getInputAndSettings :: IO (String,[String])
getInputAndSettings = do
  args <- getArgs
  case args of
    ("-":options) -> do
      input <- getContents
      return (input,options)
    (filename:options) -> do
      input <- readFile filename
      return (input,options)
    _ -> do
      name <- getProgName
      hPutStrLn stderr infoline
      hPutStrLn stderr $ "usage: " ++ name ++ " <filename> {Options}"
      hPutStrLn stderr $ "       (use filename " ++ name ++ " -  to read STDIN)"
      hPutStrLn stderr ""
      hPutStrLn stderr "  -tex   Output will be LaTeX code"
      hPutStrLn stderr ""
      hPutStrLn stderr "  -show  Write output to /tmp, generate PDF and show it."
      hPutStrLn stderr "         (implies -tex)"
      exitFailure

vividPutStrLn :: String -> IO ()
vividPutStrLn s = do
  setSGR [SetColor Foreground Vivid White]
  putStrLn s
  setSGR []

infoline :: String
infoline = "SMCDEL 16.5 by Malvin Gattinger -- https://github.com/jrclogic/SMCDEL\n"

texPrelude, texEnd :: String
texPrelude = unlines [ "\\documentclass[a4paper,12pt]{article}",
  "\\usepackage{amsmath,amssymb,tikz,graphicx,color,etex,datetime,setspace,latexsym}",
  "\\usepackage[margin=2cm]{geometry}",
  "\\usepackage[T1]{fontenc}", "\\parindent0cm", "\\parskip1em",
  "\\usepackage{hyperref}",
  "\\hypersetup{pdfborder={0 0 0}}",
  "\\title{Results}",
  "\\author{\\href{https://github.com/jrclogic/SMCDEL}{SMCDEL}}",
  "\\begin{document}",
  "\\maketitle" ]
texEnd = "\\end{document}"
```

8.2 Lexer and Parser

To read and interpret the text files we use Alex (`haskell.org/alex`) and Happy (`haskell.org/happy`).
The file `../src/SMCDEL/Internal/Token.hs`:

```
module SMCDEL.Internal.Token where
data Token a -- == AlexPn
  = TokenVARS               {apn :: a}
  | TokenLAW                {apn :: a}
  | TokenOBS                {apn :: a}
  | TokenVALIDQ             {apn :: a}
  | TokenWHEREQ             {apn :: a}
  | TokenColon              {apn :: a}
  | TokenComma              {apn :: a}
  | TokenStr {fooS::String, apn :: a}
  | TokenInt {fooI::Int,    apn :: a}
  | TokenTop                {apn :: a}
  | TokenBot                {apn :: a}
  | TokenPrp                {apn :: a}
```

```
  | TokenNeg               {apn :: a}
  | TokenOB                {apn :: a}
  | TokenCB                {apn :: a}
  | TokenCOB               {apn :: a}
  | TokenCCB               {apn :: a}
  | TokenLA                {apn :: a}
  | TokenRA                {apn :: a}
  | TokenExclam            {apn :: a}
  | TokenQuestm            {apn :: a}
  | TokenBinCon            {apn :: a}
  | TokenBinDis            {apn :: a}
  | TokenCon               {apn :: a}
  | TokenDis               {apn :: a}
  | TokenXor               {apn :: a}
  | TokenImpl              {apn :: a}
  | TokenEqui              {apn :: a}
  | TokenForall            {apn :: a}
  | TokenExists            {apn :: a}
  | TokenInfixKnowWhether  {apn :: a}
  | TokenInfixKnowThat     {apn :: a}
  | TokenInfixCKnowWhether {apn :: a}
  | TokenInfixCKnowThat    {apn :: a}
  deriving (Eq,Show)
```

The file `../src/SMCDEL/Internal/Lex.x`:

```
{
{-# OPTIONS_GHC -fno-warn-tabs -fno-warn-missing-signatures #-}
module SMCDEL.Internal.Lex where
import SMCDEL.Internal.Token
}

%wrapper "posn"

$dig = 0-9       -- digits
$alf = [a-zA-Z] -- alphabetic characters

tokens :-
  -- ignore whitespace and comments:
  $white+              ;
  "--".*               ;
  -- keywords and punctuation:
  "VARS"               { \ p _ -> TokenVARS            p }
  "LAW"                { \ p _ -> TokenLAW             p }
  "OBS"                { \ p _ -> TokenOBS             p }
  "VALID?"             { \ p _ -> TokenVALIDQ          p }
  "WHERE?"             { \ p _ -> TokenWHEREQ          p }
  ":"                  { \ p _ -> TokenColon           p }
  ","                  { \ p _ -> TokenComma           p }
  "("                  { \ p _ -> TokenOB              p }
  ")"                  { \ p _ -> TokenCB              p }
  "["                  { \ p _ -> TokenCOB             p }
  "]"                  { \ p _ -> TokenCCB             p }
  "<"                  { \ p _ -> TokenLA              p }
  ">"                  { \ p _ -> TokenRA              p }
  "!"                  { \ p _ -> TokenExclam          p }
  "?"                  { \ p _ -> TokenQuestm          p }
  -- DEL Formulas:
  "Top"                { \ p _ -> TokenTop             p }
  "Bot"                { \ p _ -> TokenBot             p }
  "~"                  { \ p _ -> TokenNeg             p }
  "Not"                { \ p _ -> TokenNeg             p }
  "not"                { \ p _ -> TokenNeg             p }
  "&"                  { \ p _ -> TokenBinCon          p }
  "|"                  { \ p _ -> TokenBinDis          p }
  "->"                 { \ p _ -> TokenImpl            p }
  "iff"                { \ p _ -> TokenEqui            p }
  "AND"                { \ p _ -> TokenCon             p }
  "OR"                 { \ p _ -> TokenDis             p }
  "XOR"                { \ p _ -> TokenXor             p }
  "ForAll"             { \ p _ -> TokenForall          p }
  "Forall"             { \ p _ -> TokenForall          p }
```

```
  "Exists"           { \ p _ -> TokenExists             p }
  "knows whether"    { \ p _ -> TokenInfixKnowWhether   p }
  "knows that"       { \ p _ -> TokenInfixKnowThat      p }
  "comknow whether" { \ p _ -> TokenInfixCKnowWhether p }
  "comknow that"     { \ p _ -> TokenInfixCKnowThat     p }
  -- Integers and Strings:
  $dig+              { \ p s -> TokenInt (read s)       p }
  $alf [$alf $dig]* { \ p s -> TokenStr s               p }
```

The file ../src/SMCDEL/Internal/Parse.y:

```
{
module SMCDEL.Internal.Parse where
import SMCDEL.Internal.Token
import SMCDEL.Internal.Lex
import SMCDEL.Language
}

%name parse CheckInput
%tokentype { Token AlexPosn }
%error { parseError }

%token
  VARS   { TokenVARS   _ }
  LAW    { TokenLAW    _ }
  OBS    { TokenOBS    _ }
  VALIDQ { TokenVALIDQ _ }
  WHEREQ { TokenWHEREQ _ }
  COLON  { TokenColon  _ }
  COMMA  { TokenComma  _ }
  TOP    { TokenTop    _ }
  BOT    { TokenBot    _ }
  '('    { TokenOB     _ }
  ')'    { TokenCB     _ }
  '['    { TokenCOB    _ }
  ']'    { TokenCCB    _ }
  '<'    { TokenLA     _ }
  '>'    { TokenRA     _ }
  '!'    { TokenExclam _ }
  '?'    { TokenQuestm _ }
  '&'    { TokenBinCon _ }
  '|'    { TokenBinDis _ }
  '~'    { TokenNeg    _ }
  '->'   { TokenImpl   _ }
  CON    { TokenCon    _ }
  DIS    { TokenDis    _ }
  XOR    { TokenXor    _ }
  STR    { TokenStr $$ _ }
  INT    { TokenInt $$ _ }
  'iff'  { TokenEqui   _ }
  KNOWSTHAT     { TokenInfixKnowThat      _ }
  KNOWSWHETHER { TokenInfixKnowWhether   _ }
  CKNOWTHAT     { TokenInfixCKnowThat     _ }
  CKNOWWHETHER { TokenInfixCKnowWhether _ }
  'Forall'      { TokenForall             _ }
  'Exists'      { TokenExists             _ }

%left '&'
%left '|'
%nonassoc '~'

%%

CheckInput : VARS IntList LAW Form OBS ObserveSpec JobList { CheckInput $2 $4 $6 $7 }
           | VARS IntList LAW Form OBS ObserveSpec { CheckInput $2 $4 $6 [] }
IntList : INT { [$1] }
        | INT COMMA IntList { $1:$3 }
Form : TOP { Top }
     | BOT { Bot }
     | '(' Form ')' { $2 }
     | '~' Form { Neg $2 }
     | CON '(' FormList ')' { Conj $3 }
```

```
      | Form '&' Form { Conj [$1,$3] }
      | Form '|' Form { Disj [$1,$3] }
      | Form '->' Form { Impl $1 $3 }
      | DIS '(' FormList ')' { Disj $3 }
      | XOR '(' FormList ')' { Xor $3 }
      | Form 'iff' Form { Equi $1 $3 }
      | INT { PrpF (P $1) }
      | String KNOWSTHAT Form { K $1 $3 }
      | String KNOWSWHETHER Form { Kw $1 $3 }
      | StringList CKNOWTHAT Form { Ck $1 $3 }
      | StringList CKNOWWHETHER Form { Ckw $1 $3 }
      | '(' StringList ')' CKNOWTHAT Form { Ck $2 $5 }
      | '(' StringList ')' CKNOWWHETHER Form { Ckw $2 $5 }
      | '[' '!' Form ']'      Form { PubAnnounce  $3 $5 }
      | '[' '?' '!' Form ']' Form { PubAnnounceW $4 $6 }
      | '<' '!' Form '>'      Form { Neg (PubAnnounce  $3 (Neg $5)) }
      | '<' '?' '!' Form '>' Form { Neg (PubAnnounceW $4 (Neg $6)) }
      -- announcements to a group:
      | '[' StringList '!' Form ']'      Form { Announce $2 $4 $6 }
      | '[' StringList '?' '!' Form ']' Form { AnnounceW $2 $5 $7 }
      | '<' StringList '!' Form '>'      Form { Neg (Announce  $2 $4 (Neg $6)) }
      | '<' StringList '?' '!' Form '>' Form { Neg (AnnounceW $2 $5 (Neg $7)) }
      -- boolean quantifiers:
      | 'Forall' IntList Form { Forall (map P $2) $3 }
      | 'Exists' IntList Form { Exists (map P $2) $3 }
FormList : Form { [$1] } | Form COMMA FormList { $1:$3 }
String : STR { $1 }
StringList : String { [$1] } | String COMMA StringList { $1:$3 }
ObservLine : STR COLON IntList { ($1,$3) }
ObserveSpec : ObservLine { [$1] } | ObservLine ObserveSpec { $1:$2 }
JobList : JobLine { [$1] } | JobLine JobList { $1:$2 }
JobLine : VALIDQ Form { Left $2 } | WHEREQ Form { Right $2 }

{
data CheckInput = CheckInput [Int] Form [(String,[Int])] [Either Form Form] deriving (Show,
    Eq,Ord)
type IntList = [Int]
type FormList = [Form]
type ObserveLine = (String,IntList)
type ObserveSpec = [ObserveLine]

parseError :: [Token AlexPosn] -> a
parseError (t:ts) = error ("Parse error in line " ++ show lin ++ ", column " ++ show col)
  where (AlexPn abs lin col) = apn t
}
```

9 The Web Interface

We use *Scotty* from `https://github.com/scotty-web/scotty`.

9.1 The Main routine

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Prelude
import Control.Monad.IO.Class (liftIO)
import Control.Arrow
import Data.List (intercalate)
import Web.Scotty
import qualified Data.Text.Lazy as T
import SMCDEL.Internal.Lex
import SMCDEL.Internal.Parse
import SMCDEL.Internal.Files
import SMCDEL.Symbolic.HasCacBDD
import SMCDEL.Internal.TexDisplay
import SMCDEL.Translations
import SMCDEL.Language
import Data.HasCacBDD.Visuals (svgGraph)

main :: IO ()
main = do
  putStrLn "Please open this link: http://localhost:3000/index.html"
  scotty 3000 $ do
    get "" $ redirect "index.html"
    get "/" $ redirect "index.html"
    get "/index.html" $ html $ T.fromStrict $ getFileContent "index.html"
    get "/getExample" $ do
      this <- param "filename"
      html $ T.fromStrict $ getFileContent this
    post "/check" $ do
      smcinput <- param "smcinput"
      let (CheckInput vocabInts lawform obs jobs) = parse $ alexScanTokens smcinput
      let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
      knstring <- liftIO $ showStructure mykns
      let results = concatMap (\j -> "<p>" ++ doJob mykns j ++ "</p>") jobs
      html $ mconcat
        [ T.pack knstring
        , "<hr />\n"
        , T.pack results ]
    post "/knsToKripke" $ do
      smcinput <- param "smcinput"
      let (CheckInput vocabInts lawform obs _) = parse $ alexScanTokens smcinput
      let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
      _ <- liftIO $ showStructure mykns -- this moves parse errors to scotty
      if numberOfStates mykns > 20
        then html $ T.pack $ "Sorry, I will not draw " ++ show (numberOfStates mykns) ++ "
          states!"
        else do
          let myKripke = knsToKripke (mykns, head $ statesOf mykns) -- FIXME: how to pick
              actual world?
          html $ T.pack ("<p>" ++ svg myKripke ++ "</p>")

-- FIXME: merge with doJob in MainCLI.hs
doJob :: KnowStruct -> Either Form Form -> String
doJob mykns (Left f) = unlines
  [ "\\( \\mathcal{F} "
  , if validViaBdd mykns f then "\\vDash" else "\\not\\vDash"
  , (texForm.simplify) f
  , "\\)" ]
doJob mykns (Right f) = unlines
  [ "At which states is \\("
  , (texForm.simplify) f
  , "\\) true?<br /> \\("
  , intercalate "," $ map tex (whereViaBdd mykns f)
```

```
  , "\\)" ]

showStructure :: KnowStruct -> IO String
showStructure (KnS props lawbdd obs) = do
  svgString <- svgGraph lawbdd
  return $ "$$ \\mathcal{F} = \\left( \n"
    ++ tex props ++ ", "
    ++ " \\begin{array}{l} {"++ " \\href{javascript:toggleLaw()}{\\theta} " ++"} \\end{
        array}\n "
    ++ ", \\begin{array}{l}\n"
    ++ intercalate " \\\\\n " (map (\(i,os) -> ("O_{"++i++"}=" ++ tex os)) obs)
    ++ "\\end{array}\n"
    ++ " \\right) $$ \n <div class='lawbdd' style='display:none;'> where \\(\\theta\\) is
        this BDD:<br /><p align='center'>" ++ svgString ++ "</p></div>"
```

10 Future Work

We are planning to extend *SMCDEL* and continue our research in the following ways.

Non-S5 Models

Currently *SMCDEL* can only work on models where the epistemic accessibility relation is an equivalence relation. This is because only those can be described by sets of observational variables. And in fact not even every S5 relation on distinctly valuated worlds can be modeled with observational variables – this is why our translation procedure in Definition 15 has to use additional atomic propositions.

To overcome this limitation, we will generalize the definition of knowledge structures. Using well-known methods from temporal model checking, arbitrary relations can also be represented as BDDs. See for example [23]. Remember that in a knowledge structure we can identify states with boolean assignments and those are just sets of propositions. Hence a relation on states with unique valuations can be seen as a relation between sets of propositions. We can therefore represent it with the BDD of a characteristic function on a double vocabulary, as described in [7, Section 5.2]. Intuitively, we construct (the BDD of) a formula which is true exactly for the pairs of boolean assignments that are connected by the relation.

For a start, see the experimental module `SMCDEL.Other.NonS5` in the appendix.

Increase Usability

Our language syntax is globally fixed and contains only one enumerated set of atomic propositions. In contrast, the model checker DEMO(-S5) allows the user to parameterize the valuation function and the language according to her needs. For example, the muddy children can be represented with worlds of the type `[Bool]`, a list indicating their status. To allow symbolic model checking on Kripke models specified in this way we have to map user specified propositions to variables in the BDD package. In parallel, formulas using the general syntax should be translated to BDDs.

Reduction to SAT Solving

Instead of representing boolean functions with BDDs also SAT solvers are being used in model checking for temporal logics and provide an alternative approach for system verification. In our case we could do the following: Instead of translating DEL formulas to boolean formulas represented as BDDs we translate them to conjunctive or disjunctive normal forms of boolean formulas. These – probably very lengthy – boolean formulas can then be fed into a SAT solver, or in case we need to know whether they are tautologies, their negation.

Abstraction and Modal Logic

Epistemic and temporal logics have been connected before and also concrete translation methods have been proposed, see [3, 12]. Also similar to our observational variables are the "mental programs" recently presented in [4]. These and other ideas could also be implemented and their performance and applicability be compared to our approach.

Another direction would be to lift the symbolic representations of Kripke models for epistemic logics to modal logic in general and explore whether this gives new insights or better complexity results. A concrete example will be to enable symbolic methods for Epistemic Crypto Logic [16]. Our methods could then also be used to analyze cryptographic protocols.

Appendix: Helper Functions

```haskell
module SMCDEL.Internal.Help (alleq,anydiff,alldiff,apply,choose,powerset,restrict,rtc,Erel,
    bl,fusion,seteq) where
import Data.List (nub,union,sort,foldl',(\\))

type Rel a b = [(a,b)]
type Erel a = [[a]]

alleq :: Eq a => Eq b => (a -> b) -> [a] -> Bool
alleq _ [] = True
alleq f (x:xs) = all (f x ==) (map f xs)

anydiff :: Eq a => (a -> Bool) -> [a] -> Bool
anydiff _ [] = False
anydiff f (x:xs) = any (f x /=) (map f xs)

alldiff :: Eq a => [a] -> Bool
alldiff [] = True
alldiff (x:xs) = notElem x xs && alldiff xs

apply :: Show a => Show b => Eq a => Rel a b -> a -> b
apply rel left = case lookup left rel of
  Nothing -> error ("apply: Relation " ++ show rel ++ " not defined at " ++ show left)
  (Just this) -> this

powerset :: [a] -> [[a]]
powerset []     = [[]]
powerset (x:xs) = map (x:) pxs ++ pxs where pxs = powerset xs

concatRel :: Eq a => Rel a a -> Rel a a -> Rel a a
concatRel r s = nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)

dom :: Eq a => Rel a a -> [a]
dom r = nub (foldr (\ (x,y) -> ([x,y]++)) [] r)

restrict :: Ord a => [a] -> Erel a -> Erel a
restrict domain =  nub . filter (/= []) . map (filter (`elem` domain))

rtc :: Eq a => Rel a a -> Rel a a
rtc r = lfp (\ s -> s `union` concatRel r s) [(x,x) | x <- dom r ]

merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = case compare x y of
  EQ -> x : merge xs ys
  LT -> x : merge xs (y:ys)
  GT -> y : merge (x:xs) ys

mergeL :: Ord a => [[a]] -> [a]
mergeL = foldl' merge []

overlap :: Ord a => [a] -> [a] -> Bool
overlap [] _  = False
overlap _  [] = False
overlap (x:xs) (y:ys) = case compare x y of
  EQ -> True
  LT -> overlap xs (y:ys)
  GT -> overlap (x:xs) ys

bl :: Eq a => Erel a -> a -> [a]
bl r x = head (filter (elem x) r)

fusion :: Ord a => [[a]] -> Erel a
fusion [] = []
fusion (b:bs) = let
    cs = filter (overlap b) bs
```

```
    xs = mergeL (b:cs)
    ds = filter (overlap xs) bs
  in if cs == ds then xs : fusion (bs \\ cs) else fusion (xs : bs)

choose :: Integral a => a -> a -> a
choose _ 0 = 1
choose 0 _ = 0
choose n k = choose (n-1) (k-1) * n `div` k

seteq :: Ord a => Eq a => [a] -> [a] -> Bool
seteq as bs = sort as == sort bs
```

Appendix: Muddy Children on the Number Triangle

This module implements [22]. The main idea is to not distinguish children who are in the same state which also means that their observations are the same. The number triangle can then be used to solve the Muddy Children puzzle on a Kripke frame with less worlds than needed in the classical analysis, namely $2n + 1$ instead of 2^n for n children.

```
module SMCDEL.Other.MCTRIANGLE where
```

We start with two type definitions: States are pairs of integers indicating how many children are (clean,muddy). A muddy children model consists of three things: A list of observational states, a list of factual states and a current state.

```
type State = (Int,Int)
data McModel = McM [State] [State] State deriving Show
```

Next are functions to create a muddy children model, to get the available successors of a state in a model, to get the observational state of an agent and to get all states deemed possible by an agent.

```
mcModel :: State -> McModel
mcModel cur@(c,m) = McM ostates fstates cur where
  total   = c + m
  ostates = [ ((total-1)-m',m') | m'<-[0..(total-1)] ] -- observational states
  fstates = [ (total-m',    m') | m'<-[0..total    ] ] -- factual states

posFrom :: McModel -> State -> [State]
posFrom (McM _ fstates _) (oc,om) = filter (`elem` fstates) [ (oc+1,om), (oc,om+1) ]

obsFor :: McModel -> Bool -> State
obsFor (McM _ _ (curc,curm)) False = (curc-1,curm)
obsFor (McM _ _ (curc,curm)) True = (curc,curm-1)

posFor :: McModel -> Bool -> [State]
posFor m muddy = posFrom m $ obsFor m muddy
```

Note that instead of naming or enumerating agents we only distinguish two kinds, the muddy and non-muddy ones, represented by Haskells constants `True` and `False` which allow pattern matching.
The following is a type for quantifiers on the number triangle, instantiated by `some`.

```
type Quantifier = State -> Bool

some :: Quantifier
some (_,b) = b > 0
```

The paper does not give a formal language definition, so here is our suggestion:

$$\varphi ::= \neg\varphi \mid \bigwedge \Phi \mid Q \mid K_b \mid \overline{K}_b$$

where Φ ranges over finite sets of formulas, b over $\{0, 1\}$ and Q over generalized quantifiers.

```
data McFormula = Neg McFormula    -- negations
               | Conj [McFormula] -- conjunctions
               | Qf Quantifier    -- quantifiers
               | KnowSelf Bool    -- all b agents DO    know their status
               | NotKnowSelf Bool -- all b agents DON'T know their status
```

Note that when there are no agents of kind `b`, the formulas `KnowSelf b` and `NotKnowSelf b` are both true. Hence `Neg (KnowSelf b)` and `NotKnowSelf b` are not the same!
Below are the formulas for "Nobody knows their own state." and "Everybody knows their own state." Note that in contrast to the standard DEL language these formulas are independent of how many children there are. This is due to our identification of agents with the same state and observations.

```
nobodyknows,everyoneKnows :: McFormula
nobodyknows   = Conj [ NotKnowSelf False, NotKnowSelf True ]
everyoneKnows = Conj [    KnowSelf False,    KnowSelf True ]
```

The semantics for our minimal language are implemented as follows.

```
eval :: McModel -> McFormula -> Bool
eval m (Neg f)          = not $ eval m f
eval m (Conj fs)        = all (eval m) fs
eval (McM _ _ s) (Qf q) = q s
eval m@(McM _ _ (_,curm)) (KnowSelf    True ) = curm==0 || length (posFor m True ) == 1
eval m@(McM _ _ (curc,_)) (KnowSelf    False) = curc==0 || length (posFor m False) == 1
eval m@(McM _ _ (_,curm)) (NotKnowSelf True ) = curm==0 || length (posFor m True ) == 2
eval m@(McM _ _ (curc,_)) (NotKnowSelf False) = curc==0 || length (posFor m False) == 2
```

The four nullary knowledge operators can be thought of as “All agents who are (not) muddy do (not) know their own state.” Hence they are vacuously true whenever there are no such agents. If there are, the agents do know their state iff they consider only one possibility (i.e. their observational state has only one successor).

Finally, we need a function to update models with a formula:

```
update :: McModel -> McFormula -> McModel
update (McM ostates fstates cur) f =
  McM ostates' fstates' cur where
    fstates' = filter (\s -> eval (McM ostates fstates s) f) fstates
    ostates' = filter (not . null . posFrom (McM [] fstates' cur)) ostates
```

The following function shows the update steps of the puzzle, given an actual state:

```
step :: State -> Int -> McModel
step s 0 = update (mcModel s) (Qf some)
step s n = update (step s (n-1)) nobodyknows

showme :: State -> IO ()
showme s@(_,m) = mapM_ (\n -> putStrLn $ show n ++ ": " ++ show (step s n)) [0..(m-1)]
```

```
*MCTRIANGLE> showme (1,2)
m0: McM [(2,0),(1,1),(0,2)] [(2,1),(1,2),(0,3)] (1,2)
m1: McM [(1,1),(0,2)] [(1,2),(0,3)] (1,2)
```

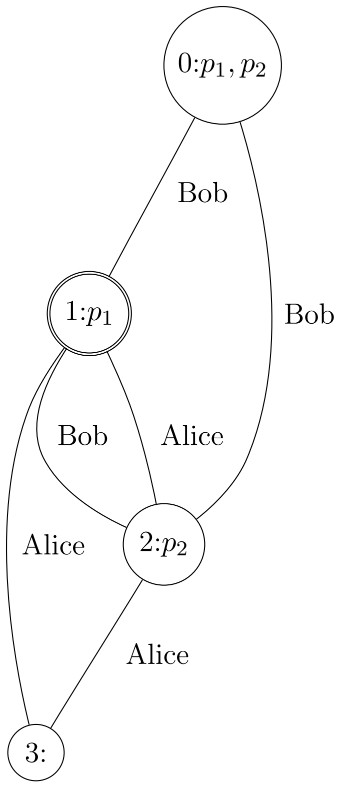
Appendix: Non-S5 and arbitrary Kripke Models

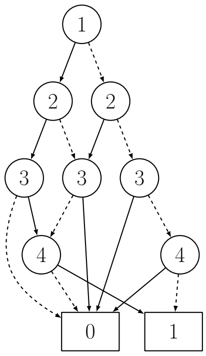
This experimental module shows how we can replace observational variables by BDDs to represent arbitrary relations.

```
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances, MultiParamTypeClasses,
    FlexibleContexts #-}

module SMCDEL.Other.NonS5 where
import Control.Arrow ((&&&))
import Data.HasCacBDD hiding (Top,Bot)
import Data.List (nub,intercalate,sort,(\\),delete)
import Data.Map.Strict (Map,fromList,elems,(!),mapMaybeWithKey,mapKeys,union)
import qualified Data.Map.Strict
import Data.Maybe (fromJust)
import SMCDEL.Language
import SMCDEL.Symbolic.HasCacBDD (Scenario,KnState,texBDD,boolBddOf)
import SMCDEL.Explicit.Simple (PointedModel,KripkeModel(KrM),State)
import SMCDEL.Translations hiding (voc)
import SMCDEL.Internal.Help (alleq)
import SMCDEL.Internal.TexDisplay
```

10.1 The limits of observational variables

In [2] we encoded Kripke frames using observational variables. This restricts our framework to S5 relations. In fact not even every S5 relation on distinctly valuated worlds can be modeled with observational variables as the following example shows. Here the knowledge of Alice is given by an equivalence relation but it can not be described by saying which subset of the vocabulary $V = \{p_1, p_2\}$ she observes. We would want to say that she observes $p \land q$ and our existing approach does this by adding an additional variable:



$\Big(\{p_1, p_2, p_3, p_4\},$  $, \begin{matrix}\{p_3\}\\ \{p_4\}\end{matrix} \Big), \{p_1, p_4\}$

```
problemPM :: PointedModel
problemPM = ( KrM [0,1,2,3] [ (alice,[[0],[1,2,3]]), (bob,[[0,1,2],[3]]) ]
  [ (0,[(P 1,True ),(P 2,True )]), (1,[(P 1,True ),(P 2,False)])
  , (2,[(P 1,False),(P 2,True )]), (3,[(P 1,False),(P 2,False)]) ], 1::State )

problemKNS :: Scenario
problemKNS = kripkeToKns problemPM
```

The following is an attempt to overcome this limitation. We will replace observational variables with BDDs for every agent that describe their relation between worlds as relations between sets of true propositions.

10.2 Translating relations to BDDs

To represent relations as BDDs we use the following well-known method from model checking. Remember that in a knowledge structure we can identify states with boolean assignments. Furthermore, if we fix a global set of variables, those are just sets of propositions. Hence `Rel KnState = [(KnState,KnState)] = [([Prp],[Prp])]`, i.e. a relation on KNS states is in fact a relation on sets of propositions. We can therefore represent it with the OBDD of a characteristic function on a double vocabulary, as described in [7, Section 5.2]. Intuitively we construct (the BDD of) a formula which is true exactly for the pairs of boolean assignments that are connected by the relation.

To do so, we consider a doubled vocabulary. For example, $(\{p, p_3\}, \{p_2\}) \in R$ should be represented by the fact the assignment $\{p, p_3, p_2'\}$ satisfies the formula representing R.

While in normal notation we can just write p' instead of p and p_2' instead of p_2 and so on, in the implementation some more work is needed. In particular we have to choose an ordering of all variables in the double vocabulary. The two candidates are interleaving order or stacking all primed variables above/below all unprimed ones.

We choose the interleaving order because it has two advantages: (i) Relations in epistemic models are often already decided by a difference in one specific propositional variable. Hence p and p' should be close to each other to keep the BDD small. (ii) Using infinite lists we can write general functions to go back and forth between the vocabularies, independent of how many variables we will actually use.

Variable	Single vocabulary	Double vocabulary
p	`P 0`	`P 0`
p'		`P 1`
p_1	`P 1`	`P 2`
p_1'		`P 3`
p_2	`P 2`	`P 4`
p_2'		`P 5`
$\vdots$	$\vdots$	$\vdots$

Table 1: Implementation of single and double vocabulary.

```
type RelBDD = Bdd
```

To switch between the normal and the double vocabulary, we use three helper functions:

```
mv :: [Prp] -> [Prp]
mv = map (fromJust . (`lookup` [ (P n, P  (2*n)      ) | n <- [0..] ])) -- represent p in
    the double vocabulary
cp :: [Prp] -> [Prp]
cp = map (fromJust . (`lookup` [ (P n, P ((2*n) + 1) ) | n <- [0..] ])) -- represent p' in
    the double vocabulary
unprime :: [Prp] -> [Prp]
-- Go from p' in double vocabulary to p in single vocabulary:
unprime = map f where
  f (P m) | even m = error "unprime failed: Number is even!"
          | otherwise = P $ fromJust (lookup m [ ((2*n) + 1, n) | n <- [0..] ])
```

Suppose we have a BDD representing a formula in the single vocabulary. The following function relabels the BDD to represent the formula with primed propositions in the double vocabulary.

```
cpBdd :: Bdd -> Bdd
cpBdd b | b == bot  = b
        | b == top  = b
        | otherwise = relabel [ (n, (2*n) + 1) | n <- [0..m] ] b where (Just m) = maxVarOf
            b
```

And with the unprimed ones in the double:

```
mvBdd :: Bdd -> Bdd
mvBdd b | b == bot  = b
        | b == top  = b
        | otherwise = relabel [ (n, 2*n) | n <- [0..(fromJust $ maxVarOf b)] ] b
```

And this function takes a BDD which uses unprimed propositions in the double vocabulary and returns a Bdd representing the same formula in the single vocabulary.

```
unmvBdd :: Bdd -> Bdd
unmvBdd b | b == bot  = b
          | b == top  = b
          | otherwise = relabel [ (2 * n, n) | n <- [0..(fromJust $ maxVarOf b)] ] b
```

The double vocabulary is therefore obtained as follows:

```
>>> SMCDEL.Other.NonS5.mv [(P 0)..(P 3)]
```

```
[P 0,P 2,P 4,P 6]
```

1.97 seconds

```
>>> SMCDEL.Other.NonS5.cp [(P 0)..(P 3)]
```

```
[P 1,P 3,P 5,P 7]
```

2.14 seconds

Let $(\varphi)'$ denote the formula obtained by priming all propositions in φ.
We model a relation R between sets of propositions using the following BDD:

$$\mathsf{Bdd}(R) := \bigvee_{(s,t)\in R} \left((s \sqsubseteq \mathsf{V}) \land (t \sqsubseteq \mathsf{V})' \right)$$

```
propRel2bdd :: [Prp] -> Map KnState [KnState] -> RelBDD
propRel2bdd voc rel = disSet (elems $ Data.Map.Strict.mapWithKey linkbdd rel) where
  linkbdd here theres =
    con (booloutof (mv here) (mv voc))
        (disSet [ booloutof (cp there) (cp voc) | there<-theres ] )
```

The following example is from [23, p. 136].

```
samplerel ::  Map KnState [KnState]
samplerel = fromList [
  ( []         , [ [],[P 1],[P 2],[P 1, P 2] ] ),
  ( [P 1]      , [    [P 1],      [P 1, P 2] ] ),
  ( [P 2]      , [    [P 2],      [P 1, P 2] ] ),
  ( [P 1, P 2], [                 [P 1, P 2] ] )  ]
```

```
>>> SMCDEL.Other.NonS5.propRel2bdd [P 1, P 2] SMCDEL.Other.NonS5.samplerel

Var 2 (Var 3 (Var 4 (Var 5 Top Bot) Top) Bot) (Var 4 (Var 5 Top Bot) Top)

2.38 seconds
```

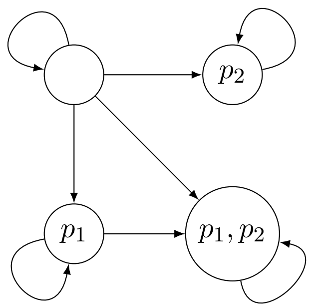


Figure 2: The original graph of `samplerel`.

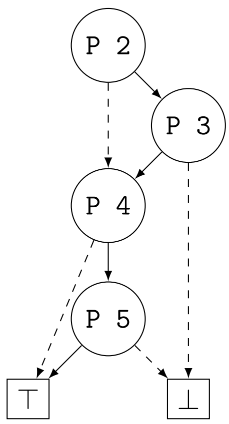


Figure 3: BDD of `samplerel` with double vocabulary labels.

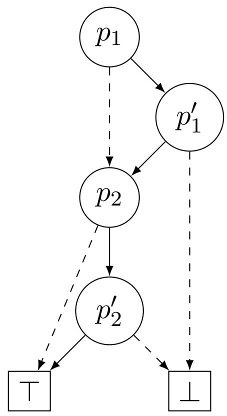


Figure 4: BDD of `samplerel` with translated labels.

Many operations and tests on relations can be done directly on their BDDs:

- The total relation is given by the constant $\top$ and the empty relation by $\bot$.
- Get the inverse: Simultaneously substitute primed for unprimed variables and vice versa.
- Test for symmetry: Is it equal to its inverse?
- Symmetric closure: Take the disjunction with the inverse.
- Test for reflexivity: Does $\bigwedge_i (p_i \leftrightarrow p'_i)$ imply the BDD of R? I.e. is the BDD of that implementation equal to $\top$?
- Reflexive closure: Take the disjunction of BDD(R) with $\bigwedge_i (p_i \leftrightarrow p'_i)$.

10.3 General non-S5 Kripke Models

In non-S5 Kripke models every agent has an arbitrary relation on the states, not necessarily and equivalence relation. Moreover, we generalize our Kripke models and let worlds be anything, not just integers.

Hence, a general Kripke model is a map from worlds to pairs of (i) assignment, i.e. maps from propositions to $\top$ or $\bot$, and (ii) reachability, i.e. maps from agents to sets of worlds.

```
type GeneralKripkeModel a = Map a (Map Prp Bool, Map Agent [a])

type GeneralPointedModel a = (GeneralKripkeModel a, a)

exampleModel :: GeneralKripkeModel Int
exampleModel = fromList
  [ (1, (fromList [(P 0,True ),(P 1,True )], fromList [(alice,[1]), (bob,[1])] ) )
  , (2, (fromList [(P 0,False),(P 1,True )], fromList [(alice,[1]), (bob,[2])] ) ) ]

examplePointedModel :: GeneralPointedModel Int
examplePointedModel = (exampleModel,1)

distinctVal :: GeneralKripkeModel a -> Bool
distinctVal m = Data.Map.Strict.size m == length (nub (map fst (elems m)))

vocOf :: GeneralKripkeModel a -> [Prp]
vocOf m = Data.Map.Strict.keys $ fst (head (Data.Map.Strict.elems m))

agentsOf :: GeneralKripkeModel a -> [Agent]
agentsOf m = Data.Map.Strict.keys $ snd (head (Data.Map.Strict.elems m))

relOfIn :: Agent -> GeneralKripkeModel a -> Map a [a]
relOfIn i = Data.Map.Strict.map (\x -> snd x ! i)

truthsInAt :: Ord a => GeneralKripkeModel a -> a -> [Prp]
truthsInAt m w = Data.Map.Strict.keys (Data.Map.Strict.filter id (fst (m ! w)))
```

As a reference, we also implement general Kripke semantics.

```
eval :: Ord a => GeneralPointedModel a -> Form -> Bool
eval _     Top            = True
eval _     Bot            = False
eval (m,w) (PrpF p)       = p `elem` truthsInAt m w
eval pm    (Neg f)        = not $ eval pm f
eval pm    (Conj fs)      = all (eval pm) fs
eval pm    (Disj fs)      = any (eval pm) fs
eval pm    (Xor  fs)      = odd $ length (filter id $ map (eval pm) fs)
eval pm    (Impl f g)     = not (eval pm f) || eval pm g
eval pm    (Equi f g)     = eval pm f == eval pm g
eval pm    (Forall ps f) = eval pm (foldl singleForall f ps) where
  singleForall g p = Conj [ substit p Top g, substit p Bot g ]
eval pm    (Exists ps f) = eval pm (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
eval (m,w) (K   i f)      = all (\w' -> eval (m,w') f) (snd (m ! w) ! i)
eval _     (Ck  _ _)      = error "eval: Ck not implemented "
eval (m,w) (Kw  i f)      = alleq (\w' -> eval (m,w') f) (snd (m ! w) ! i)
eval _     (Ckw _ _)      = error "eval: Ck not implemented "
eval (m,w) (PubAnnounce f g) = not (eval (m,w) f) || eval (pubAnnounceNonS5 m f,w) g
eval (m,w) (PubAnnounceW f g) = eval (pubAnnounceNonS5 m aform, w) g where
  aform | eval (m,w) f = f
        | otherwise      = Neg f
eval (m,w) (Announce is f g) =
  not (eval (m,w) f) || eval (announceNonS5 m is f, (w,True)) g
eval (m,w) (AnnounceW is f g) = eval (announceNonS5 m is aform, (w,True)) g where
  aform | eval (m,w) f = f
        | otherwise     = Neg f

pubAnnounceNonS5 :: Ord a => GeneralKripkeModel a -> Form -> GeneralKripkeModel a
pubAnnounceNonS5 m f = newm where
  newm = mapMaybeWithKey isin m
  isin w' (v,rs) | eval (m,w') f = Just (v,Data.Map.Strict.map newr rs)
                 | otherwise     = Nothing
```

```
  newr = filter (`elem` Data.Map.Strict.keys newm)

announceNonS5 :: Ord a => GeneralKripkeModel a -> [Agent] -> Form -> GeneralKripkeModel (a,
    Bool)
announceNonS5 m is f = oldm `union` addm where
  oldm         = mapKeys (\k -> (k,False)) $ Data.Map.Strict.map fixr m
  fixr (v,rs) = (v, Data.Map.Strict.map (map (\k -> (k,False))) rs)
  addm         = mapKeys (\k -> (k,True)) $ mapMaybeWithKey copy m
  copy w' (v,rs) | eval (m,w') f = Just (v,Data.Map.Strict.mapWithKey copyr rs)
                 | otherwise     = Nothing
  copyr i rs | i `elem` is = map (\k -> (k,True)) rs
             | otherwise   = map (\k -> (k,False)) rs
```

10.4 Describing non-S5 Kripke Models with BDDs

We now want to use BDDs to represent the relations of multiple agents in a general Kripke Model. Suppose we have a model for the vocabulary V in which the valuation function assigns to every state a distinct set of true propositions. To simplify the notation we also write s for the set of propositions true at s. Thereby we translate a relation of states to a relation of sets of propositions:

```
relBddOfIn :: Ord a =>  Agent -> GeneralKripkeModel a -> RelBDD
relBddOfIn i m
  | not (distinctVal m) =  error "m does not have distinct valuations."
  | otherwise = disSet (elems $ Data.Map.Strict.map linkbdd m) where
    linkbdd (mapPropBool,mapAgentReach)  =
      con
        (booloutof (mv here) (mv voc))
        (disSet [ booloutof (cp there) (cp voc) | there<-theres ] )
      where
        voc = Data.Map.Strict.keys mapPropBool
        here = Data.Map.Strict.keys (Data.Map.Strict.filter id mapPropBool)
        theres = map (truthsInAt m) (mapAgentReach ! i)
```

```
>>> map (flip SMCDEL.Other.NonS5.relBddOfIn SMCDEL.Other.NonS5.exampleModel)
[alice,bob]

[Var 1 (Var 2 (Var 3 Top Bot) Bot) Bot,Var 0 (Var 1 (Var 2 (Var 3 Top Bot) Bot) Bot) (
    Var 1 Bot (Var 2 (Var 3 Top Bot) Bot))]

2.13 seconds
```

It seems good to use an interleaving variable order, i.e. $p_1, p_1', p_2, p_2', \ldots, p_n, p_n'$. This way a BDD will consider differences in the valuation per proposition and be more compact if we have (almost) observational-variable-like situations.

10.5 General Knowledge Structures

```
data GenKnowStruct = GenKnS [Prp] Bdd (Map Agent RelBDD) deriving (Eq,Show)

type GenScenario = (GenKnowStruct,[Prp])
```

Rewriting all formulas to BDDs that are equivalent on a given general knowledge structure.

```
bddOf :: GenKnowStruct -> Form -> Bdd
bddOf _   Top            = top
bddOf _   Bot            = bot
bddOf _   (PrpF (P n))   = var n
bddOf kns (Neg form)     = neg $ bddOf kns form
bddOf kns (Conj forms)   = conSet $ map (bddOf kns) forms
bddOf kns (Disj forms)   = disSet $ map (bddOf kns) forms
bddOf kns (Xor  forms)   = xorSet $ map (bddOf kns) forms
bddOf kns (Impl f g)     = imp (bddOf kns f) (bddOf kns g)
bddOf kns (Equi f g)     = equ (bddOf kns f) (bddOf kns g)
```

```
bddOf kns (Forall ps f) = forallSet (map fromEnum ps) (bddOf kns f)
bddOf kns (Exists ps f) = existsSet (map fromEnum ps) (bddOf kns f)
```

Note the following notations for boolean assignments and formulas.

- Suppose s is a boolean assignment and φ is a boolean formula in the vocabulary of s. Then we write $s \vDash \varphi$ to say that s makes φ true.
- If s is an assignment for a given vocabulary, we write s' for the same assignment for a primed copy of the vocabulary. For example take $\{p_1, p_3\}$ as an assignment over $V = \{p_1, p_2, p_3, p_4\}$, hence $\{p_1, p_3\}' = \{p_1', p_3'\}$ is an assignment over $\{p_1', p_2', p_3', p_4'\}$.
- If φ is a boolean formula, write $(\varphi)'$ for the result of priming all propositions in φ. For example, $(p_1 \to (p_3 \land \neg p_2))' = (p_1' \to (p_3' \land \neg p_2'))$.
- If s and t are boolean assignments for distinct vocabularies and φ is a vocabulary in the combined vocabulary, we write $(st) \vDash \varphi$ to say that $s \cup t$ makes φ true.

We can now show how to find boolean equivalents of K-formulas:

$$
\begin{array}{rcl}
\mathcal{F}, s \vDash K_i\varphi & \iff & \text{For all } t \in \mathcal{F} : \text{If } sR_it \text{ then } \mathcal{F}, t \vDash \varphi \\
& \iff & \text{For all } t : \text{If } t \in \mathcal{F} \text{ and } sR_it \text{ then } \mathcal{F}, t \vDash \varphi \\
& \iff & \text{For all } t : \text{If } t \vDash \theta \text{ and } (st') \vDash \Omega_i(\vec{p}, \vec{p'}) \text{ then } t \vDash |\varphi|_{\mathcal{F}} \\
& \iff & \text{For all } t : \text{If } t' \vDash \theta' \text{ and } (st') \vDash \Omega_i(\vec{p}, \vec{p'}) \text{ then } t' \vDash (|\varphi|_{\mathcal{F}})' \\
& \iff & \text{For all } t : \text{If } (st') \vDash \theta' \text{ and } (st') \vDash \Omega_i(\vec{p}, \vec{p'}) \text{ then } (st') \vDash (|\varphi|_{\mathcal{F}})' \\
& \iff & \text{For all } t : (st') \vDash \theta' \to (\Omega_i(\vec{p}, \vec{p'}) \to (|\varphi|_{\mathcal{F}})') \\
& \iff & s \vDash \forall \vec{p'}(\theta' \to (\Omega_i(\vec{p}, \vec{p'}) \to (|\varphi|_{\mathcal{F}})'))
\end{array}
$$

This is exactly what the following lines do, together with the variable management described above.

```
bddOf kns@(GenKnS allprops lawbdd obdds) (K i form) = unmvBdd result where
  result = forallSet ps' (cpBdd lawbdd `imp` (omegai `imp` cpBdd (bddOf kns form)))
  ps'    = map fromEnum $ cp allprops
  omegai = obdds ! i
```

Knowing whether is just the disjunction of knowing that and knowing that not.

```
bddOf kns@(GenKnS allprops lawbdd obdds) (Kw i form) = unmvBdd result where
  result = disSet (map part [form, Neg form])
  part f = forallSet ps' (cpBdd lawbdd `imp` (omegai `imp` cpBdd (bddOf kns f)))
  ps'    = map fromEnum $ cp allprops
  omegai = obdds ! i
```

We do not interpret common knowledge on non-S5 structures:

```
bddOf _ (Ck _ _)  = error "bddOf: Ck not implemented"
bddOf _ (Ckw _ _) = error "bddOf: Ckw not implemented"
```

Public announcements only restrict the lawbdd:

```
bddOf kns (PubAnnounce f g) =
  imp (bddOf kns f) (bddOf (pubAnnounce kns f) g)
bddOf kns (PubAnnounceW f g) =
  ifthenelse (bddOf kns f)
    (bddOf (pubAnnounce kns f      ) g)
    (bddOf (pubAnnounce kns (Neg f)) g)
```

Announcements to a group now are really secret, see `announce` below.

```
bddOf kns@(GenKnS props _ _) (Announce ags f g) =
  imp (bddOf kns f) (restrict bdd2 (k,True)) where
    bdd2  = bddOf (announce kns ags f) g
    (P k) = freshp props

bddOf kns@(GenKnS props _ _) (AnnounceW ags f g) =
  ifthenelse (bddOf kns f) bdd2a bdd2b where
    bdd2a = restrict (bddOf (announce kns ags f) g) (k,True)
    bdd2b = restrict (bddOf (announce kns ags f) g) (k,False)
    (P k) = freshp props
```

Validity and Truth: A formula φ is valid on a knowledge structures iff it is true at all states. This is equivalent to the condition that the boolean equivalent formula $|\varphi|_{\mathcal{F}}$ is true at all states of $\mathcal{F}$. Furthermore, this is equivalent to saying that the law θ of $\mathcal{F}$ implies $|\varphi|_{\mathcal{F}}$. Hence, checking for validity can be done by checking if the BDD of $\theta \rightarrow |\varphi|_{\mathcal{F}}$ is equivalent=identical to the $\top$ BDD.

```
validViaBdd :: GenKnowStruct -> Form -> Bool
validViaBdd kns@(GenKnS _ lawbdd _) f = top == imp lawbdd (bddOf kns f)
```

Similarly, to check if a formula φ is true at a given state s of a knowledge structure $\mathcal{F}$, we take its boolean equivalent $|\varphi|_{\mathcal{F}}$ and check if the assignment s satisfies this BDD. We fail with an error message in case the BDD is not decided by the given assignment.

```
evalViaBdd :: GenScenario -> Form -> Bool
evalViaBdd (kns@(GenKnS allprops _ _),s) f = let
    b    = restrictSet (bddOf kns f) list
    list = [ (n,True) | (P n) <- s ] ++ [ (n,False) | (P n) <- allprops \\ s ]
  in
    case (b==top,b==bot) of
      (True,_) -> True
      (_,True) -> False
      _         -> error "evalViaBdd failed: Composite BDD leftover."
```

Above we already used the following functions for public and group announcements, adapted to belief structures.

```
pubAnnounce :: GenKnowStruct -> Form -> GenKnowStruct
pubAnnounce kns@(GenKnS allprops lawbdd obs) f =
  GenKnS allprops (con lawbdd (bddOf kns f)) obs

pubAnnounceOnScn :: GenScenario -> Form -> GenScenario
pubAnnounceOnScn (kns,s) psi = if evalViaBdd (kns,s) psi
                                 then (SMCDEL.Other.NonS5.pubAnnounce kns psi,s)
                                 else error "Liar!"

announce :: GenKnowStruct -> [Agent] -> Form -> GenKnowStruct
announce kns@(GenKnS props lawbdd obdds) ags psi = GenKnS newprops newlawbdd newobdds where
  proppsi@(P k) = freshp props
  [P k'] = cp [proppsi]
  newprops  = proppsi:props
  newlawbdd = con lawbdd (imp (var k) (bddOf kns psi))
  newobdds  = Data.Map.Strict.mapWithKey newOfor obdds
  newOfor i oi | i `elem` ags = con oi (equ (var k) (var k'))
               | otherwise    = con oi (neg (var k')) -- p_psi'
```

```
statesOf :: GenKnowStruct -> [KnState]
statesOf (GenKnS allprops lawbdd _) = map (sort.translate) resultlists where
  resultlists = map (map convToProp) $ allSatsWith (map (\(P n) -> n) allprops) lawbdd ::
      [[(Prp, Bool)]] -- FIXME ugly
  convToProp (n,bool) = (P n,bool)
  translate l = map fst (filter snd l)
```

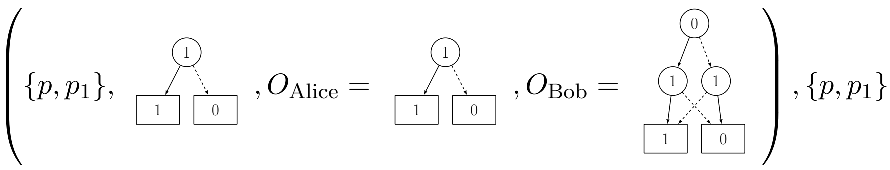
Visualizing general Knowledge Structures:

```
instance TexAble GenScenario where
  tex (GenKnS props lawbdd obdds, state) = concat
    [ " \\left( \n"
    , tex props, ", "
    , bddprefix, texBDD lawbdd, bddsuffix
    , ", "
    , intercalate ", " obddstrings
    , " \\right) , "
    , tex state
    ] where
        (bddprefix,bddsuffix) = ("\\begin{array}{l} \\scalebox{0.3}{", "} \\end{array} \n")
        obddstrings =
          map ( (\(i,os) -> "O_{\\text{" ++ i ++ "}} = " ++ bddprefix ++ os ++ bddsuffix).(
              fst &&& (texBDD . snd)) ) (Data.Map.Strict.toList obdds)
```

10.6 Converting General Kripke Models to General Knowledge Structures

Assuming we already have distinct valuations!

```
genKrp2Kns :: Ord a => GeneralPointedModel a -> GenScenario
genKrp2Kns (m, cur) = (GenKnS voc lawbdd obdds, truthsInAt m cur) where
  voc    = vocOf m
  lawbdd = disSet [ booloutof (truthsInAt m w) voc | w <- Data.Map.Strict.keys m ]
  obdds  = fromList [ (i, restrictLaw (relBddOfIn i m) (con (mvBdd lawbdd) (cpBdd lawbdd)))
       | i <- agents ]
  agents = agentsOf m

exampleGenScn :: GenScenario
exampleGenScn = genKrp2Kns examplePointedModel

exampleGenStruct :: GenKnowStruct
exampleGenState :: KnState
(exampleGenStruct,exampleGenState) = exampleGenScn
```



(If voc is just P 0) We can see that Alice's relation only depends on the valuation at the destination point: In her BDD only the variable p' is checked.

Additionally, both agent BDDs do not care about p_1 or p'_1. This is because of our use of `restrictLaw`. This ensures our relation bdds do not become unnecessarily large. The BDDs generated by relBddOfIn include a check that both parts of the related pair are actually state of our model/structure but we do not need this information in the agents bdd.

Muddy Children with observational BDDs:

```
allsamebdd :: [Prp] -> RelBDD
allsamebdd ps = conSet [boolBddOf $ PrpF p `Equi` PrpF p' | (p,p') <- zip (mv ps) (cp ps)]

mudGenScnInit :: Int -> Int -> GenScenario
mudGenScnInit n m = (GenKnS vocab law obs, actual) where
  vocab  = [P 1 .. P n]
  law    = boolBddOf Top
  obs    = fromList [(show i, allsamebdd $ delete (P i) vocab) | i <- [1..n]]
  actual = [P 1 .. P m]

myMudGenScnInit :: GenScenario
myMudGenScnInit = mudGenScnInit 3 3
```

References

[1] Baltag, A., Moss, L.S., Solecki, S.: The logic of public announcements, common knowledge, and private suspicions. In: Bilboa, I. (ed.) TARK'98. pp. 43–56 (1998)

[2] van Benthem, J., van Eijck, J., Gattinger, M., Su, K.: Symbolic model checking for dynamic epistemic logic. In: Proceedings of The Fifth International Conference on Logic, Rationality and Interaction (LORI-V) (2015)

[3] van Benthem, J., Gerbrandy, J., Hoshi, T., Pacuit, E.: Merging frameworks for interaction. Journal of Philosophical Logic 38(5), 491–526 (2009)

[4] Charrier, T., Schwarzentruber, F.: Arbitrary public announcement logic with mental programs. In: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems. pp. 1471–1479. IFAAMAS (2015)

[5] Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1(1), 65–75 (1988)

[6] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Computer Aided Verification. pp. 359–364. Springer (2002)

[7] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts, USA (1999)

[8] Cordón-Franco, A., van Ditmarsch, H., Fernández-Duque, D., Soler-Toscano, F.: A geometric protocol for cryptography with cards. Designs, Codes and Cryptography 74(1), 113–125 (2015), `http://dx.doi.org/10.1007/s10623-013-9855-y`

[9] van Ditmarsch, H.: The russian cards problem. Studia Logica 75(1), 31–62 (2003)

[10] van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic epistemic logic, vol. 1. Springer Heidelberg (2007)

[11] van Ditmarsch, H., van der Hoek, W., van der Meyden, R., Ruan, J.: Model Checking Russian Cards. Electr. Notes Theor. Comput. Sci. 149(2), 105–123 (2006)

[12] van Ditmarsch, H., van der Hoek, W., Ruan, J.: Connecting dynamic epistemic and temporal epistemic logics. Logic Journal of IGPL 21(3), 380–403 (2013)

[13] Duque, D.F., Goranko, V.: Secure aggregation of distributed information. CoRR abs/1407.7582 (2014), `http://arxiv.org/abs/1407.7582`

[14] van Eijck, J.: DEMO—a demo of epistemic modelling. In: Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London. vol. 1, pp. 303–362 (2007)

[15] van Eijck, J.: DEMO-S5. Tech. rep., CWI (2014)

[16] van Eijck, J., Gattinger, M.: Elements of epistemic crypto logic (extended abstract). In: Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015). IFAAMAS (2015)

[17] Engesser, T., Bolander, T., Nebel, B.: Cooperative epistemic multi-agent planning with implicit coordination. Distributed and Multi-Agent Planning (DMAP-15) p. 68 (2015)

[18] Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge, vol. 4. MIT press Cambridge (1995)

[19] Freudenthal, H.: Formulering van het 'som-en-product'-probleem. Nieuw Archief voor Wiskunde 17, 152 (1969)

[20] Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Computer Aided Verification. pp. 479–483. Springer (2004)

[21] Gattinger, M.: HasCacBDD. `https://github.com/m4lvin/HasCacBDD` (2015)

[22] Gierasimczuk, N., Szymanik, J.: A note on a generalization of the Muddy Children puzzle. In: Apt, K.R. (ed.) TARK'11. pp. 257–264. ACM (2011)

[23] Gorogiannis, N., Ryan, M.D.: Implementation of Belief Change Operators Using BDDs. Studia Logica 70(1), 131–156 (2002)

[24] Littlewood, J.: A Mathematician's Miscellany. Methuen, London (1953)

[25] Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer pp. 1–22 (2015)

[26] Luo, X., Su, K., Sattar, A., Chen, Y.: Solving Sum and Product Riddle via BDD-Based Model Checking. In: Web Intel./IAT Workshops. pp. 630–633. IEEE (2008)

[27] Lv, G., Su, K., Xu, Y.: CacBDD: A BDD Package with Dynamic Cache Management. In: Proceedings of the 25th International Conference on Computer Aided Verification. pp. 229–234. CAV'13, Springer-Verlag, Berlin, Heidelberg (2013)

[28] Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.5.0 (2012)

[29] Van Ditmarsch, H., Ruan, J.: Model checking logic puzzles. Annales du Lamsade 8 (2007), `https://hal.archives-ouvertes.fr/hal-00188953`