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

Malvin Gattinger

Bernoulli Institute, University of Groningen
(previously ILLC, University of Amsterdam)

malvin@w4eg.eu

Last Update: Tuesday 3rd December, 2019

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 [Ben+15; Ben+17]. This report is structured as follows.

We first only consider S5 variants of DEL based on Kripke models with equivalence relations. In Section 1 we recapitulate the syntax and intended meaning of DEL and define a data type for formulas. Section 2 describes and implements the well-known semantics for DEL on Kripke models. This implementation of explicit model checking is later used 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.

Sections 5 and 6 then generalize our implementation to general Kripke models where the accessibility relations need not be equivalence relations, and their symbolic equivalents called belief structures. Again we implement translations in Section 7.

To check that the implementations are correct we provide methods for automated randomized testing in Section 8 using QuickCheck and HSpec.

In Section 9 we provide some helper functions for epistemic planning.

Section 10 shows how to use SMCDEL. 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, Sum and Product etc. Some of the examples also suggest themselves as benchmarks which we do in Section 11 to compare the different versions of our model checker to the existing tools DEMO-S5 and MCMAS.

In Section 12.1 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 12.2 implements a web interface.

The last Section 13 discusses future work, both improvements for the implementation and on theoretical aspects of our framework.

The appendix consists of a module with more helper functions, an implementation of the number triangle analysis of the Muddy Children problem from [GS11], and a copy of DEMO-S5 from [Eij14].

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 GNU General Public License v2.0.

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

Contents

1 The Language of Dynamic Epistemic Logic

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

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

module SMCDEL.Language where

import Data.List (nub,intercalate,(\\))
import Data.Dynamic
import Data.Maybe (fromMaybe)

import Test.QuickCheck
import SMCDEL.Internal.Help (powerset)
import SMCDEL.Internal.TexDisplay
```

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

```
newtype Prp = P Int deriving (Eq,Ord,Show)

instance Enum Prp where
  toEnum = P
  fromEnum (P n) = n

defaultVocabulary :: [Prp]
defaultVocabulary = map P [0..4]

instance Arbitrary Prp where
  arbitrary = elements defaultVocabulary

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

class HasVocab a where
  vocabOf :: a -> [Prp]

type Agent = String

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

defaultAgents :: [Agent]
defaultAgents = map show [(1::Integer)..5]

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

instance Arbitrary AgAgent where
  arbitrary = elements $ map Ag defaultAgents

class HasAgents a where
  agentsOf :: a -> [Agent]

class Pointed a b

instance (HasVocab a, Pointed a b) => HasVocab (a,b) where vocabOf = vocabOf . fst

instance (HasAgents a, Pointed a b) => HasAgents (a,b) where agentsOf = agentsOf . fst

newtype Group = Group [Agent] deriving (Eq,Ord,Show)

-- generate a random group, always including agent 1
instance Arbitrary Group where
  arbitrary = fmap (Group.("1":)) $ sublistOf $ defaultAgents \\ ["1"]
```

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 $p \in V$, $P \subseteq V$, $|P| < \omega$, $\Phi \subseteq \mathcal{L}(V)$, $|\Phi| < \omega$, $i \in I$ and $\Delta \subset I$. We also write $\varphi \wedge \psi$ for $\bigwedge\{\varphi, \psi\}$ and $\varphi \vee \psi$ 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. 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 — 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 -- TODO remove, define as
    abbreviations!
  | Announce [Agent] Form Form     -- ^ (Semi-)Private announcement that
  | AnnounceW [Agent] Form Form    -- ^ (Semi-)Private announcement whether
  | Dia DynamicOp Form             -- ^ Dynamic Diamond
  deriving (Eq,Ord,Show)

box :: DynamicOp -> Form -> Form
box op f = Neg (Dia op (Neg f))

data DynamicOp = Dyn String Dynamic

instance Eq DynamicOp where
  Dyn a _ == Dyn b _ = a == b
instance Ord DynamicOp where
```

```haskell
  compare (Dyn a _) (Dyn b _) = compare a b
instance Show DynamicOp where
  show (Dyn a _) = "Dyn " ++ show a ++ " _"

class HasVocab a => Semantics a where
  isTrue :: a -> Form -> Bool

(|=) :: Semantics a => a -> Form -> Bool
(|=) = isTrue

class Optimizable a where
  optimize :: [Prp] -> a -> a

class HasPrecondition a where
  preOf :: a -> Form

-- | Formulas used as public announcements are their own precondition.
instance HasPrecondition Form where
  preOf = id

class (Show a, Show b, HasAgents a, Semantics a, HasPrecondition b) => Update a b where
  {-# MINIMAL unsafeUpdate #-}
  unsafeUpdate :: a -> b -> a
  checks :: [a -> b -> Bool]
  checks = [preCheck]
  preCheck :: a -> b -> Bool
  preCheck x y = isTrue x (preOf y)
  update :: a -> b -> a
  update x y = if and checkResults
                 then unsafeUpdate x y
                 else error . concat $
                   [ "Update failed."
                   , "\n  x = ", show x
                   , "\n  y = ", show y
                   , "\n  preOf y = ", show (preOf y)
                   , "\n  preCheck y = ", show (preCheck x y)
                   , "\n  checkResults: ", show checkResults ]
               where checkResults = [ c x y | c <- checks ]

updates :: Update a b => a -> [b] -> a
updates = foldl update

haveSameAgents :: (HasAgents a, HasAgents b) => a -> b -> Bool
haveSameAgents x y = agentsOf x == agentsOf y

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

-- | Pretty print a formula, possibly with a translation for atoms:
ppForm :: Form -> String
ppForm = ppFormWith (\(P n) -> show n)

ppFormWith :: (Prp -> String)-> Form -> String
ppFormWith _     Top              = "T"
ppFormWith _     Bot              = "F"
ppFormWith trans (PrpF p)         = trans p
ppFormWith trans (Neg f)          = "~" ++ ppFormWith trans f
ppFormWith trans (Conj fs)        = "(" ++ intercalate " & " (map (ppFormWith trans) fs) ++ ")
    "
ppFormWith trans (Disj fs)        = "(" ++ intercalate " | " (map (ppFormWith trans) fs) ++ ")
    "
ppFormWith trans (Xor fs)         = "XOR{" ++ showSet (map (ppFormWith trans) fs) ++ "}"
ppFormWith trans (Impl f g)       = "(" ++ ppFormWith trans f ++ "->" ++ ppFormWith trans g ++
     ")"
ppFormWith trans (Equi f g)       = ppFormWith trans f ++ "=" ++ ppFormWith trans g
ppFormWith trans (Forall ps f)  = "Forall {" ++ showSet ps ++ "}: " ++ ppFormWith trans f
ppFormWith trans (Exists ps f)  = "Exists {" ++ showSet ps ++ "}: " ++ ppFormWith trans f
ppFormWith trans (K i f)          = "K " ++ i ++ " " ++ ppFormWith trans f
ppFormWith trans (Ck is f)        = "Ck " ++ showSet is ++ " " ++ ppFormWith trans f
ppFormWith trans (Kw i f)         = "Kw " ++ i ++ " " ++ ppFormWith trans f
ppFormWith trans (Ckw is f)       = "Ckw " ++ showSet is ++ " " ++ ppFormWith trans f
ppFormWith trans (PubAnnounce f g)  = "[! " ++ ppFormWith trans f ++ "] " ++ ppFormWith
    trans g
```

```haskell
ppFormWith trans (PubAnnounceW f g) = "[?! " ++ ppFormWith trans f ++ "] " ++ ppFormWith
    trans g
ppFormWith trans (Announce is f g)  = "[" ++ intercalate ", " is ++ " ! " ++ ppFormWith
    trans f ++ "]" ++ ppFormWith trans g
ppFormWith trans (AnnounceW is f g) = "[" ++ intercalate ", " is ++ " ?! " ++ ppFormWith
    trans f ++ "]" ++ ppFormWith trans g
ppFormWith trans (Dia (Dyn s _) f)  = "⟨" ++ s ++ "⟩" ++ ppFormWith trans f
```

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`.

```haskell
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.

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

We define a list of subformulas as follows, including the given formula itself. In particular this is used in the `shrink` function for QuickCheck.

```haskell
subformulas :: Form -> [Form]
subformulas Top            = [Top]
subformulas Bot            = [Bot]
subformulas (PrpF p)       = [PrpF p]
subformulas (Neg f)        = Neg f : subformulas f
subformulas (Conj fs)      = Conj fs : nub (concatMap subformulas fs)
subformulas (Disj fs)      = Disj fs : nub (concatMap subformulas fs)
subformulas (Xor fs)       = Xor  fs : nub (concatMap subformulas fs)
subformulas (Impl f g)     = Impl f g : nub (concatMap subformulas [f,g])
subformulas (Equi f g)     = Equi f g : nub (concatMap subformulas [f,g])
subformulas (Forall ps f) = Forall ps f : subformulas f
subformulas (Exists ps f) = Exists ps f : subformulas f
subformulas (K i f)        = K i f : subformulas f
subformulas (Ck is f)      = Ck is f : subformulas f
subformulas (Kw i f)       = Kw i f : subformulas f
subformulas (Ckw is f)     = Ckw is f : subformulas f
subformulas (PubAnnounce  f g) = PubAnnounce  f g : nub (subformulas f ++ subformulas g)
subformulas (PubAnnounceW f g) = PubAnnounceW f g : nub (subformulas f ++ subformulas g)
subformulas (Announce  is f g) = Announce  is f g : nub (subformulas f ++ subformulas g)
subformulas (AnnounceW is f g) = AnnounceW is f g : nub (subformulas f ++ subformulas g)
subformulas (Dia dynop f)      = Dia dynop f : subformulas f

shrinkform :: Form -> [Form]
shrinkform f =
  if f /= simplify f
    then [simplify f]
    else (subformulas f \\ [f]) ++ otherShrinks f
  where
    otherShrinks (Conj      fs) = [Conj      gs | gs <- powerset fs \\ [fs]]
    otherShrinks (Disj      fs) = [Disj      gs | gs <- powerset fs \\ [fs]]
    otherShrinks (Xor       fs) = [Xor       gs | gs <- powerset fs \\ [fs]]
    otherShrinks (Ck     is g) = [Ck     js g | js <- powerset is \\ [is]]
    otherShrinks (Ckw    is g) = [Ckw    js g | js <- powerset is \\ [is]]
    otherShrinks (Forall ps g) = [Forall qs g | qs <- powerset ps \\ [ps]]
    otherShrinks (Exists ps g) = [Exists qs g | qs <- powerset ps \\ [ps]]
    otherShrinks _ = []
```

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.

```haskell
substit :: Prp -> Form -> Form -> Form
substit _ _   Top             = Top
```

```haskell
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 ++ " "
      ++ show f)
  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 ++ " "
      ++ show f)
  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)
substit _ _   (Dia _ _)            = undefined -- TODO needs substit in dynop! Dia dynop (
    substit q psi f)
```

The function `substitSet` applies multiple substitutions after each other. Note that this is *not* the same as simultaneous substitution. However, it is equivalent to simultaneous substitution if none of the replaced propositions occurs in the replacement formulas.

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

We also implement an “out of” subsittution $[A \sqsubseteq B]\varphi$.

```haskell
substitOutOf :: [Prp] -> [Prp] -> Form -> Form
substitOutOf truths allps = substitSet [(p, if p `elem` truths then Top else Bot) | p <-
    allps]
```

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

```haskell
replPsInP :: [(Prp,Prp)] -> Prp -> Prp
replPsInP repl p = fromMaybe p (lookup p repl)

replPsInF :: [(Prp,Prp)] -> Form -> Form
replPsInF _       Top       = Top
replPsInF _       Bot       = Bot
replPsInF repl (PrpF p)     = PrpF $ replPsInP repl 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 (replPsInP repl) ps) (replPsInF repl f)
replPsInF repl (Exists ps f) = Exists (map (replPsInP 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)
replPsInF _    (Dia _ _)            = undefined -- TODO needs propsIn dynop!
```

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
propsInForm (Dia _dynOp _f)     = undefined -- TODO needs HasVocab dynop!

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 $ concatMap unpack fs) where
                          unpack Top = []
                          unpack (Conj subfs) = map simStep $ filter (Top /=) subfs
                          unpack f = [simStep f]
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 $ concatMap unpack fs) where
                          unpack Bot = []
                          unpack (Disj subfs) = map simStep $ filter (Bot /=) subfs
                          unpack f = [simStep f]
simStep (Xor  [])       = Bot
simStep (Xor  [Bot])    = Bot
simStep (Xor  [f])      = simStep f
simStep (Xor  fs)       = Xor (map simStep $ filter (Bot /=) 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 _   Top)  = Top
simStep (Ck _   Bot)  = Bot
simStep (Ck ags f)    = Ck ags (simStep f)
simStep (Ckw _   Top) = Top
simStep (Ckw _   Bot) = Top
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)
simStep (Dia dynop f)       = Dia dynop (simStep f)
```

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
texForm (Dia (Dyn s _) f)   = " \\langle " ++ s ++ " \\rangle " ++ texForm f

instance TexAble Form where
  tex = removeDoubleSpaces . 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:

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

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

```
newtype BF = BF Form deriving (Eq,Ord,Show)

instance Arbitrary BF where
  arbitrary = sized $ randomboolformWith [P 1 .. P 100]
  shrink (BF f) = map BF $ shrinkform f

randomboolformWith :: [Prp] -> Int -> Gen BF
randomboolformWith allprops sz = BF <$> bf' sz where
  bf' 0 = PrpF <$> elements allprops
  bf' n = oneof [ pure SMCDEL.Language.Top
                , pure SMCDEL.Language.Bot
                , PrpF <$> elements allprops
                , Neg <$> st
                , (\x y -> Conj [x,y]) <$> st <*> st
                , (\x y z -> Conj [x,y,z]) <$> st <*> st <*> st
                , (\x y -> Disj [x,y]) <$> st <*> st
                , (\x y z -> Disj [x,y,z]) <$> st <*> st <*> st
                , Impl <$> st <*> st
                , Equi <$> st <*> st
                , (\x -> Xor [x]) <$> st
                , (\x y -> Xor [x,y]) <$> st <*> st
                , (\x y z -> Xor [x,y,z]) <$> st <*> st <*> st
                -- , (\p f -> Exists [p] f) <$> elements allprops <*> st
                -- , (\p f -> Forall [p] f) <$> elements allprops <*> st
                ]
    where
      st = bf' (n `div` 3)
```

The following is a general Arbitrary instance for formulas. It is used in Section 8.1 below.

```
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'
                   , 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` 5
```

```haskell
        arbitraryAg = (\(Ag i) -> i) <$> arbitrary
        arbitraryAgs = sublistOf (map show [1..(5::Integer)]) `suchThat` (not . null)
  shrink = shrinkform

newtype SimplifiedForm = SF Form deriving (Eq,Ord,Show)

instance Arbitrary SimplifiedForm where
  arbitrary = SF . simplify <$> arbitrary
  shrink (SF f) = nub $ map (SF . simplify) (shrinkform f)
```

2 S5 Kripke Models

We start with a summary of the standard semantics for DEL on Kripke models. The module of this section provides a simple explicit model checker. It is the basis for the translation methods in Section 4 and not meant to be used in practice otherwise. A more advanced and user-friendly explicit model checker for DEL is DEMO-S5 from [Eij14] which we include as `SMCDEL.Explicit.DEMO_S5`.

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

module SMCDEL.Explicit.S5 where

import Control.Arrow (second,(&&&))
import Data.Dynamic
import Data.GraphViz
import Data.List
import Data.Tuple (swap)
import Data.Maybe (fromMaybe)

import SMCDEL.Language
import SMCDEL.Internal.TexDisplay
import SMCDEL.Internal.Help (alleqWith,fusion,apply,(!),lfp)
import Test.QuickCheck
```

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 are represented by integers. Equivalence relations are modeled as partitions, i.e. lists of lists of worlds.

```
type World = Int

class HasWorlds a where
  worldsOf :: a -> [World]

instance (HasWorlds a, Pointed a b) => HasWorlds (a,b) where worldsOf = worldsOf . fst

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

data KripkeModelS5 = KrMS5 [World] [(Agent,Partition)] [(World,Assignment)] deriving (Eq,
    Ord,Show)

instance Pointed KripkeModelS5 World
type PointedModelS5 = (KripkeModelS5,World)

instance Pointed KripkeModelS5 [World]
type MultipointedModelS5 = (KripkeModelS5,[World])

instance HasAgents KripkeModelS5 where
  agentsOf (KrMS5 _ rel _) = map fst rel

instance HasVocab KripkeModelS5 where
  vocabOf (KrMS5 _ _ val) = map fst $ snd (head val)

instance HasWorlds KripkeModelS5 where
```

```
  worldsOf (KrMS5 ws _ _) = ws

newtype PropList = PropList [Prp]

withoutWorld :: KripkeModelS5 -> World -> KripkeModelS5
withoutWorld (KrMS5 worlds parts val) w = KrMS5
  (delete w worlds)
  (map (second (filter (/=[]) . map (delete w))) parts)
  (filter ((/=w).fst) val)

withoutProps :: KripkeModelS5 -> [Prp] -> KripkeModelS5
withoutProps (KrMS5 worlds parts val) dropProps = KrMS5
  worlds
  parts
  (map (second $ filter ((`notElem` dropProps) . fst)) val)

instance Arbitrary PropList where
  arbitrary = do
    moreprops <- sublistOf (map P [1..10])
    return $ PropList $ P 0 : moreprops

randomPartFor :: [World] -> Gen Partition
randomPartFor worlds = do
  indices <- infiniteListOf $ choose (1, length worlds)
  let pairs = zip worlds indices
  let parts = [ sort $ map fst $ filter ((==k).snd) pairs | k <- [1 .. (length worlds)] ]
  return $ sort $ filter (/=[]) parts

instance Arbitrary KripkeModelS5 where
  arbitrary = do
    nonActualWorlds <- sublistOf [1..8]
    let worlds = 0 : nonActualWorlds
    val <- mapM (\w -> do
      myAssignment <- zip defaultVocabulary <$> infiniteListOf (choose (True,False))
      return (w,myAssignment)
      ) worlds
    parts <- mapM (\i -> do
      myPartition <- randomPartFor worlds
      return (i,myPartition)
      ) defaultAgents
    return $ KrMS5 worlds parts val
  shrink m@(KrMS5 worlds _ _) =
    [ m `withoutWorld` w | not (null worlds), w <- worlds ]
```

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. $(\mathcal{M}, w) \vDash \varphi \wedge \psi$ *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 $\pi^{\mathcal{M}^{\Delta}_{\psi}}(w) := \pi^{\mathcal{M}}(w)$.

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.

```haskell
eval :: PointedModelS5 -> Form -> Bool
eval _ Top = True
eval _ Bot = False
eval (KrMS5 _ _ 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@(KrMS5 _ rel _),w) (K ag form) = all (\w' -> eval (m,w') form) vs where
  vs = concat $ filter (elem w) (apply rel ag)
eval (m@(KrMS5 _ rel _),w) (Kw ag form) = alleqWith (\w' -> eval (m,w') form) vs where
  vs = concat $ filter (elem w) (apply rel ag)
eval (m@(KrMS5 _ 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@(KrMS5 _ rel _),w) (Ckw ags form) = alleqWith (\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 (update pm form1) form2
eval pm (PubAnnounceW form1 form2) =
  if eval pm form1
    then eval (update pm form1) form2
    else eval (update 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
eval pm (Dia (Dyn dynLabel d) f) = case fromDynamic d of
  Just pactm -> eval pm (preOf (pactm :: PointedActionModelS5)) && eval (pm `update` pactm)
      f
  Nothing    -> case fromDynamic d of
    Just mpactm -> eval pm (preOf (mpactm :: MultipointedActionModelS5)) && eval (pm `
        update` mpactm) f
    Nothing     -> error $ "cannot update S5 Kripke model with '" ++ dynLabel ++ "':\n  "
        ++ show d

valid :: KripkeModelS5 -> Form -> Bool
valid m@(KrMS5 worlds _ _) f = all (\w -> eval (m,w) f) worlds

instance Semantics KripkeModelS5 where
  isTrue m f = all (\w -> isTrue (m,w) f) (worldsOf m)

instance Semantics PointedModelS5 where
  isTrue = eval

instance Semantics MultipointedModelS5 where
  isTrue (m,ws) f = all (\w -> isTrue (m,w) f) ws
```

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.

```haskell
instance Update KripkeModelS5 Form where
  unsafeUpdate m@(KrMS5 sts rel val) form = KrMS5 newsts newrel newval 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

instance Update PointedModelS5 Form where
  unsafeUpdate (m,w) f = (unsafeUpdate m f, w)

instance Update MultipointedModelS5 Form where
  unsafeUpdate (m,ws) f =
    let newm = unsafeUpdate m f in (newm, ws `intersect` worldsOf newm)

announce :: PointedModelS5 -> [Agent] -> Form -> PointedModelS5
announce pm@(m@(KrMS5 sts rel val), cur) ags form =
  if eval pm form then (KrMS5 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)
```

```
announceAction :: [Agent] -> [Agent] -> Form -> PointedActionModelS5
announceAction everyone listeners f = (am, 1) where
  am = ActMS5 -- [(Action,(Form,PostCondition))] [(Agent,Partition)]
    [ (1, (f, [])), (2, (Top, [])) ]
    [ (i, if i `elem` listeners then [[1],[2]] else [[1,2]] ) | i <- everyone ]
```

With a few lines we can also visualize our models using the module SMCDEL.Internal.TexDisplay. For example output, see Sections 10.1.1 and 10.1.2.

```
instance KripkeLike KripkeModelS5 where
  directed = const False
  getNodes (KrMS5 ws _ val) = map (show &&& labelOf) ws where
    labelOf w = tex $ apply val w
  getEdges (KrMS5 _ rel _) =
    nub [ (a,show x,show y) | a <- map fst rel, part <- apply rel a, x <- part, y <- part,
        x < y ]

instance TexAble KripkeModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike PointedModelS5 where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, cur) = [show cur]

instance TexAble PointedModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike MultipointedModelS5 where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, curs) = map show curs

instance TexAble MultipointedModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot
```

2.2 Bisimulations

```
type Bisimulation = [(World,World)]
```

```haskell
checkBisim :: Bisimulation -> KripkeModelS5 -> KripkeModelS5 -> Bool
checkBisim [] _                                _                        = False
checkBisim z  m1@(KrMS5 _ rel1 val1) m2@(KrMS5 _ rel2 val2) =
  all (\(w1,w2) ->
        (val1 ! w1 == val2 ! w2)  -- same valuation
    && and [ any (\v2 -> (v1,v2) `elem` z) (concat $ filter (elem w2) (rel2 ! ag)) -- forth
           | ag <- agentsOf m1, v1 <- concat $ filter (elem w1) (rel1 ! ag) ]
    && and [ any (\v1 -> (v1,v2) `elem` z) (concat $ filter (elem w1) (rel1 ! ag)) -- back
           | ag <- agentsOf m2, v2 <- concat $ filter (elem w2) (rel2 ! ag) ]
      ) z

checkBisimPointed :: Bisimulation -> PointedModelS5 -> PointedModelS5 -> Bool
checkBisimPointed z (m1,w1) (m2,w2) = (w1,w2) `elem` z && checkBisim z m1 m2
```

2.3 Minimization

The generated submodel of a pointed model is the smallest submudel closed under following the epistemic relation.

```haskell
generatedSubmodel :: PointedModelS5 -> PointedModelS5
generatedSubmodel (KrMS5 oldWorlds rel val, cur) =
  if cur `notElem` oldWorlds
    then error "Actual world is not in the model!"
    else (KrMS5 newWorlds newrel newval, cur) where
      newWorlds :: [World]
      newWorlds = lfp follow [cur] where
        follow xs = sort . nub $ concat [ part | (_,parts) <- rel, part <- parts, any (`
            elem` part) xs ]
      newrel = map (second $ filter (any (`elem` newWorlds))) rel
      newval = filter (\p -> fst p `elem` newWorlds) val
```

To find a possibly smaller but bisimilar model, we use the following version of partition refinement. The initial partition is given by the valuation. Then we `foldl` through all agents once, splitting the existing blocks with their relation. As we only have equivalence relations, one pass is enough and no looping until we reach a fixpoint is needed.

```haskell
bisimClasses :: KripkeModelS5 -> [[World]]
bisimClasses m@(KrMS5 _ rel val) = refine sameAssignmentPartition where
  sameAssignmentPartition =
    map (map snd)
      $ groupBy (\x y -> fst x == fst y)
        $ sort (map swap val)
  refine parts = sort $ map sort $ foldl splitByAgent parts (agentsOf m)
  splitByAgent parts a =
    concat [ filter (not . null) [ ws `intersect` aPart | aPart <- rel ! a ] | ws <- parts
        ]

checkBisimClasses :: KripkeModelS5 -> Bool
checkBisimClasses m =
  and [ checkBisimPointed (swapZ w1 w2) (m,w1) (m,w2)
      | part <- bisimClasses m, w1 <- part, w2 <-part, w1 /= w2 ] where
    swapZ w1 w2 = sort $ [(w1,w2),(w2,w1)] ++ [ (w,w) | w <- worldsOf m \\ [w1,w2] ]

bisiminimize :: PointedModelS5 -> PointedModelS5
bisiminimize (m,w) =
  if all ((==1) . length) (bisimClasses m)
    then (m,w) -- nothing to minimize
    else (KrMS5 newWorlds newRel newVal, copyFct w) where
      KrMS5 _ oldRel oldVal = m
      copyRel       = zip (bisimClasses m) [1..]
      copyFct wOld = snd $ head $ filter ((wOld `elem`) . fst) copyRel
      newWorlds     = map snd copyRel
      newRel        = [ (a,newRelFor a) | a <- agentsOf m ]
      newRelFor a   = [ nub [ copyFct wOld | wOld <- part ] |  part <- oldRel ! a ]
      newVal        = [ (wNew, oldVal ! wOld) | (wOld:_,wNew) <- copyRel ]
```

We now optimize a Kripke model by taking the generated submodel and then minimizing under bisimulation. Note that we do not use the given vocabulary.

```
instance Optimizable PointedModelS5 where
  optimize _ = bisiminimize . generatedSubmodel
```

2.4 S5 Action Models

To model epistemic and ontic events in general we use action models from [BMS98].

```
type Action = Int
type PostCondition = [(Prp,Form)]

data ActionModelS5 = ActMS5 [(Action,(Form,PostCondition))] [(Agent,Partition)]
  deriving (Eq,Ord,Show)

instance HasAgents ActionModelS5 where
  agentsOf (ActMS5 _ rel) = map fst rel

-- | A safe way to `lookup` all postconditions
safepost :: PostCondition -> Prp -> Form
safepost posts p = fromMaybe (PrpF p) (lookup p posts)

instance Pointed ActionModelS5 Action
type PointedActionModelS5 = (ActionModelS5, Action)

instance HasPrecondition ActionModelS5 where
  preOf _ = Top

instance HasPrecondition PointedActionModelS5 where
  preOf (ActMS5 acts _, actual) = fst (acts ! actual)

instance Pointed ActionModelS5 [World]
type MultipointedActionModelS5 = (ActionModelS5,[Action])

instance HasPrecondition MultipointedActionModelS5 where
  preOf (am, actuals) = Disj [ preOf (am, a) | a <- actuals ]

instance KripkeLike ActionModelS5 where
  directed = const False
  getNodes (ActMS5 acts _) = map labelOf acts where
    labelOf (a,(pre,posts)) = (show a, concat
      [ "$\\begin{array}{c} ? " , tex pre, "\\\\"
      , intercalate "\\\\" (map showPost posts)
      , "\\end{array}$" ])
    showPost (p,f) = tex p ++ " := " ++ tex f
  getEdges am@(ActMS5 _ rel) =
    nub [ (a, show x, show y) | a <- agentsOf am, part <- rel ! a, x <- part, y <- part, x
        < y ]
  getActuals _ = [ ]
  nodeAts _ True  = [shape BoxShape, style solid]
  nodeAts _ False = [shape BoxShape, style dashed]

instance TexAble ActionModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike PointedActionModelS5 where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, cur) = [show cur]

instance TexAble PointedActionModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike MultipointedActionModelS5 where
```

```haskell
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, curs) = map show curs

instance TexAble MultipointedActionModelS5 where
  tex           = tex.ViaDot
  texTo         = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance Arbitrary ActionModelS5 where
  arbitrary = do
    BF f <- sized $ randomboolformWith [P 0 .. P 4]
    BF g <- sized $ randomboolformWith [P 0 .. P 4]
    BF h <- sized $ randomboolformWith [P 0 .. P 4]
    myPost <- (\_ -> do
      proptochange <- elements [P 0 .. P 4]
      postconcon <- elements $ [Top,Bot] ++ map PrpF [P 0 .. P 4]
      return [ (proptochange, postconcon) ]
      ) (0::Action)
    return $
      ActMS5
        [ (0,(Top,[]))
        , (1,(f  ,[]))
        , (2,(g  ,myPost))
        , (3,(h  ,[]))    ]
        ( ("0",[[0],[1],[2],[3]]):[(show k,[[0..3::Int]]) | k<-[1..5::Int] ])

instance Update KripkeModelS5 ActionModelS5 where
  checks = [haveSameAgents]
  unsafeUpdate m am@(ActMS5 acts _) =
    let (newModel,_) = unsafeUpdate (m, worldsOf m) (am, map fst acts) in newModel

instance Update PointedModelS5 PointedActionModelS5 where
  checks = [haveSameAgents,preCheck]
  unsafeUpdate (m, w) (actm, a) =
    let (newModel,[newWorld]) = unsafeUpdate (m, [w]) (actm, [a]) in (newModel,newWorld)

instance Update PointedModelS5 MultipointedActionModelS5 where
  checks = [haveSameAgents,preCheck]
  unsafeUpdate (m, w) mpactm =
    let (newModel,[newWorld]) = unsafeUpdate (m, [w]) mpactm in (newModel,newWorld)

instance Update MultipointedModelS5 PointedActionModelS5 where
  checks = [haveSameAgents] -- do not check precondition!
  unsafeUpdate mpm (actm, a) = unsafeUpdate mpm (actm, [a])

instance Update MultipointedModelS5 MultipointedActionModelS5 where
  checks = [haveSameAgents] -- do not check precondition!
  unsafeUpdate (m@(KrMS5 oldWorlds oldrel oldval), oldcurs) (ActMS5 acts actrel, factions)
      =
    (KrMS5 newWorlds newrel newval, newcurs) where
      startcount         = maximum oldWorlds + 1
      copiesOf (s,a)     = [ (s, a, a * startcount + s) | eval (m, s) (fst $ acts ! a) ]
      newWorldsTriples   = concat [ copiesOf (s,a) | s <- oldWorlds, (a,_) <- acts ]
      newWorlds          = map (\(_,_,x) -> x) newWorldsTriples
      newval             = map (\(s,a,t) -> (t, newValAt (s,a))) newWorldsTriples where
        newValAt sa = [ (p, newValAtFor sa p) | p <- vocabOf m ]
        newValAtFor (s,a) p = case lookup p (snd (acts ! a)) of
          Just postOfP -> eval (m, s) postOfP
          Nothing      -> (oldval ! s) ! p
      listFor ag         = cartProd (apply oldrel ag) (apply actrel ag)
      newPartsFor ag     = [ cartProd as bs | (as,bs) <- listFor ag ]
      translSingle pair = filter (`elem` newWorlds) $ 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 ]
      newcurs            = concat [ map (\(_,_,x) -> x) $ copiesOf (s,a) | s <- oldcurs, a
          <- factions ]
      cartProd xs ys     = [ (x,y) | x <- xs, y <- ys ]
```

3 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 currently can be used with two different BDD packages. Both are written in other languages than Haskell and have to be used via bindings:

1. *CacBDD* [LSX13], a modern BDD package with dynamic cache management implemented in C++. We use it via the library *HasCacBDD* [Gat17] which provides Haskell-to-C-to-C++ bindings.
2. *CUDD* [Som12], probably the best-known BDD library which is used many in other model checkers, including MCMAS [LQR15], MCK [GM04] and NuSMV [Cim+02]. 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.S5` and `SMCDEL.Symbolic.S5_CUDD`. Here we list the *CacBDD* variant.

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

module SMCDEL.Symbolic.S5 where

import Control.Arrow (first,second,(***))
import Data.Char (isSpace)
import Data.Dynamic
import Data.HasCacBDD hiding (Top,Bot)
import Data.HasCacBDD.Visuals
import Data.List ((\\),delete,dropWhile,dropWhileEnd,intercalate,intersect,nub,sort)
import Data.Tagged
import System.IO (hPutStr, hGetContents, hClose)
import System.IO.Unsafe (unsafePerformIO)
import System.Process (runInteractiveCommand)
import Test.QuickCheck

import SMCDEL.Internal.Help ((!),alleqWith,apply,applyPartial,lfp,powerset,rtc,seteq)
import SMCDEL.Internal.TaggedBDD
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Other.BDD2Form
```

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."

boolEvalViaBdd :: [Prp] -> Form -> Bool
boolEvalViaBdd truths = bddEval truths . boolBddOf

bddEval :: [Prp] -> Bdd -> Bool
bddEval truths querybdd = evaluateFun querybdd (\n -> P n `elem` truths)

relabelWith :: [(Prp,Prp)] -> Bdd -> Bdd
relabelWith r = relabel (sort $ map (fromEnum *** fromEnum) r)
```

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 [Ben+15].

Definition 5. *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]              -- vocabulary
                      Bdd                -- state law
                      [(Agent,[Prp])]    -- observational variables
                    deriving (Eq,Show)

type State = [Prp]

instance Pointed KnowStruct State
type KnowScene = (KnowStruct,State)

instance Pointed KnowStruct Bdd
type MultipointedKnowScene = (KnowStruct,Bdd)

statesOf :: KnowStruct -> [State]
statesOf (KnS props lawbdd _) = map (sort.translate) resultlists where
  resultlists :: [[(Prp, Bool)]]
  resultlists = map (map (first toEnum)) $ allSatsWith (map fromEnum props) lawbdd
  translate l = map fst (filter snd l)

instance HasAgents KnowStruct where
  agentsOf (KnS _ _ obs)= map fst obs

instance HasVocab KnowStruct where
  vocabOf (KnS props _ _) = props

numberOfStates :: KnowStruct -> Int
numberOfStates (KnS props lawbdd _) = satCountWith (map fromEnum props) lawbdd

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

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

Definition 6. *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. $(\mathcal{F}, s) \vDash \varphi \wedge \psi$ *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 7 and*

$$\mathcal{F}^\psi := (V, \theta \wedge \|\psi\|_\mathcal{F}, O_1, \dots, O_n)$$

7. $(\mathcal{F}, s) \vDash [\psi]_\Delta\varphi$ *iff* $(\mathcal{F}, s) \vDash \psi$ *implies* $(\mathcal{F}^\Delta_\psi, s \cup \{p_\psi\}) \vDash \varphi$ *where* p_ψ *is a new propositional variable,* $\|\psi\|_\mathcal{F}$ *is a boolean formula given by Definition 7 and*

$$\mathcal{F}^\Delta_\psi := (V \cup \{p_\psi\}, \theta \wedge (p_\psi \leftrightarrow \|\psi\|_\mathcal{F}), O^*_1, \dots, O^*_n)$$

where $O^*_i := \begin{cases} O_i \cup \{p_\psi\} & \text{if } i \in \Delta \\ O_i & \text{otherwise} \end{cases}$

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 6. 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 :: KnowScene -> 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 [ SMCDEL.Language.substit p Top g, SMCDEL.Language.substit p Bot
      g ]
eval scn       (Exists ps f) = eval scn (foldl singleExists f ps) where
  singleExists g p = Disj [ SMCDEL.Language.substit p Top g, SMCDEL.Language.substit p Bot
      g ]
eval (kns@(KnS _ _ obs),s) (K i form) = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (s' `intersect` oi) (s `intersect` oi)) (statesOf kns)
  oi = obs ! i
eval (kns@(KnS _ _ obs),s) (Kw i form) = alleqWith (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (s' `intersect` oi) (s `intersect` oi)) (statesOf kns)
  oi = obs ! i
eval (kns,s) (Ck ags form)  = all (\s' -> eval (kns,s') form) theres where
  theres = [ s' | (s0,s') <- comknow kns ags, s0 == s ]
eval (kns,s) (Ckw ags form)  = alleqWith (\s' -> eval (kns,s') form) theres where
  theres = [ s' | (s0,s') <- comknow kns ags, s0 == s ]
eval scn (PubAnnounce form1 form2) =
  not (eval scn form1) || eval (update scn form1) form2
eval (kns,s) (PubAnnounceW form1 form2) =
  if eval (kns, s) form1
    then eval (update kns form1, s) form2
    else eval (update 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
eval scn (Dia (Dyn dynLabel d) f) = case fromDynamic d of
  Just event -> eval scn (preOf (event :: Event))
                && eval (scn `update` event) f
  Nothing    -> error $ "cannot update knowledge structure with '" ++ dynLabel ++ "':\n  "
      ++ show d
```

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 6.

```haskell
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, obs ! i ++ [proppsi | i `elem` ags]) | i <- map fst obs]

announceOnScn :: KnowScene -> [Agent] -> Form -> KnowScene
announceOnScn (kns@(KnS props _ _),s) ags psi
  | eval (kns,s) psi = (announce kns ags psi, sort $ 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, but generates a BDD. It only makes calls to itself, the announcement functions and the boolean operations provided by the BDD package.

Definition 7. *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 $\|K_i\psi\|_{\mathcal{F}} := \forall(V \setminus O_i)(\theta \to \|\psi\|_{\mathcal{F}})$.*
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:*

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

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

where $\mathcal{F}^\psi$ and $\mathcal{F}^\Delta_\psi$ are as given by Definition 6.

```haskell
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 \\ 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 \\ 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 \\ 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 (update kns form1) form2)
bddOf kns (PubAnnounceW form1 form2) =
  ifthenelse (bddOf kns form1) newform2a newform2b where
    newform2a = bddOf (update kns form1) form2
    newform2b = bddOf (update kns (Neg form1)) form2
```

The last case of bddOf extends our boolean translation to dynamic operators with knowledge transformers. In two subcases we deal with pointed events like $(\mathcal{X}, x)$ and multipointed events like $(\mathcal{X}, \sigma)$. For pointed events, an explanation of the chain of substitutions can be found in [Gat18, p. 74/5]. The multipointed case only differs in step 3 where instead of a single event $x \subseteq V^+$ a set of events described by $\sigma \in \mathcal{L}_B(V^+)$ is simulated.

```
bddOf kns (Dia (Dyn dynLabel d) f) =
    con (bddOf kns preCon)                   -- 5. Prefix with "precon AND ..." (diamond!)
    . relabelWith copyrelInverse             -- 4. Copy back changeProps V_° to V_
    . simulateActualEvents                   -- 3. Simulate actual event(s) [see below]
    . substitSimul [ (k, changeLaw ! p)      -- 2. Replace changeProps V_ with postcons
                   | p@(P k) <- changeProps] --    (no "relabelWith copyrel", undone in 4)
    . bddOf (kns `update` trf)               -- 1. boolean equivalent wrt new struct
    $ f
  where
    changeProps = map fst changeLaw
    copychangeProps = [(freshp $ vocabOf kns ++ addProps)..]
    copyrelInverse  = zip copychangeProps changeProps
    (trf@(KnTrf addProps addLaw changeLaw _), shiftrel) = shiftPrepare kns trfUnshifted
    (preCon,trfUnshifted,simulateActualEvents) =
      case fromDynamic d of
        -- 3. For a single event, simulate actual event x outof V+
        Just ((t,x) :: Event) -> ( preOf (t,x), t, (`restrictSet` actualAss) )
          where actualAss = [(newK, P k `elem` x) | (P k, P newK) <- shiftrel]
        Nothing -> case fromDynamic d of
          -- 3. For a multipointed event, simulate a set of actual events by ...
          Just ((t,xsBdd) :: MultipointedEvent) ->
              ( preOf (t,xsBdd), t
              , existsSet (map fromEnum addProps)  -- ... replacing addProps with
                  assigments
                . con actualsBdd                   -- ... that satisfy actualsBdd
                . con (bddOf kns addLaw)           -- ... and a precondition.
              ) where actualsBdd = relabelWith shiftrel xsBdd
          Nothing -> error $ "cannot update knowledge structure with '" ++ dynLabel ++ "':\
              n " ++ show d
```

Theorem 8. *Definition 7 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 :: KnowScene -> Form -> Bool
evalViaBdd (kns,s) f = evaluateFun (bddOf kns f) (\n -> P n `elem` s)

instance Semantics KnowStruct where
  isTrue = validViaBdd

instance Semantics KnowScene where
  isTrue = evalViaBdd

instance Semantics MultipointedKnowScene where
  isTrue (kns@(KnS _ lawBdd _),statesBdd) f =
```

```haskell
    let a = lawBdd `imp` (statesBdd `imp` bddOf kns f)
     in a == top

instance Update KnowStruct Form where
  checks = [ ] -- unpointed structures can always be updated with anything
  unsafeUpdate kns@(KnS props lawbdd obs) psi =
    KnS props (lawbdd `con` bddOf kns psi) obs

instance Update KnowScene Form where
  unsafeUpdate (kns,s) psi = (unsafeUpdate kns psi,s)
```

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 9. *Definition 7 preserves and reflects validity. That is, for any formula φ and any knowledge structure $\mathcal{F}$ with the state law θ we have that $\mathcal{F} \models \varphi$ iff $\theta \to \|\varphi\|_{\mathcal{F}}$ is a boolean tautology.*

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

```haskell
whereViaBdd :: KnowStruct -> Form -> [State]
whereViaBdd kns@(KnS props lawbdd _) f =
 map (sort . map (toEnum . fst) . filter snd) $
   allSatsWith (map fromEnum props) $ con lawbdd (bddOf kns f)
```

3.2 Minimization and Optimization

Knowledge structures can contain unnecessary vocabulary, i.e. atomic propositions that are determined by the state law and not used as observational propositions.

```haskell
determinedVocabOf :: KnowStruct -> [Prp]
determinedVocabOf strct =
  filter (\p -> validViaBdd strct (PrpF p) || validViaBdd strct (Neg $ PrpF p)) (vocabOf
      strct)

nonobsVocabOf  :: KnowStruct -> [Prp]
nonobsVocabOf (KnS vocab _ obs) = filter (`notElem` usedVars) vocab where
  usedVars = sort $ concatMap snd obs

equivExtraVocabOf :: [Prp] -> KnowStruct -> [(Prp,Prp)]
equivExtraVocabOf mainVocab kns =
  [ (p,q) | p <- vocabOf kns \\ mainVocab, q <- vocabOf kns, p > q, validViaBdd kns (PrpF p
       `Equi` PrpF q) ]

replaceWithIn :: (Prp,Prp) -> KnowStruct -> KnowStruct
replaceWithIn (p,q) (KnS oldProps oldLaw oldObs) =
  KnS
    (delete p oldProps)
    (Data.HasCacBDD.substit (fromEnum p) (var (fromEnum q)) oldLaw)
    [(i, if p `elem` os then sort $ nub (q : delete p os) else os) | (i,os) <- oldObs]

replaceEquivExtra :: [Prp] -> KnowStruct -> (KnowStruct,[(Prp,Prp)])
replaceEquivExtra mainVocab startKns = lfp step (startKns,[]) where
  step (kns,replRel) = case equivExtraVocabOf mainVocab kns of
                 []         -> (kns,replRel)
                 ((p,q):_) -> (replaceWithIn (p,q) kns, (p,q):replRel)
```

Removing those atomic propositions from a structure will make the state law smaller.

```haskell
withoutProps :: [Prp] -> KnowStruct -> KnowStruct
withoutProps propsToDel (KnS oldProps oldLawBdd oldObs) =
  KnS
    (oldProps \\ propsToDel)
    (existsSet (map fromEnum propsToDel) oldLawBdd)
    [(i,os \\ propsToDel) | (i,os) <- oldObs]
```

Putting all these helper functions togehter, we can optimiz a knowledge structure as follows, given a main vocabulary that we want to keep.

```
instance Optimizable MultipointedKnowScene where
  optimize myVocab (oldKns,oldStatesBdd) = (newKns,newStatesBdd) where
    intermediateKns = withoutProps ((determinedVocabOf oldKns `intersect` nonobsVocabOf
        oldKns) \\ myVocab) oldKns
    removedProps = vocabOf oldKns \\ vocabOf intermediateKns
    intermediateStatesBdd = existsSet (map fromEnum removedProps) oldStatesBdd
    (newKns,replRel) = replaceEquivExtra myVocab intermediateKns
    newStatesBdd = foldr (uncurry Data.HasCacBDD.substit) intermediateStatesBdd [ (fromEnum
        p, var (fromEnum q)) | (p,q) <-replRel ]
```

The equivalent of a generated submodel is not always an optmization on symbolic structures. For further discussion, see Section 2.2 from [Gat18]. Still, there are cases where generated substructures are useful, hence we implement them. In paritcular in combination with the other optimizations above, smaller structures can be obtained.

```
generatedSubstructure :: MultipointedKnowScene -> MultipointedKnowScene
generatedSubstructure kns@(KnS props oldLaw obs, curBdd) = (KnS props newLaw obs, curBdd)
    where
  newLaw = oldLaw `con` disSet (curBdd : [ existsSet (map fromEnum $ props \\ obs ! i)
      curBdd
                                         | i <- agentsOf kns ])
```

3.3 Symbolic Bisimulations for S5

See Section 2.11 from [Gat18] for details.

To distinguish explicit and symbolic bisimulations in the implementation we call symbolic bisimulations "propulations".

```
type Propulation = Tagged Quadrupel Bdd

($$) :: Monad m => ([a] -> b) -> [m a] -> m b
($$) f xs = f <$> sequence xs

checkPropu :: Propulation -> KnowStruct -> KnowStruct -> [Prp] -> Bool
checkPropu propu kns1@(KnS _ law1 obs1) kns2@(KnS _ law2 obs2) voc =
  pure top == (imp <$> lhs <*> rhs) where
    lhs = conSet $$ [mv law1, cp law2, propu]
    rhs = conSet $$ [propAgree, forth, back]
    propAgree = allsamebdd voc
    forth = conSet $$ [ forallSet (nonObs i obs1) <$>
                          (imp <$> mv law1 <*> (existsSet (nonObs i obs2) <$> (con <$> cp
                              law2 <*> propu)))
                      | i <- agentsOf kns1 ]
    back  = conSet $$ [ forallSet (nonObs i obs1) <$>
                          (imp <$> mv law2 <*> (existsSet (nonObs i obs1) <$> (con <$> cp
                              law1 <*> propu)))
                      | i <- agentsOf kns2 ]
    nonObs i obs = map (\(P n) -> n) $ voc \\ obs ! i
```

3.4 Knowledge Transformers

The symbolic model checking method can be extended to cover other kinds of events. What action models are to Kripke models, the following knowledge transformers are to knowledge structures. The analog of product update is knowledge transformation.

Definition 10. *A* knowledge transformer *for a given vocabulary V and set of agents $I = \{1, \dots, n\}$ is a tuple $\mathcal{X} = (V^+, \theta^+, O_1, \dots, O_n)$ 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 KnowTransformer = KnTrf
  [Prp]                -- addProps
  Form                 -- addLaw
  [(Prp,Bdd)]          -- changeLaw
  [(Agent,[Prp])]      -- addObs
  deriving (Show)

noChange :: ([Prp] -> Form -> [(Prp,Bdd)] -> [(Agent,[Prp])] -> KnowTransformer)
          -> [Prp] -> Form                  -> [(Agent,[Prp])] -> KnowTransformer
noChange kntrf addprops addlaw = kntrf addprops addlaw []

instance HasAgents KnowTransformer where
  agentsOf (KnTrf _ _ _ obdds) = map fst obdds

instance HasPrecondition KnowTransformer where
  preOf _ = Top

instance Pointed KnowTransformer State
type Event = (KnowTransformer,State)

instance HasPrecondition Event where
  preOf (KnTrf addprops addlaw _ _, x) = simplify $ substitOutOf x addprops addlaw

instance Pointed KnowTransformer Bdd
type MultipointedEvent = (KnowTransformer,Bdd)

instance HasPrecondition MultipointedEvent where
  preOf (KnTrf addprops addlaw _ _, xsBdd) =
    simplify $ Exists addprops (Conj [ formOf xsBdd, addlaw ])
```

The easiest example of a knowledge transformer is the one for public announcements:

```
publicAnnounce :: [Agent] -> Form -> Event
publicAnnounce agents f = (noChange KnTrf [] f myobs, []) where
  myobs = [ (i,[]) | i <- agents ]
```

The following defines S5 transformation with factual change.

```
-- | shift addprops to ensure that props and newprops are disjoint:
shiftPrepare :: KnowStruct -> KnowTransformer -> (KnowTransformer, [(Prp,Prp)])
shiftPrepare (KnS props _ _) (KnTrf addprops addlaw changelaw eventObs) =
  (KnTrf shiftaddprops addlawShifted changelawShifted eventObsShifted, shiftrel) where
    shiftrel = sort $ zip addprops [(freshp props)..]
    shiftaddprops = map snd shiftrel
    -- apply the shifting to addlaw, changelaw and eventObs:
    addlawShifted     = replPsInF shiftrel addlaw
    changelawShifted = map (second (relabelWith shiftrel)) changelaw
    eventObsShifted  = map (second (map (apply shiftrel))) eventObs

instance Update KnowScene Event where
  unsafeUpdate (kns@(KnS props _ _),s) (ctrf, eventFactsUnshifted) = (KnS newprops newlaw
      newobs, news) where
    -- PART 1: SHIFTING addprops to ensure props and newprops are disjoint
    (KnTrf addprops _ changelaw _, shiftrel) = shiftPrepare kns ctrf
```

```
    -- the actual event:
    eventFacts = map (apply shiftrel) eventFactsUnshifted
    -- PART 2: COPYING the modified propositions
    changeprops = map fst changelaw
    copyrel = zip changeprops [(freshp $ props ++ addprops)..]
    -- do the pointless update and calculate new actual state
    KnS newprops newlaw newobs = unsafeUpdate kns ctrf
    news = sort $ concat
            [ s \\ changeprops
            , map (apply copyrel) $ s `intersect` changeprops
            , eventFacts
            , filter (\ p -> bddEval (s ++ eventFacts) (changelaw ! p)) changeprops ]
```

Using laziness we can also define the pointless update of a structure with a transformer.

```
instance Update KnowStruct KnowTransformer where
  checks = [haveSameAgents]
  unsafeUpdate kns ctrf = KnS newprops newlaw newobs where
    (KnS newprops newlaw newobs, _) = unsafeUpdate (kns,undefined::Bdd) (ctrf,undefined::
        Bdd) -- using laziness!

instance Update MultipointedKnowScene MultipointedEvent where
  unsafeUpdate (kns@(KnS props law obs),statesBdd) (ctrf,eventsBddUnshifted)  =
    (KnS newprops newlaw newobs, newStatesBDD) where
      (KnTrf addprops addlaw changelaw eventObs, shiftrel) = shiftPrepare kns ctrf
      -- apply the shifting to eventsBdd:
      eventsBdd = relabelWith shiftrel eventsBddUnshifted
      -- PART 2: COPYING the modified propositions
      changeprops = map fst changelaw
      copyrel = zip changeprops [(freshp $ props ++ addprops)..]
      copychangeprops = map snd copyrel
      newprops = sort $ props ++ addprops ++ copychangeprops -- V ∪ V+ ∪ V°
      newlaw = conSet $ relabelWith copyrel (con law (bddOf kns addlaw))
                    : [var (fromEnum q) `equ` relabelWith copyrel (changelaw ! q) | q <-
                        changeprops]
      newobs = [ (i , sort $ map (applyPartial copyrel) (obs ! i) ++ eventObs ! i) | i <-
          map fst obs ]
      newStatesBDD = conSet [ relabelWith copyrel statesBdd, eventsBdd ]
```

Note that in the last line we do not say anything about the `changeprops`. This works because new actual states are given by the conjunction of the `newlaw` and `newstatesBDD`. Hence the new state law will determine the values of the (un)changed variables in the new actual states.

We end this module with helper functions to generate LaTeX code that shows a given knowledge structure, including a BDD of the state law. See Section 10.1 for examples of what the output looks like.

```
texBddWith :: (Int -> String) -> Bdd -> String
texBddWith myShow b = unsafePerformIO $ do
  (i,o,_,_) <- runInteractiveCommand "dot2tex --figpreamble=\"\\huge\" --figonly -traw"
  hPutStr i (genGraphWith myShow b ++ "\n")
  hClose i
  result <- hGetContents o
  return $ dropWhileEnd isSpace $ dropWhile isSpace result

texBDD :: Bdd -> String
texBDD = texBddWith show

newtype WrapBdd = Wrap Bdd

instance TexAble WrapBdd where
  tex (Wrap b) = texBDD b

instance TexAble KnowStruct where
  tex (KnS props lawbdd obs) = 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)" ]

instance TexAble KnowScene where
  tex (kns, state) = tex kns ++ " , " ++ tex state

instance TexAble MultipointedKnowScene where
  tex (kns, statesBdd) = concat
    [ " \\left( \n"
    , tex kns ++ ", "
    , " \\begin{array}{l} \\scalebox{0.4}{"
    , texBDD statesBdd
    , "} \\end{array}\n "
    , " \\right)" ]

instance TexAble KnowTransformer where
  tex (KnTrf addprops addlaw changelaw eventObs) = concat
    [ " \\left( \n"
    , tex addprops, ", \\ "
    , tex addlaw, ", \\ "
    , intercalate ", " $ map texChange changelaw
    , ", \\ \\begin{array}{l}\n"
    , intercalate " \\\\\n " (map (\(_,os) -> tex os) eventObs)
    , "\\end{array}\n"
    , " \\right) \n"
    ] where
        texChange (prop,changebdd) = concat
          [ tex prop ++ " := "
          , " \\begin{array}{l} \\scalebox{0.4}{"
          , texBDD changebdd
          , "} \\end{array}\n " ]

instance TexAble Event where
  tex (trf, eventFacts) = concat
    [ " \\left( \n", tex trf, ", \\ ", tex eventFacts, " \\right) \n" ]

instance TexAble MultipointedEvent where
  tex (trf, eventsBdd) = concat
    [ " \\left( \n"
    , tex trf ++ ", \\ "
    , " \\begin{array}{l} \\scalebox{0.4}{"
    , texBDD eventsBdd
    , "} \\end{array}\n "
    , " \\right)" ]
```

3.5 Reduction axioms for knowledge transformers

Adding knowledge transformers does not increase expressivity because we have the following reductions.

For now we do not implement a separate type of formulas with dynamic operators but instead implement the reduction axioms directly as a function which takes an event and "pushes it through" a formula.

The following takes an event $\mathcal{X}, x$ and a formula φ and then "pushes" $[\mathcal{X}, x]$ through all boolean and epistemic operators in φ until it disappears in front of atomic propositons. This translation is global, i.e. if there is a reduced formula, then it is equivalent to the original on all structures.

```
reduce :: Event -> Form -> Maybe Form
reduce _ Top           = Just Top
reduce e Bot           = pure $ Neg (preOf e)
reduce e (PrpF p)      = Impl (preOf e) <$> Just (PrpF p) -- FIXME use change!
reduce e (Neg f)       = Impl (preOf e) <$> (Neg <$> reduce e f)
reduce e (Conj fs)     = Conj <$> mapM (reduce e) fs
reduce e (Disj fs)     = Disj <$> mapM (reduce e) fs
reduce e (Xor fs)      = Impl (preOf e) <$> (Xor <$> mapM (reduce e) fs)
reduce e (Impl f1 f2) = Impl <$> reduce e f1 <*> reduce e f2
reduce e (Equi f1 f2) = Equi <$> reduce e f1 <*> reduce e f2
```

```
reduce _ (Forall _ _) = Nothing
reduce _ (Exists _ _) = Nothing
reduce event@(trf@(KnTrf addprops _ _ obs), x) (K a f) =
  Impl (preOf event) <$> (Conj <$> sequence
    [ K a <$> reduce (trf,y) f | y <- powerset addprops
                                , (x `intersect` (obs ! a)) `seteq` (y `intersect` (obs ! a)
                                    )
    ])
reduce e (Kw a f)      = reduce e (Disj [K a f, K a (Neg f)])
reduce _ Ck  {}        = Nothing
reduce _ Ckw {}        = Nothing
reduce _ PubAnnounce  {} = Nothing
reduce _ PubAnnounceW {} = Nothing
reduce _ Announce     {} = Nothing
reduce _ AnnounceW    {} = Nothing
reduce _ Dia          {} = Nothing
```

3.6 Random Knowledge Structures

```
instance Arbitrary KnowStruct where
  arbitrary = do
    numExtraVars <- choose (0,3)
    let myVocabulary = defaultVocabulary ++ take numExtraVars [freshp defaultVocabulary ..]
    (BF statelaw) <- sized (randomboolformWith myVocabulary) `suchThat` (\(BF bf) ->
        boolBddOf bf /= bot)
    obs <- mapM (\i -> do
      obsVars <- sublistOf myVocabulary
      return (i,obsVars)
      ) defaultAgents
    return $ KnS defaultVocabulary (boolBddOf statelaw) obs
  shrink kns = [ withoutProps [p] kns | length (vocabOf kns) > 1, p <- vocabOf kns \\
      defaultVocabulary ]
```

4 Connecting S5 Kripke Models and Knowledge Structures

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.S5 where

import Control.Arrow (second)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List (groupBy,sort,(\\),elemIndex,intersect,nub)
import Data.Maybe (listToMaybe)

import SMCDEL.Language
import SMCDEL.Symbolic.S5
import SMCDEL.Explicit.S5
import SMCDEL.Internal.Help (anydiffWith,alldiff,alleqWith,apply,powerset,(!),seteq,
    subseteq)
import SMCDEL.Other.BDD2Form
```

Lemma 11. *Suppose we have a knowledge structure $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$ and a finite S5 Kripke model $M = (W, \pi, \mathcal{K}_1, \ldots, \mathcal{K}_n)$ 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 $(\mathcal{M}, w) \vDash \varphi$.

The following is an implementation of Lemma 11: Given a pointed model, a KnowScene and a function g, we check whether the conditions C1 to C3 are fulfilled.

```
type StateMap = World -> State

equivalentWith :: PointedModelS5 -> KnowScene -> StateMap -> Bool
equivalentWith (KrMS5 ws rel val, actw) (kns@(KnS _ _ obs), curs) g =
  c1 && c2 && c3 && g actw == curs where
    c1 = all (\l -> knsLink l == kriLink l) linkSet where
      linkSet = nub [ (i,w1,w2) | w1 <- ws, w2 <- ws, w1 <= w2, i <- map fst rel ]
      knsLink (i,w1,w2) = let oi = obs ! i in (g w1 `intersect` oi) `seteq` (g w2 `
          intersect` oi)
      kriLink (i,w1,w2) = any (\p -> w1 `elem` p && w2 `elem` p) (rel ! i)
    c2 = and [ (p `elem` g w) == ((val ! w) ! p) | w <- ws, p <- map fst (snd $ head val) ]
    c3 = statesOf kns `seteq` nub (map g ws)
```

Given only a pointed model and a KnowScene, we can also try to find a g that links them according to the three conditions. A fully naive approach would be to consider all functions g mapping worlds to subsets of U, but we already know that for $C2$ we need $\{p \mid \pi(w)(p) = \top\} \subseteq g(w)$. Hence the following only generates all possible choices for the propositions which are in the vocabulary of the knowledge structure but not in that of the Kripke Model. Finally, we filter out the good maps passing the equivalentWith test and connecting the given actual world and state.

```
findStateMap :: PointedModelS5 -> KnowScene -> Maybe StateMap
findStateMap pm@(KrMS5 _ _ val, w) scn@(kns, s)
  | vocabOf pm `subseteq` vocabOf kns = listToMaybe goodMaps
  | otherwise = error "vocabOf pm not subseteq vocabOf kns"
  where
    extraProps = vocabOf kns \\ vocabOf pm
    allFuncs :: Eq a => [a] -> [b] -> [a -> b]
```

```haskell
    allFuncs []      _  = [ const undefined ]
    allFuncs (x:xs) ys = [ \a -> if a == x then y else f a | y <- ys, f <- allFuncs xs ys ]
    allMaps, goodMaps :: [StateMap]
    baseMap   = map fst . filter snd . (val !)
    allMaps   = [ \v -> baseMap v ++ restf v | restf <- allFuncs (worldsOf pm) (powerset
        extraProps) ]
    goodMaps = filter (\g -> g w == s && equivalentWith pm scn g) allMaps
```

4.1 From Knowledge Structures to S5 Kripke Models

Definition 12. *For any knowledge structure $\mathcal{F} = (V, \theta, O_1, \dots, O_n)$, 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 13. *For any knowledge structure $\mathcal{F}$, any state s of $\mathcal{F}$, and any φ we have $(\mathcal{F}, s) \vDash \varphi$ iff $(M(\mathcal{F}), s) \vDash \varphi$.*

```haskell
knsToKripke :: KnowScene -> PointedModelS5
knsToKripke = fst . knsToKripkeWithG

knsToKripkeWithG :: KnowScene -> (PointedModelS5, StateMap)
knsToKripkeWithG (kns@(KnS ps _ obs),currentState) =
  ((KrMS5 worlds rel val, cur), g) where
    g w     = statesOf kns !! w
    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 -> (s `intersect` (obs ! i), lav ! s)) (statesOf kns)
    worlds = map snd lav
    cur     | currentState `elem` statesOf kns = lav ! currentState
            | otherwise = error "knsToKripke failed: Invalid state."

knsToKripkeMulti :: MultipointedKnowScene -> MultipointedModelS5
knsToKripkeMulti (kns,statesBdd) = (m, ws) where
  ((m,_),g) = knsToKripkeWithG (kns,undefined) -- FIXME uh oh
  ws = filter (\w -> evaluateFun statesBdd (\k -> P k `elem` g w)) (worldsOf m)
```

4.2 From S5 Kripke Models to Knowledge Structures

Definition 14. *For any S5 model $\mathcal{M} = (W, \pi, \mathcal{K}_1, \dots, \mathcal{K}_n)$ with the set of atomic 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 15. *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 :: PointedModelS5 -> KnowScene
kripkeToKns = fst . kripkeToKnsWithG

kripkeToKnsWithG :: PointedModelS5 -> (KnowScene, StateMap)
kripkeToKnsWithG (KrMS5 worlds rel val, cur) = ((KnS ps law obs, curs), g) where
  v          = map fst (val ! cur)
  ags        = map fst rel
  newpstart  = fromEnum $ freshp v -- start counting new propositions here
  amount i   = ceiling (logBase 2 (fromIntegral $ length (rel ! i)) :: Float) -- = |O_i|
  newpstep   = maximum [ amount i | i <- ags ]
  newps i    = map (\k -> P (newpstart + (newpstep * inum) +k)) [0..(amount i - 1)] -- O_i
    where (Just inum) = elemIndex i (map fst rel)
  copyrel i = zip (rel ! i) (powerset (newps i)) -- label equiv.classes with P(O_i)
  gag i w    = snd $ head $ filter (\(ws,_) -> w `elem` ws) (copyrel i)
  g w        = filter (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 :: KripkeModelS5 -> Bool
uniqueVals (KrMS5 _ _ val) = alldiff (map snd 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 :: KripkeModelS5 -> Agent -> ([Prp],[Prp])
obsnobs m@(KrMS5 _ rel val) i = (obs,nobs) where
  propsets = map (map (map fst . filter snd . apply val)) (apply rel i)
  obs = filter (\p -> all (alleqWith (elem p)) propsets) (vocabOf m)
  nobs = filter (\p -> any (anydiffWith (elem p)) propsets) (vocabOf m)

-- | Test if all relations can be described using observariables.
descableRels :: KripkeModelS5 -> Bool
descableRels m@(KrMS5 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 (vocabOf 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 :: PointedModelS5 -> Maybe KnowScene
smartKripkeToKns (m, cur) =
  if uniqueVals m && descableRels m
    then Just (smartKripkeToKnsWithoutChecks (m, cur))
    else Nothing

smartKripkeToKnsWithoutChecks :: PointedModelS5 -> KnowScene
smartKripkeToKnsWithoutChecks (m@(KrMS5 worlds rel val), cur) =
  (KnS ps law obs, curs) where
    ps = vocabOf 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 S5 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 12 and 14 and their soundness also follows from Lemma 11. The implementation below works on pointed models, to simplify tracking the actual world and action.

Definition 16. *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*

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

where $\sqsubseteq$ *is the "out of" abbreviation from Definition 14. 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 17. *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 12 and* $g_{\mathcal{A}}$ *is from the construction of* $\mathsf{Trf}(\mathcal{A})$ *in Definition 16.*

```
actionToEvent :: PointedActionModelS5 -> Event
actionToEvent (ActMS5 acts actrel, faction) = (KnTrf addprops addlaw changelaw addobs,
    efacts) where
  actions = map fst acts
  ags           = map fst actrel
  addprops      = actionprops ++ actrelprops
  (P fstnewp)   = freshp . propsInForms $ concat [ pre : map snd posts | (_,(pre,posts)) <-
      acts] -- avoid props in pre- and postconditions
  actionprops   = [P fstnewp..P maxactprop] -- new props to distinguish all actions
  maxactprop    = fstnewp + ceiling (logBase 2 (fromIntegral $ length actions) :: Float) -1
  actpropsRel   = zip actions (powerset actionprops)
  ell           = apply actpropsRel -- label actions with subsets of actionprops
  happens a     = booloutofForm (ell a) actionprops -- boolean formula to say that a happens
  actform       = Disj [ Conj [ happens a, pre ] | (a,(pre,_)) <- acts ] -- connect new
      propositions to preconditions
  actrelprops   = concat [ newps i | i <- ags ] -- new props to distinguish actions for i
  actrelpstart  = maxactprop + 1
  newps i       = map (\k -> P (actrelpstart + (newpstep * inum) +k)) [0..(amount i - 1)]
    where (Just inum) = elemIndex i (map fst actrel)
  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)) -- label equclasses-of-actions
      with subsets-of-newps
  actrelfs i    = [ Equi (booloutofForm (apply (copyactrel i) as) (newps i)) (Disj (map
      happens as)) | as <- apply actrel i ]
  actrelforms   = concatMap actrelfs ags
  factsFor i    = snd $ head $ filter (\(as,_) -> faction `elem` as) (copyactrel i)
  efacts        = ell faction ++ concatMap factsFor ags
  addlaw        = simplify $ Conj (actform : actrelforms)
  changeprops   = sort $ nub $ concatMap (\(_,(_,posts)) -> map fst posts) acts --
      propositions to be changed
  changelaw     = [ (p, changeFor p) | p <- changeprops ] -- encode postconditions
```

```
  changeFor p   = disSet [ boolBddOf $ Conj [ happens a, safepost posts p ] | (a,(_,posts))
      <- acts ]
  addobs        = [ (i,newps i) | i<- ags ]
```

4.4 From Knowledge Transformers to S5 Action Models

Definition 18. *For any Knowledge Transformer $\mathcal{X} = (V^+, \theta^+, O_1^+, \ldots, O_n^+)$ 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 19. *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. The implementation below follows the definition in `eventToAction'` and then removes all actions where $\mathsf{pre}(\alpha) = \bot$ in `eventToAction`.

```
eventToAction' :: Event -> PointedActionModelS5
eventToAction' event@(KnTrf addprops addlaw changelaw addobs, efacts) = (ActMS5 acts actrel
    , faction) where
  actlist = zip (powerset addprops) [0..(2 ^ length addprops - 1)]
  acts    = [ (a, (simplify $ preFor ps, postsFor ps)) | (ps,a) <- actlist ] where
    preFor ps = substitSet (zip ps (repeat Top) ++ zip (addprops\\ps) (repeat Bot)) addlaw
    postsFor ps =
      [ (q, formOf $ restrictSet (changelaw ! q) [(p, P p `elem` ps) | (P p) <- addprops])
          | q <- map fst changelaw ]
  actrel     = [(i,rFor i) | i <- agentsOf event] where
    rFor i   = map (map snd) (groupBy ( \ (x,_) (y,_) -> x==y ) (pairs i))
    pairs i = sort $ map (\(set,a) -> (intersect set $ addobs ! i,a)) actlist
  faction    = apply actlist efacts

eventToAction :: Event -> PointedActionModelS5
eventToAction e = (ActMS5 acts actrel, faction) where
  (ActMS5 acts' actrel', faction) = eventToAction' e
  acts      = filter (\(_,(pre,_)) -> pre /= Bot) acts' -- remove actions w/ contradictory
      precon
  actrel    = map (second restrictRel) actrel' where
    restrictRel r  = filter ([]/=) $ map (filter (`elem` map fst acts)) r
```

5 General Kripke Models

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

module SMCDEL.Explicit.K where

import Control.Arrow ((&&&))
import Data.Dynamic
import Data.List (nub,sort,(\\),delete,intercalate,intersect)
import qualified Data.Map.Strict as M
import Data.Map.Strict ((!))
import Data.Maybe (isJust,isNothing)
import Test.QuickCheck

import SMCDEL.Language
import SMCDEL.Explicit.S5 (Action,Bisimulation,HasWorlds,World,worldsOf)
import SMCDEL.Internal.Help (alleqWith,lfp)
import SMCDEL.Internal.TexDisplay
```

In non-S5 Kripke models every agent has an arbitrary relation on the states, not necessarily an equivalence relation.

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.

```
newtype KripkeModel = KrM (M.Map World (M.Map Prp Bool, M.Map Agent [World]))
  deriving (Eq,Ord,Show)

instance Pointed KripkeModel World
type PointedModel = (KripkeModel, World)

instance Pointed KripkeModel [World]
type MultipointedModel = (KripkeModel,[World])

distinctVal :: KripkeModel -> Bool
distinctVal (KrM m) = M.size m == length (nub (map fst (M.elems m)))

instance HasWorlds KripkeModel where
  worldsOf (KrM m) = M.keys m

instance HasVocab KripkeModel where
  vocabOf (KrM m) = M.keys $ fst (head (M.elems m))

instance HasAgents KripkeModel where
  agentsOf (KrM m) = M.keys $ snd (head (M.elems m))

relOfIn :: Agent -> KripkeModel -> M.Map World [World]
relOfIn i (KrM m) = M.map (\x -> snd x ! i) m

truthsInAt :: KripkeModel -> World -> [Prp]
truthsInAt (KrM m) w = M.keys (M.filter id (fst (m ! w)))

instance KripkeLike KripkeModel where
  directed = const True
  getNodes m = map (show . fromEnum &&& labelOf) (worldsOf m) where
    labelOf w = "$" ++ tex (truthsInAt m w) ++ "$"
  getEdges m =
    concat [ [ (a, show $ fromEnum w, show $ fromEnum v) | v <- relOfIn a m ! w ] | w <-
        worldsOf m, a <- agentsOf m ]
  getActuals = const []

instance KripkeLike PointedModel where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals = return . show . fromEnum . snd

instance KripkeLike MultipointedModel where
  directed = directed . fst
  getNodes = getNodes . fst
```

```
  getEdges = getEdges . fst
  getActuals = map (show . fromEnum) . snd

instance TexAble KripkeModel       where
  tex          = tex.ViaDot
  texTo        = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance TexAble PointedModel      where
  tex          = tex.ViaDot
  texTo        = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance TexAble MultipointedModel where
  tex          = tex.ViaDot
  texTo        = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot
```

The following generates random Kripke models.

```
instance Arbitrary KripkeModel where
  arbitrary = do
    nonActualWorlds <- sublistOf [1..8]
    let worlds = 0 : nonActualWorlds
    m <- mapM (\w -> do
      myAssignment <- zip defaultVocabulary <$> infiniteListOf (choose (True,False))
      myRelations <- mapM (\a -> do
        reachables <- sublistOf worlds
        return (a,reachables)
        ) defaultAgents
      return (w, (M.fromList myAssignment, M.fromList myRelations)) -- FIXME avoid fromList
          , build M.map directly?
      ) worlds
    return $ KrM $ M.fromList m
  shrink krm = [ krm `withoutWorld` w | length (worldsOf krm) > 1, w <- delete 0 (worldsOf
      krm) ]

withoutWorld :: KripkeModel -> World -> KripkeModel
withoutWorld (KrM m) w = KrM $ M.map (\(a, r) -> (a, M.map (delete w) r)) $ M.delete w m
```

We now implement the standard Kripke semantics.

```
eval :: PointedModel -> 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 (KrM m,w) (K   i f) = all (\w' -> eval (KrM m,w') f) (snd (m ! w) ! i)
eval (KrM m,w) (Kw  i f) = alleqWith (\w' -> eval (KrM m,w') f) (snd (m ! w) ! i)
eval (m,w) (Ck ags form) = all (\w' -> eval (m,w') form) (groupRel m ags w)
eval (m,w) (Ckw ags form) = alleqWith (\w' -> eval (m,w') form) (groupRel m ags w)
eval (m,w) (PubAnnounce f g) = not (eval (m,w) f) || eval (update (m,w) f) g
eval (m,w) (PubAnnounceW f g) = eval (update m aform, w) g where
  aform | eval (m,w) f = f
        | otherwise      = Neg f
eval (m,w) (Announce listeners f g) = not (eval (m,w) f) || eval newm g where
  newm = (m,w) `update` announceAction (agentsOf m) listeners f
eval (m,w) (AnnounceW listeners f g) = eval newm g where
  newm = (m,w) `update` announceAction (agentsOf m) listeners aform
  aform | eval (m,w) f = f
        | otherwise     = Neg f
eval pm (Dia (Dyn dynLabel d) f) = case fromDynamic d of
```

```
  Just pactm -> eval pm (preOf (pactm :: PointedActionModel)) && eval (pm `update` pactm) f
  Nothing    -> error $ "cannot update Kripke model with '" ++ dynLabel ++ "':\n  " ++ show
      d

instance Semantics PointedModel where
  isTrue = eval

instance Semantics KripkeModel where
  isTrue m = isTrue (m, worldsOf m)

instance Semantics MultipointedModel where
  isTrue (m,ws) f = all (\w -> isTrue (m,w) f) ws

groupRel :: KripkeModel -> [Agent] -> World -> [World]
groupRel (KrM m) ags w = lfp extend (oneStepReachFrom w) where
  oneStepReachFrom x = concat [ snd (m ! x) ! a | a <- ags ]
  extend xs = sort . nub $ xs ++ concatMap oneStepReachFrom xs

instance Update KripkeModel Form where
  checks = [ ] -- unpointed models can always be updated with any formula
  unsafeUpdate (KrM m) f = KrM newm where
    newm = M.mapMaybeWithKey isin m
    isin w' (v,rs) | eval (KrM m,w') f = Just (v, M.map newr rs)
                   | otherwise         = Nothing
    newr = filter (`elem` M.keys newm)

instance Update PointedModel Form where
  unsafeUpdate (m,w) f = (unsafeUpdate m f, w)

instance Update MultipointedModel Form where
  unsafeUpdate (m,ws) f =
    let newm = unsafeUpdate m f in (newm, ws `intersect` worldsOf newm)

announceAction :: [Agent] -> [Agent] -> Form -> PointedActionModel
announceAction everyone listeners f = (ActM am, 1) where
  am = M.fromList
    [ (1, Act { pre = f,   post = M.fromList [], rel = M.fromList $ [(i,[1]) | i <-
        listeners] ++ [(i,[2]) | i <- everyone \\ listeners] } )
    , (2, Act { pre = Top, post = M.fromList [], rel = M.fromList [(i,[2]) | i <- everyone]
        } )
    ]
```

Note that group announcements are implemented using action models which are described below in subsection 5.3.

5.1 Bisimulations and Distinguishing Formulas

The following function checks that a given relation is a bisimulation.

```
checkBisim :: Bisimulation -> KripkeModel -> KripkeModel -> Bool
checkBisim [] _  _  = False
checkBisim z  m1 m2 =
  all (\(w1,w2) ->
        (truthsInAt m1 w1 == truthsInAt m2 w2)  -- same valuation
    && and [ any (\v2 -> (v1,v2) `elem` z) (relOfIn ag m2 ! w2) -- forth
           | ag <- agentsOf m1, v1 <- relOfIn ag m1 ! w1 ]
    && and [ any (\v1 -> (v1,v2) `elem` z) (relOfIn ag m1 ! w1) -- back
           | ag <- agentsOf m2, v2 <- relOfIn ag m2 ! w2 ]
      ) z

checkBisimPointed :: Bisimulation -> PointedModel -> PointedModel -> Bool
checkBisimPointed z (m1,w1) (m2,w2) = (w1,w2) `elem` z && checkBisim z m1 m2
```

The following is an adaptation of Algorithm 1 in [GK16] which given two Kripke models creates a *status map* saying which worlds are bisimilar or can be distinguished.

In a status map `statusMap (w1,w2) == `Nothing`` means that `w1` and `w2` are bisimilar or (during the run of `diff`) the status is not (yet) known. In contrast, `statusMap (w1,w2) == `Just f`` means that formula `f` holds at `w1` but not at `w2`.

The updates to the status map are monotone in the sense that `Nothing` can be changed to `Just f`, but not vice versa. Hence we can use `lfp` to iterate the update until a fixpoint is reached, instead of updating a fixed number of times as done in [GK16]

```
type Status = Maybe Form
type StatusMap = M.Map (World,World) Status

diff :: KripkeModel -> KripkeModel -> StatusMap
diff m1 m2 = lfp step start where
  -- initialize using differences in atomic propositions given by valuation
  start = M.fromList [ ((w1,w2), propDiff (truthsInAt m1 w1) (truthsInAt m2 w2))
                     | w1 <- worldsOf m1, w2 <- worldsOf m2 ]
  propDiff ps qs | ps \\ qs /= []  = Just $       PrpF $ head (ps \\ qs)
                 | qs \\ ps /= []  = Just $ Neg $ PrpF $ head (qs \\ ps)
                 | otherwise       = Nothing
  -- until a fixpoint is reached, update the map using all relations
  step curMap = M.mapWithKey (updateAt curMap) curMap
  updateAt _      _       (Just f) = Just f
  updateAt curMap (w1,w2) Nothing  = case
    -- forth
    [ Neg . K i . Neg . Conj $ [ f | w2' <- w2's, let Just f = curMap ! (w1',w2') ]
    | i <- agentsOf m1
    , let w2's = relOfIn i m2 ! w2
    , w1' <- relOfIn i m1 ! w1
    , all (\w2' -> isJust $ curMap ! (w1',w2')) w2's
    ]
    ++
    -- back
    [ K i . Disj $ [ f | w1' <- w1's, let Just f = curMap ! (w1',w2') ]
    | i <- agentsOf m1
    , let w1's = relOfIn i m1 ! w1
    , w2' <- relOfIn i m2 ! w2
    , all (\w1' -> isJust $ curMap ! (w1',w2')) w1's
    ]
    of
      [] -> Nothing
      (f:_) -> Just f
```

Given two pointed models we can thus either find a bisimulation or a distinguishing formula.

```
diffPointed :: PointedModel -> PointedModel -> Either Bisimulation Form
diffPointed (m1,w1) (m2,w2) =
  case diff m1 m2 ! (w1,w2) of
    Nothing -> Left $ M.keys $ M.filter isNothing (diff m1 m2)
    Just f -> Right f
```

5.2 Minimization of Kripke Models

```
generatedSubmodel :: PointedModel -> PointedModel
generatedSubmodel (KrM m, cur) = (KrM newm, cur) where
  newm = M.mapMaybeWithKey isin m
  isin w' (v,rs) | w' `elem` reachable = Just (v, M.map newr rs)
                 | otherwise           = Nothing
  newr = filter (`elem` M.keys newm)
  reachable = lfp follow [cur] where
    follow xs = sort . nub $ concat [ snd (m ! x) ! a | x <- xs, a <- agentsOf (KrM m) ]
```

5.3 Action Models

We now will now implement *epistemic* and *factual* change. On standard Kripke models this is done with *action models* which contain pre- and postconditions to describe the two sorts of change.

What is the type of postconditions? A function `Prp -> Form` seems natural, however it would not give us a way to check the domain and would always have to be applied to all the propositions — there would be nothing particular about the trivial postcondition `\p -> PrpF p`. To capture the partiality we could use lists of tuples `[(Prp,Form)]`. However, not every such list is a substitution and thus a valid postcondition, for it might contain two tuples with the same left part. Hence we will use the type `Map Prp Form` which captures partial functions.

```haskell
type PostCondition = M.Map Prp Form

data Act = Act {pre :: Form, post :: PostCondition, rel :: M.Map Agent [Action]}
  deriving (Eq,Ord,Show)

-- | Extend ‘post‘ to all propositions
safepost :: Act -> Prp -> Form
safepost ch p | p ‘elem‘ M.keys (post ch) = post ch ! p
              | otherwise = PrpF p

newtype ActionModel = ActM (M.Map Action Act)
  deriving (Eq,Ord,Show)

instance HasAgents ActionModel where
  agentsOf (ActM am) = M.keys $ rel (head (M.elems am))

instance HasPrecondition ActionModel where
  preOf _ = Top

instance Pointed ActionModel Action
type PointedActionModel = (ActionModel, Action)

instance HasPrecondition PointedActionModel where
  preOf (ActM am, actual) = pre (am ! actual)

instance Pointed ActionModel [Action]
type MultipointedActionModel = (ActionModel, [Action])

instance HasPrecondition MultipointedActionModel where
  preOf (ActM am, as) = Disj [ pre (am ! a) | a <- as ]

instance Update KripkeModel ActionModel where
  checks = [haveSameAgents]
  unsafeUpdate m (ActM am) =
    let (newModel,_) = unsafeUpdate (m, worldsOf m) (ActM am, M.keys am) in newModel

instance Update PointedModel PointedActionModel where
  checks = [haveSameAgents,preCheck]
  unsafeUpdate (m, w) (actm, a) =
    let (newModel,[newWorld]) = unsafeUpdate (m, [w]) (actm, [a]) in (newModel,newWorld)

instance Update PointedModel MultipointedActionModel where
  checks = [haveSameAgents,preCheck]
  unsafeUpdate (m, w) mpactm =
    let (newModel,[newWorld]) = unsafeUpdate (m, [w]) mpactm in (newModel,newWorld)

instance Update MultipointedModel PointedActionModel where
  checks = [haveSameAgents] -- do not check precondition!
  unsafeUpdate mpm (actm, a) = unsafeUpdate mpm (actm, [a])

instance Update MultipointedModel MultipointedActionModel where
  checks = [haveSameAgents]
  unsafeUpdate (KrM m, ws) (ActM am, facts) =
    (KrM $ M.fromList (map annotate worldPairs), newActualWorlds) where
      worldPairs = zip (concat [ [ (s, a) | eval (KrM m, s) (pre $ am ! a) ] | s <- M.keys
          m, a <- M.keys am ]) [0..]
      newActualWorlds = [ k | ((w,a),k) <- worldPairs, w ‘elem‘ ws, a ‘elem‘ facts ]
      annotate ((s,a),news) = (news , (newval, M.fromList (map reachFor (agentsOf (KrM m)))
          )) where
        newval = M.mapWithKey applyPC (fst $ m ! s)
        applyPC p oldbit
          | p ‘elem‘ M.keys (post (am ! a)) = eval (KrM m,s) (post (am ! a) ! p)
          | otherwise = oldbit
        reachFor i = (i, [ news’ | ((s’,a’),news’) <- worldPairs, s’ ‘elem‘ snd (m !  s) !
            i, a’ ‘elem‘ rel (am ! a) ! i ])
```

Generate a somewhat random action model with change: We have four actions where one has a trivial and the other random preconditions. All four actions change one randomly selected atomic proposition to a random constant or the value of another randomly selected atomic proposition. Agent 0 can distinguish all events, the other agents have random accessibility relations.

Note that for now we only use boolean preconditions.

```
instance Arbitrary ActionModel where
  arbitrary = do
    let allactions = [0..3]
    BF f <- sized $ randomboolformWith defaultVocabulary
    BF g <- sized $ randomboolformWith defaultVocabulary
    BF h <- sized $ randomboolformWith defaultVocabulary
    let myPres  = Top : map simplify [f,g,h]
    myPosts <- mapM (\_ -> do
      proptochange <- elements defaultVocabulary
      postconcon <- elements $ [Top,Bot] ++ map PrpF defaultVocabulary
      return $ M.fromList [ (proptochange, postconcon) ]
      ) allactions
    myRels <- mapM (\k -> do
      reachList <- sublistOf allactions
      return $ M.fromList $ ("0",[k]) : [(ag,reachList) | ag <- defaultAgents]
      ) allactions
    return $ ActM $ M.fromList
      [ (k::Action, Act (myPres !! k) (myPosts !! k) (myRels !! k)) | k <- allactions ]
  shrink (ActM am) = [ ActM $ removeFromRels k $ M.delete k am | k <- M.keys am, k /= 0 ]
      where
    removeFromRels = M.map . removeFrom where
      removeFrom k c = c { rel = M.map (delete k) (rel c) }
```

Finally, we also provide functions to visualize action models.

```
instance KripkeLike ActionModel where
  directed = const True
  getNodes (ActM am) = map (show &&& labelOf) (M.keys am) where
    labelOf a = concat
      [ "$\\begin{array}{c} ? " , tex (pre (am ! a)) , "\\\\"
      , intercalate "\\\\" (map showPost (M.toList $ post (am ! a)))
      , "\\end{array}$" ]
    showPost (p,f) = tex p ++ " := " ++ tex f
  getEdges (ActM am) =
    concat [ [ (i, show a, show b) | b <- rel (am ! a) ! i ] | a <- M.keys am, i <-
        agentsOf (ActM am) ]
  getActuals = const [ ]

instance TexAble ActionModel where
  tex = tex.ViaDot
  texTo = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike PointedActionModel where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, a) = [show a]

instance TexAble PointedActionModel where
  tex = tex.ViaDot
  texTo = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

instance KripkeLike MultipointedActionModel where
  directed = directed . fst
  getNodes = getNodes . fst
  getEdges = getEdges . fst
  getActuals (_, as) = map show as

instance TexAble MultipointedActionModel where
  tex = tex.ViaDot
  texTo = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot
```

5.4 From S5 to K

In this module we convert S5 models and structures to their more general K equivalents.

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

module SMCDEL.Translations.Convert where

import qualified Data.Map.Strict as M

import SMCDEL.Language (agentsOf)
import SMCDEL.Internal.Help
import SMCDEL.Explicit.K
import SMCDEL.Explicit.S5
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.S5

class Convertable a b where
  convert :: a -> b
```

Mathematically, every S5 Kripke model is already a Kripke model by definition. Still, in our implementation we still need to replace each partition with a proper relation.

```
instance Convertable PointedModelS5 PointedModel where
  convert s5m@(KrMS5 worlds rels vals, cur) = (KrM m, cur) where
    m = M.fromList [ (w,(valFor w, relsFor w)) | w <- worlds ]
    valFor w = M.fromList (vals ! w)
    relsFor w = M.fromList [(i, concat $ filter (elem w) (rels ! i))
                           | i <- agentsOf s5m ]
```

To convert a knowledge structure to a belief structure, we replace each O_i with $\Omega_i := \bigwedge_{p \in O_i}(p \leftrightarrow p')$.

```
instance Convertable KnowScene BelScene where
  convert (KnS voc law obs, s) = (BlS voc law obsLaws, s) where
    obsLaws = M.fromList [ (i, allsamebdd ob) | (i,ob) <- obs ]
```

6 Belief Structures

The implementation in the previous chapters 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. 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 14 has to add additional atomic propositions.

To overcome this limitation, we will generalize the definition of knowledge structures in this chapter. Using well-known methods from temporal model checking, arbitrary relations can also be represented as BDDs. See for example [GR02]. 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 [CGP99, 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.

Our symbolic model checker can then also be used for non-S5 models.

For further explanations, see [Ben+17, Section 8].

```
{-# LANGUAGE FlexibleInstances, TypeOperators, MultiParamTypeClasses, ScopedTypeVariables #
    -}

module SMCDEL.Symbolic.K where

import Data.Tagged

import Control.Arrow ((&&&),first)
import Data.Dynamic (fromDynamic)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List (intercalate,sort,intersect,(\\))
import qualified Data.Map.Strict as M
import Data.Map.Strict ((!))
import Test.QuickCheck

import SMCDEL.Explicit.K
import SMCDEL.Internal.Help (apply,lfp,powerset)
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Other.BDD2Form
import SMCDEL.Symbolic.S5 (State,texBDD,boolBddOf,texBddWith,bddEval,relabelWith)
import SMCDEL.Translations.S5 (booloutof)
```

6.1 Translating relations to type-safe BDDs

To represent relations as BDDs we use the following well-known method from temporal 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 State = [(State,State)] = [([Prp],[Prp])]`, i.e. a relation over states is in fact a relation on sets of propositions. We can therefore represent a relation with the BDD of a characteristic function on a double vocabulary, as described in [CGP99, 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 that the assignment $\{p,p_3,p_2'\}$ satisfies the formula representing R.

In our notation we can just write p' instead of p and p_2' instead of p_2 and so on, bu in the implementation 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. Notably, these functions are 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.

To switch between the normal and the double vocabulary, we use the helper functions `mv`, `cp` and their inverses. Figure 1 gives an overview of what they do.

```
mvP, cpP :: Prp -> Prp
mvP (P n) = P  (2*n)      -- represent p  in the double vocabulary
cpP (P n) = P ((2*n) + 1) -- represent p' in the double vocabulary

unmvcpP :: Prp -> Prp
unmvcpP (P m) | even m    = P $ m `div` 2
              | otherwise = P $ (m-1) `div` 2

mv, cp :: [Prp] -> [Prp]
mv = map mvP
cp = map cpP

unmv, uncp :: [Prp] -> [Prp]
-- | Go from p in double vocabulary to p in single vocabulary.
unmv = map f where
  f (P m) | odd m     = error "unmv failed: Number is odd!"
          | otherwise = P $ m `div` 2
-- | Go from p' in double vocabulary to p in single vocabulary.
uncp = map f where
  f (P m) | even m    = error "uncp failed: Number is even!"
          | otherwise = P $ (m-1) `div` 2
```

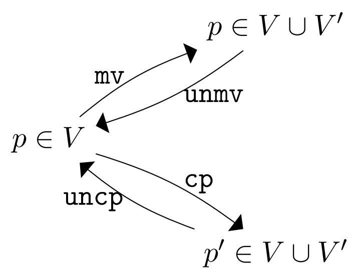


Figure 1: The functions mv, cp, unmv and uncp

The following type `RelBDD` is in fact just a newtype of `Bdd`. Tags (aka labels) from the module Data.Tagged can be used to distinguish objects of the same type which should not be combined or mixed. Making these differences explicit at the type level can rule out certain mistakes already at compile time which otherwise might only be discovered at run time or not at all.

The use case here is to distinguish BDDs for formulas over different vocabularies, i.e. sets of atomic propositions. For example, the BDD of p_1 in the standard vocabulary V uses the variable 1, but in the vocabulary of $V \cup V'$ the proposition p_1 is mapped to variable 3 while p'_1 is mapped to 4. This is implemented in the mv and cp functions above which we are now going to lift to BDDs.

If RelBDD and Bdd were synonyms (as it was the case in a previous version of this file) then it would be up to us to ensure that BDDs meant for different vocabularies would not be combined. Taking the conjunction of the BDD of p in V and the BDD of p_2 in $V \cup V'$ just makes no sense — one BDD first needs to be translated to the vocabulary of the other — but as long as the types match Haskell would happily generate the chaotic conjunction.

To catch these problems at compile time we now distinguish Bdd and RelBDD. In principle this could be done with a simple newtype, but looking ahead we will need even more different vocabularies (for factual change and symbolic bisimulations). It would become tedious to write the same instances of Functor, Applicative and Monad each time we add a new vocabulary. Fortunately, `Data.Tagged` already provides us with an instance of Functor for `Tagged t` for any type `t`.

Also note that `Dubbel` is an empty type, isomorphic to `()`.

```
data Dubbel
type RelBDD = Tagged Dubbel Bdd

totalRelBdd, emptyRelBdd :: RelBDD
totalRelBdd = pure $ boolBddOf Top
emptyRelBdd = pure $ boolBddOf Bot

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

class TagBdd a where
  tagBddEval :: [Prp] -> Tagged a Bdd -> Bool
  tagBddEval truths querybdd = evaluateFun (untag querybdd) (\n -> P n `elem` truths)

instance TagBdd Dubbel
```

Now that `Tagged Dubbel` is an applicative functor, we can lift all the Bdd functions to RelBDD using standard notation. Instead of `con (var 1) (var 3) :: RelBDD` we will now write

`con <$> (pure $ var 1) <*> (pure $ var 3)`.

On the other hand, something like `con <$> (var 1) <*> (pure $ var 3)` would fail and will prevent us from accidentaly mixing up BDDs in different vocabularies.

Now 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. It also changes the type to reflect this change.

```
cpBdd :: Bdd -> RelBDD
cpBdd b = Tagged $ relabelFun (\n -> (2*n) + 1) b
```

And with the unprimed ones in the double:

```
mvBdd :: Bdd -> RelBDD
mvBdd b = Tagged $ relabelFun (2 *) b
```

The next function translates a BDD using unprimed propositions in the double vocabulary to a Bdd representing the same formula in the single vocabulary.

```
unmvBdd :: RelBDD -> Bdd
unmvBdd (Tagged b) =
  relabelFun (\n -> if even n then n `div` 2 else error ("Not even: " ++ show n)) b
```

The double vocabulary is therefore obtained as follows:

```
>>> SMCDEL.Symbolic.K.mv [(P 0)..(P 3)]

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

0.00 seconds
```

```
>>> SMCDEL.Symbolic.K.cp [(P 0)..(P 3)]

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

0.00 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:

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

```
propRel2bdd :: [Prp] -> M.Map State [State] -> RelBDD
propRel2bdd props relation = pure $ disSet (M.elems $ M.mapWithKey linkbdd relation) where
  linkbdd here theres =
    con (booloutof (mv here) (mv props))
        (disSet [ booloutof (cp there) (cp props) | there <- theres ] )
```

The following example is from [GR02, p. 136].

```
samplerel ::  M.Map State [State]
samplerel = M.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.Symbolic.K.propRel2bdd [P 1, P 2] SMCDEL.Symbolic.K.samplerel

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

0.04 seconds
```

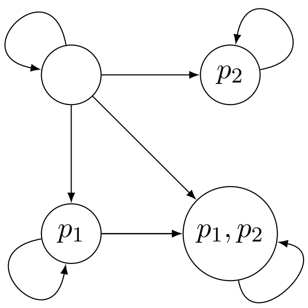


Figure 2: The original graph of `samplerel`.

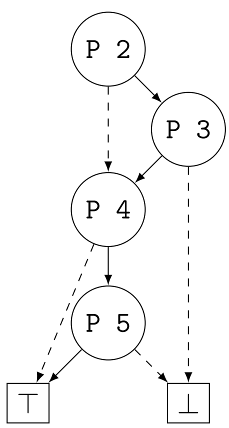


Figure 3: BDD of `samplerel` with double vocabulary labels.

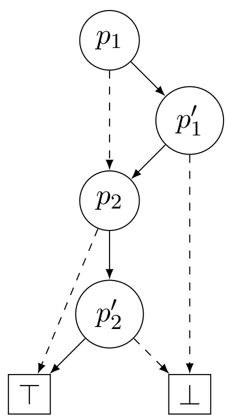


Figure 4: BDD of `samplerel` with translated labels.

Many operations and tests on relations can be done directly on their BDDs, see [Gat18, page 62].

6.2 Describing 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 :: Agent -> KripkeModel -> RelBDD
relBddOfIn i (KrM m)
  | not (distinctVal (KrM m)) = error "m does not have distinct valuations."
  | otherwise = pure $ disSet (M.elems $ M.map linkbdd m) where
    linkbdd (mapPropBool,mapAgentReach)  =
      con
        (booloutof (mv here) (mv props))
        (disSet [ booloutof (cp there) (cp props) | there<-theres ] )
      where
        props = M.keys mapPropBool
        here = M.keys (M.filter id mapPropBool)
        theres = map (truthsInAt (KrM m)) (mapAgentReach ! i)
```

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

6.3 Belief Structures

```
data BelStruct = BlS [Prp]              -- vocabulary
                     Bdd                -- state law
                     (M.Map Agent RelBDD) -- observation laws
                  deriving (Eq,Show)

instance Pointed BelStruct State
type BelScene = (BelStruct,State)

instance Pointed BelStruct Bdd
type MultipointedBelScene = (BelStruct,Bdd)

instance HasVocab BelStruct where
  vocabOf (BlS voc _ _) = voc

instance HasAgents BelStruct where
  agentsOf (BlS _ _ obdds) = M.keys obdds
```

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

```
bddOf :: BelStruct -> Form -> Bdd
bddOf _   Top             = top
bddOf _   Bot             = bot
bddOf _   (PrpF (P n))    = var n
bddOf bls (Neg form)      = neg $ bddOf bls form
bddOf bls (Conj forms)    = conSet $ map (bddOf bls) forms
bddOf bls (Disj forms)    = disSet $ map (bddOf bls) forms
bddOf bls (Xor  forms)    = xorSet $ map (bddOf bls) forms
bddOf bls (Impl f g)      = imp (bddOf bls f) (bddOf bls g)
bddOf bls (Equi f g)      = equ (bddOf bls f) (bddOf bls g)
bddOf bls (Forall ps f) = forallSet (map fromEnum ps) (bddOf bls f)
bddOf bls (Exists ps f) = existsSet (map fromEnum ps) (bddOf bls 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:

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

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

```
bddOf bls@(BlS allprops lawbdd obdds) (K i form) = unmvBdd result where
  result = forallSet ps' <$> (imp <$> cpBdd lawbdd <*> (imp <$> omegai <*> cpBdd (bddOf bls
       form)))
  ps'    = map fromEnum $ cp allprops
  omegai = obdds ! i
```

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

```
bddOf bls@(BlS allprops lawbdd obdds) (Kw i form) = unmvBdd result where
  result = dis <$> part form <*> part (Neg form)
  part f = forallSet ps' <$> (imp <$> cpBdd lawbdd <*> (imp <$> omegai <*> cpBdd (bddOf bls
       f)))
  ps'    = map fromEnum $ cp allprops
  omegai = obdds ! i
```

We can also interpret the epistemic group operator on the general structures as follows. Note that we still write `Ck` and `Ckw` but this should be read as "common belief".

```
bddOf bls@(BlS voc lawbdd obdds) (Ck ags form) = lfp lambda top where
  ps' = map fromEnum $ cp voc
  lambda :: Bdd -> Bdd
  lambda z = unmvBdd $
    forallSet ps' <$>
      (imp <$> cpBdd lawbdd <*>
        (imp <$> (disSet <$> sequence [obdds ! i | i <- ags]) <*>
          cpBdd (con (bddOf bls form) z)))

bddOf bls (Ckw ags form) = dis (bddOf bls (Ck ags form)) (bddOf bls (Ck ags (Neg form)))
```

Public announcements only restrict the lawbdd:

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

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

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

bddOf bls@(BlS props _ _) (AnnounceW ags f g) =
  ifthenelse (bddOf bls f) bdd2a bdd2b where
    bdd2a = restrict (bddOf (announce bls ags f      ) g) (k,True)
    bdd2b = restrict (bddOf (announce bls ags (Neg f)) g) (k,True)
    (P k) = freshp props
```

The following deals with diamonds containing dynamic operators. The implementation here is the same as for S5 in Section 3.

```
bddOf bls (Dia (Dyn dynLabel d) f) =
    con (bddOf bls preCon)                   -- 5. Prefix with "precon AND ..." (diamond!)
    . relabelWith copyrelInverse             -- 4. Copy back changeProps V_° to V_
    . simulateActualEvents                   -- 3. Simulate actual event(s) [see below]
    . substitSimul [ (k, changeLaw ! p)      -- 2. Replace changeProps V_ with postcons
                   | p@(P k) <- changeProps] --    (no "relabelWith copyrel", undone in 4)
    . bddOf (bls `update` trf)               -- 1. boolean equivalent wrt new struct
    $ f
  where
    changeProps = M.keys changeLaw
    copychangeProps = [(freshp $ vocabOf bls ++ addProps)..]
    copyrelInverse  = zip copychangeProps changeProps
    (trf@(Trf addProps addLaw changeLaw _), shiftrel) = shiftPrepare bls trfUnshifted
    (preCon,trfUnshifted,simulateActualEvents) =
      case fromDynamic d of
        -- 3. For a single pointed event, simulate actual event x outof V+
        Just ((t,x) :: Event) -> ( preOf (t,x), t, (`restrictSet` actualAss) )
          where actualAss = [(newK, P k `elem` x) | (P k, P newK) <- shiftrel]
        Nothing -> case fromDynamic d of
          -- 3. For a multipointed event, simulate a set of actual events by ...
          Just ((t,xsBdd) :: MultipointedEvent) ->
              ( preOf (t,xsBdd), t
              , existsSet (map fromEnum addProps)  -- ... replacing addProps with
                  assigments
                . con actualsBdd                   -- ... that satisfy actualsBdd
                . con (bddOf bls addLaw)           -- ... and a precondition.
              ) where actualsBdd = relabelWith shiftrel xsBdd
          Nothing -> error $ "cannot update belief structure with '" ++ dynLabel ++ "':\n
              " ++ show d
```

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 $\theta \to \|\varphi\|_{\mathcal{F}}$ is equivalent=identical to the $\top$ BDD.

```
validViaBdd :: BelStruct -> Form -> Bool
validViaBdd bls@(BlS _ lawbdd _) f = top == imp lawbdd (bddOf bls 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. This usually indicates that the given formula uses propositional variables outside the vocabulary of the given structure.

```
evalViaBdd :: BelScene -> Form -> Bool
evalViaBdd (bls@(BlS allprops _ _),s) f = let
    bdd  = bddOf bls f
    b    = restrictSet bdd list
    list = [ (n, P n `elem` s) | (P n) <- allprops ]
  in
    case (b==top,b==bot) of
      (True,_) -> True
      (_,True) -> False
      _        -> error $ "evalViaBdd failed: Composite BDD leftover!\n"
```

```
        ++ "  bls:  " ++ show bls ++ "\n"
        ++ "  s:    " ++ show s ++ "\n"
        ++ "  form: " ++ show f ++ "\n"
        ++ "  bdd:  " ++ show bdd ++ "\n"
        ++ "  list: " ++ show list ++ "\n"
        ++ "  b:    " ++ show b ++ "\n"

instance Semantics BelStruct where
  isTrue = validViaBdd

instance Semantics BelScene where
  isTrue = evalViaBdd
```

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

```
pubAnnounce :: BelStruct -> Form -> BelStruct
pubAnnounce bls@(BlS allprops lawbdd obs) f =
  BlS allprops (con lawbdd (bddOf bls f)) obs

pubAnnounceOnScn :: BelScene -> Form -> BelScene
pubAnnounceOnScn (bls,s) psi = if evalViaBdd (bls,s) psi
                                 then (pubAnnounce bls psi,s)
                                 else error "Liar!"

announce :: BelStruct -> [Agent] -> Form -> BelStruct
announce bls@(BlS props lawbdd obdds) ags psi = BlS newprops newlawbdd newobdds where
  (P k)     = freshp props
  newprops  = sort $ P k : props
  newlawbdd = con lawbdd (imp (var k) (bddOf bls psi))
  newobdds  = M.mapWithKey newOfor obdds
  newOfor i oi | i `elem` ags = con <$> oi <*> (equ <$> mvBdd (var k) <*> cpBdd (var k))
               | otherwise    = con <$> oi <*> (neg <$> cpBdd (var k)) -- p_psi'
```

```
statesOf :: BelStruct -> [State]
statesOf (BlS allprops lawbdd _) = map (sort.getTrues) prpsats where
  bddvars = map fromEnum allprops
  bddsats = allSatsWith bddvars lawbdd
  prpsats = map (map (first toEnum)) bddsats
  getTrues = map fst . filter snd
```

Visualizing Belief Structures:

```
texRelBDD :: RelBDD -> String
texRelBDD (Tagged b) = texBddWith texRelProp b where
  texRelProp n
    | even n    = show (n `div` 2)
    | otherwise = show ((n - 1) `div` 2) ++ "'"

bddprefix, bddsuffix :: String
bddprefix = "\\begin{array}{l} \\scalebox{0.3}{"
bddsuffix = "} \\end{array} \n"

instance TexAble BelStruct where
  tex (BlS props lawbdd obdds) = concat
    [ " \\left( \n"
    , tex props, ", "
    , bddprefix, texBDD lawbdd, bddsuffix
    , ", "
    , intercalate ", " obddstrings
    , " \\right) \n"
    ] where
      obddstrings = map (bddstring . (fst &&& (texRelBDD . snd))) (M.toList obdds)
      bddstring (i,os) = "\\Omega_{\\text{" ++ i ++ "}} = " ++ bddprefix ++ os ++
        bddsuffix

instance TexAble BelScene where
  tex (bls, state) = concat
    [ " \\left( \n", tex bls, ", ", tex state, " \\right) \n" ]
```

```
instance TexAble MultipointedBelScene where
  tex (bls, statesBdd) = concat
    [ " \\left( \n"
    , tex bls ++ ", "
    , " \\begin{array}{l} \\scalebox{0.4}{"
    , texBDD statesBdd
    , "} \\end{array}\n "
    , " \\right)" ]
```

6.4 Minimization of Belief Structures

We can restrict the observational laws with the state law without loosing any information, obtaining an equivalent belief structure.

```
cleanupObsLaw :: BelScene -> BelScene
cleanupObsLaw (BlS vocab law obs, s) = (BlS vocab law (M.map clean obs), s) where
  clean relbdd = restrictLaw <$> relbdd <*> (con <$> cpBdd law <*> mvBdd law)
```

To reduce the vocabulary, it is relevant which part is unused. For example, some variables might be determined by the state law and others might not occur in the observation laws.

```
determinedVocabOf :: BelStruct -> [Prp]
determinedVocabOf strct = filter (\p -> validViaBdd strct (PrpF p) || validViaBdd strct (
    Neg $ PrpF p)) (vocabOf strct)

nonobsVocabOf  :: BelStruct -> [Prp]
nonobsVocabOf (BlS vocab _law obs) = filter (`notElem` usedVars) vocab where
  usedVars =
    map unmvcpP
    $ sort
    $ concatMap (map P . Data.HasCacBDD.allVarsOf . untag . snd)
    $ M.toList obs
```

```
withoutProps :: [Prp] -> BelStruct -> BelStruct
withoutProps propsToDel (BlS oldProps oldLawBdd oldObs) =
  BlS
    (oldProps \\ propsToDel)
    (existsSet (map fromEnum propsToDel) oldLawBdd)
    (M.map (fmap $ existsSet (map fromEnum propsToDel)) oldObs)
```

6.5 Random Belief Structures

```
instance Arbitrary BelStruct where
  arbitrary = do
    numExtraVars <- choose (0,3)
    let myVocabulary = defaultVocabulary ++ take numExtraVars [freshp defaultVocabulary ..]
    (BF statelaw) <- sized (randomboolformWith myVocabulary) `suchThat` (\(BF bf) ->
        boolBddOf bf /= bot)
    obs <- mapM (\i -> do
      BF obsLaw <- sized $ randomboolformWith (sort $ mv myVocabulary ++ cp myVocabulary)
          -- FIXME should rather be a random BDD?
      return (i,pure $ boolBddOf obsLaw)
      ) defaultAgents
    return $ BlS myVocabulary (boolBddOf statelaw) (M.fromList obs)
  shrink bls = [ withoutProps [p] bls | length (vocabOf bls) > 1, p <- vocabOf bls \\
      defaultVocabulary ]
```

6.6 Symbolic Bisimulations

Definition 20. *Suppose we have two structures* $\mathcal{F}_1 = (V, \theta, \Omega_1, \ldots, \Omega_n)$ *and* $\mathcal{F}_2 = (V, \theta, \Omega_1, \ldots, \Omega_n)$.

A boolean formula $\beta \in \mathcal{L}(V \cup V^*)$ *where* $V = V_1 \cap V_2$ *is a* symbolic bisimulation *iff:*

- $\beta \to \bigwedge_{p\in V}(p \leftrightarrow p^*)$ *is a tautology (i.e. its BDD is equal to* $\top$*)*
- *Take any states* s_1 *of* F_1 *and* s_2 *of* F_2 *such that* $s_1 \cup (s_2^*) \vDash \beta$*, any agents* i *and any state* t_1 *of* F_1 *such that* $s_1 \cup t_1' \vDash \Omega_1^i$ *in* F_1*. Then there is a state* t_2 *of* F_2 *such that* $t_1 \cup (t_2^*) \vDash \beta$ *and* $s_2 \cup t_2' \vDash \Omega_2^i$ *in* F_2*.*
- *Vice versa.*

Again, this can also be expressed as a boolean formula. However, we need four copies of variables now. Note that the standard definition of bisimulation can also be translated to first order logic with four variables. In fact, three variables are enough and we could also overwrite V *instead of using* $V^{*\prime}$ *but this will not improve performance.*

Condition (ii):

$$\forall (V \cup V^*) : \beta \to \bigwedge_i \left(\forall V' : \Omega_i^1 \to \exists V^{*\prime} : \beta' \wedge (\Omega_i^2)^*\right)$$

6.7 Transformers

```
data Transformer = Trf
  [Prp] -- addprops
  Form  -- event law
  (M.Map Prp Bdd) -- changelaw
  (M.Map Agent RelBDD) -- eventObs
  deriving (Eq,Show)

instance HasAgents Transformer where
  agentsOf (Trf _ _ _ obdds) = M.keys obdds

instance HasPrecondition Transformer where
  preOf _ = Top

instance Pointed Transformer State
type Event = (Transformer,State)

instance HasPrecondition Event where
  preOf (Trf addprops addlaw _ _, x) = simplify $ substitOutOf x addprops addlaw

instance Pointed Transformer [State]
type MultipointedEvent = (Transformer,Bdd)

instance HasPrecondition MultipointedEvent where
  preOf (Trf addprops addlaw _ _, xsBdd) =
    simplify $ Exists addprops (Conj [ formOf xsBdd, addlaw ])

instance TexAble Transformer where
  tex (Trf addprops addlaw changelaw eventObs) = concat
    [ " \\left( \n"
    , tex addprops, ", "
    , tex addlaw, ", "
    , tex changeprops, ", "
    , intercalate ", " $ map snd . M.toList $ M.mapWithKey texChange changelaw, ", "
    , intercalate ", " eobddstrings
    , " \\right) \n"
    ] where
        changeprops = M.keys changelaw
        texChange prop changebdd = tex prop ++ " := " ++ tex (formOf changebdd)
        eobddstrings = map (bddstring . (fst &&& (texRelBDD . snd))) (M.toList eventObs)
        bddstring (i,os) = "\\Omega^+_{\\text{" ++ i ++ "}} = " ++ bddprefix ++ os ++
            bddsuffix

instance TexAble Event where
  tex (trf, eventFacts) = concat
    [ " \\left( \n", tex trf, ", ", tex eventFacts, " \\right) \n" ]

instance TexAble MultipointedEvent where
```

```
  tex (trf, eventStates) = concat
    [ " \\left( \n"
    , tex trf ++ ", \\ "
    , " \\begin{array}{l} \\scalebox{0.4}{"
    , texBDD eventStates
    , "} \\end{array}\n "
    , " \\right)" ]
```

```
-- | shift addprops to ensure that props and newprops are disjoint:
shiftPrepare :: BelStruct -> Transformer -> (Transformer, [(Prp,Prp)])
shiftPrepare (BlS props _ _) (Trf addprops addlaw changelaw eventObs) =
  (Trf shiftaddprops addlawShifted changelawShifted eventObsShifted, shiftrel) where
    shiftrel = sort $ zip addprops [(freshp props)..]
    shiftaddprops = map snd shiftrel
    -- apply the shifting to addlaw, changelaw and eventObs:
    addlawShifted    = replPsInF shiftrel addlaw
    changelawShifted = M.map (relabelWith shiftrel) changelaw
    -- to shift addObs we need shiftrel in the double vocabulary:
    shiftrelMVCP = sort $ zip (mv addprops) (mv shiftaddprops)
                       ++ zip (cp addprops) (cp shiftaddprops)
    eventObsShifted  = M.map (fmap $ relabelWith shiftrelMVCP) eventObs

instance Update BelScene Event where
  unsafeUpdate (bls@(BlS props law obdds),s) (trf, eventFactsUnshifted) = (BlS newprops
      newlaw newobs, news) where
    -- PART 1: SHIFTING addprops to ensure props and newprops are disjoint
    (Trf addprops addlaw changelaw addObs, shiftrel) = shiftPrepare bls trf
    -- the actual event:
    eventFacts = map (apply shiftrel) eventFactsUnshifted
    -- PART 2: COPYING the modified propositions
    changeprops = M.keys changelaw
    copyrel = zip changeprops [(freshp $ props ++ addprops)..]
    copychangeprops = map snd copyrel
    copyrelMVCP = sort $ zip (mv changeprops) (mv copychangeprops)
                      ++ zip (cp changeprops) (cp copychangeprops)
    -- PART 3: actual transformation
    newprops = sort $ props ++ addprops ++ copychangeprops
    newlaw = conSet $ relabelWith copyrel (con law (bddOf bls addlaw))
                    : [var (fromEnum q) `equ` relabelWith copyrel (changelaw ! q) | q <-
                      changeprops]
    newobs = M.mapWithKey (\i oldobs -> con <$> (relabelWith copyrelMVCP <$> oldobs) <*> (
        addObs ! i)) obdds
    news = sort $ concat
            [ s \\ changeprops
            , map (apply copyrel) $ s `intersect` changeprops
            , eventFacts
            , filter (\ p -> bddEval (s ++ eventFacts) (changelaw ! p)) changeprops ]
```

Using laziness we can also define the pointless update of a structure with a transformer.

```
instance Update BelStruct Transformer where
  checks = [haveSameAgents]
  unsafeUpdate bls ctrf = BlS newprops newlaw newobs where
    (BlS newprops newlaw newobs, _) = unsafeUpdate (bls,undefined::State) (ctrf,undefined::
        State) -- using laziness!
```

We also define a multipointed version of this update. In a `MultipointedEvent` the set of possible events is encoded by a BDD σ over V^+. An event is allowed by σ iff $x \vDash \sigma$. Which of the events actually happens is then decided by evaluating the event law: x is possible to happen at $\mathcal{F}, s$ iff $\mathcal{F}, s \vDash [x \sqsubseteq V^+]\theta^+$. Putting both together we get equivalent to the boolean condition $x \vDash \sigma \wedge [s \sqsubseteq V]\|\theta^+\|_{\{}$.

To update a belief scene with a multipointed event the event law has to make the events mutually exclusive. Only then can we return again a belief scene with exactly one actual state. This is analogous to updating a singlepointed Kripke model with a multi-pointed action model. Also there the preconditions need to be mutually exclusive to get a single-pointed model again.

```
instance Update BelScene MultipointedEvent where
  unsafeUpdate (bls,s) (trfUnshifted, eventFactsBddUnshifted) =
```

```
    update (bls,s) (trf,selectedEventState) where
      (trf@(Trf addprops addlaw _ _), shiftRel) = shiftPrepare bls trfUnshifted
      eventFactsBdd = relabelWith shiftRel eventFactsBddUnshifted
      selectedEventState :: State
      selectedEventState = map (P . fst) $ filter snd selectedEvent
      selectedEvent = case
                       allSatsWith
                         (map fromEnum addprops)
                         (eventFactsBdd `con` restrictSet (bddOf bls addlaw) [ (k, P k `
                             elem` s) | P k <- vocabOf bls ])
                     of
                       []     -> error "no selected event"
                       [this] -> this
                       more   -> error $ "too many selected events: " ++ show more
```

6.8 Reduction Axioms for Transformers

```
trfPost :: Event -> Prp -> Bdd
trfPost (Trf addprops _ changelaw _, x) p
  | p `elem` M.keys changelaw = restrictLaw (changelaw ! p) (booloutof x addprops)
  | otherwise                 = boolBddOf $ PrpF p

reduce :: Event -> Form -> Maybe Form
reduce _ Top           = Just Top
reduce e Bot           = Just $ Neg $ preOf e
reduce e (PrpF p)      = Impl (preOf e) <$> Just (formOf $ trfPost e p)
reduce e (Neg f)       = Impl (preOf e) <$> (Neg <$> reduce e f)
reduce e (Conj fs)     = Conj <$> mapM (reduce e) fs
reduce e (Disj fs)     = Disj <$> mapM (reduce e) fs
reduce e (Xor fs)      = Impl (preOf e) <$> (Xor <$> mapM (reduce e) fs)
reduce e (Impl f1 f2) = Impl <$> reduce e f1 <*> reduce e f2
reduce e (Equi f1 f2) = Equi <$> reduce e f1 <*> reduce e f2
reduce _ (Forall _ _) = Nothing
reduce _ (Exists _ _) = Nothing
reduce e@(t@(Trf addprops _ _ eventObs), x) (K a f) =
  Impl (preOf e) <$> (Conj <$> sequence
    [ K a <$> reduce (t,y) f | y <- powerset addprops -- FIXME is this a bit much?
                             , tagBddEval (mv x ++ cp y) (eventObs ! a)
    ])
reduce e (Kw a f)      = reduce e (Disj [K a f, K a (Neg f)])
reduce _ Ck  {}        = Nothing
reduce _ Ckw {}        = Nothing
reduce _ PubAnnounce  {} = Nothing
reduce _ PubAnnounceW {} = Nothing
reduce _ Announce     {} = Nothing
reduce _ AnnounceW    {} = Nothing
reduce _ Dia          {} = Nothing
```

We also define the following boolean translation for formulas prefixed with a dynamic operator containing a transformer. In `evalViaBddReduce` we then use this translation to evaluate such formulas symbolically:

$$\|[\mathcal{X}, x]\varphi\|_{\mathcal{F}} := \|[x \sqsubseteq V^+]\theta^+\|_{\mathcal{F}} \to [V_-^\circ \mapsto V_-][x \sqsubseteq V^+][V_- \mapsto \theta_-(V_-)]\|\varphi\|_{\mathcal{F}\times\mathcal{X}}$$

```
bddReduce :: BelScene -> Event -> Form -> Bdd
bddReduce scn@(oldBls,_) event@(Trf addprops _ changelaw _, eventFacts) f =
  let
    changeprops = M.keys changelaw
    -- same as in 'transform', to ensure props and addprops are disjoint
    shiftaddprops = [(freshp $ vocabOf scn)..]
    shiftrel      = sort $ zip addprops shiftaddprops
    -- apply the shifting to addlaw and changelaw:
    changelawShifted = M.map (relabelWith shiftrel) changelaw
    (newBlS,_) = update scn event
    -- the actual event, shifted
    actualAss  = [ (shifted, P orig `elem` eventFacts) | (P orig, P shifted) <- shiftrel ]
    postconrel = [ (n, changelawShifted ! P n) | (P n) <- changeprops ]
```

```
    -- reversing V° to V
    copychangeprops = [(freshp $ vocabOf scn ++ map snd shiftrel)..]
    copyrelInverse  = zip copychangeprops changeprops
  in
    imp (bddOf oldBls (preOf event)) $ -- 0. check if precondition holds
      relabelWith copyrelInverse $      -- 4. changepropscopies -> original changeprops
        (`restrictSet` actualAss) $     -- 3. restrict to actual event x outof V+
          substitSimul postconrel $     -- 2. replace changeprops with postconditions
            bddOf newBlS f              -- 1. boolean equivalent wrt new structure

evalViaBddReduce :: BelScene -> Event -> Form -> Bool
evalViaBddReduce (bls,s) event f = evaluateFun (bddReduce (bls,s) event f) (\n -> P n `elem
    ` s)
```

Note that in step 2 above we do a simultaneous substitution with ‘substitSimul‘ from HasCacBDD.

7 Connecting General Kripke Models and Belief Structures

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

module SMCDEL.Translations.K where

import Data.HasCacBDD hiding (Top,Bot)
import Data.List ((\\),elemIndex,nub,sort)
import Data.Map.Strict ((!))
import qualified Data.Map.Strict as M

import SMCDEL.Language
import SMCDEL.Explicit.S5 (worldsOf)
import SMCDEL.Explicit.K
import SMCDEL.Internal.Help (apply,powerset,groupSortWith)
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Translations.S5 (booloutof)
import SMCDEL.Other.BDD2Form
```

7.1 From Belief Structures to Kripke Models

This is the easy direction.

```
blsToKripke :: BelScene -> PointedModel
blsToKripke (f@(BlS _ _ obdds), curs) = (m, cur) where
  links = zip (statesOf f) [0..]
  m = KrM $ M.fromList
    [ (w, ( M.fromList [(p, p `elem` s) | p <- vocabOf f]
          , M.fromList [(a, map (apply links) $ reachFromFor s a) | a <- agentsOf f] ) )
    | (s,w) <- links ]
  reachFromFor s a = filter (\t -> tagBddEval (mv s ++ cp t) (obdds ! a)) (statesOf f)
  (Just cur) = lookup curs links
```

7.2 From Kripke Models to Belief Structures

Assuming we already have distinct valuations!

```
kripkeToBls :: PointedModel -> BelScene
kripkeToBls pm@(m,_) | distinctVal m = kripkeToBlsUnsafe pm
                     | otherwise     = kripkeToBlsUnsafe (ensureDistinctVal pm)

kripkeToBlsUnsafe :: PointedModel -> BelScene
kripkeToBlsUnsafe (m, cur) = (BlS vocab lawbdd obdds, truthsInAt m cur) where
  vocab  = vocabOf m
  lawbdd = disSet [ booloutof (truthsInAt m w) vocab | w <- worldsOf m ]
  obdds  :: M.Map Agent RelBDD
  obdds  = M.fromList [ (i, restrictLaw <$> relBddOfIn i m <*> (con <$> mvBdd lawbdd <*>
      cpBdd lawbdd)) | i <- agents ]
  agents = agentsOf m
```

If valuations are not unique, we need to add propositions. This can be done in different ways, leading to different numbers of propositions to be added. In the method below, if there maximally k many worlds with the same valuation, then we add $\log_2 k$ many new atomic propositions. This is optimal in the sense that less propositions will not be enough to distinguish all worlds. However, this also includes bisimilar worlds which we would not want to be distignuished anyway. Hence the input model should first be minimized and then converted. Alternatively, the output structure can be optimized after the conversion.

```
ensureDistinctVal :: PointedModel -> PointedModel
ensureDistinctVal (krm@(KrM m), cur) = if distinctVal krm then (krm,cur) else (KrM newM,cur
    ) where
  sameVals = groupSortWith (truthsInAt krm) (worldsOf krm)
```

```
  indexOf w = let Just k = elemIndex w (head $ filter (elem w) sameVals) in k
  numAddProps = ceiling $ logBase (2::Double) (fromIntegral $ maximum (map length sameVals)
       + 1)
  addProps = take numAddProps [freshp (vocabOf krm) ..]
  addValForIndex k = M.fromList [ (p, p `elem` (reverse (powerset addProps) !! k) ) | p <-
      addProps ]
  newM = M.mapWithKey (\w (val,r) -> (M.union val (addValForIndex (indexOf w)),r)) m
```

7.3 From Action Models to Transformers

This generalizes `actionToEvent` from Section 4. Note that we don't need extra propositions for the action relation any longer.

```
actionToEvent :: PointedActionModel -> Event
actionToEvent (ActM am, faction) = (Trf addprops addlaw changelaw eventObs, efacts) where
  actions       = M.keys am
  (P fstnewp)   = freshp $ concatMap -- avoid props in pre and postconditions
                  (\c -> propsInForms (pre c : M.elems (post c)) ++ M.keys (post c)) (M.
                      elems am)
  addprops      = [P fstnewp..P maxactprop] -- new props to distinguish all actions
  maxactprop    = fstnewp + ceiling (logBase 2 (fromIntegral $ length actions) :: Float) - 1
  ell           = apply $ zip actions (powerset addprops) -- injectively label actions with
      sets of propositions
  addlaw        = simplify $ Disj [ Conj [ booloutofForm (ell a) addprops, pre $ am ! a ] |
      a <- actions ]
  changeprops   = sort $ nub $ concatMap M.keys . M.elems $ M.map post am -- propositions to
       be changed
  changelaw     = M.fromList [ (p, changeFor p) | p <- changeprops ] -- encode
      postconditions
  changeFor p   = disSet [ booloutof (ell k) addprops `con` boolBddOf (safepost (am ! k) p)
      | k <- actions ]
  eventObs      = M.fromList [ (i, obsLawFor i) | i <- agentsOf (ActM am) ]
  obsLawFor i   = pure $ disSet (M.elems $ M.mapWithKey (link i) am)
  link i k ch   = booloutof (mv $ ell k) (mv addprops) `con` -- encode relations
                  disSet [ booloutof (cp $ ell there) (cp addprops) | there <- rel ch ! i ]
  efacts        = ell faction
```

7.4 From Transformers to Action Models

```
eventToAction :: Event -> PointedActionModel
eventToAction (t@(Trf addprops addlaw changelaw eventObs), efacts) = (ActM am, faction)
    where
  actlist     = zip (powerset addprops) [0..]
  am          = M.fromList [ (a, Act (preFor ps) (postFor ps) (relFor ps)) | (ps,a) <-
      actlist, preFor ps /= Bot ]
  preFor ps   = simplify $ substitSet (zip ps (repeat Top) ++ zip (addprops \\ ps) (repeat
      Bot)) addlaw
  postFor ps = M.fromList [ (q, formOf $ (changelaw ! q) `restrictSet` [(p, P p `elem` ps)
      | (P p) <- addprops]) | q <- M.keys changelaw ]
  relFor ps   = M.fromList [(i,rFor i) | i <- agentsOf t] where
    rFor i    = concatMap (\(qs,b) -> [ b | tagBddEval (mv ps ++ cp qs) (eventObs ! i),
        preFor qs /= Bot ]) actlist
  faction     = apply actlist efacts
```

Note that with `preFor ps /= Bot` we already filter out actions that will never happen.

8 Automated Testing

8.1 Translation tests in S5

In this section 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 (main) where

import Data.Dynamic (toDyn)
import Data.List (sort)
import Test.Hspec
import Test.Hspec.QuickCheck

import SMCDEL.Internal.Help (alleq)
import SMCDEL.Language
import SMCDEL.Symbolic.S5 as Sym
import SMCDEL.Explicit.S5 as Exp
import SMCDEL.Translations.S5
import SMCDEL.Examples
import SMCDEL.Internal.TaggedBDD

main :: IO ()
main = hspec $
  describe "SMCDEL.Translations" $ do
    prop "semantic equivalence"     semanticEquivTest
    prop "semantic validity"        semanticValidTest
    prop "lemma equivalence Kripke" lemmaEquivTestKr
    prop "lemma equivalence KnS"    lemmaEquivTestKnS
    prop "number of states"         numOfStatesTest
    prop "public announcement"      pubAnnounceTest
    prop "group announcement"       (\sf gl sg  -> alleq $ announceTest sf gl sg)
    prop "single action"            (\am f -> alleq $ singleActionTest am f)
    prop "propulations"             propulationTest
```

8.1.1 Semantic Equivalence

The following creates a Kripke model and a knowledge structure which are equivalent to each other by Lemma 11. In this model/structure Alice knows everything and the other agents do not know anything. We then check for a given formula whether the implementations of the different semantics and translation methods agree on whether the formula holds on the model or the structure.

```
mymodel :: PointedModelS5
mymodel = (KrMS5 ws rel val, 0) where
  buildTable partrows p = [ (p,v):pr | v <- [True,False], pr <- partrows ]
  table = foldl buildTable [[]] [P 0 .. P 4]
  val   = zip [0..] (map sort table)
  ws    = map fst val
  rel   = ("0", map (:[]) ws) : [ (show i,[ws]) | i <- [1..5::Int] ]

myscn :: KnowScene
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 = alleq
  [ Exp.eval mymodel f                      -- evaluate directly on Kripke
  , Sym.eval myscn (simplify f)             -- evaluate directly on KNS (slow!)
  , Sym.evalViaBdd myscn f                  -- evaluate equivalent BDD on KNS
  , Exp.eval (knsToKripke myscn) f          -- evaluate on corresponding Kripke
  , Sym.evalViaBdd (kripkeToKns mymodel) f  -- evaluate on corresponding KNS
  ]

semanticValidTest :: Form -> Bool
```

```haskell
semanticValidTest f = alleq
  [ Exp.valid (fst mymodel) f                           -- evaluate directly on Kripke
  , Sym.validViaBdd (fst myscn) f                       -- evaluate equivalent BDD on KNS
  , Exp.valid (fst $ knsToKripke myscn) f               -- evaluate on corresponding Kripke
  , Sym.validViaBdd (fst $ kripkeToKns mymodel) f   -- evaluate on corresponding KNS
  , Sym.whereViaBdd (fst $ kripkeToKns mymodel) f == Sym.statesOf (fst $ kripkeToKns
     mymodel)
  ]
```

Given a Kripke model, we check the knowledge structure obtained using Definition 14: The number of states should be the same as the number of worlds in an equivalent Kripke model and they should be equivalent according to Lemma 11.

```haskell
numOfStatesTest :: KripkeModelS5 -> Bool
numOfStatesTest m@(KrMS5 oldws _ _) = numberOfStates kns == length news where
  scn@(kns, _) = kripkeToKns (m, head oldws)
  (KrMS5 news _ _, _) = knsToKripke scn

lemmaEquivTestKr :: KripkeModelS5 -> Bool
lemmaEquivTestKr m@(KrMS5 ws _ _) = equivalentWith pm kns g where
  pm = (m, head ws)
  (kns,g) = kripkeToKnsWithG pm

lemmaEquivTestKnS :: KnowStruct -> Bool
lemmaEquivTestKnS kns = equivalentWith pm scn g where
  scn = (kns, head $ statesOf kns)
  (pm,g) = knsToKripkeWithG scn
```

8.1.2 Public and Group Announcements

We can do public announcements in various ways as described in Section 10.1.3. The following tests check that the results of all methods are the same.

```haskell
pubAnnounceTest :: Prp -> SimplifiedForm -> Bool
pubAnnounceTest prp (SF g) = alleq
  [ Exp.eval mymodel (PubAnnounce f g)
  , Sym.eval (kripkeToKns mymodel) (PubAnnounce f g)
  , Sym.evalViaBdd (kripkeToKns mymodel) (PubAnnounce f g)
  , Sym.eval (update (kripkeToKns mymodel) (actionToEvent (pubAnnounceAction (agentsOf
     mymodel) f))) g
  , Exp.eval mymodel (Dia (Dyn dynName (toDyn $ pubAnnounceAction (agentsOf mymodel) f)) g)
  , Sym.eval (kripkeToKns mymodel) (Dia (Dyn dynName (toDyn $ actionToEvent $
     pubAnnounceAction (agentsOf mymodel) f)) g)
  , Sym.evalViaBdd (kripkeToKns mymodel) (Dia (Dyn dynName (toDyn $ actionToEvent $
     pubAnnounceAction (agentsOf mymodel) f)) g)
  ] where
     f = PrpF prp
     dynName = "publicly announce " ++ show prp

announceTest :: SimplifiedForm -> Group -> SimplifiedForm -> [Bool]
announceTest (SF f) (Group listeners) (SF g) =
  [ Exp.eval mymodel (Announce listeners f g) -- directly on Kripke
  , let -- apply action model to Kripke
     precon   = Exp.eval mymodel f
     action   = groupAnnounceAction (agentsOf mymodel) listeners f
     newModel = update mymodel action
    in not precon || Exp.eval newModel g
  , Exp.eval mymodel (box (Dyn ("announce " ++ show f ++ " to " ++ show listeners) (toDyn $
      groupAnnounceAction (agentsOf mymodel) listeners f)) g)
  , Sym.eval (kripkeToKns mymodel) (Announce listeners f g) -- on equivalent kns
  , Sym.evalViaBdd (kripkeToKns mymodel) (Announce listeners f g) -- BDD on equivalent kns
  , let -- apply equivalent transformer to equivalent kns
     precon  = Sym.eval (kripkeToKns mymodel) f
     equiTrf = actionToEvent (groupAnnounceAction (agentsOf mymodel) listeners f)
     newKns  = update (kripkeToKns mymodel) equiTrf
    in not precon || Sym.eval newKns g
  ]
```

8.1.3 Random Action Models

The Arbitrary instance for action models in Section 2 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.

```
singleActionTest :: ActionModelS5 -> Form -> [Bool]
singleActionTest myact f = [a,b,c,d] where
  a = Exp.eval (update mymodel (myact,0::Action)) f
  b = Sym.evalViaBdd (update (kripkeToKns mymodel) (actionToEvent (myact,0::Action))) f
  c = Exp.eval (update mymodel (eventToAction (actionToEvent (myact,0::Action)))) f
  d = case reduce (actionToEvent (myact,0::Action)) f of
    Nothing -> c
    Just g  -> Sym.evalViaBdd (kripkeToKns mymodel) g
```

8.2 Bisimulations

```
propulationTest :: KripkeModelS5 -> Bool
propulationTest m = checkPropu (allsamebdd (vocabOf kns1)) (fst kns1) (fst kns2) (vocabOf
    kns1) where
  kns1 = kripkeToKns (m,head $ worldsOf m)
  kns2 = kripkeToKns (knsToKripke kns1)
```

8.3 Examples

This module uses Hspec and QuickCheck to easily check some properties of our implementations. For example, we check that simplification of formulas does not change their meaning and we replicate some of the results listed in the module `SMCDEL.Examples` from Section 10.1.

```
module Main (main) where

import Data.List
import Test.Hspec
import Test.Hspec.QuickCheck
import Test.QuickCheck (expectFailure,(===))

import SMCDEL.Examples
import SMCDEL.Examples.DiningCrypto
import SMCDEL.Examples.DrinkLogic
import SMCDEL.Examples.MuddyChildren
import SMCDEL.Examples.DoorMat
import SMCDEL.Examples.Prisoners
import SMCDEL.Examples.RussianCards
import SMCDEL.Examples.SumAndProduct
import SMCDEL.Examples.WhatSum
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Other.BDD2Form
import SMCDEL.Other.Planning
import SMCDEL.Symbolic.S5
import SMCDEL.Translations.S5
import qualified SMCDEL.Explicit.S5 as Exp
import qualified SMCDEL.Symbolic.K as SymK
import qualified SMCDEL.Explicit.K as ExpK

main :: IO ()
main = hspec $ do
  describe "SMCDEL.Language" $ do
    prop "freshp returns a fresh proposition" $
      \props -> freshp props `notElem` props
    prop "simplifying a boolean formula yields something equivalent" $
      \(BF bf) -> boolBddOf bf === boolBddOf (simplify bf)
    prop "simplifying a boolean formula only removes propositions" $
      \(BF bf) -> all (`elem` propsInForm bf) (propsInForm (simplify bf))
```

```
    prop "list of subformulas is already nubbed" $
      \f -> nub (subformulas f) === subformulas f
    prop "formulas are identical iff their show strings are" $
      \f g -> ((f::Form) == g) === (show f == show g)
    prop "boolean formulas with same prettyprint are equivalent" $
      \(BF bf) (BF bg) -> (ppForm bf /= ppForm bg) || boolBddOf bf == boolBddOf bg
    prop "boolean formulas with same LaTeX are equivalent" $
      \(BF bf) (BF bg) -> (tex bf /= tex bg) || boolBddOf bf == boolBddOf bg
    it "we can LaTeX the testForm" $ tex testForm === intercalate "\n"
        [ " \\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} \\} ) " ]
    it "svgViaTex works for modelA" $
        isInfixOf "stroke-linecap:butt" (svgViaTex modelA)
    prop "svgViaTex can yield strings of different length" $
      expectFailure (\m1 m2 ->
            length (svgViaTex (m1::Exp.KripkeModelS5,0::Exp.World))
        === length (svgViaTex (m2::Exp.KripkeModelS5,0::Exp.World)) )
  describe "SMCDEL.Symbolic.S5" $
    prop "boolEvalViaBdd agrees on simplified formulas" $
      \(BF bf) props -> let truths = nub props in
        boolEvalViaBdd truths bf === boolEvalViaBdd truths (simplify bf)
  describe "SMCDEL.Other.BDD2Form" $ do
    prop "boolBddOf . formOf == id" $
      \b -> b === boolBddOf (formOf b)
    prop "boolBddOf (Equi bf (formOf (boolBddOf bf))) == boolBddOf Top" $
      \(BF bf) -> boolBddOf (Equi bf (formOf (boolBddOf bf))) === boolBddOf Top
  describe "SMCDEL.Explicit.S5" $ do
    prop "generatedSubmodel preserves truth" $
      \m f -> let pm = (m::Exp.KripkeModelS5, head $ Exp.worldsOf m)
              in isTrue pm f === isTrue (Exp.generatedSubmodel pm) f
    prop "optimize preserves truth" $
      \m f -> let pm = (m::Exp.KripkeModelS5, head $ Exp.worldsOf m)
              in isTrue pm f === isTrue (optimize (vocabOf m) pm) f
    prop "optimize can shrink the model" $
      expectFailure (\m -> let pm = (m::Exp.KripkeModelS5, head $ Exp.worldsOf m)
                           in length (Exp.worldsOf pm) <= length (Exp.worldsOf (optimize (
                               vocabOf m) pm)) )
  describe "SMCDEL.Examples" $ do
    it "modelA: bob knows p, alice does not" $
      Exp.eval modelA $ Conj [K bob (PrpF (P 0)), Neg $ K alice (PrpF (P 0))]
    it "modelB: bob knows p, alice does not know whether he knows whether p" $
      Exp.eval modelB $ Conj [K bob (PrpF (P 0)), Neg $ Kw alice (Kw bob (PrpF (P 0)))]
    it "knsA has two states while knsB has three" $
      [2,3] === map (length . statesOf . fst) [knsA,knsB]
    it "redundantModel and minimizedModel are bisimilar" $
      Exp.checkBisim
        [(0,0),(1,0),(2,1)]
        (fst redundantModel)
        (fst minimizedModel `Exp.withoutProps` [P 2])
    it "bisiminimizing redundantModel removes world 0" $
      Exp.bisiminimize redundantModel === (fst redundantModel `Exp.withoutWorld` 0, 1)
    it "findStateMap works for modelB and knsB" $
      let (Just g) = findStateMap modelB knsB in equivalentWith modelB knsB g
    it "findStateMap works for redundantModel and myKNS" $
      let (Just g) = findStateMap redundantModel myKNS in equivalentWith redundantModel
          myKNS g
    it "findStateMap works for minimizedModel and myKNS" $
      let (Just g) = findStateMap minimizedModel myKNS in equivalentWith minimizedModel
          myKNS g
    describe "Three Muddy Children" $ do
      it "mudScn0: nobodyknows 3" $ evalViaBdd mudScn0 (nobodyknows 3)
      it "mudScn1: nobodyknows 3" $ evalViaBdd mudScn1 (nobodyknows 3)
      it "mudScn2: everyone knows" $ evalViaBdd mudScn2 (Conj [knows i | i <- [1..3]])
      it "mudKns2 has one state" $ length (SMCDEL.Symbolic.S5.statesOf mudKns2) === 1
      it "build result == mudScnInit 3 3" $ buildResult === mudScnInit 3 3
    it "Thirsty Logicians: valid for up to 10 agents" $
      all thirstyCheck [3..10]
    it "Dining Crypto: valid for up to 9 agents" $
      dcValid && all genDcValid [3..9]
    it "Dining Crypto, dcScn2: Only Alice knows that she paid:" $
      evalViaBdd dcScn2 $
```

```
        Conj [K "1" (PrpF (P 1)), Neg $ K "2" (PrpF (P 1)), Neg $ K "3" (PrpF (P 1))]
    it "Epistemic Planning: door mat plan succeeds" $
      reachesOn (Do "tryTake" tryTake (Check dmGoal Stop)) dmGoal dmStart
    it "Three Prisoners: Explicit Version reaches the goal" $
      endOf prisonExpStory `isTrue` prisonGoal
    it "Three Prisoners: Symbolic Version reaches the goal" $
      endOf prisonSymStory `isTrue` prisonGoal
    it "Russian Cards: all checks"
      SMCDEL.Examples.RussianCards.rcAllChecks
    it "Russian Cards: 102 solutions" $
      length (filter checkSet allHandLists) === 102
    it "Russian Cards: rusSCNfor (3,3,1) == rusSCN" $
      rusSCNfor (3,3,1) === rusSCN
    it "Sum and Product: There is exactly one solution." $
      length sapSolutions === 1
    it "Sum and Product: (4,13) is a solution." $
      validViaBdd sapKnStruct (Impl (Conj [xIs 4, yIs 13]) sapProtocol)
    it "Sum and Product: (4,13) is the only solution." $
      validViaBdd sapKnStruct (Impl sapProtocol (Conj [xIs 4, yIs 13]))
    it "Sum and Product: explaining the solution." $
      map sapExplainState sapSolutions `shouldBe` ["x = 4, y = 13, x+y = 17 and x*y = 52"]
    it "What Sum: There are 330 solutions." $
      length SMCDEL.Examples.WhatSum.wsSolutions === 330
    it "What Sum: The first solution is [('a',1),('b',3),('c',2)]" $
      wsExplainState (head wsSolutions) `shouldBe` [('a',1),('b',3),('c',2)]
  let ags = map show [1::Int,2,3]
  describe "SMCDEL.Explicit.K" $ do
    it "3MC genKrp: Top is Ck and Bot is not Ck" $
      ExpK.eval myMudGenKrpInit $ Conj [Ck ags Top, Neg (Ck ags Bot)]
    it "3MC genKrp: It is not common knowledge that someone is muddy" $
      ExpK.eval myMudGenKrpInit $
        Neg $ Ck (map show [1::Int,2,3]) $ Disj (map (PrpF . P) [1,2,3])
    it "3MC genKrp: after announcing makes it common knowledge that someone is muddy" $
      ExpK.eval myMudGenKrpInit $
        PubAnnounce (Disj (map (PrpF . P) [1,2,3])) $ Ck (map show [1::Int,2,3]) $ Disj (
            map (PrpF . P) [1,2,3])
  describe "SMCDEL.Symbolic.K" $ do
    it "3MC genScn: Top is Ck and Bot is not Ck" $
      SymK.evalViaBdd SMCDEL.Examples.MuddyChildren.myMudBelScnInit $ Conj [Ck ags Top, Neg
          (Ck ags Bot)]
    it "3MC genScn: It is not common knowledge that someone is muddy" $
      SymK.evalViaBdd SMCDEL.Examples.MuddyChildren.myMudBelScnInit $
        Neg $ Ck (map show [1::Int,2,3]) $ Disj (map (PrpF . P) [1,2,3])
    it "3MC genScn: after announcing makes it common knowledge that someone is muddy" $
      SymK.evalViaBdd SMCDEL.Examples.MuddyChildren.myMudBelScnInit $
        PubAnnounce (Disj (map (PrpF . P) [1,2,3])) $ Ck (map show [1::Int,2,3]) $ Disj (
            map (PrpF . P) [1,2,3])
```

8.4 Testing for K

```
module Main (main) where

import Data.Dynamic (toDyn)
import Data.Map.Strict (fromList)
import Data.List (sort)
import Test.Hspec
import Test.Hspec.QuickCheck

import SMCDEL.Internal.Help (alleq)
import SMCDEL.Language
import SMCDEL.Explicit.S5 (Action)
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Explicit.K as ExpK
import SMCDEL.Symbolic.K as SymK
import SMCDEL.Translations.K as TransK

main :: IO ()
main = hspec $ do
  describe "hardcoded myMod and myScn" $ do
    prop "semanticEquivalence" $ alleq . semanticEquivalenceTest
```

```haskell
    prop "singleChange: random action model with change" $ \ a f -> alleq $
        singleChangeTest a f
  describe "random Kripke models" $ do
    prop "Ck i -> K i" $ \(Ag i) krm f -> ExpK.eval (krm,0) (Ck [i] f `Impl` Ck [i] f)
    prop "semanticEquivExpToSym" $ \krm f -> alleq $ semanticEquivExpToSym (krm,0) f
    prop "diff" $ \krmA krmB -> diffTest (krmA,0) (krmB,0)
  describe "random belief structures" $
    prop "semanticEquivSymToExp" $ \bls f -> alleq $ semanticEquivSymToExp (bls, head (
        statesOf bls)) f

myMod :: ExpK.PointedModel
myMod = (ExpK.KrM $ fromList wlist, 0) where
  wlist = [ (w, (fromList val, fromList $ relFor val)) | (val,w) <- wvals ]
  vals  = map sort (foldl buildTable [[]] [P 0 .. P 4])
  wvals = zip vals [0..]
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr <- partrows ]
  relFor val = [ (show i, map snd $ seesFrom i val) | i <- [0..5::Int] ]
  seesFrom i val = filter (\(val',_) -> samefor i val val') wvals
  samefor 0 ps qs = ps == qs  -- knows everything
  samefor 1 _  _  = False     -- insane
  samefor _ _  _  = True

myScn :: SymK.BelScene
myScn =
  let allprops = [P 0 .. P 4]
  in (SymK.BlS allprops
                (boolBddOf Top)
                (fromList $ ("0", SymK.allsamebdd allprops)  -- knows everything
                          : ("1", SymK.emptyRelBdd)          -- insane
                          : [(show i, SymK.totalRelBdd) | i<-[2..5::Int]])
     , allprops)

semanticEquivalenceTest :: SimplifiedForm -> [Bool]
semanticEquivalenceTest (SF f) =
  [ ExpK.eval myMod f                              -- evaluate directly on Kripke
  , SymK.evalViaBdd myScn f                        -- evaluate equivalent BDD on BlS
  , ExpK.eval (TransK.blsToKripke myScn) f          -- evaluate on corresponding Kripke
  , SymK.evalViaBdd (TransK.kripkeToBls myMod) f   -- evaluate on corresponding BlS
  ]

singleChangeTest :: ActionModel -> SimplifiedForm -> [Bool]
singleChangeTest myact (SF f) =
  [ not (ExpK.eval                         myMod  (preOf                              (
      myact,0::Action)))
      || ExpK.eval      (update            myMod                                      (
          myact,0::Action)  ) f
  , not (SymK.evalViaBdd         (kripkeToBls myMod) (preOf                (actionToEvent (
      myact,0))))
      || SymK.evalViaBdd (update (kripkeToBls myMod)                      (actionToEvent (
          myact,0)))  f
  , not (ExpK.eval                         myMod  (preOf (eventToAction (actionToEvent (
      myact,0)))))
      || ExpK.eval      (update            myMod         (eventToAction (actionToEvent (
          myact,0)))) f
  , not (ExpK.eval               (blsToKripke myScn) (preOf (eventToAction (actionToEvent (
      myact,0)))))
      || ExpK.eval      (update (blsToKripke myScn)      (eventToAction (actionToEvent (
          myact,0)))) f
  , not (SymK.evalViaBdd                   myScn  (preOf                (actionToEvent (
      myact,0))))
      || SymK.evalViaBdd (update           myScn                       (actionToEvent (
          myact,0)) ) f
  -- using dynamic operators:
  , ExpK.eval       myMod (box (Dyn "(myact,0)"               (toDyn                  (myact
      ,0::Action))) f)
  , SymK.evalViaBdd myScn (box (Dyn "actionToEvent (myact,0)" (toDyn $ actionToEvent (myact
      ,0::Action))) f)
  ]
  ++ case SymK.reduce (actionToEvent (myact,0)) f of
      Nothing -> []
      Just g  -> pure $ SymK.evalViaBdd (kripkeToBls myMod) (simplify g)
  ++ [ SymK.evalViaBddReduce myScn (actionToEvent (myact,0)) f ]
```

The following tests are made with random models and structures.

```
semanticEquivExpToSym :: PointedModel -> SimplifiedForm -> [Bool]
semanticEquivExpToSym pm (SF f) =
  [ ExpK.eval pm f
  , SymK.evalViaBdd (TransK.kripkeToBls pm) f
  ]

semanticEquivSymToExp :: BelScene -> SimplifiedForm -> [Bool]
semanticEquivSymToExp scn (SF f) =
  [ SymK.evalViaBdd scn f
  , ExpK.eval (TransK.blsToKripke scn) f
  ]

diffTest :: PointedModel -> PointedModel -> Bool
diffTest pmA pmB =
  case diffPointed pmA pmB of
    Left  b -> checkBisimPointed b pmA pmB
    Right f -> isTrue pmA f && not (isTrue pmB f)
```

9 Epistemic Planning

Here we provide some wrapper functions for epistemic planning via model checking. Our main inspiration for this is [Eng+17].
NOTE: This module is highly experimental and under development.

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

module SMCDEL.Other.Planning where

import Data.Dynamic
import Data.HasCacBDD hiding (Top,Bot)
import Data.List (intersect,nub,sort,(\\))
import qualified Data.Map as M

import SMCDEL.Internal.Help (apply)
import SMCDEL.Language
import qualified SMCDEL.Symbolic.S5 as Sym
import qualified SMCDEL.Symbolic.K as SymK
import qualified SMCDEL.Explicit.S5 as Exp
import qualified SMCDEL.Explicit.K as ExpK
```

9.1 Offline Plans with Public Announcements

We first consider very simple plans which are list of formulas to be publicly and truthfully announced, one after each other. We write σ for any plan and ϵ for the empty plan which always succeeds. The following then defines a formula which says that the given plan reaches a given goal γ:

$$\begin{aligned} \mathsf{reaches}(\epsilon)(\gamma) &:= \gamma \\ \mathsf{reaches}(\varphi;\sigma)(\gamma) &:= \langle\varphi\rangle(\mathsf{succeeds}(\sigma)(\gamma)) \end{aligned}$$

```
type OfflinePlan = [Form] -- list of announcements to be made

class IsPlan a where
  reaches :: a -> Form -> Form
  reachesOn :: (Semantics o) => a -> Form -> o -> Bool
  reachesOn plan f start = start |= reaches plan f

instance IsPlan OfflinePlan where
  reaches []          goal = goal
  reaches (step:rest) goal = Conj [step, PubAnnounce step (reaches rest goal)]
```

A simple function to search for plans:

```
offlineSearch :: (Eq a, Semantics a, Update a Form) =>
   Int ->     -- maximum number of actions
   a ->       -- the starting model or structure
   [Form] -> -- the available actions
   [Form] -> -- intermediate goals / safety formulas
   Form ->    -- the goal formula (when to stop)
   [OfflinePlan]
offlineSearch roundsLeft now acts safety goal
  | now |= goal = [ [] ] -- done, goal reached
  | roundsLeft == 0 = [    ] -- give up
  | otherwise = [ a : rest
                | a <- acts
                , now |= a                -- only allow truthful announcements!
                , let new = now `update` a   -- the new state
                , new /= now                 -- ignore useless actions
                , all (new |=) safety
                -- depth-first search:
                , rest <- offlineSearch (roundsLeft-1) new acts safety goal ]
```

9.2 Online Planning with General Actions

An *online plan* in contrast can include tests and choices, to decide which action to do depending on the results of previous announcements. To represent such plans we use a tree where non-terminal nodes are actions to be done or formulas to be checked. Each action node has one outgoing edge and each checking node has two outgoing edges, leading to the follow-up plans. Terminal-nodes are stop signals.

In contrast to the previous section we now also allow more general actions, namely any type in the `Update` class, for example action models and transformers.

```
data Plan a = Stop
            | Do String a (Plan a)
            | Check Form (Plan a)
            | IfThenElse Form (Plan a) (Plan a)
  deriving (Eq,Ord,Show)

execute :: Update state a => Plan a -> state -> Maybe state
execute Stop                 s = Just s
execute (Do _ action rest)   s | s |= preOf action = execute rest (s `update` action)
                               | otherwise         = Nothing
execute (Check f rest)       s = if s |= f then execute rest s else Nothing
execute (IfThenElse f pa pb) s = if s |= f then execute pa   s else execute pb s
```

Again we can define a formula that describes that a given plan reaches a given goal.

```
instance IsPlan (Plan Form) where
  reaches Stop goal = goal
  reaches (Do _ toBeAn next) goal = Conj [toBeAn, PubAnnounce toBeAn (reaches next goal)]
  reaches (Check toBeChecked next) goal = Conj [toBeChecked, reaches next goal]
  reaches (IfThenElse condition planA planB) goal =
    Conj [ condition `Impl` reaches planA goal, Neg condition `Impl` reaches planB goal ]

instance IsPlan (Plan Sym.MultipointedEvent) where
  reaches Stop goal = goal
  reaches (Do actLabel action next) goal = dix (Dyn actLabel (toDyn action)) (reaches next
      goal)
  reaches (Check toBeChecked next) goal = Conj [toBeChecked, reaches next goal]
  reaches (IfThenElse check planA planB) goal =
    Conj [ check `Impl` reaches planA goal, Neg check `Impl` reaches planB goal ]
```

This uses the $((\alpha))\varphi := \langle\alpha\rangle \land [\alpha]\varphi$ abbreviation from [Eng+17]. We call it `dix` because it is a combination of *di*amond and bo*x*.

```
dix :: DynamicOp -> Form -> Form
dix op f = Conj [Dia op Top, box op f]
```

9.3 Planning Tasks

A planning task (also called a planning problem) is given by a start, a list of actions and a goal. Note that `state` and `action` here are type variables, because we use polymorphic functions for explicit and symbolic planning in K and S5.

```
data Task state action = Task state [(String,action)] Form
  deriving (Eq,Ord,Show)
```

Given a maximal search depth and a task, we search for a plan as follows.

```
findPlan :: (Eq state, Update state action) => Int -> Task state action -> [Plan action]
findPlan d (Task now acts goal)
  | now |= goal = [ Stop ]
  | d == 0      = [      ]
  | otherwise   = [ Do lbl act continue
                  | (lbl,act) <- acts
                  , isTrue now (preOf act)
                  , now /= update now act -- ignore useless actions
```

```
                   , continue <- findPlan (d-1) (Task (update now act) acts goal) ]
```

9.4 Perspective Shifts

```
class Eq o => HasPerspective o where
  asSeenBy :: o -> Agent -> o
  isLocalFor :: o -> Agent -> Bool
  isLocalFor state i = state `asSeenBy` i == state
```

Given a multipointed S5 model $(\mathcal{M}, s)$, we define the local state of an agent i like this:

$$s^i := \{w \in W \mid \exists v \in s : v \sim_i w\}$$

This is the definition given in [Eng+17].

```
instance HasPerspective Exp.MultipointedModelS5 where
  asSeenBy (m@(Exp.KrMS5 _ rel _), actualWorlds) agent = (m, seenWorlds) where
    seenWorlds = sort $ concat $ filter (not . null . intersect actualWorlds) (apply rel
        agent)
```

The authors of [Eng+17] only consider S5. Here we also implement perspective shifts in K.
Given a multipointed Kripke model $(\mathcal{M}, s)$, let the local state of i be:

$$s^i := \{w \in W \mid \exists v \in s : vR_iw\}$$

Intuitively, these are all worlds the agent considers possible if the current state is s.

```
instance HasPerspective ExpK.MultipointedModel where
  asSeenBy (ExpK.KrM m, actualWorlds) agent = (ExpK.KrM m, seenWorlds) where
    seenWorlds = sort $ nub $ M.foldlWithKey
      (\ vs w (_,rel) -> vs ++ concat [ rel M.! agent | w `elem` actualWorlds ])
      []
      m
```

Note that in S5 we always have $s \subseteq s^i$, but this is not the case in general for K. Also note that $(\cdot)^i$ is no longer idempotent if R_i is not transitive.

We can also make perspective shifts symbolically.
In the S5 setting we can exploit symmetry as follows. Suppose we have a multipointed scene $(\mathcal{F}, \sigma)$ where $\sigma \in \mathcal{L}_B(V)$ encodes the set of actual states. As usual, we assume that agent i has the observational variables O_i. Then the perspective of i is given by:

$$\sigma^i := \exists(V \setminus O_i)\sigma$$

On purpose we do not use θ in order to avoid redundancy in the resulting multipointed scene.

```
instance HasPerspective Sym.MultipointedKnowScene where
  asSeenBy (Sym.KnS props lawbdd obs, statesBdd) agent =
    (Sym.KnS props lawbdd obs, seenStatesBdd) where
      seenStatesBdd = existsSet otherps statesBdd
      otherps = map fromEnum (props \\ apply obs agent)
```

For the K setting we first need a way to flip the direction of the encoded relation Ω_i. For this we simultaneously prime and unprime all its variables. Let $\Omega_i^\smile$ denote the resulting BDD. Formally: $\Omega_i^\smile := [V \cup V' \mapsto V' \cup V]\Omega_i$.

```
flipRelBdd :: [Prp] -> SymK.RelBDD -> SymK.RelBDD
flipRelBdd props = fmap $ Sym.relabelWith [(SymK.mvP p, SymK.cpP p) | p <- props ]
```

Now again suppose we have a set of actual states encoded by $\sigma \in \mathcal{L}_B(V)$. Then the perspective of i is given by:

$$\sigma^i := \exists V'(\Omega_i^{\smile} \wedge \sigma')$$

The following chain of equivalences shows that this definition does indeed what we want. A state s satisifes the encoding of the local state σ^i iff it can be reached from a state t which satisfies the encoding of the global state σ.

$$\begin{aligned} & s \vDash \sigma^i \\ \iff & s \vDash \exists V'(\Omega_i^{\smile} \wedge \sigma') \\ \iff & \exists t \subseteq V : s \cup t' \vDash (\Omega_i^{\smile} \wedge \sigma') \\ \iff & \exists t \subseteq V : t \cup s' \vDash (\Omega_i \wedge \sigma) \\ \iff & \exists t \subseteq V : t \vDash \sigma \text{ and } t \cup s' \vDash \Omega_i \end{aligned}$$

Again we do not include θ here to avoid redundancy in the multipointed structure.

```
instance HasPerspective SymK.MultipointedBelScene where
  asSeenBy (SymK.BlS props lawbdd obsBdds, statesBdd) agent =
    (SymK.BlS props lawbdd obsBdds, seenStatesBdd) where
      flippedObsBdd = flipRelBdd props (obsBdds M.! agent)
      seenStatesBdd = SymK.unmvBdd $ existsSet (map fromEnum $ SymK.cp props) <$>
                        (con <$> SymK.cpBdd statesBdd <*> flippedObsBdd)
```

9.5 Cooperation

We now introduce owner functions as discussed in [Eng+17] and based on [LPW11].

```
data CoopTask state action = CoopTask state [Owned action] Form
  deriving (Eq,Ord,Show)

instance (HasPerspective state, Eq action) => HasPerspective (CoopTask state action) where
  asSeenBy (CoopTask start acts goal) agent = CoopTask (start `asSeenBy` agent) acts goal
```

Implicitly coordinated sequential plans. As done in section 3.2 of [Eng+17].

```
type Labelled a = (String,a)

type Owned action = (Agent,Labelled action)

type ICPlan action = [Owned action]
-- note: there is no check that the action is actually local for the agent!

ppICPlan :: ICPlan action -> String
ppICPlan [] = ""
ppICPlan [(agent,(label,_))] = agent ++ ":" ++ label ++ "."
ppICPlan ((agent,(label,_)):rest) = agent ++ ":" ++ label ++ "; " ++ ppICPlan rest

icSolves :: (Typeable action, Semantics state) => CoopTask state action -> ICPlan action ->
     Bool
icSolves (CoopTask start acts goal) plan =
  all (`elem` map fst acts) (map fst plan) && start |= succForm plan where
    succForm [] = goal
    succForm ((agent,(label,action)):rest) = K agent (dix (Dyn label (toDyn action)) (
        succForm rest))
```

We now give a simple search algorithm to find sequential IC plans (again depth-first!)

```haskell
findSequentialIcPlan :: (Typeable action, Eq state, Update state action) => Int -> CoopTask
     state action -> [ICPlan action]
findSequentialIcPlan d (CoopTask now acts goal)
  | now |= goal = [ [] ] -- goal reached
  | d == 0      = [    ] -- give up
  | otherwise   = [ (agent,(label, act)) : continue
                  | a@(agent,(label,act)) <- acts
                  , now |= preOf act           -- action must be executable
                  , now |= K agent (preOf act) -- agent must know that it is executable!
                  , now /= update now act      -- ignore useless actions
                  , continue <- findSequentialIcPlan (d-1) (CoopTask (update now act) acts
                    goal) -- DFS!
                  , icSolves (CoopTask now acts goal) (a:continue) ]
```

To find a shortest plan, we also implement breadth-first search:

```haskell
findSequentialIcPlanBFS :: (Typeable action, Eq state, Update state action) => Int ->
    CoopTask state action -> Maybe (ICPlan action)
findSequentialIcPlanBFS maxDepth (CoopTask start acts goal) = loop [([],start)] where
  loop [] = Nothing
  loop ((done,now):rest)
    | now |= goal = Just done -- FIXME: need stronger condition >> icSolves (CoopTask now
        acts goal) (done)
    | otherwise   = loop $ rest ++
                      [ (done ++ [a], update now act) -- TODO optimize (vocabOf start) ?
                      | length done < maxDepth      -- do not use more than maxDepth actions
                      , a@(agent,(_,act)) <- acts
                      , now |= preOf act            -- action must be executable
                      , now |= K agent (preOf act) -- agent must know that it is executable
                        !
                      , now /= update now act       -- ignore useless actions
                      ]
```

10 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.

10.1 Small Examples

```
{-# LANGUAGE FlexibleInstances #-}

module SMCDEL.Examples where

import Data.List ((\\),sort)

import SMCDEL.Explicit.S5
import SMCDEL.Internal.TaggedBDD
import SMCDEL.Language
import SMCDEL.Symbolic.S5
import SMCDEL.Translations.S5
```

10.1.1 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.

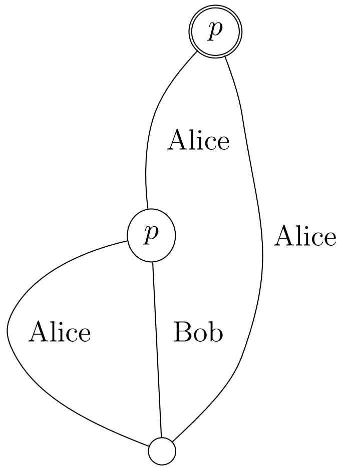
```
modelA :: PointedModelS5
modelA = (KrMS5 [0,1] [(alice,[[0,1]]),(bob,[[0],[1]])] [ (0,[(P 0,True)]), (1,[(P 0,False)
    ]) ], 0)
```


```
>>> map (SMCDEL.Explicit.S5.eval modelA) [K bob (PrpF (P 0)), K alice (PrpF (P 0))]

[True,False]

0.00 seconds
```

```
>>> SMCDEL.Explicit.S5.eval modelA (K alice (Kw bob (PrpF (P 0))))

True

0.00 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 :: PointedModelS5
modelB =
  (KrMS5
    [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.S5.eval modelB (K bob (PrpF (P 0)))
```

```
True
```

0.00 seconds

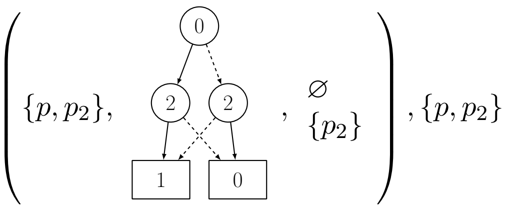
```
>>> SMCDEL.Explicit.S5.eval modelB (Kw alice (Kw bob (PrpF (P 0))))
```

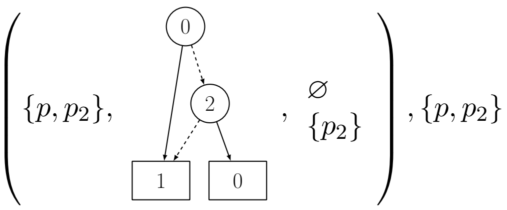
```
False
```

0.00 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 :: KnowScene
knsA = kripkeToKns modelA
knsB = kripkeToKns modelB
```





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:

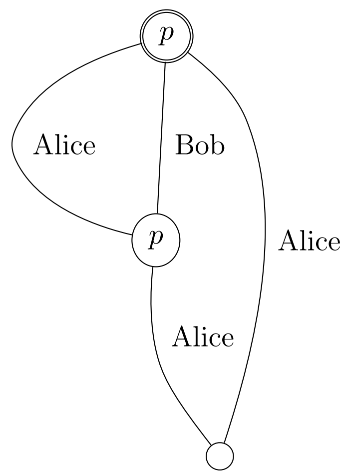
```
>>> let (structA,foo) = knsA in statesOf structA

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

0.05 seconds
```

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

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

0.05 seconds
```

10.1.2 Minimization via Translation

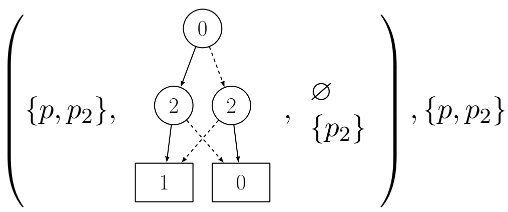
Consider the following Kripke model where **0** and **1** are bisimilar — it is redundant.

```
redundantModel :: PointedModelS5
redundantModel = (KrMS5 [0,1,2] [(alice,[[0,1,2]]),(bob,[[0,1],[2]])] [ (0,[(P 0,True)]),
    (1,[(P 0,True)]), (2,[(P 0,False)]) ], 0)
```



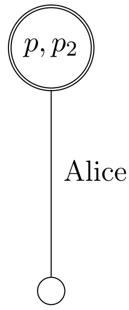
If we transform this model to a knowledge structure, we get the following:

```
myKNS :: KnowScene
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 Kripke models.

```
minimizedModel :: PointedModelS5
minimizedModel = knsToKripke myKNS
```

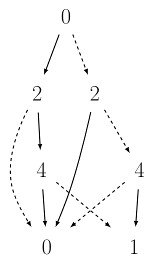


This is bisimilar to the redundant model:

```
>>> checkBisim [(0,0),(1,0),(2,1)] (fst redundantModel) (fst minimizedModel
'SMCDEL.Explicit.S5.withoutProps' [toEnum 2])

True

0.04 seconds
```

Again we can transform this model to a knowledge structure:

```
minimizedKNS :: KnowScene
minimizedKNS = kripkeToKns minimizedModel
```

$\Bigg(\{p, p_2, p_4\},$  $, \ \begin{matrix}\varnothing \\ \{p_4\}\end{matrix}\Bigg), \{p, p_2\}$

And prove them equivalent using a simple propulation:

```
myPropu :: Propulation
myPropu = allsamebdd (vocabOf myKNS)
```

```
>>> checkPropu myPropu (fst myKNS) (fst minimizedKNS) (vocabOf myKNS)

True

0.04 seconds
```

10.1.3 Different Announcements

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

```
pubAnnounceAction :: [Agent] -> Form -> PointedActionModelS5
pubAnnounceAction ags f = (ActMS5 [(0,(f,[]))] [ (i,[[0]]) | i <- ags ], 0)

examplePaAction :: PointedActionModelS5
examplePaAction = pubAnnounceAction [alice,bob] (PrpF (P 0))
```

```
>>> examplePaAction

(ActMS5 [(0,(PrpF (P 0),[]))] [("Alice",[[0]]),("Bob",[[0]])],0)

0.00 seconds
```

```
>>> actionToEvent examplePaAction

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

0.00 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 -> PointedActionModelS5
groupAnnounceAction everyone listeners f = (ActMS5 [(0,(f,[])),(1,(Neg f,[]))] actrel, 0)
  where actrel = sort $ [ (i,[[0],[1]]) | i <- listeners ]
                     ++ [ (i,[[0 , 1]]) | i <- everyone \\ listeners ]

exampleGroupAnnounceAction :: PointedActionModelS5
exampleGroupAnnounceAction = groupAnnounceAction [alice, bob] [alice] (PrpF (P 0))

eGrAnLaw :: Form
exampleGrAnnEvent :: Event
exampleGrAnnEvent@(KnTrf _ eGrAnLaw _ _, _) = actionToEvent exampleGroupAnnounceAction
```

```
>>> exampleGroupAnnounceAction

(ActMS5 [(0,(PrpF (P 0),[])),(1,(Neg (PrpF (P 0)),[]))] [("Alice",[[0],[1]]),("Bob
    ",[[0,1]])],0)

0.00 seconds
```

```
>>> exampleGrAnnEvent

(KnTrf [P 1,P 2] (Conj [Disj [Conj [PrpF (P 1),PrpF (P 0)],Conj [Neg (PrpF (P 1)),
    Neg (PrpF (P 0))]],Equi (PrpF (P 2)) (PrpF (P 1)),Equi (Neg (PrpF (P 2))) (Neg
     (PrpF (P 1)))]) [] [("Alice",[P 2]),("Bob",[])],[P 1,P 2])

0.00 seconds
```

But it is not hard to check that this is equivalent to the definition. Consider the θ^+ formula of this transformer:

$$\texttt{eGrAnLaw} = \bigwedge\{((p_1 \land p) \lor (\neg p_1 \land \neg p)), (p_2 \leftrightarrow p_1), (\neg p_2 \leftrightarrow \neg p_1)\}$$

Note that this implies $p_1 \leftrightarrow p_2$. 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 16 to translate action models to knowledge transformers: First a set of propositions is used to label all actions, then additional new observational variables are used to enumerate all equivalence classes for all agents.

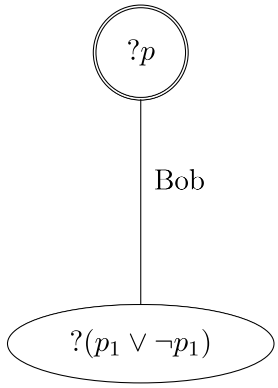
We can also turn this knowledge transformer back to an action model. The result is the same as the action model we started with, up to a renaming of action 1 to 3.

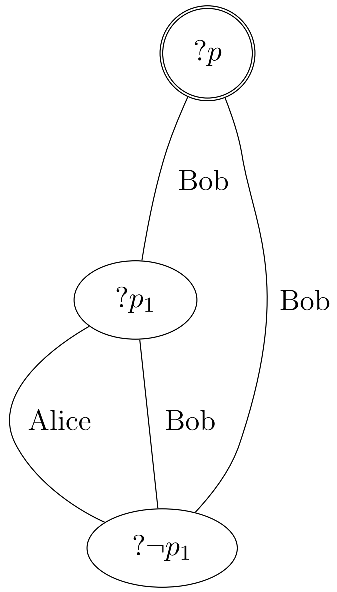
```
>>> eventToAction (actionToEvent exampleGroupAnnounceAction)

(ActMS5 [(0,(PrpF (P 0),[])),(3,(Neg (PrpF (P 0)),[]))] [("Alice",[[3],[0]]),("Bob
    ",[[0,3]])],0)

0.00 seconds
```

10.1.4 Equivalent Action Models

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

```
actionOne :: PointedActionModelS5
actionOne = (ActMS5 [(0,(p,[])),(1, (Disj [q, Neg q],[]))] [("Alice",[[0],[1]]), ("Bob"
    ,[[0,1]])], 0) where (p,q) = (PrpF $ P 0, PrpF $ P 1)

actionTwo :: PointedActionModelS5
actionTwo = (ActMS5 [(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

(KnTrf [P 2,P 3] (Conj [Disj [Conj [PrpF (P 2),PrpF (P 0)],Neg (PrpF (P 2))],Equi
    (PrpF (P 3)) (PrpF (P 2)),Equi (Neg (PrpF (P 3))) (Neg (PrpF (P 2)))]) [] [("
    Alice",[P 3]),("Bob",[])],[P 2,P 3])

0.00 seconds
```

```
>>> actionToEvent actionTwo

(KnTrf [P 2,P 3,P 4] (Conj [Disj [Conj [PrpF (P 2),PrpF (P 3),PrpF (P 0)],Conj [
    PrpF (P 2),Neg (PrpF (P 3)),PrpF (P 1)],Conj [PrpF (P 3),Neg (PrpF (P 2)),Neg
    (PrpF (P 1))]],Equi (PrpF (P 4)) (Conj [PrpF (P 2),PrpF (P 3)]),Equi (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])

0.00 seconds
```

$$\left(\left(\{p_2, p_3\},\ \bigwedge\{((p_2 \land p) \lor \neg p_2), (p_3 \leftrightarrow p_2), (\neg p_3 \leftrightarrow \neg p_2)\},\ ,\ \begin{matrix}\{p_3\}\\ \varnothing\end{matrix}\ \right),\ \{p_2, p_3\}\right)$$

$$\left(\left(\{p_2, p_3, p_4\},\ \bigwedge\{\bigvee\{\bigwedge\{p_2, p_3, p\}, \bigwedge\{p_2, \neg p_3, p_1\}, \bigwedge\{p_3, \neg p_2, \neg p_1\}\}, (p_4 \leftrightarrow (p_2 \land p_3)), (\neg p_4 \leftrightarrow ((p_2 \land \neg p_3) \lor (p_3 \land \neg p_2))), \bigvee\{(p_2 \land p_3), (p_2 \land \neg p_3), (p_3 \land \neg p_2)\}\},\ ,\ \begin{matrix}\{p_4\}\\ \varnothing\end{matrix}\ \right),\ \{p_2, p_3, p_4\}\right)$$

10.2 Cheryl's Birthday

We now solve the famous riddle from the Singapore math olympiad. For the statement and standard solution of the puzzle, see [Dit+17]. For a solution of the puzzle using the explicit model checker DEMO-S5, see `https://malv.in/posts/2015-04-20-finding-cheryls-birthday-with-DEMO.html`.

```
module SMCDEL.Examples.Cheryl where

import Data.HasCacBDD (Bdd,con,disSet)
import Data.List

import SMCDEL.Language
import SMCDEL.Symbolic.S5
import SMCDEL.Internal.Help (powerset)

type Possibility = (Int, String)

possibilities :: [Possibility]
possibilities =
  [ (15,"May"), (16,"May"), (19,"May")
  , (17,"June"), (18,"June")
  , (14,"July"), (16,"July")
  , (14,"August"), (15,"August"), (17,"August") ]

dayIs :: Int -> Prp
dayIs = P

monthIs :: String -> Prp
monthIs "May"    = P 5
monthIs "June"   = P 6
monthIs "July"   = P 7
monthIs "August" = P 8
monthIs _        = undefined

thisPos :: Possibility -> Form
thisPos (d,m) = Conj $
  (PrpF (dayIs d) : [ Neg (PrpF $ dayIs d') | d' <- nub (map fst possibilities) \\ [d] ])
    ++
  (PrpF (monthIs m) : [ Neg (PrpF $ monthIs m') | m' <- nub (map snd possibilities) \\ [m]
      ])
```

The formula saying that i knows Cheryl's birthday is defined as the disjunction over all statements of the form "Agent i knows that the birthday is s":

```
knWhich :: Agent -> Form
knWhich i = Disj [ K i (thisPos pos) | pos <- possibilities ]

start :: KnowStruct
start = KnS allprops statelaw obs where
  allprops = sort $ nub $ map (dayIs . fst) possibilities ++ map (monthIs . snd)
      possibilities
```

```
  statelaw = boolBddOf $ Conj
    [ Disj (map thisPos possibilities)
    , Conj [ Neg $ Conj [thisPos p1, thisPos p2] | p1 <- possibilities, p2 <- possibilities
        , p1 /= p2 ] ]
  obs = [ ("Albert", nub $ map (dayIs   . fst) possibilities)
        , ("Bernard", nub $ map (monthIs . snd) possibilities) ]
```

Now we update the model three times, using the function `update` which given a formula does a public announcement.

1. Albert: I don't know when Cheryl's birthday is and I know that Bernard does not know.
2. Bernard: "Now I know when Cheryl's birthday is."
3. Albert says: "Now I also know when Cheryl's birthday is."

```
round1,round2,round3 :: KnowStruct
round1 = update start (Conj [Neg $ knWhich "Albert", K "Albert" $ Neg (knWhich "Bernard")])
round2 = update round1 (knWhich "Bernard")
round3 = update round2 (knWhich "Albert")

cherylsBirthday :: String
cherylsBirthday = m ++ " " ++ show d ++ "th" where
  [(d,m)] = filter (\(d',m') -> [monthIs m', dayIs d'] `elem` statesOf round3)
      possibilities
```

```
>>> SMCDEL.Examples.Cheryl.cherylsBirthday
```

```
"June 17th"
```

0.05 seconds

10.3 Cheryl's Age

We now formalize and implement the sequel of the Cheryl's Birthday puzzle. This new puzzle concerns the age of Cheryl and two brothers. To work with such numeric variable we first define some general functions. For more information about this binary encoding, see [Gat18, Section 5.1].

```
newtype Variable = Var [Prp] deriving (Eq,Ord,Show)

bitsOf :: Int -> [Int]
bitsOf 0 = []
bitsOf n = k : bitsOf (n - 2^k) where
  k :: Int
  k = floor (logBase 2 (fromIntegral n) :: Double)

-- alternative to:  booloutofForm (powerset props !! n) props
is :: Variable -> Int -> Form
is (Var props) n = Conj [ (if i `elem` bitsOf n then id else Neg) (PrpF k)
                        | (k,i) <- zip props [(0::Int)..] ]

isBdd :: Variable -> Int -> Bdd
isBdd v = boolBddOf . is v

-- inverse of "is":
valueIn :: Variable -> State -> Int
valueIn (Var props) s = sum [ 2^i | (k,i) <- zip props [(0::Int)..], k `elem` s ]

explainState :: [Variable] -> State -> [Int]
explainState vs s = map (`valueIn` s) vs

-- an agent knows the value iff they know-whether all bits
kv :: Agent -> Variable -> Form
kv i (Var props) = Conj [ Kw i (PrpF k) | k <- props ]
```

We can now start our analysis of the puzzle by defining all possible triples.

```
--  Cheryl: I have two younger brothers. The product of all our ages is 144.
allStates :: [(Int,Int,Int)]
allStates = [ (c,b1,b2) | c  <- [1..144]
                        , b1 <- [1..(c-1)]
                        , b2 <- [1..(c-1)]
                        , c * b1 * b2 == 144 ]
```

We then use two variables with eight bits to encode the age of cheryl and one brother. In order to use fewer variables we leave the age of the other brother implicit.

```
cheryl, broOne :: Variable
cheryl = Var [P (2*k   ) | k <- [0..7] ]
broOne = Var [P (2*k +1) | k <- [0..7] ]

ageKnsStart :: KnowStruct
ageKnsStart = KnS allprops statelaw obs where
  allprops = let (Var cs, Var bs) = (cheryl, broOne) in sort $ cs ++ bs
  statelaw = disSet [ con (cheryl `isBdd` c) (broOne `isBdd` b) | (c,b,_) <- allStates ]
  obs = [("albernard",[])]
```

We now update the structure in five steps, following the dialogue given in the puzzle.

```
step1,step2,step3,step4,step5 :: KnowStruct

-- Albert: We still don't know your age. What other hints can you give us?
step1 = ageKnsStart `update` Neg (kv "albernard" cheryl)

-- Cheryl: The sum of all our ages is the bus number of this bus that we are on.
step2 = step1 `update` revealTransformer
-- For this we need a way to reveal the sum, hence we use a knowledge transformer
revealTransformer :: KnowTransformer
revealTransformer = noChange KnTrf addProps addLaw addObs where
  addProps = map P [1001..1008] -- 8 bits to label all sums
  addLaw = simplify $ Conj [ Disj [ label (c + b + a) | (c,b,a) <- allStates ]
                           , Conj [ sumIs s `Equi` label s | s <- [1..144] ] ] where
    label s = booloutofForm (powerset (map P [1001..1008]) !! s) (map P [1001..1008])
    sumIs n = Disj [ Conj [ cheryl `is` c, broOne `is` b ]
                   | (c,b,a) <- allStates, c + b + a == n ]
  addObs = [("albernard",addProps)]

-- Bernard: Of course we know the bus number, but we still don't know your age.
step3 = step2 `update` Neg (kv "albernard" cheryl)

-- Cheryl: Oh, I forgot to tell you that my brothers have the same age.
step4 = step3 `update` sameAge where
  sameAge = Disj [ Conj [cheryl `is` c, broOne `is` b ]
                 | (c,b,a) <- allStates
                 , b == a ]

-- Albert and Bernard: Oh, now we know your age.
step5 = step4 `update` kv "albernard" cheryl
```

The solution can then be found by translating true propositions in the remaining state back to integers:

```
>>> map (explainState [cheryl,broOne]) (statesOf step5)

[[9,4]]

0.09 seconds
```

10.4 Cheryl's Age in DEMO-S5

```
module SMCDEL.Examples.CherylDemo where

import Data.List
```

```
import SMCDEL.Explicit.DEMO_S5 as DEMO_S5

type MyWorld = (Int,Int,Int)

--  Cheryl: I have two younger brothers. The product of all our ages is 144.
allStates :: [MyWorld]
allStates = [ (c,b1,b2) | c  <- [1..144]
                        , b1 <- [1..(c-1)]
                        , b2 <- [1..(c-1)]
                        , c * b1 * b2 == 144 ]

start,step1,step2,step3,step4,step5 :: DEMO_S5.EpistM MyWorld

start = DEMO_S5.Mo states agents [] rels points where
  states = allStates
  agents = map DEMO_S5.Ag [1] -- a single observer agent
  rels = [ (DEMO_S5.Ag 1, [states]) ] -- nothing known
  points = allStates

cherylIs :: Int -> DemoForm MyWorld
cherylIs n = Disj [ Info (n,b1,b2) | b1 <- [1..144], b2 <- [1..144], (n,b1,b2) `elem`
    allStates ]

weKnowIt :: DemoForm MyWorld
weKnowIt = Disj [ Kn (Ag 1) (cherylIs n) | n <- [1..144]]

-- Albert: We still don't know your age. What other hints can you give us?
step1 = start `updPa` Ng weKnowIt

-- Cheryl: The sum of all our ages is the bus number of this bus that we are on.
step2 = updsPaW step1 [ sumIs n | n <- possibleSums ] where
  possibleSums = sort . nub $ map (\(c, b1, b2) -> c+b1+b2) allStates
  sumIs n = Disj (map Info (filter (\(c, b1, b2) -> c+b1+b2 == n) allStates))

-- Bernard: Of course we know the bus number, but we still don't know your age.
step3 = step2 `updPa` Ng weKnowIt

-- Cheryl: Oh, I forgot to tell you that my brothers have the same age.
step4 = step3 `updPa` broSame where
  broSame = Disj (map Info (filter (\(_, b1, b2) -> b1 == b2) allStates))

-- Albert and Bernard: Oh, now we know your age.
step5 = step4 `updPa` weKnowIt
```

The resulting Kripke model contains the solution as its only world:

```
>>> step5

Mo
  [(9,4,4)]
  [Ag 1]
  []
  [(Ag 1,[[(9,4,4)]])]
  [(9,4,4)]

1.42 seconds
```

Note that the first and the last line of the puzzle do not give us any information. Formally, we can check that these announcements do not change the model as follows:

```
>>> (start==step1, step1==step2, step2==step3, step3==step4, step4==step5)

(True,False,False,False,True)

1.38 seconds
```

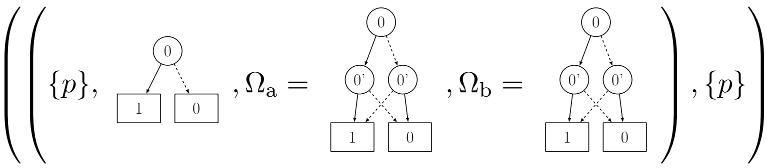
10.5 Example: Coin Flip

```
module SMCDEL.Examples.CoinFlip where

import Data.Map.Strict (fromList)
import Data.List ((\\))

import SMCDEL.Language
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Symbolic.K
```

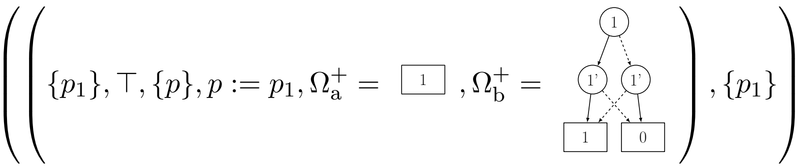
Consider a coin lying on a table with heads up: p is true and this is common knowledge. Suppose we then toss it randomly and hide the result from agent a but reveal it to agent b.

```
coinStart :: BelScene
coinStart = (BlS [P 0] law obs, actual) where
  law    = boolBddOf (PrpF $ P 0)
  obs    = fromList [ ("a", allsamebdd [P 0]), ("b", allsamebdd [P 0]) ]
  actual = [P 0]

flipRandomAndShowTo :: [Agent] -> Prp -> Agent -> Event
flipRandomAndShowTo everyone p i = (Trf [q] eventlaw changelaw obs, [q]) where
  q = freshp [p]
  eventlaw = Top
  changelaw = fromList [ (p, boolBddOf $ PrpF q) ]
  obs = fromList $
    (i, allsamebdd  [q]) :
    [ (j,totalRelBdd) | j <- everyone \\ [i] ]

coinFlip :: Event
coinFlip = flipRandomAndShowTo ["a","b"] (P 0) "b"

coinResult :: BelScene
coinResult = coinStart `update` coinFlip
```

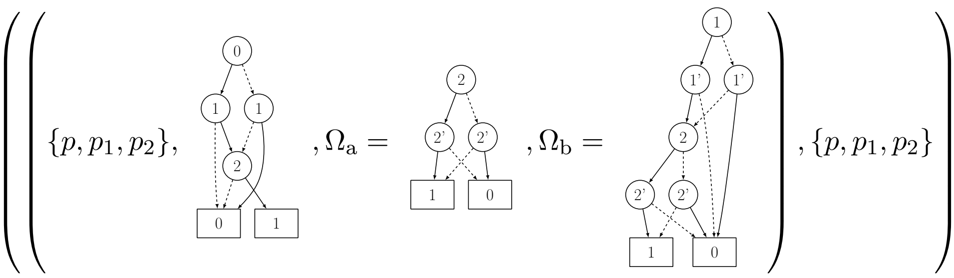
The structure ...



... transformed with `coinFlip` ...



... yields this new structure:



which has two states:

```
>>> SMCDEL.Symbolic.K.statesOf (fst SMCDEL.Examples.CoinFlip.coinResult)

 [[P 0,P 1,P 2],[P 2]]

0.05 seconds
```

10.6 Dining Cryptographers

```
module SMCDEL.Examples.DiningCrypto where

import Data.List (delete)

import SMCDEL.Language
import SMCDEL.Symbolic.S5
```

We model the scenario described in [Cha88]:

> 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 can 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.

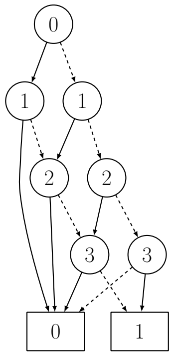
```
dcScnInit :: Int -> (Bool,Bool,Bool) -> KnowScene
dcScnInit payer (b1,b2,b3) = ( KnS props law obs , truths ) where
  props = [ P 0   -- The NSA paid
          , P 1   -- Alice paid
          , P 2   -- Bob paid
          , P 3   -- Charlie paid
          , P 4   -- shared bit of Alice and Bob
          , 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 :: KnowScene
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:

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



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

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 SMCDEL.Examples.DiningCrypto.dcScn1)
[SMCDEL.Examples.DiningCrypto.reveal 1, SMCDEL.Examples.DiningCrypto.reveal 2,
SMCDEL.Examples.DiningCrypto.reveal 3]
```

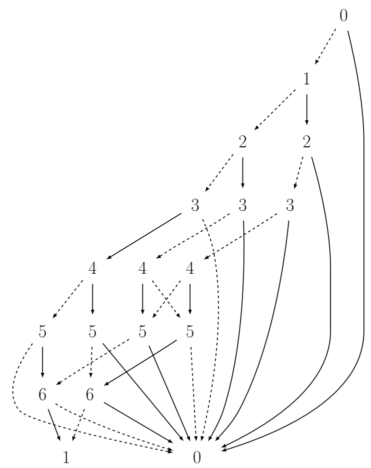
```
[True,True,True]
```

0.05 seconds

Now these three facts are announced:

```
dcScn2 :: KnowScene
dcScn2 = update dcScn1 (Conj [reveal 1, reveal 2, reveal 3])
```

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



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

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 SMCDEL.Examples.DiningCrypto.dcScn2
SMCDEL.Examples.DiningCrypto.everyoneKnowsWhetherNSApaid
```

```
True
```

0.04 seconds

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

```
>>> evalViaBdd SMCDEL.Examples.DiningCrypto.dcScn2 (K (show 1) (PrpF (P 1)))

True

0.05 seconds
```

```
>>> evalViaBdd SMCDEL.Examples.DiningCrypto.dcScn2 (K (show 2) (PrpF (P 1)))

False

0.05 seconds
```

```
>>> evalViaBdd SMCDEL.Examples.DiningCrypto.dcScn2 (K (show 3) (PrpF (P 1)))

False

0.05 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.

```
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 SMCDEL.Examples.DiningCrypto.dcScn1
SMCDEL.Examples.DiningCrypto.dcCheckForm

True

0.05 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.

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

The whole check runs within a fraction of a second:

```
>>> SMCDEL.Examples.DiningCrypto.dcValid

True

0.05 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 use it for a benchmark in Section 11.2. 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.

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

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

-- | Initial structure for Dining Cryptographers (complete graph!)
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 [genDcSomeonepaid n, genDcNotwopaid 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)

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

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

genDcNobodyknowsWhoPaid :: Int -> Form
genDcNobodyknowsWhoPaid 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 [ genDcEveryoneKnowsWhetherNSApaid n, genDcNobodyknowsWhoPaid n ]

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

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

```
>>> SMCDEL.Examples.DiningCrypto.genDcValid 4

True

0.05 seconds
```

10.7 Drinking Logicians

```
module SMCDEL.Examples.DrinkLogic where

import SMCDEL.Language
import SMCDEL.Symbolic.S5
```

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

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

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:

$$\bigwedge_i \neg \left(K^?_i \bigwedge_k P_k \right) \quad \wedge \quad [!\neg K^?_1 \bigwedge_k P_k] \dots [!\neg K^?_{n-1} \bigwedge_k P_k] \left(K_n \bigwedge_k P_k \right)$$

```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)
```

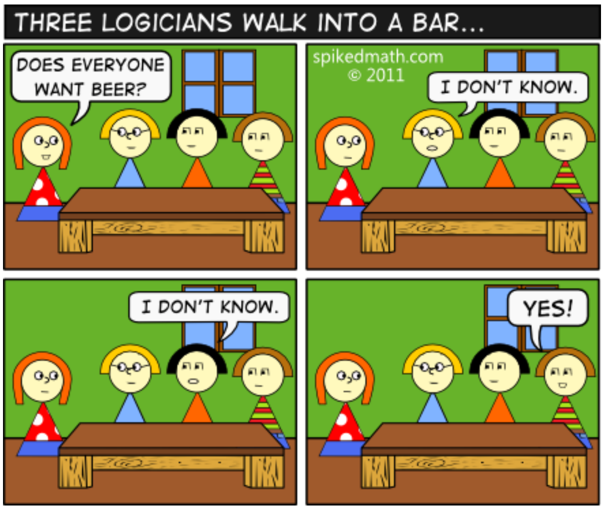
```
>>> thirstyCheck 3

True

0.05 seconds
```

```
>>> thirstyCheck 10

True

0.05 seconds
```

```
>>> thirstyCheck 100

True

0.12 seconds
```

```
>>> thirstyCheck 200

True

0.38 seconds
```

```
>>> thirstyCheck 400

True

1.77 seconds
```



http://spikedmath.com/445.html

10.8 Knowing-whether Gossip on belief structures with epistemic change

We consider the classic telephone problem where n agents each know only their own secret in the beginning and then make phone calls in which they always exchange all secrets they know. For now we only consider static and not dynamic gossip, i.e. all agents can call all others — the N relation is total.

In this section we follow the modeling used in [Att+14] and use *knowing-whether*: Agent a knows the secret of b iff $K^?_a p_b$.

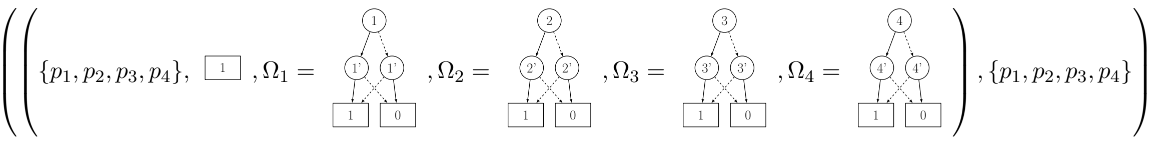
We use belief instead of knowledge structures because this makes it much easier to describe the event observation law. Otherwise we would have to add a lot more observational variables.

Still, the relations actually will be equivalences — despite the name, nobody is being deceived in the classic gossip problem as we model it here. Hence not using knowledge structures optimized for S5 is a big waste. In the next Section 10.9 we present an alternative model which is an abstraction of the one here and performs much better but has other limitations.

```
module SMCDEL.Examples.GossipKw where

import SMCDEL.Language
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Symbolic.K

import Control.Arrow ((&&&))
import Data.HasCacBDD hiding (Top)
import Data.Map.Strict (Map,fromList)
import Data.List ((\\),sort)
```

We fix the number of agents at 4 for now.

```
n :: Int
n = 4

gossipInit :: BelScene
gossipInit = (BlS vocab law obs, actual) where
  vocab  = map P [1..n]
  law    = boolBddOf Top
  obs    = fromList [ (show i, allsamebdd [P i]) | i <- [1..n] ]
  actual = vocab
```




```
willExchangeT :: (Int,Int) -> Int -> Form
willExchangeT (a,b) k | k `elem` [a,b] = PrpF (P k)
                      | otherwise      = Disj [ K (show i) $ PrpF (P k) | i <- [a,b] ]

inCall,inSecT :: Int -> Prp
inCall k = P (100+k) -- k participates in the call
inSecT k = P (200+(k*2)) -- secret k is being exchanged (as true)

call :: (Int,Int) -> [Int] -> Event
call (a,b) secSetT = (callTrf,actualSet) where
  actualSet = [inCall a, inCall b] ++ map inSecT secSetT

callTrf :: Transformer
callTrf = Trf vocplus lawplus (fromList []) obsplus where
  vocplus = sort $ map inCall [1..n] ++ map inSecT [1..n]
  lawplus = simplify $ Disj [ Conj [ thisCallHappens i j, theseSecretsAreExchanged i j ]  |
       i <- [1..n], j <- [1..n], i < j ] where
    thisCallHappens i j = Conj $ map (PrpF . inCall) [i,j] ++ map (Neg . PrpF . inCall)
       ([1..n] \\ [i,j])
    -- lnsPreCondition i j = Neg $ K (show i) (PrpF $ P j)
    theseSecretsAreExchanged i j = simplify $ Conj
      [ PrpF (inSecT k) `Equi` willExchangeT (i,j) k | k <- [1..n] ]
  obsplus :: Map Agent RelBDD
  obsplus = fromList $ map (show &&& obsfor) [1..n] where
    obsfor i = con <$> allsamebdd [ inCall i ]
                   <*> (imp <$> (mvBdd . boolBddOf . PrpF $ inCall i)
```

```
                            <*> allsamebdd (sort $ map inCall [1..n] ++ map inSecT [1..n]))
```

The following is an ad-hoc solution to calculate `secSetT` in advance. A more efficient implementation should use multi-pointed transformers.

```
toBeExchangedT :: BelScene -> (Int,Int) -> [Int]
toBeExchangedT scn (a,b) = filter (evalViaBdd scn . willExchangeT (a,b)) [1..n]

doCall :: BelScene -> (Int,Int) -> BelScene
doCall start (a,b) = cleanupObsLaw $ start `update` call (a,b) (toBeExchangedT start (a,b))

doCalls :: BelScene -> [(Int,Int)] -> BelScene
doCalls = foldl doCall

expert :: Int -> Form
expert k = Conj [ K (show k) $ PrpF (P i) | i <- [1..n] ]

allExperts :: Form
allExperts = Conj $ map expert [1..n]

whoKnowsWhat :: BelScene -> [(Int,[Int])]
whoKnowsWhat scn = [ (k, filter (knownBy k) [1..n]) | k <- [1..n] ] where
  knownBy k i = evalViaBdd scn (K (show k) $ PrpF (P i))

-- What do agents know, and what do they know about each others knowledge?
whoKnowsMeta :: BelScene -> [(Int,[(Int,String)])]
whoKnowsMeta scn = [ (k, map (meta k) [1..n] ) | k <- [1..n] ] where
  meta x y = (y, map (knowsAbout x y) [1..n])
  knowsAbout x y i
    | evalViaBdd scn (K (show x) $ PrpF (P i) `Impl` K (show y) (PrpF (P i))) = 'Y'
    | evalViaBdd scn (Neg $ K (show x) $ Neg $ K (show y) $ PrpF (P i))       = '?'
    | evalViaBdd scn (K (show x) $ Neg $ K (show y) $ PrpF (P i))             = '_'
    | otherwise                                                               = 'E'

after :: [(Int,Int)] -> BelScene
after = doCalls gossipInit

succeeds :: [(Int,Int)] -> Bool
succeeds sequ = evalViaBdd (after sequ) allExperts

allSequs :: Int -> [ [(Int,Int)] ]
allSequs 0 = [ [] ]
allSequs l = [ (i,j):rest | rest <- allSequs (l-1), i <- [1..n], j <- [1..n], i < j ]
```

For example, the following is not a success sequence for four agents:

```
>>> SMCDEL.Examples.GossipKw.succeeds [(1,2),(2,3),(3,1)]
```

```
False
```

2.41 seconds

But this is:

```
>>> SMCDEL.Examples.GossipKw.succeeds [(1,2),(3,4),(1,3),(2,4)]
```

```
True
```

15.70 seconds

10.9 Atomic-knowing Gossip on knowledge structures with factual change

This modules contains a differessnt modeling of the gossip problem: Agent a knows the secret of b iff the atomic proposition $S_a b$ is true. Learning of secrets is then modeled as factual change.

Again we only consider the classic, static version of the gossip problem where N is a total graph and thus everyone can call everyone.

```
module SMCDEL.Examples.GossipS5 where

import SMCDEL.Language
import SMCDEL.Symbolic.S5
import Data.List ((\\))
```

Most functions below take a parameter n for the number of agents.

```
gossipers :: Int -> [Int]
gossipers n = [0..(n-1)]

hasSof :: Int -> Int -> Int -> Prp
hasSof n a b | a == b     = error "Let's not even talk about that."
             | otherwise = toEnum (n * a + b)

has :: Int -> Int -> Int -> Form
has n a b = PrpF (hasSof n a b)

expert :: Int -> Int -> Form
expert n a = Conj [ PrpF (hasSof n a b) | b <- gossipers n, a /= b ]

allExperts :: Int -> Form
allExperts n = Conj [ expert n a | a <- gossipers n ]

gossipInit :: Int -> KnowScene
gossipInit n = (KnS vocab law obs, actual) where
  vocab  = [ hasSof n i j | i <- gossipers n, j <- gossipers n, i /= j ]
  law    = boolBddOf $ Conj [ Neg $ PrpF $ hasSof n i j
                            | i <- gossipers n, j <- gossipers n, i /= j]
  obs    = [ (show i, []) | i <- gossipers n ]
  actual = [ ]

thisCallProp :: (Int,Int) -> Prp
thisCallProp (i,j) | i < j     = P (100 + 10*i + j)
                   | otherwise = error $ "wrong call: " ++ show (i,j)

call :: Int -> (Int,Int) -> Event
call n (a,b) = (callTrf n, [thisCallProp (a,b)])

callTrf :: Int -> KnowTransformer
callTrf n = KnTrf eventprops eventlaw changelaws eventobs where
  thisCallHappens (i,j) = PrpF $ thisCallProp (i,j)
  isInCallForm k = Disj $ [ thisCallHappens (i,k) | i <- gossipers n \\ [k], i < k ]
                       ++ [ thisCallHappens (k,j) | j <- gossipers n \\ [k], k < j ]
  allCalls = [ (i,j) | i <- gossipers n, j <- gossipers n, i < j ]
  eventprops = map thisCallProp allCalls
  eventlaw = simplify $
    Conj [ Disj (map thisCallHappens allCalls)
         -- some call must happen, but never two at the same time:
         , Neg $ Disj [ Conj [thisCallHappens c1, thisCallHappens c2]
                      | c1 <- allCalls, c2 <- allCalls \\ [c1] ] ]
  callPropsWith k = [ thisCallProp (i,k) | i <- gossipers n, i < k ]
                 ++ [ thisCallProp (k,j) | j <- gossipers n, k < j ]
  eventobs = [(show k, callPropsWith k) | k <- gossipers n]
  changelaws =
    [(hasSof n i j, boolBddOf $              -- after a call, i has the secret of j iff
        Disj [ has n i j                     -- i already knew j, or
             , Conj (map isInCallForm [i,j]) -- i and j are both in the call or
             , Conj [ isInCallForm i         -- i is in the call and there is some k in
                    , Disj [ Conj [ isInCallForm k, has n k j ] -- the call who knew j
                           | k <- gossipers n \\ [j] ] ]
             ])
    | i <- gossipers n, j <- gossipers n, i /= j ]

doCall :: KnowScene -> (Int,Int) -> KnowScene
doCall start (a,b) = start `update` call (length $ agentsOf start) (a,b)

after :: Int -> [(Int,Int)] -> KnowScene
after n = foldl doCall (gossipInit n)

isSuccess :: Int -> [(Int,Int)] -> Bool
isSuccess n cs = evalViaBdd (after n cs) (allExperts n)
```

```
whoKnowsMeta :: KnowScene -> [(Int,[(Int,String)])]
whoKnowsMeta scn = [ (k, map (meta k) [0..maxid] ) | k <- [0..maxid] ] where
  n = length (agentsOf scn)
  maxid = n - 1
  meta x y = (y, map (knowsAbout x y) [0..maxid])
  knowsAbout x y i
    | y == i = 'X'
    | evalViaBdd scn (      K (show x) $       PrpF (hasSof n y i)) = 'Y'
    | evalViaBdd scn (Neg $ K (show x) $ Neg $ PrpF (hasSof n y i)) = '?'
    | evalViaBdd scn (      K (show x) $ Neg $ PrpF (hasSof n y i)) = '_'
    | otherwise                                                      = 'E'

allSequs :: Int -> Int -> [ [(Int,Int)] ]
allSequs _ 0 = [ [] ]
allSequs n l = [ (i,j):rest | rest <- allSequs n (l-1), i <- gossipers n, j <- gossipers n,
     i < j ]
```

For example, among three agents, after a call the non-involved agent still knows who knows what:

```
λ> mapM_ print (whoKnowsMeta (after 3 [(0,1)]))
(0,[(0,"XY_"),(1,"YX_"),(2,"__X")])
(1,[(0,"XY_"),(1,"YX_"),(2,"__X")])
(2,[(0,"XY_"),(1,"YX_"),(2,"__X")])
```

This is different for four agents, where the two non-involved agents are unsure which call happened:

```
λ> mapM_ print (whoKnowsMeta (after 4 [(0,1)]))
(0,[(0,"XY__"),(1,"YX__"),(2,"__X_"),(3,"___X")])
(1,[(0,"XY__"),(1,"YX__"),(2,"__X_"),(3,"___X")])
(2,[(0,"X?_?"),(1,"?X_?"),(2,"__X_"),(3,"??_X")])
(3,[(0,"X??_"),(1,"?X?_"),(2,"??X_"),(3,"___X")])
```

10.10 Muddy Children

```
module SMCDEL.Examples.MuddyChildren where

import Data.List
import Data.Map.Strict (fromList)

import SMCDEL.Internal.Help (seteq)
import SMCDEL.Language
import SMCDEL.Symbolic.S5
import qualified SMCDEL.Symbolic.K
import qualified SMCDEL.Explicit.K
```

We now model the story of the muddy children which is known in many versions. See for example [Lit53], [Fag+95, p. 24-30] or [DHK07, 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 -> KnowScene
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 :: KnowScene
myMudScnInit = mudScnInit 3 3
```

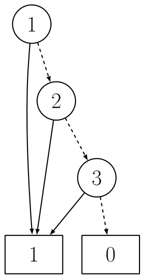
$$\left(\{p_1, p_2, p_3\}, \boxed{1}, \begin{matrix} \{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\} \end{matrix} \right), \{p_1, p_2, p_3\}$$

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.

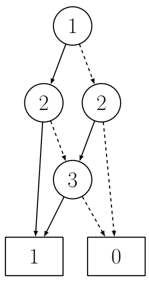
```
father :: Int -> Form
father n = Disj (map PrpF [P 1 .. P n])
mudScn0 :: KnowScene
mudScn0 = update myMudScnInit (father 3)
```

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

```
>>> evalViaBdd SMCDEL.Examples.MuddyChildren.mudScn0
(SMCDEL.Examples.MuddyChildren.nobodyknows 3)

True

0.05 seconds
```

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

```
mudScn1 :: KnowScene
mudScn1 = update mudScn0 (nobodyknows 3)
```

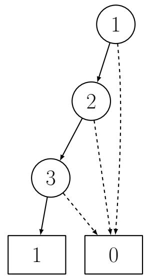
$\Big(\{p_1, p_2, p_3\},$  $, \begin{matrix} \{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\} \end{matrix} \Big), \{p_1, p_2, p_3\}$

```
>>> evalViaBdd SMCDEL.Examples.MuddyChildren.mudScn1
(SMCDEL.Examples.MuddyChildren.nobodyknows 3)

True

0.06 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 :: KnowScene
mudKns2 :: KnowStruct
mudScn2@(mudKns2,_) = update mudScn1 (nobodyknows 3)
```

$$\left(\{p_1, p_2, p_3\}, \right.$$



$$\left. , \begin{matrix} \{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\} \end{matrix} \right), \{p_1, p_2, p_3\}$$

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

```
True
```

0.05 seconds

```
>>> SMCDEL.Symbolic.S5.statesOf SMCDEL.Examples.MuddyChildren.mudKns2
```

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

0.05 seconds

We also make use of this example in the benchmarks in Section 11.

10.11 Building Muddy Children using Knowledge Transformers

We can also start modeling the muddy children story before they get muddy. The following initial knowledge structure has no atomic propositions. We then apply an `Event` in which each child can get muddy or not. Interestingly, this way of modeling the story does not need factual change. We do not change any facts, but rather introduce new ones.

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

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

buildResult :: KnowScene
buildResult = empty 3 `update` buildMC 3 3
```

This yields exactly the same knowledge structure:

```
>>> buildResult == mudScnInit 3 3
```

```
True
```

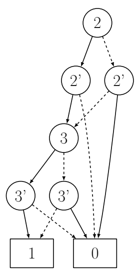
0.04 seconds

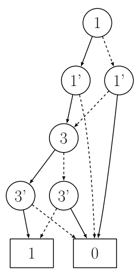
10.12 Muddy Children on general Kripke models

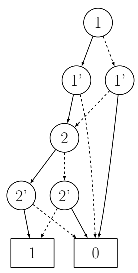
```
mudGenKrpInit :: Int -> Int -> SMCDEL.Explicit.K.PointedModel
mudGenKrpInit n m = (SMCDEL.Explicit.K.KrM $ fromList wlist, cur) where
  wlist = [ (w, (fromList (vals !! w), fromList $ relFor w)) | w <- ws ]
  ws    = [0..(2^n-1)]
  vals  = map sort (foldl buildTable [[]] [P k | k<- [1..n]])
  buildTable 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))
  (Just cur) = elemIndex curVal vals
  curVal = sort $ [(p,True) | p <- [P 1 .. P m]] ++ [(p,True) | p <- [P (m+1) .. P n]]

myMudGenKrpInit :: SMCDEL.Explicit.K.PointedModel
myMudGenKrpInit = mudGenKrpInit 3 3
```

10.13 Muddy Children on Belief Structures

```
mudBelScnInit :: Int -> Int -> SMCDEL.Symbolic.K.BelScene
mudBelScnInit n m = (SMCDEL.Symbolic.K.BlS vocab law obs, actual) where
  vocab  = [P 1 .. P n]
  law    = boolBddOf Top
  obs    = fromList [(show i, SMCDEL.Symbolic.K.allsamebdd $ delete (P i) vocab) | i <-
      [1..n]]
  actual = [P 1 .. P m]

myMudBelScnInit :: SMCDEL.Symbolic.K.BelScene
myMudBelScnInit = mudBelScnInit 3 3
```

$\Bigg(\Bigg(\{p_1, p_2, p_3\}, \boxed{1}, \Omega_1 =$ 

$, \Omega_2 =$ 

$, \Omega_3 =$ 

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

10.14 Muddy Planning

```
module SMCDEL.Examples.MuddyPlanning where

import SMCDEL.Examples.MuddyChildren
import SMCDEL.Language
import SMCDEL.Other.Planning
```

As a toy example, suppose we have three children, two of which are muddy. The available actions are public announcements that at least one child is muddy or that a specific child does not know their own state. Finally, suppose our goal is that child 1 knows whether it is muddy, while child 2 should not learn whether it is muddy. The following uses offlineSearch to find a solution.

```
toyPlan :: [OfflinePlan]
toyPlan = offlineSearch maxSteps start acts cons goal where
  maxSteps = 5 -- 2 would be enough
  start = mudScnInit 3 2
  acts = Disj [PrpF (P k) | k <- [1,2,3]] : [Neg $ Kw (show k) $ PrpF (P k) | k <- [1,2,3]]
```

```
  cons = [ Neg $ Kw "2" (PrpF $ P 2) ]
  goal = Kw "1" (PrpF $ P 1)
```

```
>>> toyPlan
```

```
[[Disj [PrpF (P 1),PrpF (P 2),PrpF (P 3)],Neg (Kw "2" (PrpF (P 2)))],[Disj [PrpF (
    P 1),PrpF (P 2),PrpF (P 3)],Neg (Kw "3" (PrpF (P 3))),Neg (Kw "2" (PrpF (P 2))
    )]]
```

```
0.04 seconds
```

10.15 Door Mat

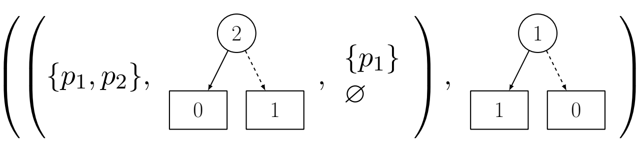
```
module SMCDEL.Examples.DoorMat where

import SMCDEL.Explicit.S5 as Exp hiding (announce)
import SMCDEL.Language
import SMCDEL.Symbolic.S5 hiding (announce)
import SMCDEL.Translations.S5
import SMCDEL.Other.Planning
```

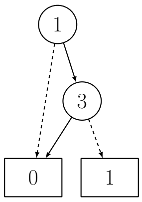
The running example from [Eng+17].

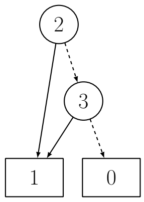
```
explain :: Prp -> String
explain (P 1) = "key-under-mat"
explain (P 2) = "bob-has-key"
explain (P k) = "prop-" ++ show k

dmStart :: MultipointedKnowScene
dmStart = (KnS voc law obs, cur) where
  voc = [ P 1, P 2 ]
  law = boolBddOf $ Neg $ PrpF $ P 2 -- it is common knowledge that Bob has no key
  obs = [ ("Anne",[P 1]), ("Bob",[ ]) ] -- Anne knows whether the key is under the mat
  cur = boolBddOf $ PrpF (P 1) -- actually, the key is under the mat

tryTake :: MultipointedEvent
tryTake = (KnTrf addprops addlaw changeLaw addObs, boolBddOf Top) where
  addprops    = [P 3]
  addlaw      = PrpF (P 3) `Equi` PrpF (P 1)
  changeLaw   = [ (P 1, boolBddOf $ Conj [PrpF (P 3) `Impl` Bot, Neg (PrpF (P 3)) `Impl`
      PrpF (P 1)])
                , (P 2, boolBddOf $ Conj [PrpF (P 3) `Impl` Top, Neg (PrpF (P 3)) `Impl`
                    PrpF (P 2)]) ]
  addObs      = [ ("Anne",[]), ("Bob",[P 3]) ]


tryTakeL :: Labelled MultipointedEvent
tryTakeL = ("tryTake", tryTake)

dmGoal :: Form
dmGoal = PrpF (P 2) -- Bob should get the key!

dmTask :: Task MultipointedKnowScene MultipointedEvent
dmTask = Task dmStart [("tryTake",tryTake)] dmGoal
```

Note how we use `P 3` to distinguish two possible events.

`dmStart` =

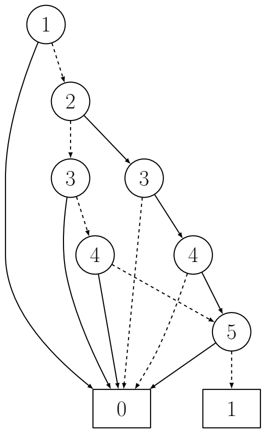


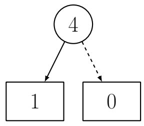
$$\texttt{tryTake} = \left(\left(\{p_3\},\ (p_3 \leftrightarrow p_1),\ p_1 := \right. \right.$$


$$, p_2 :=$$


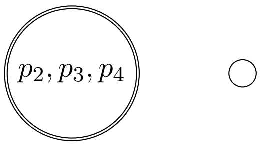
$$\left. \left. ,\ \begin{matrix} \varnothing \\ \{p_3\} \end{matrix} \right),\ \boxed{1} \right)$$

```
dmResult :: MultipointedKnowScene
dmResult =  dmStart `update` tryTake

dmResultKripke :: MultipointedModelS5
dmResultKripke = knsToKripkeMulti dmResult
```

$$\texttt{dmResult} = \left(\left(\{p_1, p_2, p_3, p_4, p_5\}, \right. \right.$$


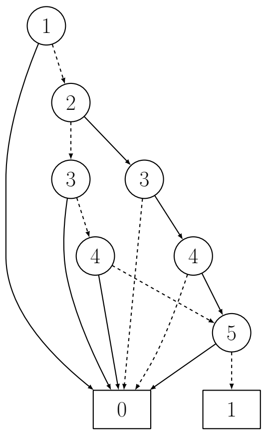
$$\left. ,\ \begin{matrix} \{p_4\} \\ \{p_3\} \end{matrix} \right),$$


$$\Big)$$

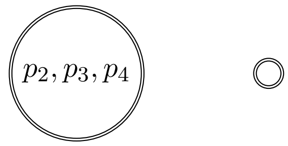
$$\texttt{knsToKripkeMulti dmResult} =$$


In the last model Bob got the key in the actual world. But before the action he did not know that this plan would succced:

```
dmResultBob :: MultipointedKnowScene
dmResultBob = (dmStart `asSeenBy` "Bob") `update` tryTake

dmResultBobKripke :: MultipointedModelS5
dmResultBobKripke = knsToKripkeMulti dmResultBob
```

$$\texttt{dmResultBob} = \left(\left(\{p_1, p_2, p_3, p_4, p_5\}, \right. \right.$$


$$\left. \left. ,\ \begin{matrix} \{p_4\} \\ \{p_3\} \end{matrix} \right),\ \boxed{1} \right)$$

dmResultBobKripke = 

```
>>> snd ((dmStart `asSeenBy` "Bob") `asSeenBy` "Anne")

Top

0.04 seconds
```

```
>>> reachesOn (Do "tryTake" tryTake (Check dmGoal Stop)) dmGoal dmStart

True

0.04 seconds
```

```
dm :: Task MultipointedKnowScene MultipointedEvent
dm = Task dmStart [ tryTakeL ] dmGoal

dmCoop :: CoopTask MultipointedKnowScene MultipointedEvent
dmCoop = CoopTask dmStart [("Bob",tryTakeL)] dmGoal
```

```
>>> findPlan 3 dm

[Do "tryTake" (KnTrf [P 3] (Equi (PrpF (P 3)) (PrpF (P 1))) [(P 1,Var 1 (Var 3 Bot
     Top) Bot),(P 2,Var 2 Top (Var 3 Top Bot))] [("Anne",[]),("Bob",[P 3])],Top)
   Stop]

0.04 seconds
```

However, this is not an implicitly coordinated (ic) plan:

```
>>> icSolves dmCoop [("Bob",tryTakeL)]

False

0.04 seconds
```

If Anne were the one to execute tryTake, then it would be an ic plan:

```
>>> icSolves (CoopTask dmStart [("Anne",tryTakeL)] dmGoal) [("Anne",tryTakeL)]

True

0.05 seconds
```

An implicitly coordinated plan also needs an announce action.

```
announce :: MultipointedEvent
announce = ( KnTrf [] (PrpF (P 1)) [] [ ("Anne",[]), ("Bob",[]) ]
           , boolBddOf Top )

announce' :: Labelled MultipointedEvent
announce' = ("announce", announce)

dmCoop2 :: CoopTask MultipointedKnowScene MultipointedEvent
dmCoop2 = CoopTask dmStart [ ("Bob" , tryTakeL )
                           , ("Anne", announce') ] dmGoal

dmPlan2 :: ICPlan MultipointedEvent
dmPlan2 = [ ("Anne",announce'), ("Bob",tryTakeL) ]
```

```
>>> icSolves dmCoop2 dmPlan2
True
0.04 seconds
```

10.16 Letter Passing

```
module SMCDEL.Examples.LetterPassing where

import Data.List (sort)

import SMCDEL.Language
import SMCDEL.Symbolic.S5 hiding (announce)
import SMCDEL.Translations.S5 (booloutof)
import SMCDEL.Other.Planning
```

This is Example 6 from [Eng+17].

```
explain :: Prp -> String
explain (P k) | odd k     = "at " ++ show ((k + 1) `div` 2)
              | otherwise = "for " ++ show (k `div` 2)

atP, forP :: Int -> Prp
atP  k = P $ k*2 - 1
forP k = P $ k*2

at, for :: Int -> Form
at  = PrpF . atP
for = PrpF . forP

letterStart :: MultipointedKnowScene
letterStart = (KnS voc law obs, cur) where
  voc = sort $ map atP [1..3] ++ map forP [1..3]
  law = boolBddOf $ Conj $
    -- letter must be at someone, but not two:
       [ Disj (map at [1..3]) ]
    ++ [ Neg $ Conj [at i, at j] | i <- [1..3], j <- [1..3], i /= j ]
    -- letter must be for someone, but not two:
    ++ [ Disj (map for [1..3]) ]
    ++ [ Neg $ Conj [for i, for j] | i <- [1..3], j <- [1..3], i /= j ]
    -- make it common knowledge that the letter is at 1, but not adressed to 1:
    ++ [ at 1, Neg (for 1) ]
  obs = [ ("1",[atP 1, forP 1, forP 2, forP 3])
        , ("2",[atP 2])
        , ("3",[atP 3]) ]
  cur = booloutof [atP 1, forP 3] voc

letterPass :: Int -> Int -> Int -> Labelled MultipointedEvent
letterPass n i j = (label, (KnTrf addprops addlaw changeLaw addObs, boolBddOf Top)) where
  offset       = n*2 -- ensure addprops does not overlap with vocabOf (letterStartFor n)
  label        = show i ++ "->" ++ show j
  addprops     = map P [(offset + 1)..(offset + n)]
  addlaw       = Conj $ at i : [ PrpF (P (offset + k)) `Equi` for k | k <- [1..n] ]
  -- publicly pass the letter from i to j:
  changeLaw    = [ (atP i, boolBddOf Bot), (atP j, boolBddOf Top) ]
  -- privately tell j who the receiver is:
  addObs       = [ (show k, if k == j then addprops else []) | k <- [1..n] ]

letterGoal :: Form
letterGoal = Conj [ for i `Impl` at i | i <- [1,2,3] ]

letter :: CoopTask MultipointedKnowScene MultipointedEvent
letter = CoopTask letterStart actions letterGoal where
  actions = [ (show i, letterPass 3 i j) | (i,j) <- [(1,2),(2,1),(2,3),(3,2)] ]
```

With a search depth of 2 we find this plan:

```
>>> ppICPlan (head (findSequentialIcPlan 2 letter))

"1:1->2; 2:2->3."

0.05 seconds
```

Note that this is depth-first search which can lead to unnecessarily long plans:

```
>>> ppICPlan (head (findSequentialIcPlan 4 letter))

"1:1->2; 2:2->1; 1:1->2; 2:2->3."

0.05 seconds
```

We can also use breadth-first search:

```
>>> fmap ppICPlan (findSequentialIcPlanBFS 2 letter)

Just "1:1->2; 2:2->3."

0.05 seconds
```

We now generalize the letter example for n agents.

```
letterStartFor :: Int -> MultipointedKnowScene
letterStartFor n = (KnS voc law obs, cur) where
  voc = sort $ map atP [1..n] ++ map forP [1..n]
  law = boolBddOf $ Conj $
    -- letter must be at someone, but not two:
       [ Disj (map at [1..n]) ]
    ++ [ Neg $ Conj [at i, at j] | i <-[1..n], j <- [1..n], i /= j ]
    -- letter must be for someone, but not two:
    ++ [ Disj (map for [1..n]) ]
    ++ [ Neg $ Conj [for i, for j] | i <-[1..n], j <- [1..n], i /= j ]
    -- make it common knowledge that letter is at 1, but not adressed to 1:
    ++ [ at 1, Neg (for 1) ]
  obs = ("1", atP 1 : map forP [1..n]) : [ (show k, [atP k]) | k <- [2..n] ]
  cur = booloutof [atP 1, forP n] voc

letterLine :: Int -> CoopTask MultipointedKnowScene MultipointedEvent
letterLine n = CoopTask (letterStartFor n) actions goal where
  actions = [ (show i, letterPass n i j) | i <- [1..n], j <- [1..n], abs(i-j) == 1 ]
  goal = Conj [ for i `Impl` at i | i <- [1..n] ]
```

10.17 Hundred Prisoners and a Lightbulb

```
module SMCDEL.Examples.Prisoners where

import Data.HasCacBDD hiding (Top,Bot)

import SMCDEL.Explicit.S5
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Symbolic.S5
```

The story, from [DEW10]:

> "A group of 100 prisoners, all together in the prison dining area, are told that they will be all put in isolation cells and then will be interrogated one by one in a room containing a light with an on/off switch. The prisoners may communicate with one another by toggling the light-switch (and that is the only way in which they can communicate). The light is initially switched off. There is no fixed order of interrogation, or fixed interval between interrogations, and the same prisoner may be interrogated again at any stage. When

interrogated, a prisoner can either do nothing, or toggle the light-switch, or announce that all prisoners have been interrogated. If that announcement is true, the prisoners will (all) be set free, but if it is false, they will all be executed. While still in the dining room, and before the prisoners go to their isolation cells, can the prisoners agree on a proto- col that will set them free (assuming that at any stage every prisoner will be interrogated again sometime)?"

The solution: Let one agent be a "counter". He turns off the light whenever he enters the room and counts until this has happend 99 times. Then he knows that everyone has been in the room, because: Everyone else turns on the light the first time they find it turned off when they are brought into the room and does not do anything else afterwards, no matter how often they are brought to the room.

We first try to implement it with three agents and let the first one be the counter in the standard solution.

We use four propositions: p says that the light is on and p_1 to p_3 say that the agents have been in the room, respectively.

The goal is $\bigvee_i K_i(p_1 \wedge p_2 \wedge p_3)$.

Now, calling an agent i into the room is an event with multiple consequences:

- the agent learns P 0, i.e. whether the light is on or off
- the agent can set the new value of P 0
- P i for that agent
- the other agents no longer know the value of P 0 and should consider it possible that anyone else has been in the room.

10.17.1 Explicit Version

We first give an explicit, Kripke model implementation, similar to the DEMO version in [DEW10]. In particular, we only model the knowledge of the counter, ignoring what the other agents might know. We also do not include a "nothing happens" event but instead model the synchronous version only. Both of these restrictions help to keep the Kripke model small.

```
n :: Int
n = 3

light :: Form
light = PrpF (P 0)

-- P 0 -- light is on
-- P i -- agent i has been in the room (and switched on the light)
-- agents: 1 is the counter, 2 and 3 are the others

prisonExpStart :: KripkeModelS5
prisonExpStart =
  KrMS5
    [1]
    [ ("1",[[1]]) ]
    [ (1, [(P k,False) | k <- [0..n] ] ) ]

prisonGoal :: Form
prisonGoal = K "1" $ Conj [ PrpF $ P k | k <- [1..n] ]

prisonAction :: ActionModelS5
prisonAction = ActMS5 actions actRels where
  p = PrpF . P
  actions =
    [ (0, (p 0      , [(P 0, Bot), (P 1, Top)]) ) -- interview 1 with light on
    , (1, (Neg (p 0), [             (P 1, Top)]) ) -- interview 1 with light off
    ]
    ++
```

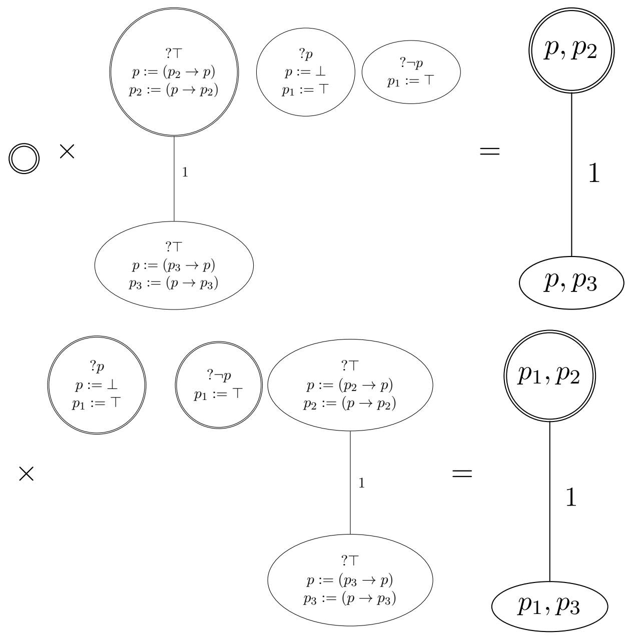
```
    [ (k, (Top, [(P 0, p k ‘Impl‘ p 0), (P k, p 0 ‘Impl‘ p k)]) ) | k <- [2..n] ] --
        interview k
  actRels = [ ("1", [[0],[1],[2..n]]) ]

prisonInterview :: Int -> MultipointedActionModelS5
prisonInterview 1 = (prisonAction, [0,1])
prisonInterview k = (prisonAction, [k])
```

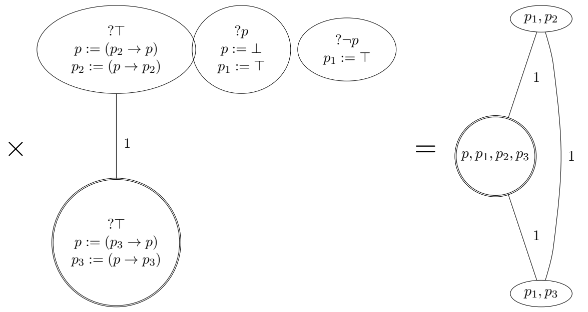
Interlude: **Story telling**. We define a general function to execute a sequence of updates on a given starting point and optimizing the intermediate steps. We also define a function to LaTeXthe whole story.

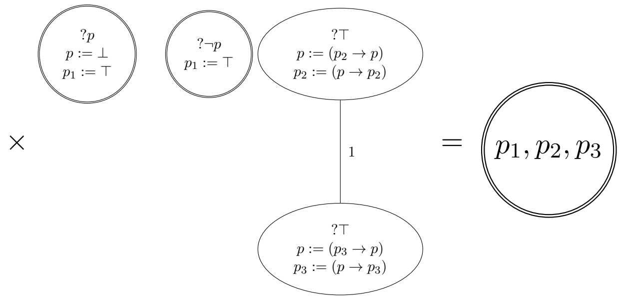
```
data Story a b = Story a [b]

endOf :: (Update a b, Optimizable a) => Story a b -> a
endOf (Story start actions) =
  foldl (\cur a -> optimize (vocabOf start) $ cur ‘update‘ a) start actions

instance (Update a b, Optimizable a, TexAble a, TexAble b) => TexAble (Story a b) where
  tex (Story start actions) = adjust (tex start) ++ loop start actions where
    adjust thing = " \\raisebox{-.5\\height}{ \\begin{adjustbox}{max height=4cm, max width
        =0.3\\linewidth} $ " ++ thing ++ " $ \\end{adjustbox} } "
    loop _       []      = ""
    loop current (a:as) =
      let
        new = optimize (vocabOf start) $ current ‘update‘ a
      in
        " \\times " ++ adjust (tex a) ++ " = " ++ adjust (tex new) ++ "\\] \\[ " ++ loop
            new as
```

The story of the prisoners could then for example be the following, in which agents 2, 1, 3 and 1 are interviewed in this order. We show only generated submodels here. The full product models are actually much larger — even more so, if we would also keep track of the knowledge of all other agents beside 1.

```
prisonExpStory :: Story PointedModelS5 MultipointedActionModelS5
prisonExpStory = Story (prisonExpStart,1) (map prisonInterview [2,1,3,1])
```







And indeed we have:

```
>>> endOf prisonExpStory ‘isTrue‘ prisonGoal

True

0.00 seconds
```

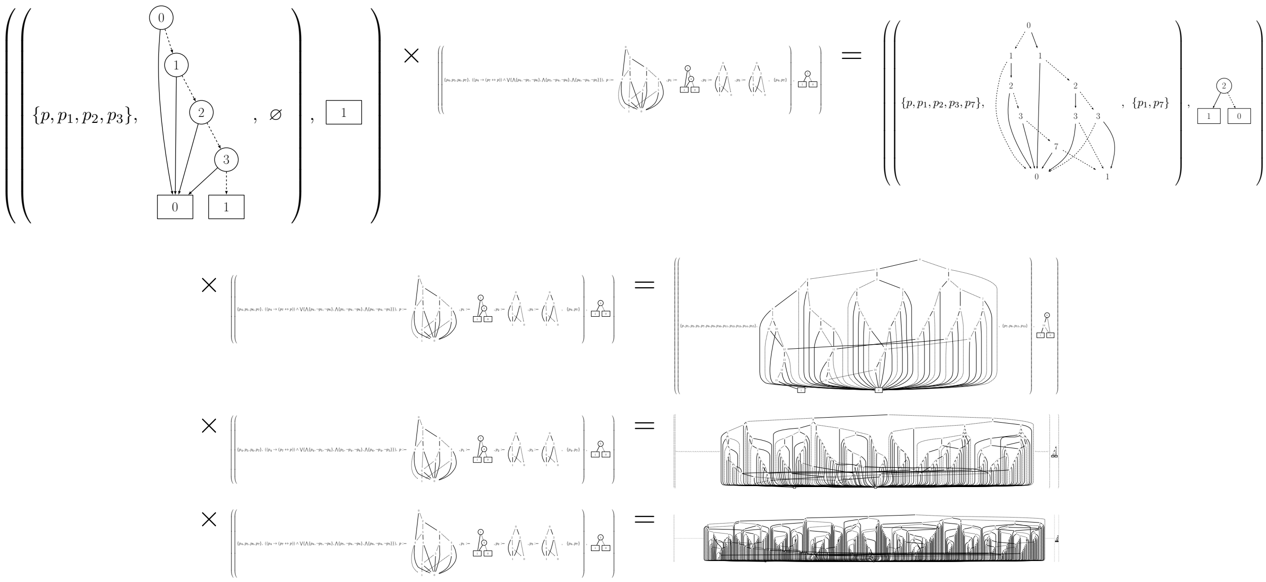
10.17.2 Symbolic Version

In the initial structure the light is off and nobody has been interviewed. This is the actual and the only state, thus common knowledge.

```
prisonSymStart :: MultipointedKnowScene
prisonSymStart = (KnS (map P [0..n]) law obs, actualStatesBdd) where
  law    = boolBddOf (Conj (Neg light : [ Neg $ wasInterviewed k | k <- [1..n] ]))
  obs    = [ ("1", []) ]
  actualStatesBdd = top

wasInterviewed, isNowInterviewed :: Int -> Form
wasInterviewed     = PrpF . P
isNowInterviewed k = PrpF (P (k + n))

lightSeenByOne :: Form
lightSeenByOne  = PrpF (P (1 + (2*n)))

prisonSymEvent :: KnowTransformer
prisonSymEvent = KnTrf -- agent 1 is interviewed
  (map P $ [ k+n | k <- [1..n] ] ++ [1+(2*n)] ) -- distinguish events
  (Conj [ isNowInterviewed 1 ‘Impl‘ (lightSeenByOne ‘Equi‘ light)
        , Disj [ Conj $ isNowInterviewed k : [Neg $ isNowInterviewed l | l <- [1..n], l /=
            k ] | k <- [1..n]]
        ] )
  -- light might be switched and visits of the agents might be recorded
  ( [ (P 0, boolBddOf $
          Conj $ isNowInterviewed 1 ‘Impl‘ Bot -- 1 turns off the light
               : concat [ [ Conj [Neg $ wasInterviewed k, isNowInterviewed k] ‘Impl‘ Top
                          , Conj [      wasInterviewed k, isNowInterviewed k] ‘Impl‘ light
                            ]
                        | k <- [2..n] ])
  , (P 1, boolBddOf $ Disj [wasInterviewed 1, Conj [           isNowInterviewed 1]])
  ]
  ++
  [ (P k, boolBddOf $ Disj [wasInterviewed k, Conj [Neg light, isNowInterviewed k]])
```

```
  | k <- [2..n] ] )
  -- agent 1 observes whether they are interviewed, and if so, also observes the light
  [ ("1", let (PrpF px, PrpF py) = (isNowInterviewed 1, lightSeenByOne) in [px, py]) ]

prisonSymInterview :: Int -> MultipointedEvent
prisonSymInterview k = (prisonSymEvent, boolBddOf (isNowInterviewed k))

prisonSymStory :: Story MultipointedKnowScene MultipointedEvent
prisonSymStory = Story prisonSymStart (map prisonSymInterview [2,1,3,1])
```

Thanks to the optimization we can still draw all BDDs in the structures, but they are not very readable.



Finally, also in the symbolic version of the story we reach the goal:

```
>>> endOf prisonSymStory `isTrue` prisonGoal

True

0.06 seconds
```

10.18 Russian Cards

```
{-# LANGUAGE FlexibleInstances #-}

module SMCDEL.Examples.RussianCards where

import Control.Monad (replicateM)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List ((\\),delete,intersect,nub,sort)
import Data.Map.Strict (fromList)

import SMCDEL.Internal.Help (powerset)
import SMCDEL.Language
import SMCDEL.Other.Planning
import SMCDEL.Symbolic.S5
import qualified SMCDEL.Symbolic.K as K
```

As another case study we analyze the Russian Cards problem. One of its first (dynamic epistemic) logical treatments was [Dit03] and the problem has since gained notable attention as an intuitive example of information-theoretically (in contrast to computationally) secure cryptography [Cor+15; DG14].

The basic version of the Russian Cards 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))

hasHand :: Agent -> [Int] -> Form
hasHand i ns = Conj $ map (i `hasCard`) ns

rcExplain :: Prp -> String
rcExplain (P k) = (rcPlayers !! i) ++ " has card " ++ show n where (n,i) = divMod k 3
```

```
>>> hasCard carol 5

PrpF (P 17)

0.00 seconds
```

```
>>> hasHand bob [1,3,5]

Conj [PrpF (P 4),PrpF (P 10),PrpF (P 16)]

0.00 seconds
```

```
>>> ppFormWith rcExplain (hasHand bob [1,3,5])

"(Bob has card 1 & Bob has card 3 & Bob has card 5)"

0.00 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 ]
```

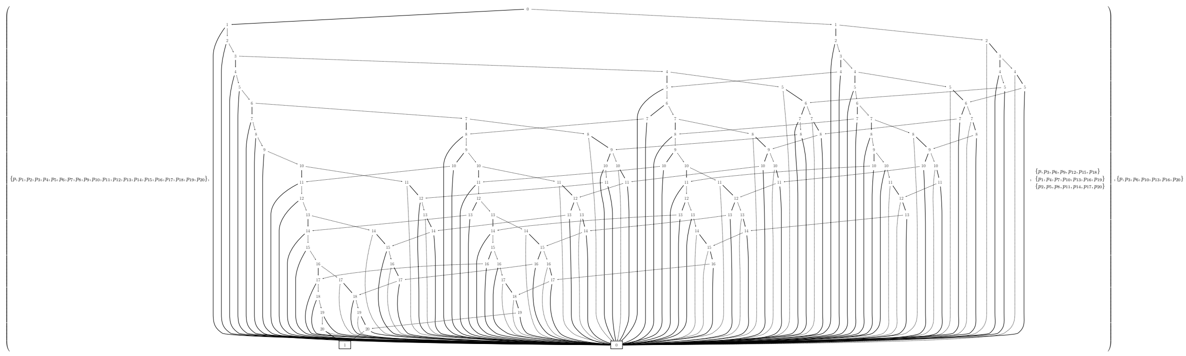
Moreover, Alice, Bob and Carol should get at least three, three and one card, respectively. As there are only seven cards in total this already implies that they can not have more.

```
distribute331 :: Form
distribute331 = Conj [ aliceAtLeastThree, bobAtLeastThree, carolAtLeastOne ] where
  triples = [ [x, y, z] | x <- rcCards, y <- delete x rcCards, z <- rcCards \\ [x,y] ]
  aliceAtLeastThree = Disj [ Conj (map (alice `hasCard`) t) | t <- triples ]
  bobAtLeastThree   = Disj [ Conj (map (bob   `hasCard`) t) | t <- triples ]
  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 :: KnowScene
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:



10.18.1 Verifying a five-hand protocol

Many different solutions for Russian Cards exist. Here we will focus on the following so-called five-hands protocols (and their extensions with six or seven hands) which are also used in [Dit+06]. 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 which cards Alice and Bob have, independent of which card Carol 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 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.", i.e. we prefix the statements with the knowledge operators of the speaker.

```haskell
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 knowing-whether `Kw` for the first two but plain `K` for the last condition.

```haskell
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.

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

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

```haskell
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.

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

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

```haskell
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.

```haskell
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 is quick.

```
>>> rcAllChecks

True

0.05 seconds
```

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 [Dit+06] needs 4 seconds to check one protocol.

10.18.2 Finding all five/six/seven-hands solutions

We can not just verify but also *find* all protocols based on a set of five, six or seven hands. To make the problem feasible we use a 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 <- filter (> x) rcCards, z <-filter (> y)
    rcCards ]

pickHandsNoCrossing :: [ [Int] ] -> Int -> [ [ [Int] ] ]
pickHandsNoCrossing _ 0 = [ [ [ ] ] ]
pickHandsNoCrossing unused 1 = [ [h] | h <- unused ]
pickHandsNoCrossing unused n = concat [ [ h:hs | hs <- pickHandsNoCrossing (myfilter h
    unused) (n-1) ]  | h <- unused ] where
  myfilter h = filter (\xs -> length (h ‘intersect‘ xs) < 2 && h < xs) -- do not allow
      intersection > 2
```

The last line includes two important restrictions to the set of possible lists of hands that we will consider. First, Proposition 32 in [Dit03] 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, safeHandLists :: [ [ [Int] ] ]
allHandLists = concatMap (pickHandsNoCrossing possibleHands) [5,6,7]
safeHandLists = sort (filter checkSet allHandLists)
```

```
>>> length allHandLists

1290

0.02 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 [Dit03, Figure 3] where it was manually generated.

```
>>> length safeHandLists

102

0.38 seconds
```

```
>>> head safeHandLists

[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6]]

0.38 seconds
```

```
>>> last safeHandLists

[[0,1,2],[0,5,6],[1,4,6],[2,3,6],[3,4,5]]

0.41 seconds
```

10.18.3 Protocol synthesis

We now adopt a more general perspective considered in [Eng+17]. Taking the perspective of Alice, we want to *find* a plan. Fix that Alice has $\{0, 1, 2\}$ and that she will announce five hands, including this one. Hence she has to pick four other hands of three cards each, i.e. she has to choose among

$$\binom{\binom{7}{3} - 1}{4} = \binom{34}{4} = 46376$$

many possible actions.

```
alicesActions :: [Form]
alicesActions = [ Disj $ map (alice `hasHand`) ([0,1,2]:otherHands) | otherHands <-
    handLists ] where
  handLists :: [ [ [Int] ] ]
  handLists = pickHands (delete [0,1,2] possibleHands) 4
  pickHands :: [ [Int] ] -> Int -> [ [ [Int] ] ]
  pickHands _     0 = [ [ [ ] ] ]
  pickHands hands 1 = [ [ h   ] | h <- hands ]
  pickHands hands n = [ h:hs    | h <- hands, hs <- pickHands (filter (h <) hands) (n-1) ]
```

```
>>> 46376 == length alicesActions

True

0.00 seconds
```

For example, the first action Alice will consider is this:

```
>>> ppFormWith rcExplain (head alicesActions)

"((Alice has card 0 & Alice has card 1 & Alice has card 2) | (Alice has card 0 &
    Alice has card 1 & Alice has card 3) | (Alice has card 0 & Alice has card 1 &
    Alice has card 4) | (Alice has card 0 & Alice has card 1 & Alice has card 5) |
     (Alice has card 0 & Alice has card 1 & Alice has card 6))"

0.00 seconds
```

Alice does not know which card Bob has, but of course she knows that he cannot have one of her cards. Hence Alice considers four possibilities for his action of saying which card Carol has.

```
bobsActions :: [Form]
bobsActions = [ carol `hasCard` n | n <- reverse [4..6] ]
```

```
rcSolutions :: [ [Form] ]
rcSolutions = [ [a, b] | a <- alicesActions, b <- bobsActions, testPlan a b ] where
  testPlan :: Form -> Form -> Bool
  testPlan aSays bSays = all (evalViaBdd rusSCN)
    -- NOTE: increasing checks are faster than one big conjunction!
    [ aSays
    , PubAnnounce aSays bKnowsAs
    , PubAnnounce aSays cIgnorant
    , PubAnnounce aSays bSays
    , PubAnnounce aSays (PubAnnounce bSays aKnowsBs)
    , PubAnnounce aSays (PubAnnounce bSays (Ck [alice,bob] $ Conj [cIgnorant,aKnowsBs,
      bKnowsAs])) ]
```

10.18.4 Via Planning

We can also model the problem using the planning definitions from subsection 9.1.

```
rcPlan :: OfflinePlan
rcPlan = [ aAnnounce, bAnnounce ]

rcGoal :: Form
rcGoal = Conj [ aKnowsBs
              , bKnowsAs
              , Ck [alice,bob] (Conj [aKnowsBs, bKnowsAs])
              , Ck [alice,bob,carol] cIgnorant ]
```

```
>>> reachesOn rcPlan rcGoal rusSCN
True
0.05 seconds
```

To find solutions we can also use the search function from section 9.

```
rcSolutionsViaPlanning :: [OfflinePlan]
rcSolutionsViaPlanning = offlineSearch maxSteps start actions constraints goal where
  maxSteps    = 2 -- We need two steps!
  start       = rusSCNfor (3,3,1)
  actions     = alicesActions ++ bobsActions
  constraints = [cIgnorant,bKnowsAs]
  goal        = Conj [aKnowsBs, bKnowsAs]
```

In fact, in both ways we find the same solution.

```
>>> map (ppFormWith rcExplain) (head rcSolutionsViaPlanning)
["((Alice has card 0 & Alice has card 1 & Alice has card 2) | (Alice has card 0 &
    Alice has card 1 & Alice has card 3) | (Alice has card 0 & Alice has card 1 &
    Alice has card 4) | (Alice has card 0 & Alice has card 1 & Alice has card 5) |
     (Alice has card 2 & Alice has card 3 & Alice has card 4))","Carol has card 6"
    ]
2.78 seconds
```

```
>>> map (ppFormWith rcExplain) (head rcSolutions)
["((Alice has card 0 & Alice has card 1 & Alice has card 2) | (Alice has card 0 &
    Alice has card 1 & Alice has card 3) | (Alice has card 0 & Alice has card 1 &
    Alice has card 4) | (Alice has card 0 & Alice has card 1 & Alice has card 5) |
     (Alice has card 2 & Alice has card 3 & Alice has card 4))","Carol has card 6"
    ]
0.06 seconds
```

```
>>> head rcSolutionsViaPlanning == head rcSolutions
True
2.81 seconds
```

We could in principle use only two propositions instead of three and encode that Carol has a card by saying that the others don't have it. Concretely, consider replacing c_n with $\neg a_n \land \neg b_n$. However, this makes it impossible to capture what Carol knows with observational variables. The more general belief structures from Section 6 provide a solution for this, see subsection 10.20.

10.19 Generalized Russian Cards

Fun fact: Even if we want to use more or less than 7 cards, we do not have to modify the function `hasCard`.

```
type RusCardProblem = (Int,Int,Int)

distribute :: RusCardProblem -> Form
distribute (na,nb,nc) = Conj [ alice `hasAtLeast` na
                             , bob   `hasAtLeast` nb
                             , carol `hasAtLeast` nc ] where
  n = na + nb + nc
  hasAtLeast :: Agent -> Int -> Form
  hasAtLeast _ 0 = Top
  hasAtLeast i 1 = Disj [ i `hasCard` k | k <- nCards n ]
  hasAtLeast i k = Disj [ Conj (map (i `hasCard`) (sort set))
                        | set <- powerset (nCards n), length set == k ]

nCards :: Int -> [Int]
nCards n = [0..(n-1)]

nCardsGiven, nCardsUnique :: Int -> Form
nCardsGiven  n = Conj [ Disj [ i `hasCard` k | i <- rcPlayers ] | k <- nCards n ]
nCardsUnique n = Conj [ Neg $ isDouble k | k <- nCards n ] where
  isDouble k = Disj [ Conj [ x `hasCard` k, y `hasCard` k ] | x <- rcPlayers, y <-
      rcPlayers, x/=y, x < y ]

rusSCNfor :: RusCardProblem -> KnowScene
rusSCNfor (na,nb,nc) = (KnS props law [ (i, obs i) | i <- rcPlayers ], defaultDeal) where
  n = na + nb + nc
  props   = [ P k | k <-[0..((length rcPlayers * n)-1)] ]
  law = boolBddOf $ Conj [ nCardsGiven n, nCardsUnique n, distribute (na,nb,nc)  ]
  obs i = [ P (3 * k + rcNumOf i) | k<-[0..6] ]
  defaultDeal =  [ let (PrpF p) = i `hasCard` k in p | i <- rcPlayers, k <- cardsFor i ]
  cardsFor "Alice" = [0..(na-1)]
  cardsFor "Bob"   = [na..(na+nb-1)]
  cardsFor "Carol" = [(na+nb)..(na+nb+nc-1)]
  cardsFor _       = error "Who is that?"
```

For the following cases it is unknown whether a multi-announcement solution exists. (It *is* known that no two-announcement solution exists.)

- (2,2,1)
- (3,2,1)
- (3,3,2)

Note also: (4,4,2) discussed in [DS11].

```
possibleHandsN :: Int -> Int -> [[Int]]
possibleHandsN n na = filter alldiff $ nub $ map sort $ replicateM na (nCards n) where
  alldiff [] = True
  alldiff (x:xs) = x `notElem` xs && alldiff xs

allHandListsN :: Int -> Int -> [ [ [Int] ] ]
allHandListsN n na = concatMap (pickHandsNoCrossing (possibleHandsN n na)) [5,6,7] -- FIXME
     how to adapt the number of hands for larger n?
```

Note that we still use the same pickHandsNoDouble. This is a problem because of the intersection constraint! The only should have striclty less than `na - nc` cards in common!

```
aKnowsBsN, bKnowsAsN, cIgnorantN :: Int -> Form
aKnowsBsN n = Conj [ alice `Kw` (bob `hasCard` k) | k <- nCards n ]
bKnowsAsN n = Conj [ bob `Kw` (alice `hasCard` k) | k <- nCards n ]
cIgnorantN n = Conj $ concat [ [ Neg $ K carol $ alice `hasCard` i
                              , Neg $ K carol $ bob   `hasCard` i ] | i <- nCards n ]
```

```haskell
checkSetFor :: RusCardProblem -> [[Int]] -> Bool
checkSetFor (na,nb,nc) set = reachesOn plan rcGoal (rusSCNfor (na,nb,nc)) where
  n = na + nb + nc
  aliceSays = K alice (Disj [ Conj $ map (alice `hasCard`) h | h <- set ])
  bobSays = K bob (carol `hasCard` last (nCards n))
  plan = [ aliceSays, bobSays ]

checkHandsFor :: RusCardProblem -> [ ( [[Int]], Bool) ]
checkHandsFor (na,nb,nc) = map (\hs -> (hs, checkSetFor (na,nb,nc) hs)) (allHandListsN n na
    ) where
  n = na + nb + nc

allCasesUpTo :: Int -> [RusCardProblem]
allCasesUpTo bound = [ (na,nb,nc) | na <- [1..bound]
                                  , nb <- [1..(bound-na)]
                                  , nc <- [1..(bound-(na+nb))]
                                  -- these restrictions are only proven
                                  -- for two announcement plans!
                                  , nc < (na - 1)
                                  , nc < nb ]
```

10.20 Russian Cards on Belief Structures with Less Atoms

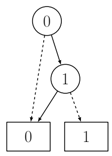
```haskell
dontChange :: [Form] -> K.RelBDD
dontChange fs = conSet <$> sequence [ equ <$> K.mvBdd b <*> K.cpBdd b | b <- map boolBddOf
    fs ]

noDoubles :: Int -> Form
noDoubles n = Neg $ Disj [ notDouble k | k <- nCards n ] where
  notDouble k = Conj [alice `hasCard` k, bob `hasCard` k]

rusBelScnfor :: RusCardProblem -> K.BelScene
rusBelScnfor (na,nb,nc) = (K.BlS props law (fromList [ (i, obsbdd i) | i <- rcPlayers ]),
    defaultDeal) where
  n = na + nb + nc
  props   = [ P k | k <-[0..((2 * n)-1)] ]
  law = boolBddOf $ Conj [ noDoubles n, distribute (na,nb,nc)  ]
  obsbdd "Alice" = dontChange [ PrpF (P $ 2*k) | k <- [0..(n-1)] ]
  obsbdd "Bob"   = dontChange [ PrpF (P $ (2*k) + 1) | k <- [0..(n-1)] ]
  obsbdd "Carol" = dontChange [ Disj [PrpF (P $ 2*k), PrpF (P $ (2*k) + 1)] | k <- [0..(n
      -1)] ]
  obsbdd _        = error "Unkown Agent"
  defaultDeal =  [ let (PrpF p) = i `hasCard` k in p | i <- [alice,bob], k <- cardsFor i ]
      where
    cardsFor "Alice" = [0..(na-1)]
    cardsFor "Bob"   = [na..(na+nb-1)]
    cardsFor "Carol" = [(na+nb)..(na+nb+nc-1)]
    cardsFor _       = error "Unkown Agent"
```

10.21 The Sally-Anne false belief task

```haskell
module SMCDEL.Examples.SallyAnne where

import Data.Map.Strict (fromList)

import SMCDEL.Language
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.S5 (boolBddOf)
```

The vocabulary is $V = \{p, t\}$ where p means that Sally is in the room and t that the marble is in the basket. The initial scene is $(\mathcal{F}_0, s_0) = ((\{p,t\}, (p \land \neg t), \top, \top), \{p\})$ where the last two components are Ω_S and Ω_A.

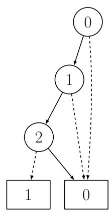
```haskell
pp, qq, tt :: Prp
pp = P 0
tt = P 1
qq = P 7 -- this number does not matter

sallyInit :: BelScene
sallyInit = (BlS [pp, tt] law obs, actual) where
  law    = boolBddOf $ Conj [PrpF pp, Neg (PrpF tt)]
  obs    = fromList [ ("Sally", totalRelBdd), ("Anne", totalRelBdd) ]
  actual = [pp]
```

$\Big(\Big(\{p, p_1\},$  $, \Omega_{\text{Anne}} = \boxed{1}, \Omega_{\text{Sally}} = \boxed{1}\Big), \{p\}\Big)$

The sequence of events is:
Sally puts the marble in the basket: $(\mathcal{X}_1 = (\varnothing, \top, \{t\}, \theta_-(t) = \top, \top, \top), \varnothing)$,

```haskell
sallyPutsMarbleInBasket :: Event
sallyPutsMarbleInBasket = (Trf [] Top
  (fromList [ (tt,boolBddOf Top) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate1 :: BelScene
sallyIntermediate1 = sallyInit `update` sallyPutsMarbleInBasket
```

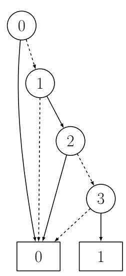
$$\Big(\Big(\varnothing, \top, \{p_1\}, p_1 := \top, \Omega^+_{\text{Anne}} = \boxed{1}, \Omega^+_{\text{Sally}} = \boxed{1}\Big), \varnothing\Big)$$

$\Big(\Big(\{p, p_1, p_2\},$  $, \Omega_{\text{Anne}} = \boxed{1}, \Omega_{\text{Sally}} = \boxed{1}\Big), \{p, p_1\}\Big)$

Sally leaves: $(\mathcal{X}_2 = (\varnothing, \top, \{p\}, \theta_-(p) = \bot, \top, \top), \varnothing)$.

```haskell
sallyLeaves :: Event
sallyLeaves = (Trf [] Top
  (fromList [ (pp,boolBddOf Bot) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate2 :: BelScene
sallyIntermediate2 = sallyIntermediate1 `update` sallyLeaves
```

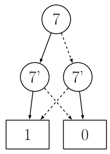
$$\Big(\Big(\varnothing, \top, \{p\}, p := \bot, \Omega^+_{\text{Anne}} = \boxed{1}, \Omega^+_{\text{Sally}} = \boxed{1}\Big), \varnothing\Big)$$

$\Big(\Big(\{p, p_1, p_2, p_3\},$  $, \Omega_{\text{Anne}} = \boxed{1}, \Omega_{\text{Sally}} = \boxed{1}\Big), \{p_1, p_3\}\Big)$

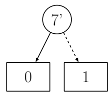
Anne puts the marble in the box, not observed by Sally: $(\mathcal{X}_2 = (\{q\}, \top, \{t\}, \theta_-(t) = (\neg q \to t) \wedge (q \to \bot), \neg q', q \leftrightarrow q'), \{q\})$.

```
annePutsMarbleInBox :: Event
annePutsMarbleInBox = (Trf [qq] Top
  (fromList [ (tt,boolBddOf $ Conj [Neg (PrpF qq) `Impl` PrpF tt, PrpF qq `Impl` Bot]) ])
  (fromList [ ("Anne", allsamebdd [qq]), ("Sally", cpBdd $ boolBddOf $ Neg (PrpF qq))  ]),
      [qq])

sallyIntermediate3 :: BelScene
sallyIntermediate3 = sallyIntermediate2 `update` annePutsMarbleInBox
```

$$\Big(\Big(\{p_7\}, \top, \{p_1\}, p_1 := (p_1 \wedge \neg p_7), \Omega^+_{\text{Anne}} =$$

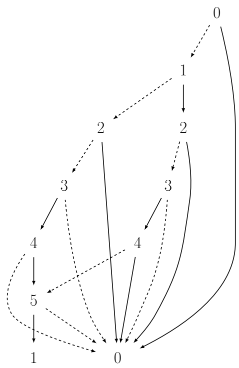


$$, \Omega^+_{\text{Sally}} =$$

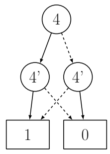


$$\Big), \{p_7\}\Big)$$

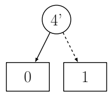
$$\Big(\Big(\{p, p_1, p_2, p_3, p_4, p_5\},$$



$$, \Omega_{\text{Anne}} =$$



$$, \Omega_{\text{Sally}} =$$



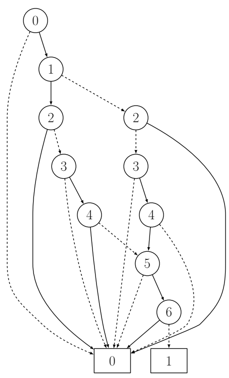
$$\Big), \{p_3, p_4, p_5\}\Big)$$

Sally comes back: $(\mathcal{X}_4 = (\varnothing, \top, \{p\}, \theta_-(p) = \top, \top, \top), \varnothing)$.

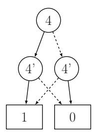
```
sallyComesBack :: Event
sallyComesBack = (Trf [] Top
  (fromList [ (pp,boolBddOf Top) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate4 :: BelScene
sallyIntermediate4 = sallyIntermediate3 `update` sallyComesBack
```

$$\Big(\Big(\varnothing, \top, \{p\}, p := \top, \Omega^+_{\text{Anne}} = \boxed{1}\ , \Omega^+_{\text{Sally}} = \boxed{1}\ \Big), \varnothing\Big)$$

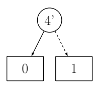
$$\Big(\Big(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$$



$$, \Omega_{\text{Anne}} =$$

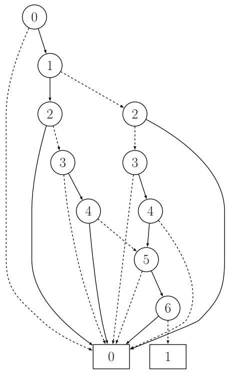
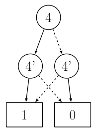
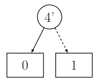


$$, \Omega_{\text{Sally}} =$$



$$\Big), \{p, p_3, p_4, p_5\}\Big)$$

```
sallyFinal :: BelScene
sallyFinal = sallyInit `updates`
  [ sallyPutsMarbleInBasket
  , sallyLeaves
  , annePutsMarbleInBox
  , sallyComesBack ]

sallyFinalCheck :: Bool
sallyFinalCheck = SMCDEL.Symbolic.K.evalViaBdd sallyFinal (K "Sally" (PrpF tt))
```

$\Bigg(\Bigg(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},$  $, \Omega_{\text{Anne}} =$  $, \Omega_{\text{Sally}} =$  $\Bigg), \{p, p_3, p_4, p_5\}\Bigg)$

```
>>> sallyFinalCheck

True

0.05 seconds
```

We check that in the last scene Sally believes the marble is in the basket:

$$\{p, q\} \vDash \Box_{\text{S}} t$$

$$\text{iff } \{p, q\} \vDash \forall V'(\theta' \to (\Omega_{\text{S}} \to t'))$$

$$\text{iff } \{p, q\} \vDash \forall \{p', t', q'\}((t' \leftrightarrow \neg q') \land p' \to (\neg q' \to t'))$$

$$\text{iff } \{p, q\} \vDash \top$$

10.22 Sum and Product

```
module SMCDEL.Examples.SumAndProduct where

import Data.List
import Data.Maybe

import SMCDEL.Language
import SMCDEL.Internal.Help
import SMCDEL.Symbolic.S5
```

Our model checker can also be used to solve the famous Sum & Product puzzle from [Fre69], translated from Dutch:

> A says to S and P: "I chose two numbers x, y such that $1 < x < y$ and $x + y \leq 100$. I will tell $s = x + y$ to S alone, and $p = xy$ to P alone. These messages will stay secret. But you should try to calculate the pair (x, y)". He does as announced. Now follows this conversation: P says: "I do not know it." S says: "I knew that." P says: "Now I know it." S says: "Now I also know it." Determine the pair (x, y).

We first need to encode the value of numbers with boolean propositions.

```haskell
-- 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 to label [2..100], because 2^6 = 64 < 100 < 128 = 2^7
xProps, yProps, sProps, pProps :: [Prp]
xProps = [(P  1)..(P  7)]
yProps = [(P  8)..(P 14)]
sProps = [(P 15)..(P 21)]
-- 12 propositions for the product, because 2^11 = 2048 < 2500 < 4096 = 2^12
pProps = [(P 22)..(P 33)]

sapAllProps :: [Prp]
sapAllProps = sort $ 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: `xIs 5` $= \bigwedge\{p_1, p_2, p_3, p_4, p_6, \neg p_5, \neg p_7\}$

```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 = Disj [ 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.

```haskell
sapSolutions :: [[Prp]]
sapSolutions = whereViaBdd 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]]

1.04 seconds
```

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

```haskell
sapExplainState :: [Prp] -> String
sapExplainState truths = concat
  [ "x = ", explain xProps, ", y = ", explain yProps, ", x+y = ", explain sProps
  , " and x*y = ", explain pProps ] where explain = show . nmbr truths

nmbr :: [Prp] -> [Prp] -> Int
nmbr truths set = fromMaybe (error "Value not found") $
  elemIndex (set `intersect` truths) (powerset set)
```

```
>>> map sapExplainState sapSolutions

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

1.05 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

1.05 seconds
```

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

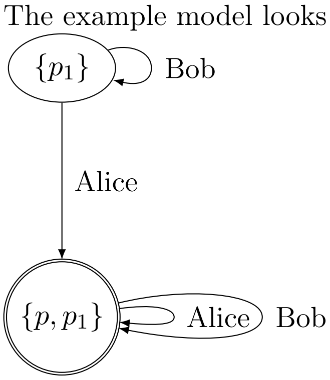
```
>>> validViaBdd sapKnStruct (Impl sapProtocol (Conj [xIs 4, yIs 13]))

True

1.03 seconds
```

Our implementation is faster than the one in [Luo+08] which also used BDDs. It is known that BBDs encoding products are relatively large and inefficient. And indeed the explicit model checker DEMO-S5 still solves this puzzle a bit faster. For a benchmark and further discussion see Section 11.3 and [Gat18, Section 4.6].

10.23 Simple Actions in K

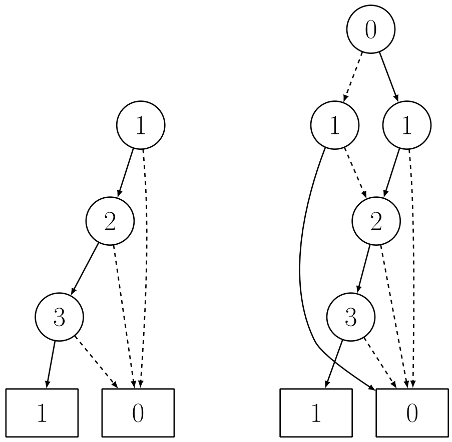
```
module SMCDEL.Examples.SimpleK where

import Data.HasCacBDD hiding (Bot,Top)
import Data.List ((\\))
import qualified Data.Map.Strict as M
import Data.Tagged (untag)

import SMCDEL.Explicit.K
import SMCDEL.Language
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Translations.K

exampleModel :: KripkeModel
exampleModel = KrM $ M.fromList
  [ (1, (M.fromList [(P 0,True ),(P 1,True )], M.fromList [(alice,[1]), (bob,[1])] ) )
  , (2, (M.fromList [(P 0,False),(P 1,True )], M.fromList [(alice,[1]), (bob,[2])] ) ) ]

examplePointedModel :: PointedModel
examplePointedModel = (exampleModel,1)
```

The example model looks as follows:



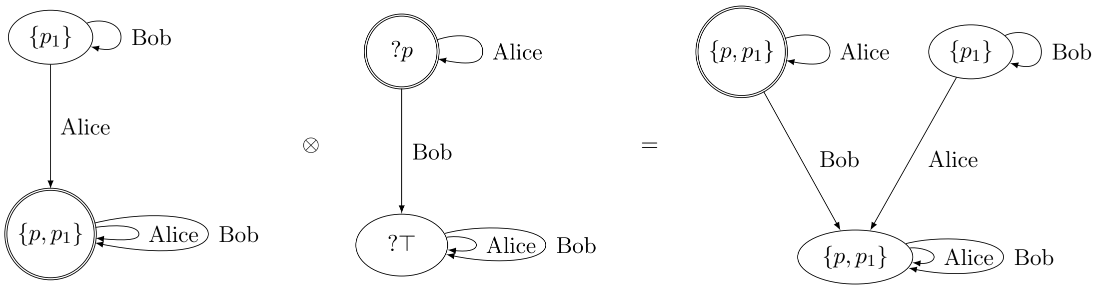
The relations in this model can be describes with these BDDs:

```
aliceBdd, bobBdd :: Bdd
[aliceBdd,bobBdd] = map (untag . flip SMCDEL.Symbolic.K.relBddOfIn exampleModel) [alice,bob
    ]
```



```
-- Privately tell alice that P 0, while bob thinks nothing happens.
exampleGenActM :: ActionModel
exampleGenActM = ActM $ M.fromList
  [ (1, Act { pre = PrpF (P 0), post = M.fromList [], rel = M.fromList [(alice,[1]), (bob
      ,[2])] } )
  , (2, Act { pre = Top       , post = M.fromList [], rel = M.fromList [(alice,[2]), (bob
      ,[2])] } )
  ]

examplePointedActM :: PointedActionModel
examplePointedActM = (exampleGenActM,1)

exampleResult :: PointedModel
exampleResult = update examplePointedModel examplePointedActM
```

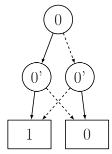
Now we can do a full example:



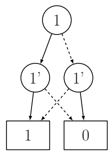
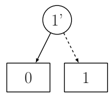
Here is another full example of belief transformation:

```
exampleStart :: BelScene
exampleStart = (BlS [P 0] law obs, actual) where
  law    = boolBddOf Top
  obs    = M.fromList [ ("1", mvBdd $ boolBddOf Top), ("2", allsamebdd [P 0]) ]
  actual = [P 0]

exampleEvent :: Event
exampleEvent = (Trf [P 1] addlaw (M.fromList []) eventObs, [P 1]) where
  addlaw = PrpF (P 1) `Impl` PrpF (P 0)
  eventObs = M.fromList [ ("1", allsamebdd [P 1]), ("2", cpBdd . boolBddOf $ Neg (PrpF $ P
      1)) ]

exampleBlTresult :: BelScene
exampleBlTresult = exampleStart `update` exampleEvent
```

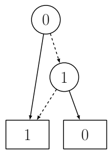
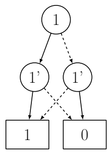
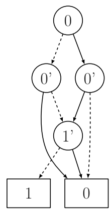
The structure

$\Big(\Big(\{p\}, \boxed{1}, \Omega_1 = \boxed{1}, \Omega_2 =$  $\Big), \{p\}\Big)$

transformed with the event

$\Big(\Big(\{p_1\}, (p_1 \to p), \varnothing, , \Omega_1^+ =$  $, \Omega_2^+ =$  $\Big), \{p_1\}\Big)$

yields this new structure:

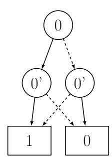
$\Big(\Big(\{p, p_1\},$  $, \Omega_1 =$  $, \Omega_2 =$  $\Big), \{p, p_1\}\Big)$

Here follows another example with factual change.

```
publicMakeFalseActM :: [Agent] -> Prp -> PointedActionModel
publicMakeFalseActM ags p =
  (ActM $ M.fromList [ (1::Int, Act myPre myPost myRel) ], 0) where
    myPre  = Top
    myPost = M.fromList [(p,Bot)]
    myRel  = M.fromList [(i,[1]) | i <- ags]
```

```
publicMakeFalseTrf :: [Agent] -> Prp -> Event
publicMakeFalseTrf agents p = (Trf [] Top changelaw eventobs, []) where
  changelaw = M.fromList [ (p,boolBddOf Bot) ]
  eventobs  = M.fromList [ (i,totalRelBdd) | i <- agents ]

myEvent :: Event
myEvent = publicMakeFalseTrf (agentsOf exampleStart) (P 0)

tResult :: BelScene
tResult = exampleStart `update` myEvent

flipOverAndShowTo :: [Agent] -> Prp -> Agent -> Event
flipOverAndShowTo everyone p i = (Trf [q] eventlaw changelaw eventobs, [q]) where
  q         = freshp [p]
  eventlaw  = PrpF q `Equi` PrpF p
  changelaw = M.fromList [ (p, boolBddOf . Neg . PrpF $ p) ]
  eventobs  = M.fromList $ (i, allsamebdd [q])
                        : [ (j,totalRelBdd) | j <- everyone \\ [i] ]

myOtherEvent :: Event
myOtherEvent = flipOverAndShowTo ["1","2"] (P 0) "1"

tResult2 :: BelScene
tResult2 = exampleStart `update` myOtherEvent
```

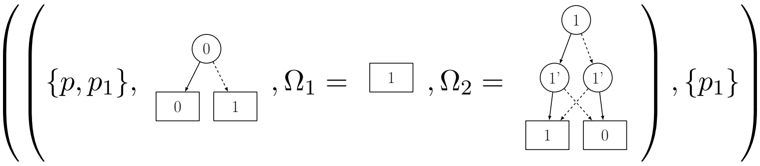
The structure ...

$\Big(\Big(\{p\}, \boxed{1}, \Omega_1 = \boxed{1}, \Omega_2 =$  $\Big), \{p\}\Big)$

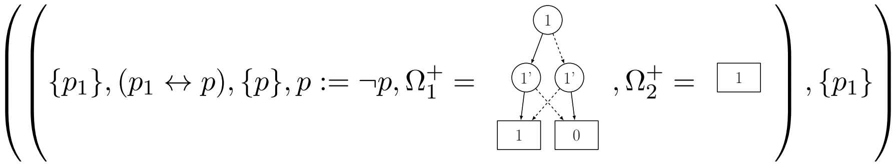
…transformed with `myEvent` …

$$\left(\left(\varnothing, \top, \{p\}, p := \bot, \Omega_1^+ = \boxed{1}, \Omega_2^+ = \boxed{1}\right), \varnothing\right)$$

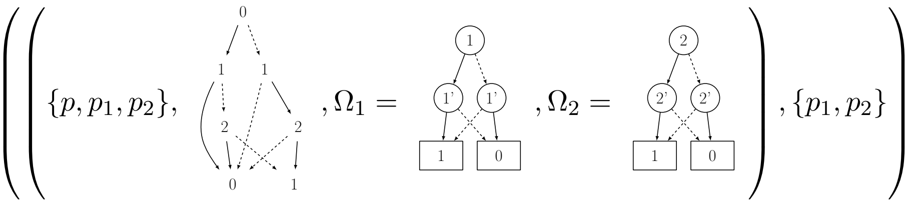
…yields this new structure:



If we instead transform it with `myOtherEvent` …

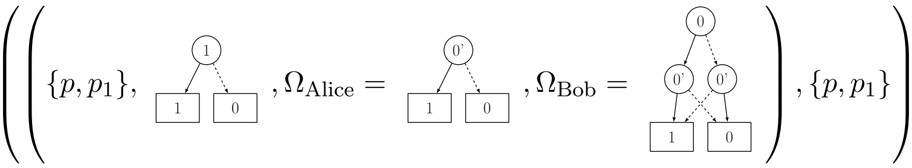


…then we get:



10.24 Translations

```
exampleBelScn :: BelScene
exampleBelScn = kripkeToBls examplePointedModel
```



(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 checks that both parts of the related pair are actually states of the structure. However, we do not need to repeat this information in the BDDs for every agent, because the state law already contains it.

10.25 Simple Actions in S5

```
module SMCDEL.Examples.SimpleS5 where

import Data.List ((\\))

import SMCDEL.Explicit.S5
import SMCDEL.Translations.S5
import SMCDEL.Symbolic.S5
import SMCDEL.Language
```

We now do a simple example with factual change: publicly make p false.

```haskell
myStart :: KnowScene
myStart = (KnS [P 0] (boolBddOf Top) [("Alice",[]),("Bob",[P 0])],[P 0])

publicMakeFalse :: [Agent] -> Prp -> Event
publicMakeFalse agents p = (KnTrf [] Top mychangelaw myobs, []) where
  mychangelaw = [ (p,boolBddOf Bot) ]
  myobs = [ (i,[]) | i <- agents ]

myEvent :: Event
myEvent = publicMakeFalse (agentsOf myStart) (P 0)

myResult :: KnowScene
myResult = myStart `update` myEvent
```

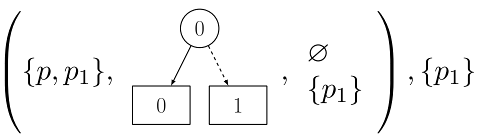
The structure ...

$$\left(\{p\}, \boxed{1}, \begin{smallmatrix} \varnothing \\ \{p\} \end{smallmatrix} \right), \{p\}$$

... transformed with ...

$$\left(\left(\varnothing, \top, \{p\}, p := \bot, \Omega_1^+ = \boxed{1}, \Omega_2^+ = \boxed{1} \right), \varnothing \right)$$

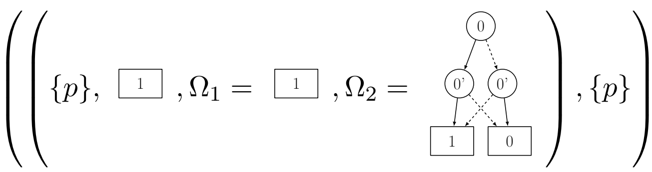
... yields this new structure:



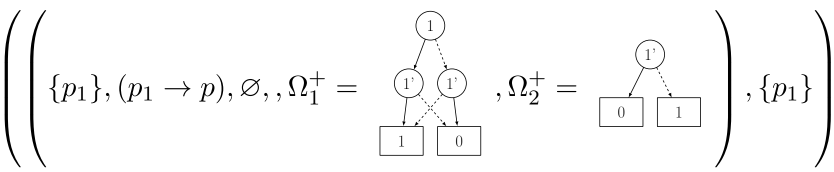
Something more involved, making it false but still keeping it secret from Alice.

```haskell
exampleStart :: KnowScene
exampleStart = (KnS [P 0] (boolBddOf Top) [("Alice",[]),("Bob",[P 0])],[P 0])

makeFalseShowTo :: [Agent] -> Prp -> [Agent] -> Event
makeFalseShowTo agents p intheknow = (KnTrf [P 99] Top examplechangelaw exampleobs, [])
    where
  examplechangelaw = [ (p,boolBddOf $ PrpF (P 99)) ]
  exampleobs = [ (i,[P 99]) | i <- intheknow            ]
            ++ [ (i,[    ]) | i <- agents \\ intheknow ]

exampleEvent :: Event
exampleEvent = makeFalseShowTo (agentsOf exampleStart) (P 0) ["Bob"]

exampleResult :: KnowScene
exampleResult = exampleStart `update` exampleEvent
```

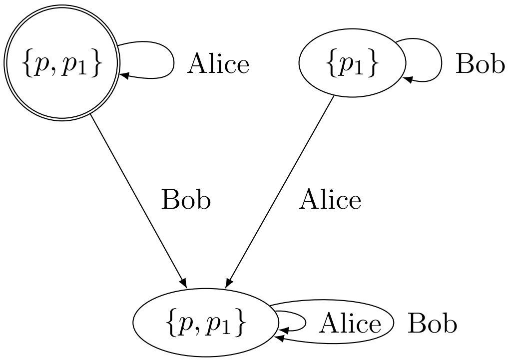
The structure ...



... transformed with ...



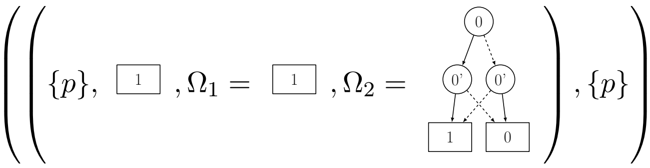
. . . yields this new structure:



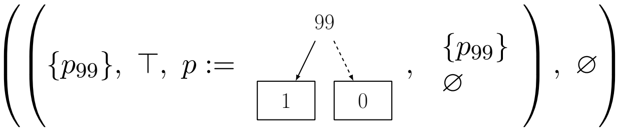
And alternatively, showing the result only to Alice:

```
thirdEvent :: Event
thirdEvent = makeFalseShowTo (agentsOf exampleStart) (P 0) ["Alice"]

thirdResult :: KnowScene
thirdResult = exampleStart `update` thirdEvent
```

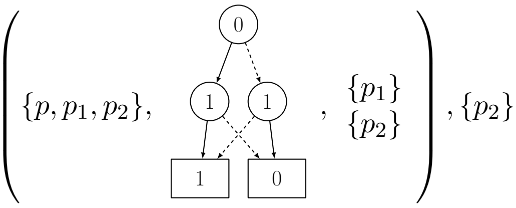
The same structure . . .



. . . transformed with . . .



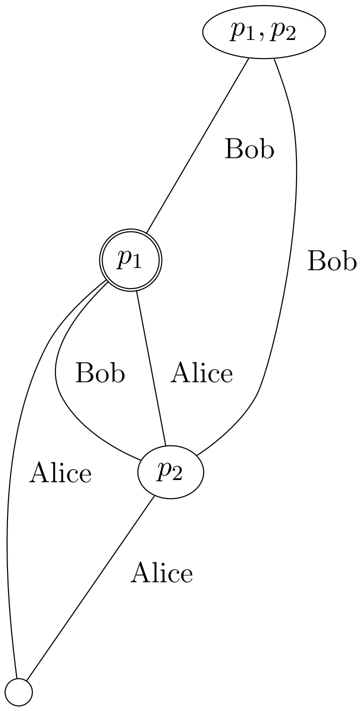
. . . yields this new structure:

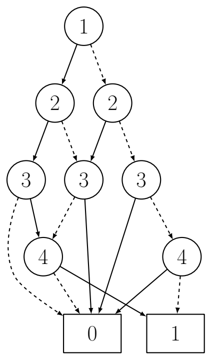


10.26 The limits of observational variables

In [Ben+15] 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 $p \wedge q$ and our existing approach does this by adding an additional variable:

```
problemPM :: PointedModelS5
problemPM = ( KrMS5 [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::World )

problemKNS :: KnowScene
problemKNS = kripkeToKns problemPM
```



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

To overcome this limitation we need to switch from knowledge structures to belief structures, where the observational variables are replaced with BDDs. These BDDs describe relations between worlds as relations between sets of true propositions.

10.27 What Sum

```
module SMCDEL.Examples.WhatSum where

import SMCDEL.Examples.SumAndProduct (nmbr)
import SMCDEL.Language
import SMCDEL.Internal.Help
import SMCDEL.Symbolic.S5
```

We quote the following "What Sum" puzzle from [DR07] where it was implemented using DEMO.

> Each of agents Anne, Bill, and Cath has a positive integer on its forehead. They can only see the foreheads of others. One of the numbers is the sum of the other two. All the previous is common knowledge. The agents now successively make the truthful announcements:
>
> Anne: "I do not know my number."
> Bill: "I do not know my number."

Cath: “I do not know my number.”
Anne: “I know my number. It is 50.”

What are the other numbers?

As we can not make our model infinite, we pick bound all numbers at 100. Note that this gives extra knowledge to the agents and thereby limits the set of solutions.

```
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 1)..(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] ]
```

The dialogue from the puzzle gives us the following conditions:

```
wsResult :: KnowStruct
wsResult = foldl update wsKnStruct [ Neg wsKnowSelfA, Neg wsKnowSelfB, Neg wsKnowSelfC ]

wsSolutions :: [State]
wsSolutions = statesOf wsResult

wsExplainState :: [Prp] -> [(Char,Int)]
wsExplainState truths =
  [ ('a', explain aProps), ('b', explain bProps), ('c', explain cProps) ] where
    explain = nmbr truths
```

Note that we use the `nmbr` function from *Sum and Product* above.
Use `fmap length (mapM (putStrLn.wsExplainState) wsSolutions)` to list and count solutions:

```
λ> fmap length (mapM (print.wsExplainState) wsSolutions)
[('a',1),('b',3),('c',2)]
[('a',1),('b',3),('c',4)]
2
(0.02 secs, 6,360,792 bytes)
```

wsBound	Runtime DEMO [DR07]	Runtime SMCDEL	# Solutions
10	1.59	0.22	2
20	30.31	0.27	36
30	193.20	0.23	100
40	n/a	0.41	198
50	n/a	0.83	330

However, this was a simplification of the original puzzle. We can also consider the following version also suggested in [DR07, p. 144].

> Each of agents Anne, Bill, and Cath has a positive integer on its forehead. They can only see the foreheads of others. One of the numbers is the sum of the other two. All the previous is common knowledge. The agents now successively make the truthful announcements:
>
> Anne: “I do not know my number.”
> Bill: “I do not know my number.”
> Cath: “I do not know my number.”
>
> What are the numbers, if Anne now knows her number and if all numbers are prime?

As we can not make our model infinite, we will still bound all numbers at some high value, say 100. Any bound still gives extra knowledge to the agents.

11 Benchmarks

We now provide two different benchmarks for SMCDEL. All experiments and benchmarks described in this chapter were done using 64-bit Debian GNU/Linux 9 with kernel 4.9.65–3, GHC 8.2.2 and g++ 6.3.0 on an Intel Core i3–2120 3.30 GHz processor and 12 GB of memory.

11.1 Muddy Children

In this section we compare the performance of different model checking functions using the Muddy Children example from Section 10.10.

- SMCDEL with two different BDD packages: CacBDD and CUDD.
- DEMO-S5, a version of the epistemic model checker DEMO optimized for S5 [Eij07; Eij14].
- MCTRIANGLE, an ad-hoc implementation of [GS11], see Appendix 1 on page 142.

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 SMCDEL.Language
import SMCDEL.Examples.MuddyChildren
import SMCDEL.Internal.Help (apply)
import qualified SMCDEL.Explicit.DEMO_S5 as DEMO_S5
import qualified SMCDEL.Explicit.S5
import qualified SMCDEL.Symbolic.S5
import qualified SMCDEL.Symbolic.S5_CUDD
import qualified SMCDEL.Translations.S5
import qualified SMCDEL.Translations.K
import qualified SMCDEL.Other.MCTRIANGLE
import qualified SMCDEL.Symbolic.K
```

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.

```haskell
findNumberCacBDD :: Int -> Int -> Int
findNumberCacBDD = findNumberWith (cacMudScnInit,SMCDEL.Symbolic.S5.evalViaBdd) where
  cacMudScnInit n m = ( SMCDEL.Symbolic.S5.KnS (mudPs n) (SMCDEL.Symbolic.S5.boolBddOf Top)
       [ (show i,delete (P i) (mudPs n)) | i <- [1..n] ], mudPs m )

findNumberCUDD :: Int -> Int -> Int
findNumberCUDD = findNumberWith (cuddMudScnInit,SMCDEL.Symbolic.S5_CUDD.evalViaBdd) where
  cuddMudScnInit n m = ( SMCDEL.Symbolic.S5_CUDD.KnS (mudPs n) (SMCDEL.Symbolic.S5_CUDD.
      boolBddOf Top) [ (show i,delete (P i) (mudPs n)) | i <- [1..n] ], mudPs m )

findNumberTrans :: Int -> Int -> Int
findNumberTrans = findNumberWith (start,SMCDEL.Symbolic.S5.evalViaBdd) where
  start n m = SMCDEL.Translations.S5.kripkeToKns $ mudKrpInit n m

mudKrpInit :: Int -> Int -> SMCDEL.Explicit.S5.PointedModelS5
mudKrpInit n m = (SMCDEL.Explicit.S5.KrMS5 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) $
      sortOn (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 ]

findNumberK :: Int -> Int -> Int
findNumberK = findNumberWith (mudBelScnInit, SMCDEL.Symbolic.K.evalViaBdd)

findNumberTransK :: Int -> Int -> Int
findNumberTransK = findNumberWith (start,SMCDEL.Symbolic.K.evalViaBdd) where
  start n m = SMCDEL.Translations.K.kripkeToBls $ mudGenKrpInit n m
```

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.

```haskell
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
```

Also the number triangle approach has to be treated separately. See [GS11] and Appendix 1 on page 142 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.

```haskell
findNumberTriangle :: Int -> Int -> Int
findNumberTriangle n m = findNumberTriangleLoop 0 start where
  start = SMCDEL.Other.MCTRIANGLE.mcUpdate (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.mcUpdate curMod SMCDEL.
        Other.MCTRIANGLE.nobodyknows)
    else count
```

In the following we use the library *Criterion* [OSu16] 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] )
  , ("K"          , findNumberK          , [3..12] )
  , ("DEMOS5"     , findNumberDemoS5     , [3..12] )
  , ("Trans"      , findNumberTrans      , [3..12] )
  , ("TransK"     , findNumberTransK     , [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
```

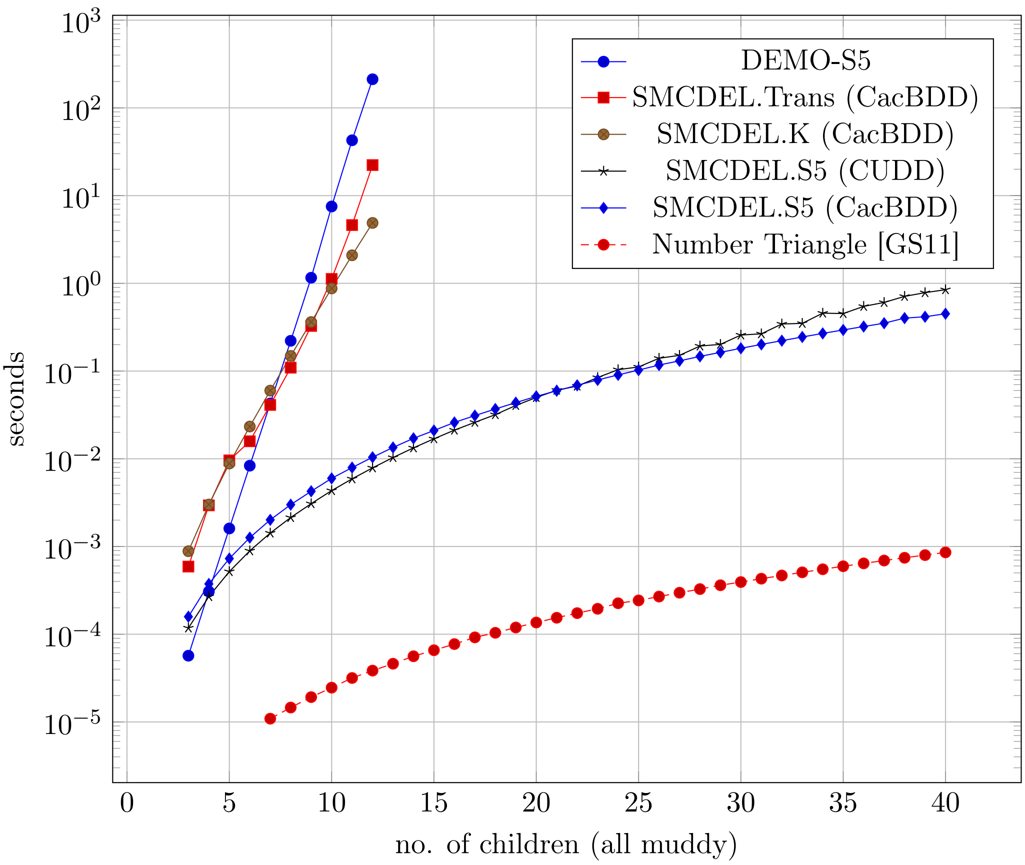


Figure 5: Benchmark Results on a logarithmic scale.

As expected we can see in Figure 5 that *SMCDEL* is faster than the explicit model checker DEMO-S5. Both BDD packages give us similar performance for S5, with a slightly better performance of CacBDD compared to CUDD. It is important to note that this difference and the performance in general also depends on the binding libraries we use. Also the more general methods for the logic K are a bit faster than DEMO-S5.

Finally, the number triangle approach from [GS11] 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 should be studied in the future 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.

11.2 Dining Cryptographers

Muddy Children has also been used to benchmark MCMAS [LQR15] 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 [Ben+09; DHR13].

A protocol which fits nicely into both frameworks are the Dining Cryptographers from [Cha88] which we implemented in Section 10.6. We will now use it to measure the performance of *SMCDEL* in a way that is more similar to [LQR15].

```
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 Text.Printf

import SMCDEL.Language
import SMCDEL.Symbolic.S5
import SMCDEL.Examples.DiningCrypto
```

The following statement was also checked with MCMAS in [LQR15].

> "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 from [Ben+09; DHR13] we formalize the same statement 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 public announcement whether the number of agents which announced a 1 is odd or even, i.e. $\psi := \bigoplus_i \bigoplus \{p \mid \text{Agent } i \text{ can observe } p\}$.

```
benchDcCheckForm :: Int -> Form
benchDcCheckForm n =
  PubAnnounceW (Xor [genDcReveal n i | i<-[1..n] ]) $
  -- pubAnnounceWhetherStack [ genDcReveal n i | i<-[1..n] ] $ -- slow!
    Impl (Neg (PrpF $ P 1)) $
      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] ] ] ]
```

Note that this formula is different from the one we checked in Section 10.6.

```
benchDcValid :: Int -> Bool
benchDcValid n = validViaBdd (genDcKnsInit n) (benchDcCheckForm 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 $ benchDcCheckForm n) ++ "\t"
  if benchDcValid 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
  printf "%.4f\n" (realToFrac result :: Double)
  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 (3:4:(5 : map (10*) [1..])) limit
```

The program outputs a table like the following, showing in five columns (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.

```
$ stack bench :bench-diningcrypto
...
n       n(prps) sz(KNS) sz(frm) time
3       7       217     339     0.1372
4       11      332     477     0.0005
5       16      483     645     0.0008
10      56      1654    1847    0.0023
20      211     6497    6289    0.0091
30      466     14572   13419   0.0283
40      821     25747   23149   0.0619
50      1276    40850   36031   0.1212
60      1831    59890   52071   0.2229
70      2486    82330   70911   0.4160
80      3241    108170  92551   0.7038
90      4096    137410  116991  1.0987
```

These results are satisfactory: While MCMAS already needs more than 10 seconds to check the interpreted system for 50 or more dining cryptographers (see [LQR15, Table 4]), *SMCDEL* can deal with the DEL model of up to 160 agents in less time. Note however, that the DEL model we use here is less detailed than a temporal model. In particular, we take synchronous perfect recall for granted and merge all broadcasts done by different agents into one public announcement.

Note that the result for three agents is slower just because we compute it first. The same happens if we start at four or five agents. The reason is that initializing the BDD package takes some time, but is done only once.

11.3 Sum and Product

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

```
module Main (main) where
import Criterion.Main
import Data.List (groupBy,sortBy)
```

```
import Data.Time (getCurrentTime, diffUTCTime)
import System.Environment (getArgs)
import SMCDEL.Explicit.DEMO_S5
import SMCDEL.Examples.SumAndProduct
import SMCDEL.Symbolic.S5
```

We use the implementation in the module `SMCDEL.Examples.SumAndProduct`, see Section 10.22. 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
alice, bob :: Agent
(alice,bob) = (Ag 0,Ag 1)

--initial pointed epistemic model
msnp :: EpistM (Int,Int)
msnp = Mo pairs [alice,bob] [] rels pairs 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')) pairs

fmrs1e, fmrp2e, fmrs3e :: DemoForm (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 <- pairs])

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

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

```
main :: IO ()
main = do
  args <- getArgs
  if args == ["checkingOnly"]
    then do
      putStrLn "Benchmarking only the checking, without model generation."
      benchCheckingOnly
    else do
      putStrLn "Benchmarking the complete run."
      benchAllOnce

benchAllOnce :: IO ()
benchAllOnce = do
  putStrLn "*** Running DEMO_S5 ***"
  start <- getCurrentTime
  print $ updsPa msnp [fmrs1e, fmrp2e, fmrs3e]
  end <- getCurrentTime
  putStrLn $ "This took " ++ show (end `diffUTCTime` start) ++ " seconds.\n"

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

benchCheckingOnly :: IO ()
benchCheckingOnly = defaultMain [
  bgroup "checkingOnly"
    [ bench "DEMO-S5" $ nf (show . updsPa msnp) [fmrs1e, fmrp2e, fmrs3e]
    , bench "SMCDEL"  $ nf (sapExplainState . head . whereViaBdd sapKnStruct) sapProtocol
    ]
  ]
```

12 Executables

12.1 CLI Interface

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

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

Is this valid on the given structure? True

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

At which states is this true? $\varnothing$

For more examples, see the `Examples` folder.

```haskell
module Main where

import Control.Arrow (second)
import Control.Monad (when,unless)
import Data.List (intercalate)
import Data.Version (showVersion)
import Paths_smcdel (version)
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.S5

main :: IO ()
main = do
  (input,options) <- getInputAndSettings
  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 $ putStrLn infoline
  when texMode $ hPutStrLn outHandle texPrelude
  case parse $ alexScanTokens input of
    Left (lin,col) -> error ("Parse error in line " ++ show lin ++ ", column " ++ show col)
    Right (CheckInput vocabInts lawform obs jobs) -> do
      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,[])::KnowScene) ++ ") \\]", "\\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 -> Job -> IO ()
doJob outHandle True mykns (ValidQ f) = do
  hPutStrLn outHandle $ "Is $" ++ texForm (simplify f) ++ "$ valid on $\\mathcal{F}$?"
  hPutStrLn outHandle (show (validViaBdd mykns f) ++ "\n\n")
doJob outHandle False mykns (ValidQ f) = do
  hPutStrLn outHandle $ "Is " ++ ppForm f ++ " valid on the given structure?"
  vividPutStrLn (show (validViaBdd mykns f) ++ "\n")
doJob outHandle True mykns (WhereQ 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 (WhereQ 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
      mapM_ (hPutStrLn stderr)
        [ infoline
        , "usage: " ++ name ++ " <filename> {options}"
        , "        (use filename - for STDIN)\n"
        , "  -tex   generate LaTeX code\n"
        , "  -show  write to /tmp, generate PDF and show it (implies -tex)\n" ]
      exitFailure

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

infoline :: String
infoline = "SMCDEL " ++ showVersion version ++ " -- 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}"
```

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 -w #-}
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 }

{
type LexResult a = Either (Int,Int) a

alexScanTokensSafe :: String -> LexResult [Token AlexPosn]
alexScanTokensSafe str = go (alexStartPos,'\n',[],str) where
  go inp@(pos,_,_,str) =
    case (alexScan inp 0) of
      AlexEOF -> Right []
      AlexError ((AlexPn _ line column),_,_,_) -> Left (line,column)
      AlexSkip  inp' len     -> go inp'
      AlexToken inp' len act -> case (act pos (take len str), go inp') of
        (_, Left lc) -> Left lc
        (x, Right y) -> Right (x : y)
}
```

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

```
{
{-# OPTIONS_GHC -w #-}
module SMCDEL.Internal.Parse where
import SMCDEL.Internal.Token
import SMCDEL.Internal.Lex
import SMCDEL.Language
}

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

%monad { ParseResult } { >>= } { Right }

%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 }
ObserveLine : STR COLON IntList { ($1,$3) }
ObserveSpec : ObserveLine { [$1] } | ObserveLine ObserveSpec { $1:$2 }
JobList : Job { [$1] } | Job JobList { $1:$2 }
Job : VALIDQ Form { ValidQ $2 } | WHEREQ Form { WhereQ $2 }

{
data CheckInput = CheckInput [Int] Form [(String,[Int])] JobList deriving (Show,Eq,Ord)
data Job = ValidQ Form | WhereQ Form deriving (Show,Eq,Ord)
type JobList = [Job]
type IntList = [Int]
type FormList = [Form]
type ObserveLine = (String,IntList)
type ObserveSpec = [ObserveLine]

type ParseResult a = Either (Int,Int) a

parseError :: [Token AlexPosn] -> ParseResult a
parseError []     = Left (1,1)
parseError (t:ts) = Left (lin,col)
  where (AlexPn abs lin col) = apn t
}
```

12.2 Web Interface

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

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

module Main where

import Prelude
import Control.Monad.IO.Class (liftIO)
import Control.Arrow
import Data.FileEmbed
import Data.List (intercalate)
import Data.Version (showVersion)
import Paths_smcdel (version)
import Web.Scotty
import qualified Data.Text as T
import qualified Data.Text.Encoding as E
import qualified Data.Text.Lazy as TL
import Data.HasCacBDD.Visuals (svgGraph)
import qualified Language.Javascript.JQuery as JQuery
import Language.Haskell.TH.Syntax

import SMCDEL.Internal.Lex
import SMCDEL.Internal.Parse
import SMCDEL.Symbolic.S5
import SMCDEL.Internal.TexDisplay
import SMCDEL.Translations.S5
import SMCDEL.Language

main :: IO ()
main = do
  putStrLn $ "SMCDEL " ++ showVersion version ++ " -- https://github.com/jrclogic/SMCDEL"
  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 . TL.fromStrict $ addVersionNumber $ embeddedFile "index.html
    "
    get "/jquery.js"   . html . TL.fromStrict $ embeddedFile "jquery.js"
    get "/ace.js"      . html . TL.fromStrict $ embeddedFile "ace.js"
    get "/viz-lite.js" . html . TL.fromStrict $ embeddedFile "viz-lite.js"
    get "/getExample" $ do
      this <- param "filename"
      html . TL.fromStrict $ embeddedFile this
    post "/check" $ do
      smcinput <- param "smcinput"
      case alexScanTokensSafe smcinput of
        Left pos -> webError "Lex" pos
        Right lexResult -> case parse lexResult of
          Left pos -> webError "Parse" pos
          Right (CheckInput vocabInts lawform obs jobs) -> do
            let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs
                )
            knstring <- liftIO $ showStructure mykns
            let results = concatMap (\j -> "<p>" ++ doJobWeb mykns j ++ "</p>") jobs
            html $ mconcat
              [ TL.pack knstring
              , "<hr />\n"
              , TL.pack results ]
    post "/knsToKripke" $ do
      smcinput <- param "smcinput"
      case alexScanTokensSafe smcinput of
        Left pos -> webError "Lex" pos
        Right lexResult -> case parse lexResult of
          Left pos -> webError "Parse" pos
          Right (CheckInput vocabInts lawform obs _) -> do
            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 > 32
              then html . TL.pack $ "Sorry, I will not draw " ++ show (numberOfStates mykns
                  ) ++ " states!"
```

```
              else do
                let (myKripke, _) = knsToKripke (mykns, head $ statesOf mykns) -- ignore
                    actual world
                html $ TL.concat
                  [ TL.pack "<div id='here'></div>"
                  , TL.pack "<script>document.getElementById('here').innerHTML += Viz('"
                  , textDot myKripke
                  , TL.pack "');</script>" ]

doJobWeb :: KnowStruct -> Job -> String
doJobWeb mykns (ValidQ f) = unlines
  [ "\\( \\mathcal{F} "
  , if validViaBdd mykns f then "\\vDash" else "\\not\\vDash"
  , (texForm.simplify) f
  , "\\)" ]
doJobWeb mykns (WhereQ 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>"

embeddedFile :: String -> T.Text
embeddedFile s = case s of
  "index.html"            -> E.decodeUtf8 $(embedFile "static/index.html")
  "viz-lite.js"           -> E.decodeUtf8 $(embedFile "static/viz-lite.js")
  "ace.js"                -> E.decodeUtf8 $(embedFile "static/ace.js")
  "jquery.js"             -> E.decodeUtf8 $(embedFile =<< runIO JQuery.file)
  "MuddyChildren"         -> E.decodeUtf8 $(embedFile "Examples/MuddyChildren.smcdel.txt")
  "DiningCryptographers"  -> E.decodeUtf8 $(embedFile "Examples/DiningCryptographers.smcdel.
      txt")
  "DrinkingLogicians"     -> E.decodeUtf8 $(embedFile "Examples/DrinkingLogicians.smcdel.txt
      ")
  _                       -> error "File not found."

addVersionNumber :: T.Text -> T.Text
addVersionNumber = T.replace "<!-- VERSION NUMBER -->" (T.pack $ showVersion version)

webError :: String -> (Int,Int) -> ActionM ()
webError kind (lin,col) = html $ TL.pack $ concat
  [ "<p class='error'>", kind, " error in line ", show lin, ", column ", show col, "</p>\n"
  , "<script>"
  , "editor.clearSelection();"
  , "editor.moveCursorTo(", show (lin - 1), ",", show col, ");"
  , "editor.renderer.scrollCursorIntoView({row: ", show (lin - 1),", column: ", show col, "
      }, 0.5);"
  , "editor.focus();"
  , "</script>"
  ]
```

13 Future Work

We are planning to extend *SMCDEL* and continue our research as follows.

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.

Temporal and Modal Logic

Epistemic and temporal logics have been connected before and translation methods have been proposed, see [Ben+09; DHR13]. Also similar to our observational variables are the "mental programs" recently presented in [CS15]. 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 would be to enable symbolic methods for Epistemic Crypto Logic [EG15]. Our methods could then also be used to analyze cryptographic protocols.

Appendix: Helper Functions

```haskell
module SMCDEL.Internal.Help (
  alleq,alleqWith,anydiff,anydiffWith,alldiff,
  groupSortWith,
  apply,applyPartial,(!),(!=),
  powerset,restrict,rtc,tc,Erel,bl,fusion,seteq,subseteq,lfp
  ) where
import Data.List ((\\),foldl',groupBy,nub,sort,sortBy,union)

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

alleq  :: Eq a => [a] -> Bool
alleq = alleqWith id

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

anydiff :: Eq a => [a] -> Bool
anydiff = anydiffWith id

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

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

groupSortWith :: (Eq a, Ord b) => (a -> b) -> [a] -> [[a]]
groupSortWith f = groupBy (\ x y -> myCompare x y == EQ) . sortBy myCompare where
  myCompare x y = compare (f x) (f y)

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

(!) :: Show a => Show b => Eq a => Rel a b -> a -> b
(!) = apply

applyPartial :: Eq a => [(a,a)] -> a -> a
applyPartial rel left = case lookup left rel of
  Nothing     -> left
  (Just this) -> this

(!=) :: Eq a => [(a,a)] -> a -> a
(!=) = applyPartial

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 ]

tc :: Eq a => Rel a a -> Rel a a
```

```
tc r = lfp (\ s -> s `union` concatRel r s) 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)

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

subseteq :: Eq a => [a] -> [a] -> Bool
subseteq xs ys = all (`elem` ys) xs
```

Appendix: Tagging BDDs for type safety

```
module SMCDEL.Internal.TaggedBDD where

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

import SMCDEL.Language

class TagForBDDs a where
  -- | How many copies of the vocabulary do we have?
  -- This is the number of markers + 1.
  multiplier :: Tagged a Bdd -> Int
  multiplier _ = 2
  -- | move back, must be without markers!
  unmvBdd :: Tagged a Bdd -> Bdd
  unmvBdd = relabelFun (\n -> if even n then n `div` 2 else error ("Odd: " ++ show n)) .
      untag
  -- | move into double vocabulary, but do not add marker
  mv :: Bdd -> Tagged a Bdd
  mv = cpMany 0
  -- | move into extended vocabulary, add one marker
  cp :: Bdd -> Tagged a Bdd
  cp = cpMany 1
  -- | move into extended vocabulary, add k many markers, MUST be available!
  cpMany :: Int -> Bdd -> Tagged a Bdd
  cpMany k b = let x = pure $ relabelFun (\n -> (2*n) + k) b
               in if k >= multiplier x then error "Not enough markers!" else x

  tagBddEval :: [Prp] -> Tagged a Bdd -> Bool
  tagBddEval truths querybdd = evaluateFun (untag querybdd) (\n -> P n `elem` truths)
  totalRelBdd, emptyRelBdd :: Tagged a Bdd
  totalRelBdd = pure top
  emptyRelBdd = pure bot
```

We provide three tags and the instances to use them.

```
data Dubbel
instance TagForBDDs Dubbel where
  multiplier = const 2

data Tripel
instance TagForBDDs Tripel where
  multiplier = const 3

data Quadrupel
instance TagForBDDs Quadrupel where
  multiplier = const 4
```

In double (or more) vocabularies we often want to say that each plain (p) and the corresponding marked (p') atom have the same value: $\bigwedge_p (p \leftrightarrow p')$. This can be defined once for all tagged BDDs.

```
allsamebdd :: TagForBDDs a => [Prp] -> Tagged a Bdd
allsamebdd ps = conSet <$> sequence [ equ <$> mv (var x) <*> cp (var x) | (P x) <- ps ]
```

Appendix: Muddy Children on the Number Triangle

This module implements [GS11]. 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 in a Kripke model 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 some type definitions: A child can be muddy or clean. 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.

```
data Kind = Muddy | Clean
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 -> Kind -> State
obsFor (McM _ _ (curc,curm)) Clean = (curc-1,curm)
obsFor (McM _ _ (curc,curm)) Muddy = (curc,curm-1)

posFor :: McModel -> Kind -> [State]
posFor m status = posFrom m $ obsFor m status
```

Note that instead of naming or enumerating agents we only distinguish two `Kind`s, the muddy and non-muddy ones, represented by Haskells constants `Muddy` and `Clean` which allow pattern matching.

The following is a type for quantifiers on the number triangle, e.g. `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 Kind     -- all b agents DO    know their status
               | NotKnowSelf Kind  -- 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 Clean, NotKnowSelf Muddy ]
everyoneKnows = Conj [    KnowSelf Clean,    KnowSelf Muddy ]
```

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    Muddy) = curm==0 || length (posFor m Muddy) == 1
eval m@(McM _ _ (curc,_)) (KnowSelf    Clean) = curc==0 || length (posFor m Clean) == 1
eval m@(McM _ _ (_,curm)) (NotKnowSelf Muddy) = curm==0 || length (posFor m Muddy) == 2
eval m@(McM _ _ (curc,_)) (NotKnowSelf Clean) = curc==0 || length (posFor m Clean) == 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:

```
mcUpdate :: McModel -> McFormula -> McModel
mcUpdate (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 = mcUpdate (mcModel s) (Qf some)
step s n = mcUpdate (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: DEMO-S5

```haskell
-- Note: This is a modified version of DEMO-S5 by Jan van Eijck.
-- For the original, see http://homepages.cwi.nl/~jve/software/demo_s5/
module SMCDEL.Explicit.DEMO_S5 where

import Control.Arrow (first,second)
import Data.List (sortBy)

import SMCDEL.Internal.Help (apply,restrict,Erel,bl)

newtype Agent = Ag Int deriving (Eq,Ord,Show)

data DemoPrp = DemoP Int | DemoQ Int | DemoR Int | DemoS Int deriving (Eq,Ord)
instance Show DemoPrp where
  show (DemoP 0) = "p"; show (DemoP i) = "p" ++ show i
  show (DemoQ 0) = "q"; show (DemoQ i) = "q" ++ show i
  show (DemoR 0) = "r"; show (DemoR i) = "r" ++ show i
  show (DemoS 0) = "s"; show (DemoS i) = "s" ++ show i

data EpistM state = Mo
             [state]
             [Agent]
             [(state,[DemoPrp])]
             [(Agent,Erel state)]
             [state]  deriving (Eq)

instance Show state => Show (EpistM state) where
  show (Mo worlds ags val accs points) = concat
    [ "Mo\n   "
    , show worlds, "\n   "
    , show ags, "\n   "
    , show val, "\n   "
    , show accs, "\n   "
    , show points, "\n"
    ]

rel :: Show a => Agent -> EpistM a -> Erel a
rel ag (Mo _ _ _ rels _) = apply rels ag

initM :: (Num state, Enum state) => [Agent] -> [DemoPrp] -> EpistM state
initM ags props = Mo worlds ags val accs points where
  worlds  = [0..(2^k-1)]
  k       = length props
  val     = zip worlds (sortL (powerList props))
  accs    = [ (ag,[worlds]) | ag  <- ags        ]
  points  = worlds

powerList  :: [a] -> [[a]]
powerList  [] = [[]]
powerList  (x:xs) =
  powerList xs ++ map (x:) (powerList xs)

sortL :: Ord a => [[a]] -> [[a]]
sortL  = sortBy
  (\ xs ys -> if length xs < length ys
                then LT
              else if length xs > length ys
                then GT
              else compare xs ys)

data DemoForm a = Top
            | Info a
            | Prp DemoPrp
            | Ng (DemoForm a)
            | Conj [DemoForm a]
            | Disj [DemoForm a]
            | Kn Agent (DemoForm a)
            | PA (DemoForm a) (DemoForm a)
            | PAW (DemoForm a) (DemoForm a)
          deriving (Eq,Ord,Show)
```

```haskell
impl :: DemoForm a -> DemoForm a -> DemoForm a
impl form1 form2 = Disj [Ng form1, form2]

-- | semantics: truth at a world in a model
isTrueAt :: (Show state, Ord state) => EpistM state -> state -> DemoForm state -> Bool
isTrueAt _ _ Top = True
isTrueAt _ w (Info x) = w == x
isTrueAt (Mo _ _ val _ _) w (Prp p) = p `elem` apply val w
isTrueAt m w (Ng f) = not (isTrueAt m w f)
isTrueAt m w (Conj fs) = all (isTrueAt m w) fs
isTrueAt m w (Disj fs) = any (isTrueAt m w) fs
isTrueAt m w (Kn ag f) = all (flip (isTrueAt m) f) (bl (rel ag m) w)
isTrueAt m w (PA f g) = not (isTrueAt m w f) || isTrueAt (updPa m f) w g
isTrueAt m w (PAW f g) = not (isTrueAt m w f) || isTrueAt (updPaW m f) w g

-- | global truth in a model
isTrue :: Show a => Ord a => EpistM a -> DemoForm a -> Bool
isTrue m@(Mo _ _ _ _ points) f = all (\w -> isTrueAt m w f) points

-- | public announcement
updPa :: (Show state, Ord state) => EpistM state -> DemoForm state -> EpistM state
updPa m@(Mo states agents val rels actual) f = Mo states' agents val' rels' actual' where
  states' = [ s | s <- states, isTrueAt m s f ]
  val'    = [ (s, ps) | (s,ps) <- val, s `elem` states' ]
  rels'   = [ (ag,restrict states' r) | (ag,r) <- rels ]
  actual' = [ s | s <- actual, s `elem` states' ]

updsPa ::  (Show state, Ord state) => EpistM state -> [DemoForm state] -> EpistM state
updsPa = foldl updPa

-- | public announcement-whether
updPaW :: (Show state, Ord state) => EpistM state -> DemoForm state -> EpistM state
updPaW m@(Mo states agents val rels actual) f = Mo states agents val rels' actual where
  rels'    = [ (ag, sortL $ concatMap split r) | (ag,r) <- rels ]
  split ws = filter (/= []) [ filter (\w -> isTrueAt m w f) ws, filter (\w -> not $
      isTrueAt m w f) ws ]

updsPaW ::  (Show state, Ord state) => EpistM state -> [DemoForm state] -> EpistM state
updsPaW = foldl updPaW

-- | safe substitutions
sub :: Show a => [(DemoPrp,DemoForm a)] -> DemoPrp -> DemoForm a
sub subst p | p `elem` map fst subst = apply subst p
            | otherwise              = Prp p

-- | public factual change
updPc :: (Show state, Ord state) => [DemoPrp] -> EpistM state -> [(DemoPrp,DemoForm state)]
     -> EpistM state
updPc ps m@(Mo states agents _ rels actual) sb = Mo states agents val' rels actual where
   val' = [ (s, [p | p <- ps, isTrueAt m s (sub sb p)]) | s <- states ]

updsPc :: Show state => Ord state => [DemoPrp] -> EpistM state
                      -> [[(DemoPrp,DemoForm state)]] -> EpistM state
updsPc ps = foldl (updPc ps)

updPi :: (state1 -> state2) -> EpistM state1 -> EpistM state2
updPi f (Mo states agents val rels actual) =
  Mo
  (map f states)
  agents
  (map (first f) val)
  (map (second (map (map f))) rels)
  (map f actual)

bTables :: Int -> [[Bool]]
bTables 0 = [[]]
bTables n = map (True:) (bTables (n-1)) ++ map (False:) (bTables (n-1))

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

```
                   tab1 <- bTables (i-1),
                   tab2 <- bTables (n-i) ]) | i <- [1..n] ]
  points = [False: replicate (n-1) True]

fatherN :: Int -> DemoForm [Bool]
fatherN n = Ng (Info (replicate n False))

kn :: Int -> Int -> DemoForm [Bool]
kn n i = Disj [Kn (Ag i) (Disj [ Info (tab1++[True]++tab2)
                               | tab1 <- bTables (i-1)
                               , tab2 <- bTables (n-i)
                               ] ),
               Kn (Ag i) (Disj [ Info (tab1++[False]++tab2)
                               | tab1 <- bTables (i-1)
                               , tab2 <- bTables (n-i)
                               ] )
              ]

dont :: Int -> DemoForm [Bool]
dont n = Conj [Ng (kn n i) | i <- [1..n] ]

knowN :: Int -> DemoForm [Bool]
knowN n = Conj [kn n i | i <- [2..n] ]

solveN :: Int -> EpistM [Bool]
solveN n = updsPa (initN n) (f:istatements ++ [knowN n]) where
  f = fatherN n
  istatements = replicate (n-2) (dont n)
```

References

[Att+14] Maduka Attamah, Hans van Ditmarsch, Davide Grossi, and Wiebe van der Hoek. "Knowledge and Gossip". In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*. Frontiers in Artificial Intelligence and Applications. Prague, Czech Republic, 2014, pp. 21–26. ISBN: 978-1-61499-418-3. DOI: 10.3233/978-1-61499-419-0-21.

[Ben+09] Johan van Benthem, Jelle Gerbrandy, Tomohiro Hoshi, and Eric Pacuit. "Merging frameworks for interaction". In: *Journal of Philosophical Logic* 38.5 (2009), pp. 491–526. DOI: 10.1007/s10992-008-9099-x.

[Ben+15] Johan van Benthem, Jan van Eijck, Malvin Gattinger, and Kaile Su. "Symbolic Model Checking for Dynamic Epistemic Logic". In: *Logic, Rationality, and Interaction: 5th International Workshop, LORI 2015, Taipei, Taiwan, October 28-30, 2015. Proceedings*. Ed. by Wiebe van der Hoek, Wesley H. Holliday, and Wen-fang Wang. Springer, 2015, pp. 366–378. ISBN: 978-3-662-48561-3. DOI: 10.1007/978-3-662-48561-3_30.

[Ben+17] Johan van Benthem, Jan van Eijck, Malvin Gattinger, and Kaile Su. "Symbolic Model Checking for Dynamic Epistemic Logic – S5 and Beyond". In: *Journal of Logic and Computation (JLC)* (2017). DOI: 10.1093/logcom/exx038. URL: https://homepages.cwi.nl/~jve/papers/16/pdfs/2016-05-23-del-bdd-lori-journal.pdf.

[BMS98] Alexandru Baltag, Lawrence S. Moss, and Slawomir Solecki. "The logic of public announcements, common knowledge, and private suspicions". In: *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge*. Ed. by I. Bilboa. TARK 1998. 1998, pp. 43–56. URL: https://dl.acm.org/citation.cfm?id=645876.671885.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. Cambridge, Massachusetts, USA: The MIT Press, 1999. ISBN: 9780262032704.

[Cha88] David Chaum. "The dining cryptographers problem: Unconditional sender and recipient untraceability". In: *Journal of Cryptology* 1.1 (1988), pp. 65–75. ISSN: 0933-2790. DOI: 10.1007/BF00206326.

[Cim+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Computer Aided Verification (CAV)*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29.

[Cor+15] Andrés Cordón-Franco, Hans van Ditmarsch, David Fernández-Duque, and Fernando Soler-Toscano. "A geometric protocol for cryptography with cards". In: *Designs, Codes and Cryptography* 74.1 (2015), pp. 113–125. ISSN: 0925-1022. DOI: 10.1007/s10623-013-9855-y.

[CS15] Tristan Charrier and François Schwarzentruber. "Arbitrary Public Announcement Logic with Mental Programs". In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS. 2015, pp. 1471–1479. URL: https://dl.acm.org/citation.cfm?id=2772879.2773340.

[DEW10] Hans van Ditmarsch, Jan van Eijck, and William Wu. "One Hundred Prisoners and a Lightbulb — Logic and Computation". In: *KR 2010*. Toronto, Ontario, Canada: AAAI Press, 2010, pp. 90–100. ISBN: 978-1-5773-545-12. URL: https://www.aaai.org/ocs/index.php/KR/KR2010/paper/view/1234.

[DG14] David Fernández Duque and Valentin Goranko. "Secure aggregation of distributed information". In: *CoRR* abs/1407.7582 (2014). URL: https://arxiv.org/abs/1407.7582.

[DHK07] Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic epistemic logic*. Springer, 2007. ISBN: 978-1-4020-5838-7. DOI: 10.1007/978-1-4020-5839-4.

[DHR13] Hans van Ditmarsch, Wiebe van der Hoek, and Ji Ruan. “Connecting dynamic epistemic and temporal epistemic logics”. In: *Logic Journal of IGPL* 21.3 (2013), pp. 380–403. DOI: 10.1093/jigpal/jzr038.

[Dit+06] Hans van Ditmarsch, Wiebe van der Hoek, Ron van der Meyden, and Ji Ruan. “Model Checking Russian Cards.” In: *Electronic Notes in Theoretical Computer Science* 149.2 (2006), pp. 105–123. DOI: 10.1016/j.entcs.2005.07.029.

[Dit+17] Hans van Ditmarsch, Michael Ian Hartley, Barteld Kooi, Jonathan Welton, and Joseph B.W. Yeo. “Cheryl’s Birthday”. In: *TARK 2017*. 2017, pp. 1–9. DOI: 10.4204/EPTCS.251.1.

[Dit03] Hans van Ditmarsch. “The Russian Cards problem”. In: *Studia Logica* 75.1 (2003), pp. 31–62. DOI: 10.1023/A:1026168632319.

[DR07] Hans van Ditmarsch and Ji Ruan. “Model Checking Logic Puzzles”. In: *Annales du Lamsade* 8 (2007). URL: https://hal.archives-ouvertes.fr/hal-00188953.

[DS11] Hans van Ditmarsch and Fernando Soler–Toscano. “Three Steps”. In: *CLIMA 2011: Computational Logic in Multi-Agent Systems*. 2011, pp. 41–57. DOI: 10.1007/978-3-642-22359-4_4.

[EG15] Jan van Eijck and Malvin Gattinger. “Elements of Epistemic Crypto Logic”. In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’15. Istanbul, Turkey: International Foundation for Autonomous Agents and Multiagent Systems, 2015, pp. 1795–1796. ISBN: 978-1-4503-3413-6. URL: https://dl.acm.org/citation.cfm?id=2773441.

[Eij07] Jan van Eijck. “DEMO—a demo of epistemic modelling”. In: *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London*. Vol. 1. 2007, pp. 303–362. URL: http://homepages.cwi.nl/~jve/papers/07/pdfs/DEMO_IL.pdf.

[Eij14] Jan van Eijck. *DEMO-S5*. Tech. rep. CWI, 2014. URL: http://homepages.cwi.nl/~jve/software/demo_s5.

[Eng+17] Thorsten Engesser, Thomas Bolander, Robert Mattmüuller, and Bernhard Nebel. “Cooperative Epistemic Multi-Agent Planning for Implicit Coordination”. In: *Methods for Modalities 2017*. Ed. by Sujata Ghosh and R. Ramanujam. 2017, pp. 75–90. DOI: 10.4204/EPTCS.243.6.

[Fag+95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about knowledge*. Vol. 4. MIT press Cambridge, 1995. ISBN: 9780262061629.

[Fre69] Hans Freudenthal. “Formulering van het ’som-en-product’-probleem”. In: *Nieuw Archief voor Wiskunde* 17 (1969), p. 152.

[Gat17] Malvin Gattinger. *HasCacBDD*. Version 0.1.0.0. Mar. 9, 2017. URL: https://github.com/m4lvin/HasCacBDD.

[Gat18] Malvin Gattinger. “New Directions in Model Checking Dynamic Epistemic Logic”. PhD thesis. University of Amsterdam, 2018. ISBN: 978-94-028-1025-7. URL: https://malv.in/phdthesis.

[GK16] Valentin Goranko and Louwe B. Kuijer. “On the Length and Depth of Temporal Formulae Distinguishing Non-bisimilar Transition Systems”. In: *23rd International Symposium on Temporal Representation and Reasoning (TIME)*. 2016, pp. 177–185. DOI: 10.1109/TIME.2016.26.

[GM04] Peter Gammie and Ron van der Meyden. “MCK: Model Checking the Logic of Knowledge”. In: *Computer Aided Verification*. Springer, 2004, pp. 479–483. DOI: 10.1007/978-3-540-27813-9_41.

[GR02] Nikos Gorogiannis and Mark D. Ryan. “Implementation of Belief Change Operators Using BDDs”. In: *Studia Logica* 70.1 (2002), pp. 131–156. ISSN: 0039-3215. DOI: 10.1023/A:1014610426691.

[GS11] Nina Gierasimczuk and Jakub Szymanik. "A note on a generalization of the Muddy Children puzzle". In: *TARK 2011*. Ed. by Krzysztof R. Apt. ACM, 2011, pp. 257–264. ISBN: 978-1-4503-0707-9. DOI: 10.1145/2000378.2000409.

[Lit53] J.E. Littlewood. *A Mathematician's Miscellany*. London: Methuen, 1953. URL: https://archive.org/details/mathematiciansmi033496mbp.

[LPW11] Benedikt Löwe, Eric Pacuit, and Andreas Witzel. "DEL Planning and Some Tractable Cases". In: *Logic, Rationality, and Interaction – LORI*. Ed. by Hans van Ditmarsch, Jérôme Lang, and Shier Ju. Lecture Notes in Computer Science. 2011, pp. 179–192. ISBN: 978-3-642-24130-7.

[LQR15] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. "MCMAS: an open-source model checker for the verification of multi-agent systems". In: *International Journal on Software Tools for Technology Transfer* (2015), pp. 1–22. ISSN: 1433-2779. DOI: 10.1007/s10009-015-0378-x.

[LSX13] Guanfeng Lv, Kaile Su, and Yanyan Xu. "CacBDD: A BDD Package with Dynamic Cache Management". In: *Proceedings of the 25th International Conference on Computer Aided Verification*. CAV'13. Saint Petersburg, Russia: Springer, 2013, pp. 229–234. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_15.

[Luo+08] Xiangyu Luo, Kaile Su, Abdul Sattar, and Yan Chen. "Solving Sum and Product Riddle via BDD-Based Model Checking". In: *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. IEEE, 2008, pp. 630–633. DOI: 10.1109/WIIAT.2008.277.

[OSu16] Bryan O'Sullivan. *Criterion*. 2016. URL: http://www.serpentine.com/criterion.

[Som12] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 2.5.0*. 2012.