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

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

Last Update: Monday 26th February, 2018

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. We add factual change in the S5 setting in Section 4. In Section 5 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 6 and 7 then generalize our implementation to general Kripke models where the accessibility relations need not be equivalence relations, and their symbolic equivalents called belief structures. In Sections 8 and 9 we implement action models with factual change and their symbolic equivalent called transformers. Again we implement translations in Sections 10 and 11.

To check that the implementations are correct we provide methods for automated randomized testing in Section 12 using QuickCheck and HSpec.

In Section 13 we provide some helper functions for epistemic planning.

Section 14 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. These examples also suggest themselves as benchmarks which we will do in Section 15 to compare the different versions of our model checker to the existing tools DEMO-S5 and MCMAS.

In Section 16.1 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 16.2 implements a web interface.

The last Section 17 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 TypeSynonymInstances, FlexibleInstances #-}

module SMCDEL.Language where
import Data.List (nub,intercalate,(\\))
import Data.Maybe (fromMaybe)
import Test.QuickCheck
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

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

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"

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

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

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

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

-- generate a non-empty group of up to 5 agents
instance Arbitrary Group where
  arbitrary = fmap (Group.("1":)) $ sublistOf $ map show [2..(5::Integer)]
```

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

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

where $p \in V$, $P \subseteq V$, $|P| < \omega$, $\Phi \subseteq \mathcal{L}_{\mathsf{DEL}}$, $|\Phi| < \omega$, $i \in I$ and $\Delta \subset I$. We also write $\varphi \wedge \psi$ 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. Another example are $C_{\{i\}}\varphi$ and $K_i\varphi$.

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

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

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

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

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

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

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

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

class HasPrecondition a where
  preOf :: a -> Form

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{" ++ intercalate "," (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 " ++ intercalate "," is ++ " " ++ ppFormWith trans f
ppFormWith trans (Kw i f)      = "Kw " ++ i ++ " " ++ ppFormWith trans f
ppFormWith trans (Ckw is f)    = "Ckw " ++ intercalate "," is ++ " " ++ ppFormWith trans f
ppFormWith trans (PubAnnounce f g)  = "[! " ++ ppFormWith trans f ++ "] " ++ ppFormWith
    trans g
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
```

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

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

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

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

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

We define a list of subformulas as follows, including the given formula itself. In particular this can be used to make QuickCheck `shrink` functions.

```
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)

shrinkform :: Form -> [Form]
shrinkform f | f == simplify f = subformulas f \\ [f]
             | otherwise       = let g = simplify f in subformulas g \\ [g]
```

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.

```
substit :: Prp -> Form -> Form -> Form
substit _ _   Top           = Top
substit _ _   Bot           = Bot
substit q psi (PrpF p)      = if p==q then psi else PrpF p
substit q psi (Neg form)    = Neg (substit q psi form)
substit q psi (Conj forms)  = Conj (map (substit q psi) forms)
substit q psi (Disj forms)  = Disj (map (substit q psi) forms)
substit q psi (Xor  forms)  = Xor  (map (substit q psi) forms)
```

```
substit q psi (Impl f g)    = Impl (substit q psi f) (substit q psi g)
substit q psi (Equi f g)    = Equi (substit q psi f) (substit q psi g)
substit q psi (Forall ps f) = if q `elem` ps
  then error ("substit failed: Substituens "++ show q ++ " in 'Forall " ++ show ps ++ " "
      ++ 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)
```

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.

```
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 $[A \sqsubseteq B]\varphi$.

```
substitOutOf :: [Prp] -> [Prp] -> Form -> Form
substitOutOf truths allps = substitSet $ [(p,Top) | p <- truths] ++ [(p,Bot) | p <- allps
    \\ truths]
```

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

```
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)
```

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

```
propsInForm :: Form -> [Prp]
propsInForm Top                = []
propsInForm Bot                = []
propsInForm (PrpF p)           = [p]
propsInForm (Neg f)            = propsInForm f
propsInForm (Conj fs)          = nub $ concatMap propsInForm fs
```

```
propsInForm (Disj fs)          = nub $ concatMap propsInForm fs
propsInForm (Xor  fs)          = nub $ concatMap propsInForm fs
propsInForm (Impl f g)         = nub $ concatMap propsInForm [f,g]
propsInForm (Equi f g)         = nub $ concatMap propsInForm [f,g]
propsInForm (Forall ps f)      = nub $ ps ++ propsInForm f
propsInForm (Exists ps f)      = nub $ ps ++ propsInForm f
propsInForm (K _ f)            = propsInForm f
propsInForm (Kw _ f)           = propsInForm f
propsInForm (Ck _ f)           = propsInForm f
propsInForm (Ckw _ f)          = propsInForm f
propsInForm (Announce _ f g)   = nub $ propsInForm f ++ propsInForm g
propsInForm (AnnounceW _ f g)  = nub $ propsInForm f ++ propsInForm g
propsInForm (PubAnnounce f g)  = nub $ propsInForm f ++ propsInForm g
propsInForm (PubAnnounceW f g) = nub $ propsInForm f ++ propsInForm g

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

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

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

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

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

simStep :: Form -> Form
simStep Top            = Top
simStep Bot            = Bot
simStep (PrpF p)       = PrpF p
simStep (Neg Top)      = Bot
simStep (Neg Bot)      = Top
simStep (Neg (Neg f)) = simStep f
simStep (Neg f)        = Neg $ simStep f
simStep (Conj [])      = Top
simStep (Conj [f])     = simStep f
simStep (Conj fs)      | Bot `elem` fs = Bot
                       | or [ Neg f `elem` fs | f <- fs ] = Bot
                       | otherwise      = Conj (nub $ 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)
```

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

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

instance TexAble Form where
  tex = 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:

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

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 (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 12.1 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
        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 5 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.GraphViz
import Data.List
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 $I = \{1, \ldots, n\}$ is a tuple $\mathcal{M} = (W, \pi, \mathcal{K}_1, \ldots, \mathcal{K}_n)$, 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 $\mathcal{K}_1, \ldots, \mathcal{K}_n$ 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]

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

data KripkeModelS5 = KrMS5 [World] [(Agent,Partition)] [(World,Assignment)] deriving (Eq,
    Ord,Show)
type PointedModelS5 = (KripkeModelS5,World)

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

instance HasAgents PointedModelS5 where
  agentsOf = agentsOf . fst

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

instance HasVocab PointedModelS5 where
  vocabOf = vocabOf . fst

instance HasWorlds KripkeModelS5 where
  worldsOf (KrMS5 ws _ _) = ws

instance HasWorlds PointedModelS5 where
  worldsOf = worldsOf . fst

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
    let agents = map show [1..(5::Int)]
    let props = map P [0..4]
    worlds <- sort . nub <$> listOf1 (elements [0..8])
    val <- mapM (\w -> do
      randomAssignment <- zip props <$> infiniteListOf (choose (True,False))
      return (w,randomAssignment)
      ) worlds
    parts <- mapM (\i -> do
      randomPartition <- randomPartFor worlds
      return (i,randomPartition)
      ) agents
    return $ KrMS5 worlds parts val
  shrink m@(KrMS5 worlds _ _) =
    [ m `withoutWorld` w | w <- worlds, length worlds > 1 ]
```

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

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

```
eval :: 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 (pubAnnounce pm form1) form2
eval pm (PubAnnounceW form1 form2) =
  if eval pm form1
    then eval (pubAnnounce pm form1) form2
    else eval (pubAnnounce pm (Neg form1)) form2
eval pm (Announce ags form1 form2) =
  not (eval pm form1) || eval (announce pm ags form1) form2
eval pm (AnnounceW ags form1 form2) =
  if eval pm form1
    then eval (announce pm ags form1) form2
    else eval (announce pm ags (Neg form1)) form2

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

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

```
pubAnnounce :: PointedModelS5 -> Form -> PointedModelS5
pubAnnounce pm@(m@(KrMS5 sts rel val), cur) form =
  if eval pm form then (KrMS5 newsts newrel newval, cur)
                  else error "pubAnnounce failed: Liar!"
  where
    newsts = filter (\s -> eval (m,s) form) sts
    newrel = map (second relfil) rel
    relfil = filter ([]/=) . map (filter (`elem` newsts))
    newval = filter (\p -> fst p `elem` newsts) val

announce :: 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)
```

With a few lines we can also visualize our models using the module `SMCDEL.Internal.TexDisplay`. For example output, see Sections 14.1.1 and 14.1.2.

```
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 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
```

2.2 Bisimulations

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

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

2.3 Minimization

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

```
generatedSubmodel :: PointedModelS5 -> PointedModelS5
generatedSubmodel (KrMS5 _ rel val, cur) = (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
```

2.4 S5 Action Models

To model epistemic change in general we use action models [BMS98]. For now we only consider S5 action models without factual change.

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

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

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

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

instance KripkeLike PointedActionModel where
  directed = const False
  getNodes (ActM as actval _, _) = map (show &&& labelOf) as where
    labelOf w = " $ ? " ++ tex (apply actval w) ++ " $ "
  getEdges (ActM _ _ rel, _) =
    nub [ (a, show x, show y) | a <- map fst rel, part <- apply rel a, x <- part, y <- part
        , x < y ]
  getActuals (ActM {}, cur) = [show cur]
  nodeAts _ True  = [shape BoxShape, style solid]
  nodeAts _ False = [shape BoxShape, style dashed]

instance TexAble PointedActionModel where tex = tex.ViaDot

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

productUpdate :: PointedModelS5 -> PointedActionModel -> PointedModelS5
productUpdate pm@(m@(KrMS5 oldWorlds oldrel oldval), oldcur) (ActM actions precon actrel,
    faction) =
  let
    startcount        = maximum oldWorlds + 1
    copiesOf (s,a)    = [ (s, a, a * startcount + s) | eval (m, s) (apply precon a) ]
    newWorldsTriples  = concat [ copiesOf (s,a) | s <- oldWorlds, a <- actions ]
    newWorlds         = map (\(_,_,x) -> x) newWorldsTriples
    newval            = map (\(s,_,t) -> (t, apply oldval s)) newWorldsTriples
    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 ]
    ((_,_,newcur):_)  = copiesOf (oldcur,faction)
    factTest          = apply precon faction
    cartProd xs ys    = [ (x,y) | x <- xs, y <- ys ]
  in case ( map fst oldrel == map fst actrel, eval pm factTest ) of
    (False, _) -> error "productUpdate failed: Agents of KrMS5 and ActM are not the same!"
    (_, False) -> error "productUpdate failed: Actual precondition is false!"
    _          -> (KrMS5 newWorlds newrel newval, newcur)
```

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 TypeSynonymInstances, FlexibleInstances #-}

module SMCDEL.Symbolic.S5 where
import Control.Arrow (first)
import Data.HasCacBDD hiding (Top,Bot)
import Data.HasCacBDD.Visuals
import Data.List (sort,intercalate,(\\))
import System.IO (hPutStr, hGetContents, hClose)
import System.IO.Unsafe (unsafePerformIO)
import System.Process (runInteractiveCommand)
import SMCDEL.Internal.Help (alleqWith,apply,(!),rtc,seteq,powerset)
import SMCDEL.Language
import SMCDEL.Internal.TexDisplay
```

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

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

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

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

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 6. *Suppose we have n agents. A* knowledge structure *is a tuple $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$ 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 $(V, \theta, O_1, \ldots, O_n)$ 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]

type KnowScene = (KnowStruct,State)

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

instance HasAgents KnowScene where agentsOf = agentsOf . fst
instance HasVocab  KnowScene where vocabOf  = vocabOf  . fst

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

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

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

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

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

1. $(\mathcal{F}, s) \vDash p$ *iff* $s \vDash p$.
2. $(\mathcal{F}, s) \vDash \neg\varphi$ *iff not* $(\mathcal{F}, s) \vDash \varphi$
3. $(\mathcal{F}, s) \vDash \varphi \wedge \psi$ *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 8 and*

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

7. $(\mathcal{F}, s) \models [\psi]_\Delta \varphi$ *iff* $(\mathcal{F}, s) \models \psi$ *implies* $(\mathcal{F}^\Delta_\psi, s \cup \{p_\psi\}) \models \varphi$ *where* p_ψ *is a new propositional variable,* $\|\psi\|_\mathcal{F}$ *is a boolean formula given by Definition 8 and*

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

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

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

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

```
eval :: 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 [ substit p Top g, substit p Bot g ]
eval scn     (Exists ps f) = eval scn (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
eval (kns@(KnS _ _ obs),s) (K i form) = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns@(KnS _ _ obs),s) (Kw i form) = alleqWith (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns,s) (Ck ags form)  = all (\s' -> eval (kns,s') form) theres where
  theres = [ 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 (pubAnnounceOnScn scn form1) form2
eval (kns,s) (PubAnnounceW form1 form2) =
  if eval (kns, s) form1
    then eval (pubAnnounce kns form1, s) form2
    else eval (pubAnnounce kns (Neg form1), s) form2
eval scn (Announce ags form1 form2) =
  not (eval scn form1) || eval (announceOnScn scn ags form1) form2
eval scn (AnnounceW ags form1 form2) =
  if eval scn form1
    then eval (announceOnScn scn ags form1      ) form2
    else eval (announceOnScn scn ags (Neg form1)) form2
```

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

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

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

announce :: KnowStruct -> [Agent] -> Form -> KnowStruct
announce kns@(KnS props lawbdd obs) ags psi = KnS newprops newlawbdd newobs where
  proppsi@(P k) = freshp props
  newprops  = proppsi:props
```

```
  newlawbdd = con lawbdd (equ (var k) (bddOf kns psi))
  newobs    = [(i, apply obs i ++ [proppsi | i `elem` ags]) | i <- map fst obs]

announceOnScn :: 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. It only makes calls to itself, the announcement functions and the boolean operations provided by the BDD package.

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

1. *For any primitive formula, let $\|p\|_{\mathcal{F}} := p$.*
2. *For negation, let $\|\neg\psi\|_{\mathcal{F}} := \neg\|\psi\|_{\mathcal{F}}$.*
3. *For conjunction, let $\|\psi_1 \wedge \psi_2\|_{\mathcal{F}} := \|\psi_1\|_{\mathcal{F}} \wedge \|\psi_2\|_{\mathcal{F}}$.*
4. *For knowledge, let $\|K_i\psi\|_{\mathcal{F}} := \forall(V \setminus O_i)(\theta \to \|\psi\|_{\mathcal{F}})$.*
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 7.

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

```
  ifthenelse (bddOf kns form1) bdd2a bdd2b where
    bdd2a = restrict (bddOf (announce kns ags form1) form2) (k,True)
    bdd2b = restrict (bddOf (announce kns ags form1) form2) (k,False)
    (P k) = freshp props
bddOf kns (PubAnnounce form1 form2) =
  imp (bddOf kns form1) (bddOf (pubAnnounce kns form1) form2)
bddOf kns (PubAnnounceW form1 form2) =
  ifthenelse (bddOf kns form1) newform2a newform2b where
    newform2a = bddOf (pubAnnounce kns form1) form2
    newform2b = bddOf (pubAnnounce kns (Neg form1)) form2
```

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

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

```
evalViaBdd :: KnowScene -> Form -> Bool
evalViaBdd (kns,s) f = evaluateFun (bddOf kns f) (\n -> P n `elem` s)

instance Semantics KnowScene where
  isTrue = evalViaBdd
```

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

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

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

```
whereViaBdd :: KnowStruct -> Form -> [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 Symbolic Bisimulations for S5

When are two knowledge structures equivalent? This question comes with a hidden parameter, namely the vocabulary for which we want them to be equivalent. If the structures have disjoint vocabularies, then there are no non-trivial formulas which can be interpreted on both. So we will assume that their vocabularies at least overlap. They do not have to be same though — for example they can use different auxiliary variables that encode epistemic relations.

The following definition describes a symbolic equivalent of bisimulations.

Definition 11. *Suppose we have two knowledge structures $\mathcal{F}_1 = (V_1, \theta_1, O_1^1, \ldots, O_1^n)$ and $\mathcal{F}_2 = (V_2, \theta_2, O_2^1, \ldots, O_2^n)$.*

A boolean formula $\beta \in \mathcal{L}_B(V \cup V^*)$ where $V := V_1 \cap V_2$ is called a bipropulation *between the two structures iff:*

- $\beta \to \bigwedge_{p \in V}(p \leftrightarrow p^*)$
- *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 $O_1^i \cap s_1 = O_1^i \cap t_1$ in F_1. Then there is a state t_2 of F_2 such that $t_1 \cup (t_2^*) \vDash \beta$ and $O_2^i \cap s_2 = O_2^i \cap t_2$ in F_2.*
- *and vice versa*

Note that all these conditions, in particular also (ii) and (iii) can be expressed as a boolean formula:

Lemma 12. *The following are all equivalent:*

- β *is a bipropulation*
- β *encodes a bisimiluation between the equivalent S5 Kripke models*
- *the following boolean formulas are tautologies, i.e. their BDDs and the single BDD of their conjunction are equal to* $\top$*:*

$$\beta \to \bigwedge_{p \in V} (p \leftrightarrow p^*)$$

$$\forall (V \cup V^*) : \beta \to \bigwedge_i \left(\forall (V \setminus O_1^i) : \exists (V^* \setminus (O_2^i)^*) : \beta' \right)$$

3.3 Knowledge Transformers

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

Definition 13. *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* $\mathcal{F} = (V, \theta, O_1, \dots, O_n)$ *with a knowledge transformer* $\mathcal{X} = (V^+, \theta^+, O_1^+, \dots, O_n^+)$ *for* V *is defined by:*

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

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 $((\varnothing, \varphi, \varnothing, \dots, \varnothing), \varnothing)$. The semi-private announcement of φ to a group of agents Δ is given by $((\{p_\varphi\}, p_\varphi \leftrightarrow \varphi, O_1^+, \dots, O_n^+), \{p_\varphi\})$ where $O_i^+ = \{p_\varphi\}$ if $i \in \Delta$ and $O_i^+ = \varnothing$ otherwise.

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

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

knowTransform :: KnowScene -> Event -> KnowScene
knowTransform (kns@(KnS props lawbdd obs),s) (KnT addprops addlaw eventobs, eventfacts) =
  (KnS (props ++ map snd shiftrel) newlawbdd newobs, s++shifteventfacts) where
    shiftrel = zip addprops [(freshp props)..]
    newobs = [ (i , sort $ apply obs i ++ map (apply shiftrel) (apply eventobs i)) | i <-
        map fst obs ]
    shiftaddlaw = replPsInF shiftrel addlaw
    newlawbdd = con lawbdd (bddOf kns shiftaddlaw)
    shifteventfacts = map (apply shiftrel) eventfacts
```

We end this module with helper functions to generate LaTeX code that shows a given knowledge structure, including a BDD of the state law. See Section 14.1 for examples of what the output looks like.

```haskell
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
  hGetContents o

texBDD :: Bdd -> String
texBDD = texBddWith show

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 Event where
  tex (KnT props changelaw obs, actual) = concat
    [ " \\left( \n"
    , tex props ++ ", "
    , tex changelaw, "\n "
    , ", \\begin{array}{l}\n"
    , intercalate " \\\\\n " (map (\(_,os) -> (tex os)) obs)
    , "\\end{array}\n"
    , " \\right) , "
    , tex actual ]
```

3.4 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.

First, we reduce Formulas to Formulas. This translation is global, i.e. if there is a reduced formula, then it s equivalent to the original on all structures.

```haskell
pre :: Event -> Form
pre (KnT addprops changelaw _, x) = substitOutOf x addprops changelaw

reduce :: Event -> Form -> Maybe Form
reduce _ Top           = Just Top
reduce e Bot           = pure $ Neg (pre e)
reduce e (PrpF p)      = Impl (pre e) <$> Just (PrpF p)
reduce e (Neg f)       = Impl (pre e) <$> (Neg <$> reduce e f)
reduce e (Conj fs)     = Conj <$> mapM (reduce e) fs
reduce e (Disj fs)     = Disj <$> mapM (reduce e) fs
reduce e (Xor fs)      = Impl (pre e) <$> (Xor <$> mapM (reduce e) fs)
reduce e (Impl f1 f2) = Impl <$> reduce e f1 <*> reduce e f2
reduce e (Equi f1 f2) = Equi <$> reduce e f1 <*> reduce e f2
reduce _ (Forall _ _) = Nothing
reduce _ (Exists _ _) = Nothing
reduce e@(t@(KnT addprops _ obs), x) (K a f) =
  Impl (pre e) <$> (Conj <$> sequence
    [ K a <$> reduce (t,y) f | y <- powerset addprops -- FIXME is this a bit much?
                             , seteq (restrictState x (obs ! a)) (restrictState y (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
```

Second, we can extend the boolean translation given by `bddOff` to also cover dynamic operators for knowledge transformers. However, this is no longer global but with respect to a given knowledge structure. Fo convenience and to use `knowTransform` we take a scene as input, but note that the actual state is not used in `bddReduce`, only in `evalViaBddReduce`.

```
bddReduce :: KnowScene -> Event -> Form -> Bdd
bddReduce scn event f = restrictSet (bddOf newKns f) actualAss where
  (newKns,_) = knowTransform scn event
  (KnT eventprops _ _, actual) = event
  actualAss = [(k, P k `elem` actual) | (P k) <- eventprops]

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

It is crucial to not use `bddOf newKns (substitOutOf actual eventprops f)` here, because the substitution function `substitOutOf` has to remove all `eventprops` again, *after bddOf* with respect to the new structure might have introduced them.

4 Knowledge Transformers with Factual Change

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

module SMCDEL.Symbolic.S5.Change where

import Control.Lens (over,both)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List
import qualified Data.Map.Strict as M
import Data.Map.Strict ((!))

import SMCDEL.Language
import SMCDEL.Internal.TexDisplay
import SMCDEL.Other.BDD2Form
import SMCDEL.Internal.Help (apply,applyPartial)
import SMCDEL.Symbolic.S5 hiding (Event)

data KnowChange = CTrf
  [Prp]               -- addprops
  Form                -- event law
  [Prp]               -- changeprops, modified subset
  (M.Map Prp Bdd)     -- changelaw
  (M.Map Agent [Prp]) -- eventObs
  deriving (Show)

instance HasAgents KnowChange where
  agentsOf (CTrf _ _ _ _ obdds) = M.keys obdds

type Event = (KnowChange,State)

instance HasAgents Event where
  agentsOf = agentsOf . fst

type MultiEvent = (KnowChange,[State])

instance TexAble KnowChange where
  tex (CTrf addprops addlaw changeprops changelaw eventObs) = concat
    [ " \\left( \n"
    , tex addprops, ", \\ "
    , tex addlaw, ", \\ "
    , tex changeprops, ", \\ "
    , intercalate ", " $ map snd . M.toList $ M.mapWithKey texChange changelaw
    , ", \\ \\begin{array}{l}\n"
    , intercalate " \\\\\n " (map (\(_,os) -> (tex os)) (M.toList eventObs))
    , "\\end{array}\n"
    , " \\right) \n"
    ] where
        texChange prop changebdd = tex prop ++ " := " ++ tex (formOf changebdd)

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

The following defines S5 transformation with factual change.

```
knowChange :: KnowScene -> Event -> KnowScene
knowChange (kns@(KnS props law obs),s) (CTrf addprops addlaw changeprops changelaw eventObs
    , eventFacts) =
  (KnS newprops newlaw newobs, news) where
    relabelWith r = relabel (sort $ map (over both fromEnum) r)
    -- PART 1: SHIFTING addprops to ensure props and newprops are disjoint
    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
    eventObsShifted  = M.map (map (apply shiftrel)) eventObs
    -- the actual event:
    x = map (apply shiftrel) eventFacts
    -- PART 2: COPYING the modified propositions
```

```
    copyrel = zip changeprops [(freshp $ props ++ shiftaddprops)..]
    copychangeprops = map snd copyrel
    newprops = sort $ props ++ shiftaddprops ++ copychangeprops -- V ∪ V⁺ ∪ V°
    newlaw = conSet $ relabelWith copyrel (con law (bddOf kns addlawShifted))
                    : [var (fromEnum q) `equ` relabelWith copyrel (changelawShifted ! q) |
                        q <- changeprops]
    newobs = [ (i , sort $ map (applyPartial copyrel) (apply obs i) ++ eventObsShifted ! i)
         | i <- map fst obs ]
    news | bddEval (s ++ x) (con law (bddOf kns addlawShifted)) = sort $ concat
            [ s \\ changeprops
            , map (apply copyrel) $ s `intersect` changeprops
            , x
            , filter (\ p -> bddEval (s ++ x) (changelawShifted ! p)) changeprops ]
         | otherwise = error "Transformer is not applicable!"
```

We now do a simple example: publicly make p false.

```
myStart :: KnowScene
myStart = (KnS [P 0] (boolBddOf Top) [("Alice",[]),("Bob",[P 0])],[P 0])

publicMakeFalse :: [Agent] -> Prp -> Event
publicMakeFalse agents p = (CTrf [] Top [p] mychangelaw myobs, []) where
  mychangelaw = M.fromList [ (p,boolBddOf Bot) ]
  myobs = M.fromList [ (i,[]) | i <- agents ]

myEvent :: Event
myEvent = publicMakeFalse (agentsOf myStart) (P 0)

myResult :: KnowScene
myResult = myStart `knowChange` myEvent
```

The structure ...

$$\left(\{p\},\ \boxed{1}\ ,\ \begin{matrix}\varnothing\\ \{p\}\end{matrix}\right),\{p\}$$

... transformed with ...

$$\left(\left(\varnothing,\ \top,\ \{p\},\ p := \bot,\ \begin{matrix}\varnothing\\ \varnothing\end{matrix}\right),\ \varnothing\right)$$

... yields this new structure:

$$\left(\{p,p_1\},\ \begin{array}{c}\overset{0}{\bigcirc}\\ \boxed{0}\quad\boxed{1}\end{array}\ ,\ \begin{matrix}\varnothing\\ \{p_1\}\end{matrix}\right),\{p_1\}$$

Something more involved, making it false but still keeping it secret from Alice.

```
exampleStart :: KnowScene
exampleStart = (KnS [P 0] (boolBddOf Top) [("Alice",[]),("Bob",[P 0])],[P 0])

makeFalseShowTo :: [Agent] -> Prp -> [Agent] -> Event
makeFalseShowTo agents p intheknow = (CTrf [P 99] Top [p] examplechangelaw exampleobs, [])
    where
  examplechangelaw = M.fromList [ (p,boolBddOf $ PrpF (P 99)) ]
  exampleobs = M.fromList $ [ (i,[P 99]) | i <- intheknow           ]
                          ++ [ (i,[    ]) | i <- agents \\ intheknow ]

exampleEvent :: Event
exampleEvent = makeFalseShowTo (agentsOf exampleStart) (P 0) ["Bob"]

exampleResult :: KnowScene
exampleResult = exampleStart `knowChange` exampleEvent
```

The structure ...

$$\left(\{p\},\ \boxed{1}\ ,\ \begin{matrix}\varnothing\\ \{p\}\end{matrix}\right),\{p\}$$

. . . transformed with . . .

$$\left(\left(\{p_{99}\},\ \top,\ \{p\},\ p := p_{99},\ \begin{matrix}\varnothing\\ \{p_{99}\}\end{matrix}\right),\ \varnothing\right)$$

. . . yields this new structure:

$$\left(\{p, p_1, p_2\},\ \ \begin{matrix}\ & 0 & \ \\ 1 & & 1 \\ 1 & & 0\end{matrix}\ \ ,\ \begin{matrix}\varnothing\\ \{p_1, p_2\}\end{matrix}\right), \{p_2\}$$

And alternatively, showing the result only to Alice:

```
thirdEvent :: Event
thirdEvent = makeFalseShowTo (agentsOf exampleStart) (P 0) ["Alice"]

thirdResult :: KnowScene
thirdResult = exampleStart `knowChange` thirdEvent
```

The same structure . . .

$$\left(\{p\},\ \boxed{1}\ ,\ \begin{matrix}\varnothing\\ \{p\}\end{matrix}\right), \{p\}$$

. . . transformed with . . .

$$\left(\left(\{p_{99}\},\ \top,\ \{p\},\ p := p_{99},\ \begin{matrix}\{p_{99}\}\\ \varnothing\end{matrix}\right),\ \varnothing\right)$$

. . . yields this new structure:

$$\left(\{p, p_1, p_2\},\ \ \begin{matrix}\ & 0 & \ \\ 1 & & 1 \\ 1 & & 0\end{matrix}\ \ ,\ \begin{matrix}\{p_1\}\\ \{p_2\}\end{matrix}\right), \{p_2\}$$

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

Lemma 14. *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 14: 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]
    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
```

5.1 From Knowledge Structures to S5 Kripke Models

Definition 15. *For any knowledge structure $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$, we define the Kripke model $\mathcal{M}(\mathcal{F}) := (W, \pi, \mathcal{K}_1, \ldots, \mathcal{K}_n)$ 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 16. *For any knowledge structure $\mathcal{F}$, any state s of $\mathcal{F}$, and any φ we have $(\mathcal{F}, s) \models \varphi$ iff $(M(\mathcal{F}), s) \models \varphi$.*

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

knsToKripkeWithG :: KnowScene -> (PointedModelS5, StateMap)
knsToKripkeWithG (kns@(KnS ps _ obs),curs) =
  if curs `elem` statesOf kns
     then ((KrMS5 worlds rel val, cur) , g)
     else error "knsToKripke failed: Invalid state."
  where
    lav    = zip (statesOf kns) [0..(length (statesOf kns)-1)]
    val    = map ( \(s,n) -> (n,state2kripkeass s) ) lav where
      state2kripkeass s = map (\p -> (p, p `elem` s)) ps
    rel    = [(i,rfor i) | i <- map fst obs]
    rfor i = map (map snd) (groupBy ( \ (x,_) (y,_) -> x==y ) (sort pairs)) where
      pairs = map (\s -> (restrictState s (obs ! i), lav ! s)) (statesOf kns)
    worlds = map snd lav
    cur    = lav ! curs
    g w    = statesOf kns !! w
```

5.2 From S5 Kripke Models to Knowledge Structures

Definition 17. *For any S5 model $\mathcal{M} = (W, \pi, \mathcal{K}_1, \ldots, \mathcal{K}_n)$ with the set of atomic propositions U we define a knowledge structure $\mathcal{F}(\mathcal{M})$ as follows. For each agent i, write $\gamma_{i,1}, \ldots, \gamma_{i,k_i}$ 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 $O_1, \ldots, O_n$, 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 $O_{\gamma_{i,1}}, \ldots, O_{\gamma_{i,k_i}}$ 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 $\mathcal{F}(\mathcal{M}) := (V, \theta_M, O_1, \ldots, O_n)$ 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 18. *For any finite pointed S5 Kripke model $(\mathcal{M}, w)$ and every formula φ, we have that $(\mathcal{M}, w) \models \varphi$ iff $(\mathcal{F}(\mathcal{M}), g(w)) \models \varphi$.*

```
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,_) -> elem w 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
```

5.3 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 15 and 17 and their soundness also follows from Lemma 14. The implementation below works on pointed models, to simplify tracking the actual world and action.

Definition 19. *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 17. 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 20. *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 15 and* $g_{\mathcal{A}}$ *is from the construction of* $\mathsf{Trf}(\mathcal{A})$ *in Definition 19.*

```
actionToEvent :: PointedActionModel -> Event
actionToEvent (ActM actions precon actrel, faction) = (KnT eprops elaw eobs, efacts) where
  ags          = map fst actrel
  eprops       = actionprops ++ actrelprops
  (P fstnewp)  = freshp $ propsInForms (map snd precon)
  actionprops  = [P fstnewp..P maxactprop] -- new props to distinguish all actions
  maxactprop   = fstnewp + ceiling (logBase 2 (fromIntegral $ length actions) :: Float) -1
  copyactprops = zip actions (powerset actionprops)
  ell          = apply copyactprops -- label actions with subsets of actionprops
  happens a    = booloutofForm (ell a) actionprops -- boolean formula to say that a happens
  actform      = Disj [ Conj [ happens a, apply precon a ] | a <- actions ] -- 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,_) -> elem faction as) (copyactrel i)
  efacts       = apply copyactprops faction ++ concatMap factsFor ags
  elaw         = simplify $ Conj (actform : actrelforms)
  eobs         = [ (i,newps i) | i<- ags ]
```

5.4 From Knowledge Transformers to S5 Action Models

Definition 21. *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 22. *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 -> PointedActionModel
eventToAction' (KnT eprops eform eobs, efacts) = (ActM actions precon actrel, faction)
    where
  actions   = [0..(2 ^ length eprops - 1)]
  actlist   = zip (powerset eprops) actions
  precon    = [ (a, simplify $ preFor ps) | (ps,a) <- actlist ] where
    preFor ps = substitSet (zip ps (repeat Top) ++ zip (eprops\\ps) (repeat Bot)) eform
  actrel    = [(i,rFor i) | i <- map fst eobs] where
    rFor i  = map (map snd) (groupBy ( \ (x,_) (y,_) -> x==y ) (pairs i))
    pairs i = sort $ map (\(set,a) -> (restrictState set $ apply eobs i,a)) actlist
  faction   = apply actlist efacts

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

6 General Kripke Models

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

module SMCDEL.Explicit.K where

import Control.Arrow ((&&&))
import Data.GraphViz
import Data.List (nub,sort,(\\),delete,elemIndex)
import Data.Map.Strict (Map,fromList,elems,(!),mapMaybeWithKey)
import qualified Data.Map.Strict

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

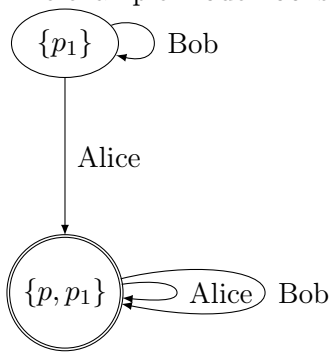
In non-S5 Kripke models every agent has an arbitrary relation on the states, not necessarily and equivalence relation.

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 (Map World (Map Prp Bool, Map Agent [World]))
  deriving (Eq,Ord,Show)

type PointedModel = (KripkeModel, World)

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

instance HasWorlds KripkeModel where
  worldsOf (KrM m) = Data.Map.Strict.keys m

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

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

instance HasWorlds PointedModel where worldsOf = worldsOf . fst
instance HasVocab PointedModel  where vocabOf  = vocabOf  . fst
instance HasAgents PointedModel where agentsOf = agentsOf . fst

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

truthsInAt :: KripkeModel -> World -> [Prp]
truthsInAt (KrM m) w = Data.Map.Strict.keys (Data.Map.Strict.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 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

exampleModel :: KripkeModel
exampleModel = KrM $ fromList
  [ (1, (fromList [(P 0,True ),(P 1,True )], fromList [(alice,[1]), (bob,[1])] ) )
  , (2, (fromList [(P 0,False),(P 1,True )], fromList [(alice,[1]), (bob,[2])] ) ) ]

examplePointedModel :: PointedModel
examplePointedModel = (exampleModel,1)
```

The example model looks as follows:



As a reference, we also implement general 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 (pubAnnounceNonS5 m f,w) g
eval (m,w) (PubAnnounceW f g) = eval (pubAnnounceNonS5 m aform, w) g where
  aform | eval (m,w) f = f
        | otherwise    = Neg f
eval (m,w) (Announce listeners f g) = not (eval (m,w) f) || eval newm g where
  newm = (m,w) `productUpdate` announceActionNonS5 (agentsOf m) listeners f
eval (m,w) (AnnounceW listeners f g) = eval newm g where
  newm = (m,w) `productUpdate` announceActionNonS5 (agentsOf m) listeners aform
  aform | eval (m,w) f = f
        | otherwise    = Neg f

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

pubAnnounceNonS5 :: KripkeModel -> Form -> KripkeModel
pubAnnounceNonS5 (KrM m) f = KrM newm where
  newm = mapMaybeWithKey isin m
  isin w' (v,rs) | eval (KrM m,w') f = Just (v,Data.Map.Strict.map newr rs)
                 | otherwise         = Nothing
  newr = filter (`elem` Data.Map.Strict.keys newm)

announceActionNonS5 :: [Agent] -> [Agent] -> Form -> PointedActionModel
```

```
announceActionNonS5 everyone listeners f = (AM am, 1) where
  am = fromList
    [ (1, (f   , fromList $ [(i,[1]) | i <- listeners] ++ [(i,[2]) | i <- everyone \\
        listeners]))
    , (2, (Top, fromList    [(i,[2]) | i <- everyone]) ) ]
```

Note that group announcements are implemented using general action models which are described below.

6.1 Muddy Children on general models

```
mudGenKrpInit :: Int -> Int -> PointedModel
mudGenKrpInit n m = (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 :: PointedModel
myMudGenKrpInit = mudGenKrpInit 3 3
```

6.2 Minimization of general models

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

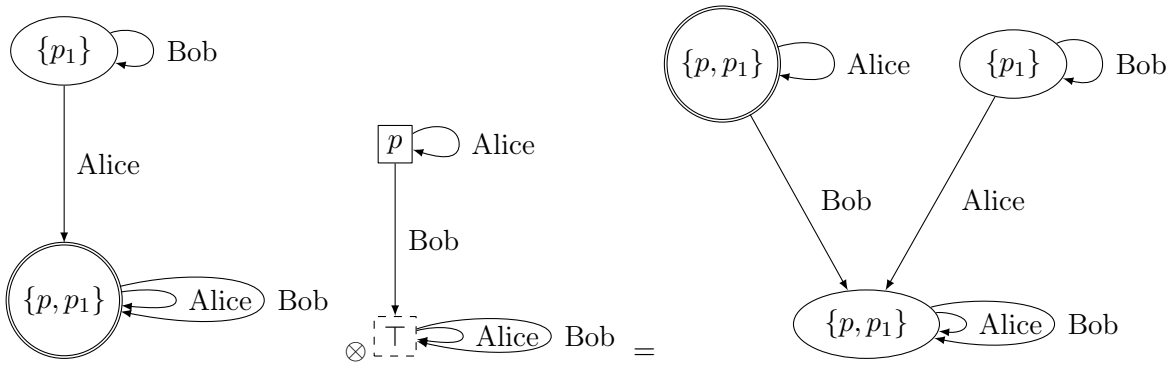
6.3 General Action Models

To model belief change we also need non-S5 action models.

```
newtype ActionModel = AM (Map Action (Form, Map Agent [Action]))
  deriving (Eq,Ord,Show)
type PointedActionModel = (ActionModel, Action)

eventsOf :: ActionModel -> [Action]
eventsOf (AM am) = Data.Map.Strict.keys am

instance HasAgents ActionModel where
  agentsOf (AM am) = Data.Map.Strict.keys $ snd (head (Data.Map.Strict.elems am))

relOfInAM :: Agent -> ActionModel -> Map Action [Action]
relOfInAM i (AM am) = Data.Map.Strict.map (\x -> snd x ! i) am

instance KripkeLike PointedActionModel where
  directed = const True
  getNodes (AM am, _) = map (show &&& labelOf) (eventsOf (AM am)) where
    labelOf a = "$" ++ tex (fst $ am ! a) ++ "$"
  getEdges (AM am, _) = concat [ [ (a,show w,show v) | v <- relOfInAM a (AM am) ! w ] | w
      <- eventsOf (AM am), a <- agentsOf (AM am) ]
  getActuals (_, cur) = [show cur]
  nodeAts _ True  = [shape BoxShape, style solid]
  nodeAts _ False = [shape BoxShape, style dashed]
```

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

productUpdate :: PointedModel -> PointedActionModel -> PointedModel
productUpdate (KrM m, oldcur) (AM am, act)
  | agentsOf (KrM m) /= agentsOf (AM am)    = error "productUpdate failed: Agents of KrM
      and GAM are not the same!"
  | not $ eval (KrM m, oldcur) (fst $ am ! act) = error "productUpdate failed: Actual
      precondition is false!"
  | otherwise = (KrM $ fromList (map annotate statePairs), newcur) where
    statePairs = zip [ (s, a) | s <- worldsOf (KrM m), a <- eventsOf (AM am), eval (KrM m,
        s) (fst $ am ! a) ] [0..]
    annotate ((s,a),news) = (news , (fst $ m ! s, fromList (map reachFor (agentsOf (KrM m))
        ))) where
      reachFor i = (i, [ news' | ((s',a'),news') <- statePairs, s' `elem` snd (m !  s) ! i,
          a' `elem` snd (am ! a) ! i ])
    (Just newcur) = lookup (oldcur,act) statePairs

-- Privately tell alice that P 0, while bob thinks nothing happens.
exampleGenActM :: ActionModel
exampleGenActM = AM $ fromList
  [ (1, (PrpF (P 0), fromList [(alice,[1]), (bob,[2])] ) )
  , (2, (Top       , fromList [(alice,[2]), (bob,[2])] ) ) ]

examplePointedActM :: PointedActionModel
examplePointedActM = (exampleGenActM,1)

exampleResult :: PointedModel
exampleResult = productUpdate examplePointedModel examplePointedActM
```

Now we can do a full example:



7 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 17 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.

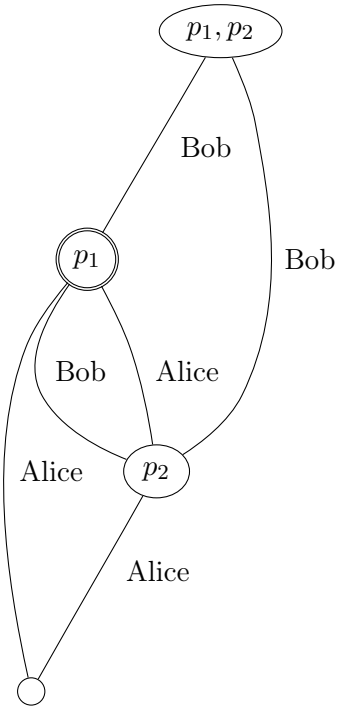
Our symbolic model checker can then also be used for non-S5 models.

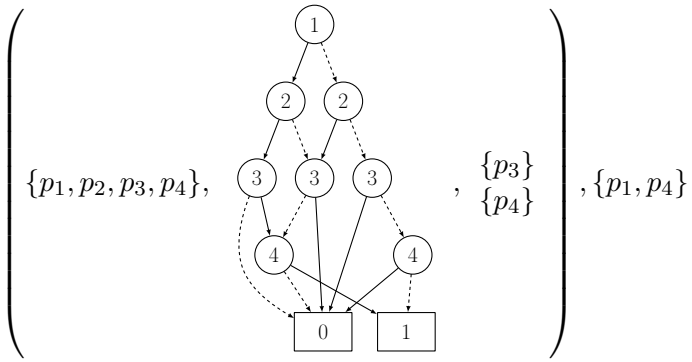
For further explanations, see [Ben+17, Section 8].

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

module SMCDEL.Symbolic.K where

import Data.Tagged

import Control.Arrow ((&&&),(***),first)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List (intercalate,sort)
import Data.Map.Strict (Map,fromList,toList,elems,(!))
import qualified Data.Map.Strict

import SMCDEL.Language
import SMCDEL.Symbolic.S5 (KnowScene,State,texBDD,boolBddOf,texBddWith)
import SMCDEL.Explicit.S5 (PointedModelS5,KripkeModelS5(KrMS5),World)
import SMCDEL.Explicit.K
import SMCDEL.Translations.S5
import SMCDEL.Internal.Help (apply,lfp)
import SMCDEL.Internal.TexDisplay
```

7.1 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
```

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

7.2 Translating relations to type-safe BDDs

To represent relations as BDDs we use the following well-known method from model checking. Remember that in a knowledge structure we can identify states with boolean assignments. Furthermore, if we fix a global set of variables, those are just sets of propositions. Hence `Rel State = [(State,State)] = [([Prp],[Prp])]`, i.e. a relation on KNS states is in fact a relation on sets of propositions. We can therefore represent it with the OBDD of a characteristic function on a double vocabulary, as described in [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 the assignment $\{p, p_3, p_2'\}$ satisfies the formula representing R.

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

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

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

Table 1: Implementation of single and double vocabulary.

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

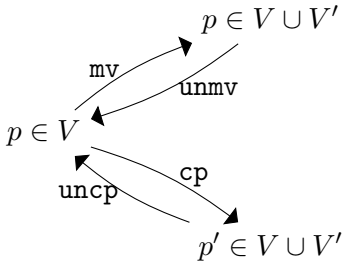


Figure 1: The functions mv, cp, unmv and uncp

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 = pure $ case maxVarOf b of
  Nothing -> b
  Just m  -> relabel [ (n, (2*n) + 1) | n <- [0..m] ] b
```

And with the unprimed ones in the double:

```
mvBdd :: Bdd -> RelBDD
mvBdd b = pure $ case maxVarOf b of
  Nothing -> b
  Just m  -> relabel [ (n, 2*n) | n <- [0..m] ] 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) = case maxVarOf b of
  Nothing -> b
  Just m  -> relabel [ (2 * n, n) | n <- [0..m] ] 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] -> Map State [State] -> RelBDD
propRel2bdd props rel = pure $ disSet (elems $ Data.Map.Strict.mapWithKey linkbdd rel)
    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 ::  Map State [State]
samplerel = fromList [
  ( []        , [ [],[P 1],[P 2],[P 1, P 2] ] ),
  ( [P 1]     , [    [P 1],      [P 1, P 2] ] ),
  ( [P 2]     , [    [P 2],      [P 1, P 2] ] ),
  ( [P 1, P 2], [                [P 1, P 2] ] )  ]
```

```
>>> SMCDEL.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)

0.13 seconds
```

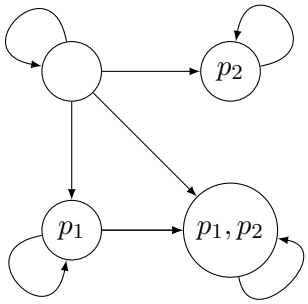


Figure 2: The original graph of `samplerel`.

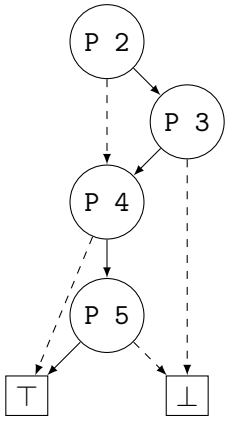


Figure 3: BDD of `samplerel` with double vocabulary labels.

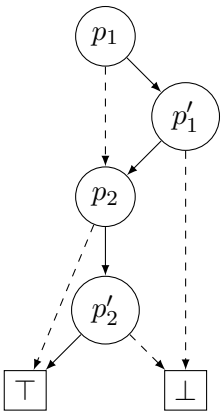


Figure 4: BDD of `samplerel` with translated labels.

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

- The total relation is given by the constant $\top$ and the empty relation by $\bot$.
- Get the inverse: Simultaneously substitute primed for unprimed variables and vice versa.
- Test for symmetry: Is it equal to its inverse?
- Symmetric closure: Take the disjunction with the inverse.
- Test for reflexivity: Does $\bigwedge_i (p_i \leftrightarrow p'_i)$ imply the BDD of R? I.e. is the BDD of that implication equal to $\top$?
- Reflexive closure: Take the disjunction of $\mathsf{Bdd}(R)$ with $\bigwedge_i (p_i \leftrightarrow p'_i)$.
- Test for transitivity: Check whether $\mathsf{Bdd}(R) \wedge \mathsf{Bdd}(R)' \rightarrow [V'/V'']\,\mathsf{Bdd}(R)$ is a tautology. Note that this is a formula over $V \cup V' \cup V''$.

7.3 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 (elems $ Data.Map.Strict.map linkbdd m) where
    linkbdd (mapPropBool,mapAgentReach)  =
      con
        (booloutof (mv here) (mv props))
        (disSet [ booloutof (cp there) (cp props) | there<-theres ] )
      where
        props = Data.Map.Strict.keys mapPropBool
        here = Data.Map.Strict.keys (Data.Map.Strict.filter id mapPropBool)
        theres = map (truthsInAt (KrM m)) (mapAgentReach ! i)
```

```
>>> map (flip SMCDEL.Symbolic.K.relBddOfIn SMCDEL.Explicit.K.exampleModel)
[alice,bob]

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

0.12 seconds
```

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

7.4 Belief Structures

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

type BelScene = (BelStruct,State)

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

instance HasVocab BelScene where
  vocabOf = vocabOf . fst

instance HasAgents BelStruct where
  agentsOf (BlS _ _ obdds) = Data.Map.Strict.keys obdds

instance HasAgents BelScene where
  agentsOf = agentsOf . fst
```

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 kns (Neg form)    = neg $ bddOf kns form
bddOf kns (Conj forms)  = conSet $ map (bddOf kns) forms
bddOf kns (Disj forms)  = disSet $ map (bddOf kns) forms
bddOf kns (Xor  forms)  = xorSet $ map (bddOf kns) forms
bddOf kns (Impl f g)    = imp (bddOf kns f) (bddOf kns g)
bddOf kns (Equi f g)    = equ (bddOf kns f) (bddOf kns g)
bddOf kns (Forall ps f) = forallSet (map fromEnum ps) (bddOf kns f)
bddOf kns (Exists ps f) = existsSet (map fromEnum ps) (bddOf kns f)
```

Note the following notations for boolean assignments and formulas.

- Suppose s is a boolean assignment and φ is a boolean formula in the vocabulary of s. Then we write $s \vDash \varphi$ to say that s makes φ true.
- If s is an assignment for a given vocabulary, we write s' for the same assignment for a primed copy of the vocabulary. For example take $\{p_1, p_3\}$ as an assignment over $V = \{p_1, p_2, p_3, p_4\}$, hence $\{p_1, p_3\}' = \{p_1', p_3'\}$ is an assignment over $\{p_1', p_2', p_3', p_4'\}$.
- If φ is a boolean formula, write $(\varphi)'$ for the result of priming all propositions in φ. For example, $(p_1 \to (p_3 \wedge \neg p_2))' = (p_1' \to (p_3' \wedge \neg p_2'))$.
- 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:

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

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

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

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

```
bddOf kns@(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 kns
       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 kns@(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 kns form) z)))

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

Public announcements only restrict the lawbdd:

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

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

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

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

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

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

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

```
evalViaBdd :: BelScene -> Form -> Bool
evalViaBdd (kns@(BlS allprops _ _),s) f = let
    bdd  = bddOf kns 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"
        ++ "  kns:  " ++ show kns ++ "\n"
        ++ "  s:    " ++ show s ++ "\n"
        ++ "  form: " ++ show f ++ "\n"
        ++ "  bdd:  " ++ show bdd ++ "\n"
        ++ "  list: " ++ show list ++ "\n"
        ++ "  b:    " ++ show b ++ "\n"
```

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

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

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

announce :: BelStruct -> [Agent] -> Form -> BelStruct
announce kns@(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 kns psi))
  newobdds  = Data.Map.Strict.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))) (toList obdds)
        bddstring (i,os) = "\\Omega_{\\text{" ++ i ++ "}} = " ++ bddprefix ++ os ++
            bddsuffix

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

7.5 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 newobs, s) where
  newobs = Data.Map.Strict.map optimize obs
  optimize 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)
    $ Data.Map.Strict.toList obs
```

7.6 Symbolic Bisimulations

Definition 23. *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):

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

7.7 Belief Transformers

To conclude this section, we discuss the symbolic version of non-S5 action models, namely *belief transformers*. They are like knowledge transformers, but instead of observed atomic propositions $O_i^+ \subseteq V^+$ we use BDDs Ω_i^+ encoding a relation on $\mathcal{P}(V^+)$. Thus we obtain a symbolic representation of events where observability need not be an equivalence relation, for example if someone is being deceived.

Definition 24. *A* belief transformer *for a given vocabulary V and set of agents $I = \{1, \ldots, n\}$ is a tuple $\mathcal{X} = (V^+, \theta^+, \Omega_1, \ldots, \Omega_n)$ where V^+ is a set of atomic propositions such that $V \cap V^+ = \varnothing$, θ^+ is a possibly epistemic formula from $\mathcal{L}(V \cup V^+)$ Ω_i^+ are boolean formulas over $V^+ \cup V^{+\prime}$. A* belief event *is a belief transformer together with a subset $x \subseteq V^+$, written as $(\mathcal{X}, x)$.*

The belief transformation *of a belief structure $\mathcal{F} = (V, \theta, O_1, \ldots, O_n)$ with a belief transformer $\mathcal{X} = (V^+, \theta^+, \Omega_1^+, \ldots, \Omega_n^+)$ for V is defined by:*

$$\mathcal{F} \times \mathcal{X} := (V \cup V^+, \theta \wedge ||\theta^+||_{\mathcal{F}}, \Omega_1 \wedge \Omega_1^+, \ldots, \Omega_n \wedge \Omega_n^+)$$

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

Note that the resulting Ωs are boolean formulas over $(V \cup V') \cup (V^+ \cup V^{+\prime}) = (V \cup V^+) \cup (V \cup V^+)'$ and describe relations on $\mathcal{P}(V \cup V^+)$.

```
data BelTransf = BlT [Prp] Form (Map Agent RelBDD) deriving (Eq,Show)
type GenEvent = (BelTransf,State)

belTransform :: BelScene -> GenEvent -> BelScene
belTransform (kns@(BlS props lawbdd obdds),s) (BlT addprops addlaw eventObs, eventFacts) =
  (BlS (props ++ map snd shiftrel) newlawbdd newobs, sort $ s ++ shiftedEventFacts) where
    shiftrel = zip addprops [(freshp props)..]
    doubleShiftrel = map (mvP *** mvP) shiftrel ++ map (cpP *** cpP) shiftrel
    doubleShiftrelVars = sort $ map (fromEnum *** fromEnum) doubleShiftrel
    shiftEventObsBDD o = relabel doubleShiftrelVars <$> o
    newobs = fromList [ (i , con <$> (obdds ! i) <*> shiftEventObsBDD (eventObs ! i)) | i
        <- Data.Map.Strict.keys obdds ]
    shiftaddlaw = replPsInF shiftrel addlaw
    newlawbdd = con lawbdd (bddOf kns shiftaddlaw)
    shiftedEventFacts = map (apply shiftrel) eventFacts

instance TexAble BelTransf where
  tex (BlT addprops addlaw eventObs) = concat
    [ " \\left( \n"
    , tex addprops, ", "
    , tex addlaw , ", "
    , intercalate ", " eobddstrings
    , " \\right) \n"
    ] where
        eobddstrings = map (bddstring . (fst &&& (texRelBDD . snd))) (toList eventObs)
        bddstring (i,os) = "\\Omega^+_{\\text{" ++ i ++ "}} = " ++ bddprefix ++ os ++
            bddsuffix

instance TexAble GenEvent where
  tex (blt, eventFacts) = concat
    [ " \\left( \n", tex blt, ", ", tex eventFacts, " \\right) \n" ]
```

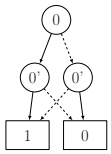
Here is a full example of belief transformation:

```
exampleStart :: BelScene
exampleStart = (BlS [P 0] law obs, actual) where
  law    = boolBddOf Top
  obs    = fromList [ ("1", mvBdd $ boolBddOf Top), ("2", allsamebdd [P 0]) ]
```

```
  actual = [P 0]

exampleEvent :: GenEvent
exampleEvent = (BlT [P 1] addlaw eventObs, [P 1]) where
  addlaw = PrpF (P 1) `Impl` PrpF (P 0)
  eventObs = fromList [ ("1", allsamebdd [P 1]), ("2", cpBdd . boolBddOf $ Neg (PrpF $ P 1)
      ) ]

exampleBlTresult :: BelScene
exampleBlTresult = belTransform exampleStart exampleEvent
```

The structure

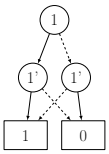
$$\left(\left(\{p\},\ \boxed{1}\ ,\Omega_1 =\ \boxed{1}\ ,\Omega_2 =\ \right.\right.$$



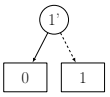
$$\left.\left.\right),\{p\}\right)$$

transformed with the event

$$\left(\left(\{p_1\},(p_1 \to p),\Omega_1^+ =\ \right.\right.$$



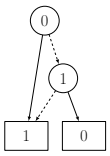
$$,\Omega_2^+ =\ $$



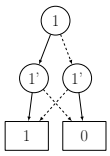
$$\left.\left.\right),\{p_1\}\right)$$

yields this new structure:

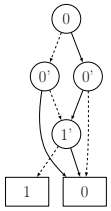
$$\left(\left(\{p,p_1\},\ \right.\right.$$



$$,\Omega_1 =\ $$



$$,\Omega_2 =\ $$



$$\left.\left.\right),\{p,p_1\}\right)$$

8 Action Models with Factual Change

Our model checker so far only considered *epistemic* change. In this module we try to also cover *factual* changes. On standard Kripke models this is represented with postconditions.

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

module SMCDEL.Explicit.K.Change where

import Control.Arrow ((&&&))
import Control.Monad
import Data.List (delete,intercalate)
import qualified Data.Map.Strict as M
import Data.Map.Strict ((!),fromList)
import Test.QuickCheck

import SMCDEL.Explicit.K
import SMCDEL.Explicit.S5 (World)
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
```

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 really captures partial functions.

```
type PostCondition = M.Map Prp Form

type Action = Int

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

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

newtype ChangeModel = ChM (M.Map Action Change)
  deriving (Eq,Ord,Show)
type PointedChangeModel = (ChangeModel, Action)

type MultipointedChangeModel = (ChangeModel, [Action])

instance HasAgents ChangeModel where
  agentsOf (ChM cm) = M.keys $ rel (head (M.elems cm))

instance HasAgents PointedChangeModel where
  agentsOf = agentsOf . fst

instance HasPrecondition PointedChangeModel where
  preOf (ChM cm, actual) = pre (cm ! actual)

productChangeWithMap :: KripkeModel -> ChangeModel -> (KripkeModel, M.Map (World, Action)
    World)
productChangeWithMap (KrM m) (ChM cm)
  | agentsOf (KrM m) /= agentsOf (ChM cm) = error "productChange failed: agentsOf differs!"
  | otherwise = (KrM $ fromList (map annotate statePairs), fromList statePairs) where
    statePairs = zip (concat [ [ (s, a) | eval (KrM m, s) (pre $ cm ! a) ] | s <- M.keys m,
         a <- M.keys cm ]) [0..]
    annotate ((s,a),news) = (news , (newval, fromList (map reachFor (agentsOf (KrM m)))))
        where
      newval = M.mapWithKey applyPC (fst $ m ! s)
      applyPC p oldbit
        | p `elem` M.keys (post (cm ! a)) = eval (KrM m,s) (post (cm ! a) ! p)
        | otherwise = oldbit
```

```
      reachFor i = (i, [ news' | ((s',a'),news') <- statePairs, s' `elem` snd (m !  s) ! i,
           a' `elem` rel (cm ! a) ! i ])

productChange :: PointedModel -> PointedChangeModel -> PointedModel
productChange (m, cur) (ChM cm, act)
  | not $ eval (m, cur) (pre $ cm ! act) = error "productChange failed: false actual
      precondition!"
  | otherwise = (newm, newcur) where
      (newm, stateMap) = productChangeWithMap m (ChM cm)
      newcur = stateMap ! (cur,act)

productChangePointless :: KripkeModel -> ChangeModel -> KripkeModel
productChangePointless (KrM m) (ChM cm) = fst $ productChangeWithMap (KrM m) (ChM cm)

productChangeMulti :: PointedModel -> MultipointedChangeModel -> PointedModel
productChangeMulti pm (ChM cm, as) = productChange pm (ChM cm, a) where
  [a] = filter (\x -> eval pm (pre $ cm ! x)) as
```

The last pattern match succeeds iff exactly one precondition holds.

```
publicMakeFalseChM :: [Agent] -> Prp -> PointedChangeModel
publicMakeFalseChM ags p = (ChM $ fromList [ (1::Action, Ch myPre myPost myRel) ], 0) where
  myPre = Top
  myPost = fromList [(p,Bot)]
  myRel = fromList [(i,[1]) | i <- ags]
```

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 ChangeModel where
  arbitrary = do
    let allactions = [0..3]
    [BF f, BF g, BF h, BF l] <- replicateM 4 (sized $ randomboolformWith [P 0 .. P 4])
    let myPres  = map simplify [f,g,h,l]
    myPosts <- mapM (\_ -> do
      proptochange <- elements [P 0 .. P 4]
      postconcon <- elements $ [Top,Bot] ++ map PrpF [P 0 .. P 4]
      return $ fromList [ (proptochange, postconcon) ]
      ) allactions
    myRels <- mapM (\k -> do
      reachList <- sublistOf allactions
      return $ fromList $ ("0",[k]) : [(show i,reachList) | i <- [1..5::Int]]
      ) allactions
    return $ ChM $ fromList
      [ (k::Action, Ch (myPres !! k) (myPosts !! k) (myRels !! k)) | k <- allactions ]
  shrink (ChM cm) = [ ChM $ removeFromRels k $ M.delete k cm | k <- M.keys cm, k /= 0 ]
      where
    removeFromRels = M.map . removeFrom where
      removeFrom k c = c { rel = M.map (delete k) (rel c) }
```

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

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

```
  texTo = texTo.ViaDot
  texDocumentTo = texDocumentTo.ViaDot

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

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

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

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

9 Transformers

Our model checker so far only considered *epistemic* change. In this module we try to also cover *factual* changes. On standard Kripke models this is represented with postconditions.

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

module SMCDEL.Symbolic.K.Change where

import Control.Arrow ((&&&))
import Control.Lens (over,both)
import Data.HasCacBDD hiding (Top,Bot)
import Data.List ((\\),intersect,intercalate,sort)
import qualified Data.Map.Strict as M
import Data.Map.Strict ((!),fromList,toList)

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

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

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

type Event = (Transformer,State)

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

type MultiEvent = (Transformer,[State])

instance TexAble Transformer where
  tex (Trf addprops addlaw changeprops changelaw eventObs) = concat
    [ " \\left( \n"
    , tex addprops, ", "
    , tex addlaw, ", "
    , tex changeprops, ", "
    , intercalate ", " $ map snd . toList $ M.mapWithKey texChange changelaw, ", "
    , intercalate ", " eobddstrings
    , " \\right) \n"
    ] where
        texChange prop changebdd = tex prop ++ " := " ++ tex (formOf changebdd)
        eobddstrings = map (bddstring . (fst &&& (texRelBDD . snd))) (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" ]
```

```
transform :: BelScene -> Event -> BelScene
transform (kns@(BlS props law obdds),s) (Trf addprops addlaw changeprops changelaw eventObs
    , eventFacts) =
  (BlS newprops newlaw newobs, news) where
    -- PART 1: SHIFTING addprops to ensure props and newprops are disjoint
    shiftaddprops = [(freshp props)..]
    shiftrel = sort $ zip addprops shiftaddprops
    relabelWith r = relabel (sort $ map (over both fromEnum) r)
```

```
    -- apply the shifting to addlaw and changelaw:
    addlawShifted = replPsInF shiftrel addlaw
    changelawShifted = M.map (relabelWith shiftrel) changelaw
    -- to apply the shifting to eventObs we need shiftrel for the double vocabulary:
    shiftrelMVCP = sort $ zip (mv addprops) (mv shiftaddprops)
                        ++ zip (cp addprops) (cp shiftaddprops)
    eventObsShifted = M.map (fmap $ relabelWith shiftrelMVCP) eventObs
    -- the actual event:
    x = map (apply shiftrel) eventFacts
    -- PART 2: COPYING the modified propositions
    copychangeprops = [(freshp $ props ++ map snd shiftrel)..]
    copyrel = zip changeprops copychangeprops
    copyrelMVCP = sort $ zip (mv changeprops) (mv copychangeprops)
    -- PART 3: actual transformation
    newprops = sort $ props ++ map snd shiftrel ++ map snd copyrel
    newlaw = conSet $ relabelWith copyrel (con law (bddOf kns addlawShifted))
                    : [var (fromEnum q) `equ` relabelWith copyrel (changelawShifted ! q) |
                       q <- changeprops]
    newobs = M.mapWithKey (\i oldobs -> con <$> (relabelWith copyrelMVCP <$> oldobs) <*> (
        eventObsShifted ! i)) obdds
    news | bddEval (s ++ x) (con law (bddOf kns addlawShifted)) = sort $ concat
           [ s \\ changeprops
           , map (apply copyrel) $ s `intersect` changeprops
           , x
           , filter (\ p -> bddEval (s ++ x) (changelawShifted ! p)) changeprops ]
         | otherwise = error "Transformer is not applicable!"
```

Analogous to `productChangeMulti` above we define `transformMulti` which takes a multi event, the symbolic equivalent of a multi-pointed action model. Different events are represented as different States. Which of them actually happens is then decided by evaluating the event law: x is possible to happen at $\mathcal{F}, s$ iff $\mathcal{F}, s \vDash [x/\top][(V^+ \setminus x)/\bot]\theta^+$. We presuppose that the event law makes the events mutually exclusive, just like preconditions in multi-pointed action models.

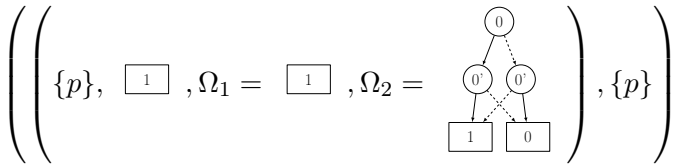
```
transformMulti :: BelScene -> MultiEvent -> BelScene
transformMulti (kns,s) (trf@(Trf addprops addlaw _ _ _), eventsFacts) =
  transform (kns,s) (trf,selectedEventFacts) where
    possible :: State -> Bool
    possible eventFact = evalViaBdd (kns,s) (substitSet subs addlaw) where
      subs = [ (p, if p `elem` eventFact then Top else Bot) |  p <- addprops ]
    selectedEventFacts :: State
    [selectedEventFacts] = filter possible eventsFacts
```

9.1 Simple Example

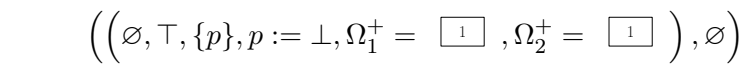
```
publicMakeFalse :: [Agent] -> Prp -> Event
publicMakeFalse agents p = (Trf [] Top [p] changelaw eventobs, []) where
  changelaw = fromList [ (p,boolBddOf Bot) ]
  eventobs  = fromList [ (i,totalRelBdd) | i <- agents ]

myEvent :: Event
myEvent = publicMakeFalse (agentsOf $ fst SMCDEL.Symbolic.K.exampleStart) (P 0)

tResult :: BelScene
tResult = SMCDEL.Symbolic.K.exampleStart `transform` myEvent

flipOverAndShowTo :: [Agent] -> Prp -> Agent -> Event
flipOverAndShowTo everyone p i = (Trf [q] eventlaw [p] changelaw eventobs, [q]) where
  q         = freshp [p]
  eventlaw  = PrpF q `Equi` PrpF p
  changelaw = fromList [ (p, boolBddOf . Neg . PrpF $ p) ]
  eventobs  = fromList $ (i, allsamebdd [q])
                       : [ (j,totalRelBdd) | j <- everyone \\ [i] ]

myOtherEvent :: Event
myOtherEvent = flipOverAndShowTo ["1","2"] (P 0) "1"

tResult2 :: BelScene
```

```
tResult2 = SMCDEL.Symbolic.K.exampleStart `transform` myOtherEvent
```

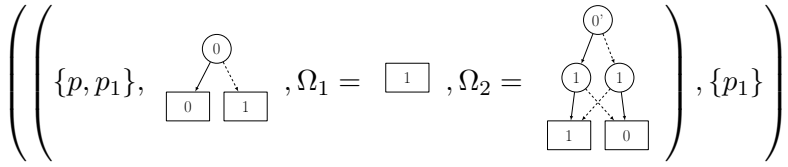
The structure ...



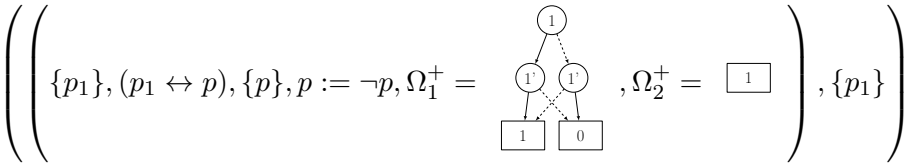
... transformed with `myEvent` ...



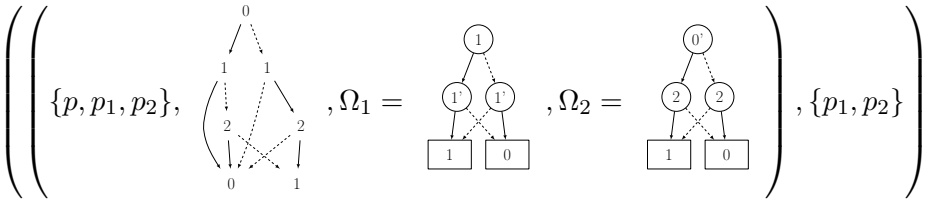
... yields this new structure:



If we instead transform it with `myOtherEvent` ...



... then we get:



9.2 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
```

We also implement the following boolean translation for formulas prefixed with a dynamic operator containint 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 _ changeprops changelaw _, eventFacts) f =
  let
    -- same as in 'transform', to ensure props and addprops are disjoint
    shiftaddprops = [(freshp $ vocabOf scn)..]
    shiftrel      = sort $ zip addprops shiftaddprops
    relabelWith r = relabel (sort $ map (over both fromEnum) r)
    -- apply the shifting to addlaw and changelaw:
    changelawShifted = M.map (relabelWith shiftrel) changelaw
    (newBlS,_) = transform 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+
          bddSubstitSimul postconrel $ -- 2. replace changeprops with postconditions
            bddOf newBlS f             -- 1. boolean equivalent wrt new structure

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

The following two functions do the necessary substitutions on BDDs. We define them here because similar functions are currently not available in HasCacBDD.

```
-- replace variable n with a BDD psi in BDD b
bddSubstit :: Int -> Bdd -> Bdd -> Bdd
bddSubstit n psi b =
  case firstVarOf b of
    Nothing -> b
    Just k  -> case compare n k of
                 LT -> b
                 EQ -> ifthenelse psi (thenOf b) (elseOf b)
                 GT -> ifthenelse (var k) (bddSubstit n psi (thenOf b)) (bddSubstit n psi
                     (elseOf b))

-- *simultaneous* substitution of BDDs for variables
-- (not the same as folding bddSubstit)
bddSubstitSimul :: [(Int,Bdd)] -> Bdd -> Bdd
bddSubstitSimul []    b = b
bddSubstitSimul repls b =
  case firstVarOf b of
    Nothing -> b
    Just k  -> case lookup k repls of
      Nothing  -> ifthenelse (var k) (bddSubstitSimul repls $ thenOf b) (bddSubstitSimul
          repls $ elseOf b)
      Just psi -> ifthenelse psi     (bddSubstitSimul repls $ thenOf b) (bddSubstitSimul
          repls $ elseOf b)
```

10 Connecting General Kripke Models and Belief Structures

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

module SMCDEL.Translations.K where

import Data.HasCacBDD hiding (Top,Bot)
import Data.Map.Strict (Map,fromList,(!))

import SMCDEL.Language
import SMCDEL.Symbolic.S5 (State)
import SMCDEL.Explicit.S5 (worldsOf)

import SMCDEL.Symbolic.K
import SMCDEL.Explicit.K
import SMCDEL.Translations.S5 (booloutof)
import SMCDEL.Internal.Help (apply)
```

10.1 From Belief Structures to Kripke Models

```
blsToKripke :: BelScene -> PointedModel
blsToKripke (f@(BlS _ _ obdds), curs) = (m, cur) where
  links = zip (statesOf f) [0..]
  m = KrM $ fromList
    [ (w, ( fromList [(p, p `elem` s) | p <- vocabOf f]
          , 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
```

10.2 From Kripke Models to Belief Structures

Assuming we already have distinct valuations!

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

exampleBelScn :: BelScene
exampleBelScn = kripkeToBls examplePointedModel

exampleBelStruct :: BelStruct
exampleBelState :: State
(exampleBelStruct,exampleBelState) = exampleBelScn
```

$$\left(\left(\{p,p_1\}, \quad \begin{array}{c} 1 \\ 1 \quad 0 \end{array} \quad , \Omega_{\text{Alice}} = \quad \begin{array}{c} 0' \\ 1 \quad 0 \end{array} \quad , \Omega_{\text{Bob}} = \quad \begin{array}{c} 0 \\ 0' \quad 0' \\ 1 \quad 0 \end{array}\right), \{p,p_1\}\right)$$

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

Additionally, both agent BDDs do not care about p_1 or p_1'. This is because of our use of `restrictLaw`. This ensures our relation bdds do not become unnecessarily large. The BDDs generated by relBddOfIn

include a check that both parts of the related pair are actually state of our model/structure but we do not need this information in the agents bdd.

11 Connecting Actions Models and Transformers

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

module SMCDEL.Translations.K.Change where

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

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

11.1 From Action Models with Change to Transformers

This generalizes `actionToEvent` from Section 5 and Section 10. Note that we don't need extra propositions for the action relation any longer.

```
actionToEvent :: PointedChangeModel -> Event
actionToEvent (ChM chm, faction) = (Trf addprops addlaw changeprops changelaw eventObs,
    efacts) where
  actions      = M.keys chm
  (P fstnewp)  = freshp $ concatMap -- avoid props in pre and postconditions
    (\c -> propsInForms (pre c : M.elems (post c)) ++ M.keys (post c)) (M.elems chm)
  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 $ chm ! a ] |
       a <- actions ]
  changeprops  = sort $ nub $ concatMap M.keys . M.elems $ M.map post chm -- propositions
      to be changed
  changelaw    = M.fromList [ (p, changeFor p) | p <- changeprops ] -- encode
      postconditions
  changeFor p  = disSet [ booloutof (ell k) addprops `con` boolBddOf (safepost (chm ! k) p)
       | k <- actions ]
  eventObs     = M.fromList [ (i, obsLawFor i) | i <- agentsOf (ChM chm) ]
  obsLawFor i  = pure $ disSet (M.elems $ M.mapWithKey (link i) chm)
  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
```

```
eventToAction' :: Event -> PointedChangeModel
eventToAction' (t@(Trf addprops addlaw changeprops changelaw eventObs), efacts) = (ChM chm,
     faction) where
  actlist    = zip (powerset addprops) [0..]
  chm        = fromList [ (a, Ch (preFor ps) (postFor ps) (relFor ps)) | (ps,a) <- actlist
      ]
  preFor ps  = simplify $
    substitSet (zip ps (repeat Top) ++ zip (addprops \\ ps) (repeat Bot)) addlaw
  postFor ps = fromList
    [ (q, formOf $ restrictSet (changelaw ! q) [(p, P p `elem` ps) | (P p) <- addprops]) |
        q <- changeprops ]
  relFor ps = fromList [(i,rFor i) | i <- agentsOf t] where
    rFor i = concatMap (\(qs,b) -> [ b | tagBddEval (mv ps ++ cp qs) (eventObs ! i) ])
        actlist
  faction    = apply actlist efacts
```

12 Automated Testing

12.1 Translation tests

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 where

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

main :: IO ()
main = hspec $
  describe "SMCDEL.Translations" $ do
    prop "semantic equivalence"     semanticEquivTest
    prop "semantic validity"        semanticValidTest
    prop "lemma equivalence Kripke" lemmaEquivTestKr
    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)
```

12.1.1 Semantic Equivalence

The following creates a Kripke model and a knowledge structure which are equivalent to each other by Lemma 14. 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
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 17: 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 14.

```
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
```

12.1.2 Public and Group Announcements

We can do public announcements in various ways as described in Section 14.1.3. The following tests check that the results of all methods are the same.

```
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 (knowTransform (kripkeToKns mymodel) (actionToEvent (pubAnnounceAction (
      agentsOf mymodel) f))) g
  ] where
      f = PrpF 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 = productUpdate mymodel action
    in not precon || Exp.eval newModel 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  = knowTransform (kripkeToKns mymodel) equiTrf
    in not precon || Sym.eval newKns g
  ]
```

12.1.3 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 :: ActionModel -> Form -> [Bool]
singleActionTest myact f = [a,b,c,d,e] where
```

```
  a = Exp.eval (productUpdate mymodel (myact,0)) f
  b = Sym.evalViaBdd (knowTransform (kripkeToKns mymodel) (actionToEvent (myact,0))) f
  c = Exp.eval (productUpdate mymodel (eventToAction (actionToEvent (myact,0)))) f
  d = case reduce (actionToEvent (myact,0)) f of
    Nothing -> c
    Just g  -> Sym.evalViaBdd (kripkeToKns mymodel) g
  e = Sym.evalViaBddReduce (kripkeToKns mymodel) (actionToEvent (myact,0)) f
```

12.2 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 14.1.

```
module Main where

import Data.List
import Test.Hspec
import Test.Hspec.QuickCheck
import SMCDEL.Examples
import SMCDEL.Examples.DiningCrypto
import SMCDEL.Examples.DrinkLogic
import SMCDEL.Examples.MuddyChildren
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.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} \\} ) " ]
  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 . boolBddOf == boolBddOf" $
      \(BF bf) -> (boolBddOf . formOf . boolBddOf) bf == boolBddOf bf
    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" $
    prop "generatedSubmodel preserves truth" $
```

```
    \m f -> Exp.eval (m, head $ Exp.worldsOf m) f == Exp.eval (Exp.generatedSubmodel (m,
        head $ Exp.worldsOf m)) f
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 "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
  it "Three Muddy Children" $
    evalViaBdd mudScn0 (nobodyknows 3) &&
    evalViaBdd mudScn1 (nobodyknows 3) &&
    evalViaBdd mudScn2 (Conj [knows i | i <- [1..3]]) &&
    length (SMCDEL.Symbolic.S5.statesOf mudKns2) == 1
  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 "Russian Cards: all checks"
    SMCDEL.Examples.RussianCards.rcAllChecks
  it "Russian Cards: 102 solutions" $
    length (filter checkSet allHandLists) == 102
  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 "What Sum: There are 330 solutions." $
    length SMCDEL.Examples.WhatSum.wsSolutions == 330
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 ExpK.myMudGenKrpInit $ Conj [Ck ags Top, Neg (Ck ags Bot)]
  it "3MC genKrp: It is not common knowledge that someone is muddy" $
    ExpK.eval ExpK.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 ExpK.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])
```

12.3 Non-S5 Testing

```
module Main where

import Test.Hspec
import Test.Hspec.QuickCheck
import SMCDEL.Internal.Help (alleq)
import SMCDEL.Language
import SMCDEL.Symbolic.S5 (boolBddOf)
import SMCDEL.Explicit.K as ExpK
import SMCDEL.Symbolic.K as SymK
import SMCDEL.Translations.K as TransK
import SMCDEL.Explicit.K.Change
import SMCDEL.Symbolic.K.Change
import SMCDEL.Translations.K.Change
import Data.Map.Strict (fromList)
import Data.List (sort)

main :: IO ()
main = hspec $ do
  describe "SMCDEL.Symbolic.K" $
    prop "semantic equivalence" $ alleq . semanticEquivTest
  describe "SMCDEL.Other.Change" $
    prop "single change" $ \ a f -> alleq $ singleChangeTest a 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)

semanticEquivTest :: SimplifiedForm -> [Bool]
semanticEquivTest (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 :: ChangeModel -> SimplifiedForm -> [Bool]
singleChangeTest myact (SF f) =
  [ not (ExpK.eval                                  myMod  (preOf
                                  (myact,0::Action)))
      || ExpK.eval       (productChange                 myMod
                                            (myact,0)  ) f
  , not (SymK.evalViaBdd                (kripkeToBls myMod) (preOf                (
      actionToEvent (myact,0))))
      || SymK.evalViaBdd (transform     (kripkeToBls myMod)                       (
          actionToEvent (myact,0)))  f
  , not (ExpK.eval                                  myMod  (preOf (eventToAction' (
      actionToEvent (myact,0)))))
      || ExpK.eval       (productChange             myMod         (eventToAction' (
          actionToEvent (myact,0)))) f
  , not (ExpK.eval                      (blsToKripke myScn) (preOf (eventToAction' (
      actionToEvent (myact,0)))))
      || ExpK.eval       (productChange (blsToKripke myScn)        (eventToAction' (
          actionToEvent (myact,0)))) f
  , not (SymK.evalViaBdd                            myScn  (preOf                (
      actionToEvent (myact,0))))
```

```
    || SymK.evalViaBdd (transform                  myScn                          (
        actionToEvent (myact,0)) ) f
]
++ case SMCDEL.Symbolic.K.Change.reduce (actionToEvent (myact,0)) f of
    Nothing -> []
    Just g  -> pure $ SymK.evalViaBdd (kripkeToBls myMod) (simplify g)
++ [ SMCDEL.Symbolic.K.Change.evalViaBddReduce myScn (actionToEvent (myact,0)) f ]
```

13 Epistemic Planning

Here we provide some wrapper functions for epistemic planning via model checking.

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

module SMCDEL.Other.Planning where

import SMCDEL.Language
```

We model an *offline plan* as a list of pairs of formulas. The first part of the first tuple should be announced truthfully and lead to a model where the second part of the tuple is true. Then we continue with the next tuple. We write σ for any plan and ϵ for the empty plan which always succeeds. The following then defines a formula which saying that the given plan succeeds:

$$\begin{aligned} \mathsf{succeeds}(\epsilon) &:= \top \\ \mathsf{succeeds}((\varphi,\psi);\sigma) &:= [\varphi](\psi \wedge \mathsf{succeeds}(\sigma)) \end{aligned}$$

```
type OfflinePlan = [(Form,Form)] -- list of (announcement,goal) tuples

class Plan a where
  succeeds :: a -> Form

instance Plan OfflinePlan where
  succeeds [] = Top
  succeeds ((step,goal):rest) =
    Conj [step, PubAnnounce step goal, PubAnnounce step (succeeds rest)]

succeedsOn :: (Semantics o, Plan a) => a -> o -> Bool
succeedsOn plan start = isTrue start (succeeds plan)
```

An *online plan* in contrast can include tests and choices, to decide which action to do depending on the results of previous announcements. The data type we use to represent this is a tree where nodes are actions

```
data OnlinePlan = Stop | DoAnnounce Form OnlinePlan | IfThenElse Form OnlinePlan OnlinePlan

instance Plan OnlinePlan where
  succeeds Stop = Top
  succeeds (DoAnnounce step next) = Conj [step, PubAnnounce step (succeeds next)]
  succeeds (IfThenElse check planA planB) =
    Conj [ check `Impl` succeeds planA, Neg check `Impl` succeeds planB ]
```

14 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.

14.1 Small Examples

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

module SMCDEL.Examples where
import Data.List ((\\),sort)
import SMCDEL.Language
import SMCDEL.Symbolic.S5
import SMCDEL.Explicit.S5
import SMCDEL.Translations.S5
```

14.1.1 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.

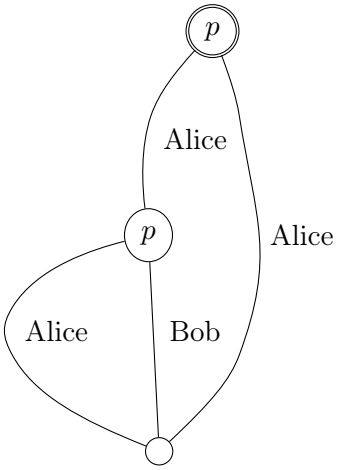
```
modelA :: PointedModelS5
modelA = (KrMS5 [0,1] [(alice,[[0,1]]),(bob,[[0],[1]])] [ (0,[(P 0,True)]), (1,[(P 0,False)
    ]) ], 0)
```


```
>>> 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
```

$$\left(\{p, p_2\}, \quad \begin{array}{c} 0 \\ 2 \quad 2 \\ 1 \quad 0 \end{array} \quad , \begin{array}{c} \varnothing \\ \{p_2\} \end{array} \right), \{p, p_2\}$$

$$\left(\{p, p_2\}, \quad \begin{array}{c} 0 \\ 2 \\ 1 \quad 0 \end{array} \quad , \begin{array}{c} \varnothing \\ \{p_2\} \end{array} \right), \{p, p_2\}$$

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

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

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

```
0.10 seconds
```

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

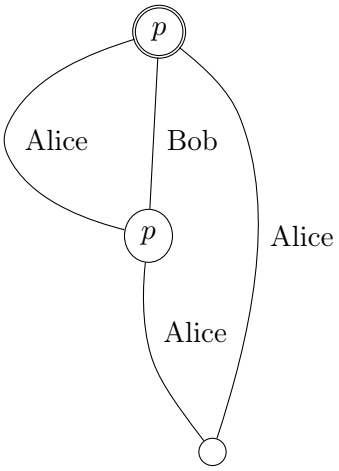
```
[[P 0],[P 0,P 2],[]]
```

```
0.12 seconds
```

14.1.2 Minimization via Translation

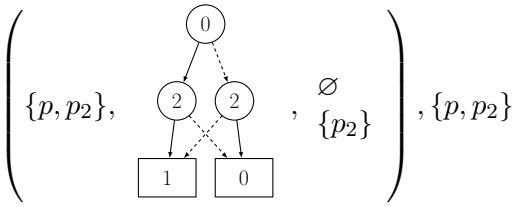
Consider the following Kripke model where **0** and **1** are bisimilar — it is redundant.

```
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)
```



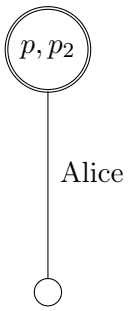
If we transform this model to a knowledge structure, we get the following:

```
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
```



14.1.3 Different Announcements

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

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

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

```
>>> examplePaAction
```

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

```
0.00 seconds
```

```
>>> actionToEvent examplePaAction
```

```
(KnT [] (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 -> PointedActionModel
groupAnnounceAction everyone listeners f = (ActM [0,1] [(0,f),(1,Neg f)] actrel, 0)
  where actrel = sort $ [ (i,[[0],[1]]) | i <- listeners ]
                     ++ [ (i,[[0 , 1]]) | i <- everyone \\ listeners ]

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

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

```
>>> exampleGroupAnnounceAction
```

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

```
0.00 seconds
```

```
>>> exampleGrAnnEvent
```

```
(KnT [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:

$$\bigwedge\{((p_1 \wedge p) \vee (\neg p_1 \wedge \neg p)), (p_2 \leftrightarrow p_1), (\neg p_2 \leftrightarrow \neg p_1)\}$$

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 19 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)
```

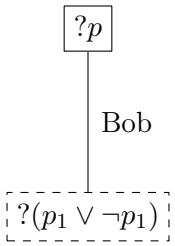
```
(ActM [0,3] [(0,PrpF (P 0)),(3,Neg (PrpF (P 0)))] [("Alice",[[3],[0]]),("Bob"
    ,[[0,3]])],0)
```

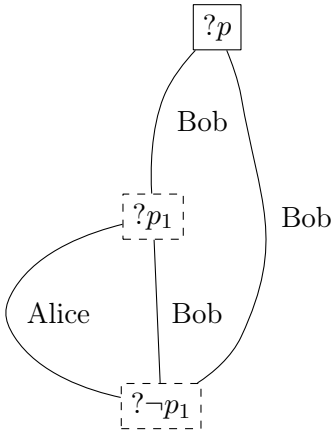
```
0.00 seconds
```

14.1.4 Equivalent Action Models

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

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

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






```
>>> actionToEvent actionOne
```

```
(KnT [P 2,P 3] (Conj [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
```

```
(KnT [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
```

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

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

14.2 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
import SMCDEL.Symbolic.K.Change
```

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 [p] 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 `transform` coinFlip
```

The structure ...

$$\left(\left(\{p\},\ \boxed{0}\!\!\begin{array}{cc}1 & 0\end{array},\Omega_{\mathrm{a}}=\ \boxed{0}\!\!\begin{array}{cc}0' & 0'\end{array}\!\!\begin{array}{cc}1 & 0\end{array},\Omega_{\mathrm{b}}=\ \boxed{0}\!\!\begin{array}{cc}0' & 0'\end{array}\!\!\begin{array}{cc}1 & 0\end{array}\right),\{p\}\right)$$

... transformed with `coinFlip` ...

$$\left(\left(\{p_1\},\top,\{p\},p:=p_1,\Omega_{\mathrm{a}}^{+}=\ \boxed{1}\ ,\Omega_{\mathrm{b}}^{+}=\ \boxed{1}\!\!\begin{array}{cc}1' & 1'\end{array}\!\!\begin{array}{cc}1 & 0\end{array}\right),\{p_1\}\right)$$

... yields this new structure:

$$\left(\left(\{p,p_1,p_2\},\ \boxed{0}\!\!\begin{array}{cc}1 & 1\end{array}\!\!\begin{array}{c}2\end{array}\!\!\begin{array}{cc}0 & 1\end{array},\Omega_{\mathrm{a}}=\ \boxed{0'}\!\!\begin{array}{cc}2 & 2\end{array}\!\!\begin{array}{cc}1 & 0\end{array},\Omega_{\mathrm{b}}=\ \boxed{0'}\!\!\begin{array}{cc}1 & 1\end{array}\!\!\begin{array}{cccc}1' & 1' & 1' & 1'\end{array}\!\!\begin{array}{cc}2 & 2\end{array}\!\!\begin{array}{cc}1 & 0\end{array}\right),\{p,p_1,p_2\}\right)$$

which has two states:

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

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

```
0.12 seconds
```

14.3 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.

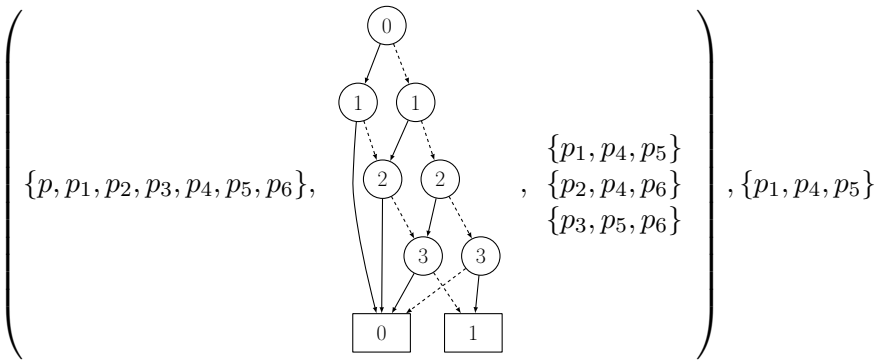
The following function generates a knowledge structure to model this story. Given an index 0, 1, 2, or 3 for who paid and three boolean values for the random coins we get the corresponding scenario.

```
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:



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])
```

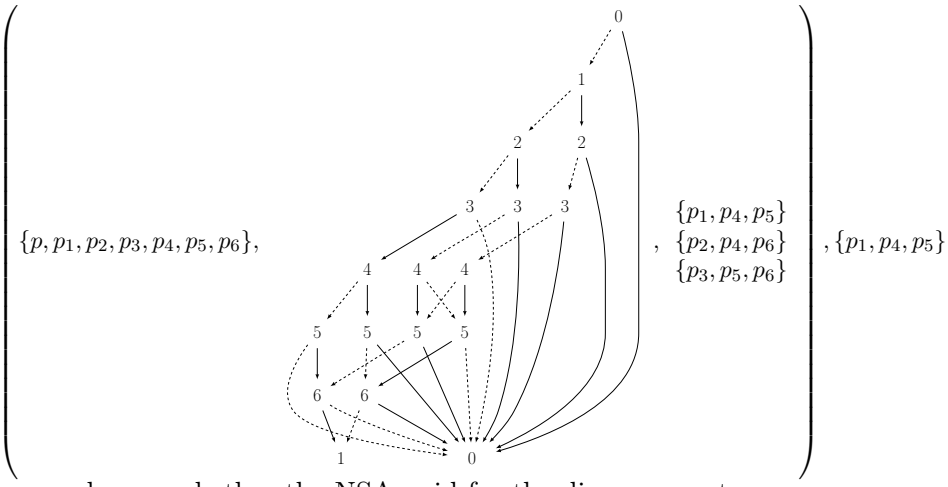
```
>>> map (evalViaBdd SMCDEL.Examples.DiningCrypto.dcScn1)
[SMCDEL.Examples.DiningCrypto.reveal 1, SMCDEL.Examples.DiningCrypto.reveal 2,
SMCDEL.Examples.DiningCrypto.reveal 3]
```

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

```
0.13 seconds
```

Now these three facts are announced:

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



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
```

```
0.12 seconds
```

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

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

```
True
```

```
0.13 seconds
```

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

```
False
```

```
0.13 seconds
```

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

```
False
```

```
0.12 seconds
```

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

```
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

0.12 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

0.13 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 15.2. 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
```

```
0.13 seconds
```

14.4 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.

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

myThirstyScene :: KnowScene
myThirstyScene = thirstyScene 3
```

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

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

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

```
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)
```

```
>>>
SMCDEL.Examples.DrinkLogic.thirstyCheck
3
```

```
True
```

```
0.12 seconds
```

```
>>>
SMCDEL.Examples.DrinkLogic.thirstyCheck
10
```

```
True
```

```
0.15 seconds
```

```
>>>
SMCDEL.Examples.DrinkLogic.thirstyCheck
100
```

```
True
```

```
0.22 seconds
```

```
>>>
SMCDEL.Examples.DrinkLogic.thirstyCheck
200
```

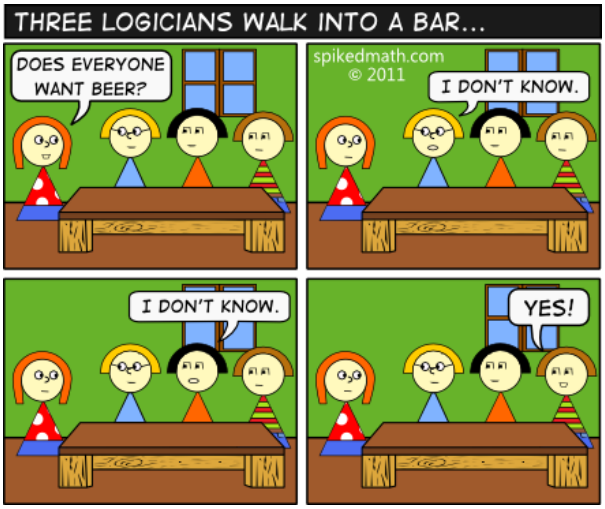
```
True
```

```
0.70 seconds
```

```
>>>
SMCDEL.Examples.DrinkLogic.thirstyCheck
400
```

```
True
```

```
2.82 seconds
```



http://spikedmath.com/445.html

14.5 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 $K_a^? p_b$.

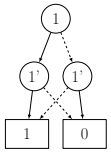
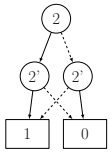
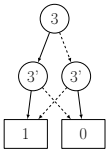
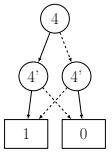
We use belief instead of knowledge structures because this makes it much easier to describe the event observation law. Otherwise we would have to add a lot more observational variables.

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 14.6 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
```

$$\left(\left(\{p_1,p_2,p_3,p_4\},\ \boxed{1}\ ,\Omega_1 = \right.\right.$$  $$,\Omega_2 =$$  $$,\Omega_3 =$$  $$,\Omega_4 =$$  $$\left.\left.\right)\right),\{p_1,p_2,p_3,p_4\}\right)$$

```
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] ]

call :: (Int,Int) -> [Int] -> GenEvent
call (a,b) secSetT = (BlT vocplus lawplus obsplus, actualSet) where
  vocplus = sort $ map inCall [1..n] ++ map inSecT [1..n]
  inCall k = P (100+k) -- k participates in the call
  inSecT k = P (200+(k*2)) -- secret k is being exchanged (as true)
  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]))

  actualSet = [inCall a, inCall b] ++ map inSecT secSetT
```

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 $ belTransform start (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

4.68 seconds
```

But this is:

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

  True

28.42 seconds
```

14.6 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 hiding (Event)
```

```
import SMCDEL.Symbolic.S5.Change
import Data.Map.Strict (fromList,keys)
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 -> KnowChange
callTrf n = CTrf eventprops eventlaw changeprops 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 = fromList [(show k, callPropsWith k) | k <- gossipers n]
  changeprops = keys changelaws
  changelaws = fromList
    [(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) = knowChange start (call (length $ agentsOf start) (a,b)) -- TODO
    optimize here!

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")])
```

14.7 Muddy Children

```
module SMCDEL.Examples.MuddyChildren where

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

import SMCDEL.Language
import SMCDEL.Symbolic.S5
import qualified SMCDEL.Symbolic.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 = pubAnnounceOnScn myMudScnInit (father 3)
```

$$\left(\{p_1,p_2,p_3\},\ \begin{array}{c}1\\2\\3\\\boxed{1}\quad\boxed{0}\end{array}\ ,\ \begin{matrix}\{p_2,p_3\}\\ \{p_1,p_3\}\\ \{p_1,p_2\}\end{matrix} \right), \{p_1,p_2,p_3\}$$

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

  True

0.13 seconds
```

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

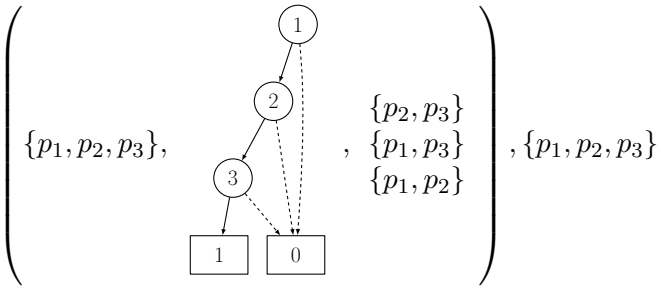
```
mudScn1 :: KnowScene
mudScn1 = pubAnnounceOnScn mudScn0 (nobodyknows 3)
```

$$\left(\{p_1,p_2,p_3\},\ \begin{array}{c}1\\2\quad2\\3\\\boxed{1}\quad\boxed{0}\end{array}\ ,\ \begin{matrix}\{p_2,p_3\}\\ \{p_1,p_3\}\\ \{p_1,p_2\}\end{matrix} \right), \{p_1,p_2,p_3\}$$

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

  True

0.14 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,_) = pubAnnounceOnScn mudScn1 (nobodyknows 3)
```




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

```
True
```

```
0.13 seconds
```

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

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

```
0.14 seconds
```

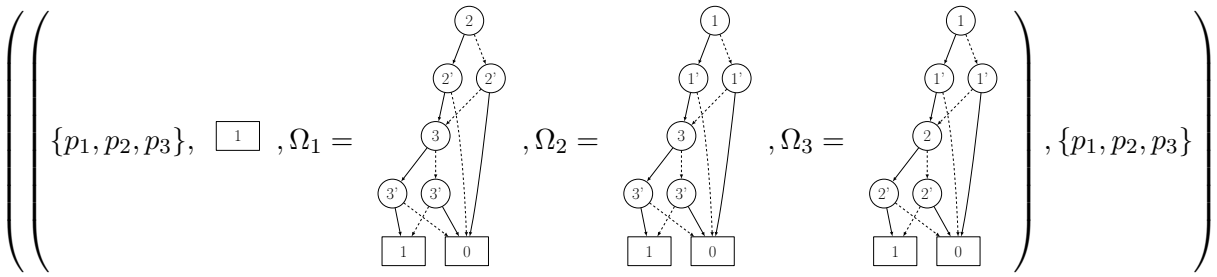
We also make use of this example in the benchmarks in Section 15.

14.8 Building Muddy Children using Knowledge Transformers

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

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

14.9 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
```



14.10 Hundred Prisoners and a Lightbulb

```
module SMCDEL.Examples.Prisoners where

import Data.HasCacBDD hiding (Top,Bot)
import Data.Map.Strict (fromList)

import SMCDEL.Explicit.K
import SMCDEL.Explicit.K.Change
import SMCDEL.Internal.TexDisplay
import SMCDEL.Language
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.K.Change
import SMCDEL.Symbolic.S5 (boolBddOf)
```

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.

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. However, we do not include a "nothing happens" event but instead model the synchronous version only.

```
-- 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 :: KripkeModel
prisonExpStart =
  KrM $ fromList [ (1, (fromList [(P 0,False),(P 1,False),(P 2,False),(P 3,False)],
      fromList [("1",[1])])) ]

prisonGoal :: Form
prisonGoal = Disj [ K i everyoneWasInTheRoom | i <- ["1"] ] where
  everyoneWasInTheRoom = Conj [ PrpF $ P k | k <- [1,2,3] ]

prisonAction :: ChangeModel
prisonAction = ChM $ fromList actions where
  p = PrpF (P 0)
  [p2,p3] = map (PrpF . P) [2,3]
  actions =
    [ (1, Ch p       (fromList [(P 0, Bot), (P 1, Top)                ]) (fromList [("1",[1
            ])]))
    , (2, Ch (Neg p) (fromList [            (P 1, Top)                ]) (fromList [("1",[2
            ])]))
    , (3, Ch Top     (fromList [(P 0, p2 `Impl` p), (P 2, p `Impl` p2)]) (fromList [("1"
        ,[3,4,5])])) -- interview 2
    , (4, Ch Top     (fromList [(P 0, p3 `Impl` p), (P 3, p `Impl` p3)]) (fromList [("1"
        ,[3,4,5])])) -- interview 3
    -- , (5, Ch Top  (fromList [ ]) (fromList [("1",[3,4,5])])) -- nothing happens, ignored
         for now
    ]

prisonInterview :: Integer -> MultipointedChangeModel
prisonInterview 1 = (prisonAction, [1,2])
prisonInterview 2 = (prisonAction, [3])
prisonInterview 3 = (prisonAction, [4])
prisonInterview _ = undefined
```

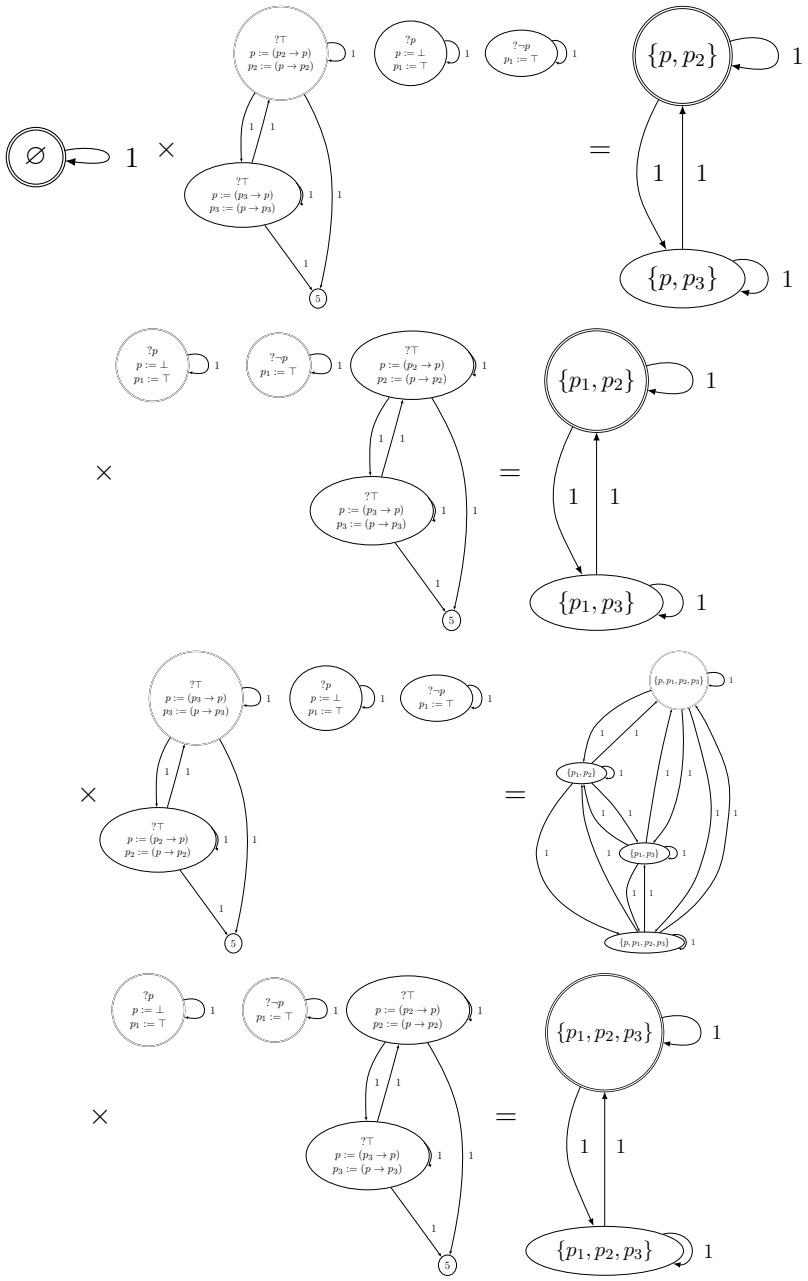
Interlude: **Story telling with Kripke models**. We define a general function to execute a sequence of multi-pointed action models on a given initial model and LaTeX the whole story.

```
newtype KripkeStory = KrpStory (PointedModel,[MultipointedChangeModel])

endOfStory :: KripkeStory -> PointedModel
endOfStory (KrpStory (start,actions)) = foldl productChangeMulti start actions

instance TexAble KripkeStory where
  tex (KrpStory (start,actions)) = adjust (tex start) ++ loop start actions where
    adjust thing = "\\raisebox{-.5\\height}{\\begin{adjustbox}{max height=4cm, max width=7
        cm}" ++ thing ++ "\\end{adjustbox}}"
    loop _       [] = ""
    loop current (a:as) =
      let
        new = generatedSubmodel $ current `productChangeMulti` 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. Note that the full product models are actually much larger — we show only generated submodels here.

```
prisonExp :: KripkeStory
```

```
prisonExp = KrpStory ((prisonExpStart,1), map prisonInterview [2,1,3,1])

prisonExpResult :: PointedModel
prisonExpResult = endOfStory prisonExp
```



And indeed we have:

```
>>> SMCDEL.Explicit.K.eval SMCDEL.Examples.Prisoners.prisonExpResult
SMCDEL.Examples.Prisoners.prisonGoal
```

```
True
```

```
0.00 seconds
```

```
prisonSymStart :: BelScene
prisonSymStart = (BlS (map P [0..3]) law obs, actual) where
  law    = boolBddOf (Conj [ Neg $ PrpF $ P k | k <- [0..3]])
  -- Light off, nobody has been in the room, this is common knowledge.
  obs    = fromList [ ("1", pure top), ("2", pure top), ("3", pure top) ]
  actual = []

prisonSymInterview :: Int -> MultiEvent
prisonSymInterview 1 = (prisonSymEvent, [undefined])
prisonSymInterview 2 = (prisonSymEvent, [undefined])
prisonSymInterview 3 = (prisonSymEvent, [undefined])
prisonSymInterview _ = undefined

prisonSymEvent :: Transformer
prisonSymEvent = Trf -- agent 1 is interviewd
  (map P [7,8,9]) -- distinguish six events using fresh variables:
  -- p7 iff the light was on
  -- p8 p9 - interview 1
  -- p8    - interview 2
  --    p9 - interview 3
  --       - forbid!
  (PrpF (P 7) `Equi` PrpF (P 0))  -- p7 hapens iff light is on aka p0
  [P 0, P 1, P 2, P 3] -- light might be turned off and visits of agents recorded
  (fromList [ (P 0, boolBddOf $ Conj [PrpF (P 7) `Impl` Bot, Neg (PrpF (P 7)) `Impl` PrpF (
      P 0)])
            , (P 1, top) ]) -- changelaw

  (fromList
    -- agent 1 observes whether (p8 && p9) and if true, observes p7
    [("1", allsamebdd [P 7])
    ,("2", pure top),("3", pure top)])

prisonSymEvent' :: Transformer
prisonSymEvent' = Trf -- agent 2 or 3 is interviewd
  [P 7] Top -- p7 iff interviewee is 2
  [P 0, P 2, P 3] -- light p0 might be turned on and visits of 2 and 3 recorded
  (fromList [ (P 0, boolBddOf $ Conj
                    [      PrpF (P 7)  `Impl` (PrpF (P 2) `Impl` PrpF (P 0))
                    , Neg (PrpF (P 7)) `Impl` (PrpF (P 3) `Impl` PrpF (P 0)) ] )
            , (P 2, boolBddOf $ Conj
                    [      PrpF (P 7)  `Impl` (PrpF (P 0) `Impl` PrpF (P 2))
                    , Neg (PrpF (P 7)) `Impl` PrpF (P 2) ] )
            , (P 3, boolBddOf $ Conj
                    [ Neg (PrpF (P 7)) `Impl` (PrpF (P 0) `Impl` PrpF (P 3))
                    ,      PrpF (P 7)  `Impl` PrpF (P 3) ] )
            ]) -- changelaw
  (fromList [("1", pure top), ("2", allsamebdd [P 7]), ("3", allsamebdd [P 7])]) -- agent 2
       and 3 observe
```

Same interlude for symbolic stories:

```
type Story = (BelScene,[MultiEvent])

prisonSym :: Story
prisonSym = (prisonSymStart, map prisonSymInterview [2,1,3,1])
```

14.11 Russian Cards

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

module SMCDEL.Examples.RussianCards where

import Control.Monad (replicateM)
import Data.List (delete,intersect,nub,sort)
import Data.Map.Strict (fromList)
import Data.HasCacBDD hiding (Top,Bot)

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))

-- use this in ppFormWith
rcExplain :: Prp -> String
rcExplain (P k) = show (rcPlayers !! i) ++ " `hasCard` " ++ show n where (n,i) = divMod k 3
```

```
>>> SMCDEL.Examples.RussianCards.hasCard carol 5

  PrpF (P 17)

0.01 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 ]
```

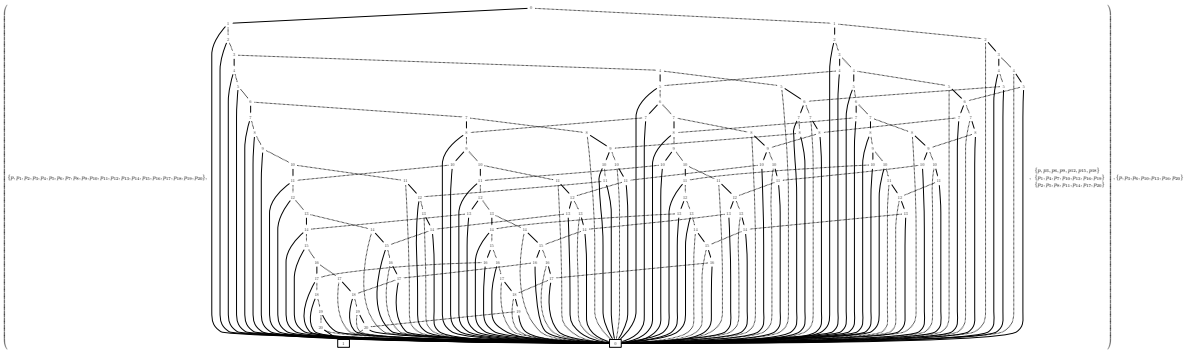
Moreover, Alice, Bob and Carol should get at least three, three and one card, respectively. As there are only seven cards in total this already implies that they can not have more.

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

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

```
rusSCN :: 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:



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 anything, independent of which card it has. In contrast, Bob will be able to rule out all but one of the hands in the list because of his own hand. Hence the second and last step of the protocol is that Bob says which card Carol has. For example, if Bob's hand is 345 he would finish the protocol with "Carol has card 6.".

To verify this protocol with our model checker we first define the two formulas for Alice saying "My hand is one of 012, 034, 056, 135 and 246." and Bob saying "Carol holds card 6". Note 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Verifying this protocol for the fixed deal 012|345|6 with our symbolic model checker takes about one second. Moreover, checking multiple protocols in a row does not take much longer because the BDD package caches results. Compared to that, the DEMO implementation from [Dit+06] needs 4 seconds to check one protocol.

```
>>> SMCDEL.Examples.RussianCards.rcAllChecks

  True

0.13 seconds
```

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

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

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

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

The last line includes two important restrictions to the set of possible lists of hands that we will consider. First, Proposition 32 in [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 :: [ [ [Int] ] ]
allHandLists = concatMap (pickHands possibleHands) [5,6,7]
```

```
>>> length SMCDEL.Examples.RussianCards.allHandLists

1290

0.03 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.

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

```
>>> length (filter SMCDEL.Examples.RussianCards.checkSet
SMCDEL.Examples.RussianCards.allHandLists)

102

0.60 seconds
```

Protocol synthesis . We now adopt a more general perspective considered in [Eng+15]. Fix that Alice has $\{0, 1, 2\}$ and that she will announce 5 hands, including this one. Hence she has to pick 4 other hands of three cards each, i.e. she has to choose among 46376 possible actions.

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

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

```
>>> 46376 == length SMCDEL.Examples.RussianCards.alicesActions
```

```
True
```

```
0.01 seconds
```

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

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

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

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

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

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

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

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

Note that we could in principle use only two propositions instead of three, and simply encode that Cath has a card by saying that the others don't have it. So we could replace c_n with $\neg a_n \wedge \neg b_n$. However, this makes it impossible to capture what Cath knows with observational variables. The more general belief structures from Section 7 could provide a solution for this in the future.

14.12 Generalized Russian Cards

Fun fact: Even if we want to use more or less than 7 cards, we do not have to modify the hasCard function.

```
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 2 = Disj [ Conj (map (i `hasCard`) [x, y]) | x <- nCards n, y <- nCards n, x
      /=y ]
  hasAtLeast i 3 = Disj [ Conj (map (i `hasCard`) [x, y, z]) | x<-nCards n, y<-nCards n, z
      <-nCards n, x/=y, x/=z, y/=z ]
  hasAtLeast i k = Disj [ Conj (map (i `hasCard`) set) | set <- sets ] where
    sets = filter alldiff $ nub $ map sort $ replicateM k (nCards n) where
      alldiff [] = True
      alldiff (x:xs) = x `notElem` xs && alldiff xs

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)

We model a deterministic *plan* as a list of pairs of formulas. The first part of the first tuple should be announced truthfully and lead to a model where the second part of the tuple is true. Then we continue with the next tuple. We now use the definition and implementation of plans and success from Section 13.

```
-- the plan for (3,3,1)
basicPlan :: OfflinePlan
basicPlan =
  [ (aAnnounce, Conj [ bKnowsAs, Ck [alice,bob] bKnowsAs, Ck [alice,bob,carol] cIgnorant ]
     )
  , (bAnnounce, Conj [ aKnowsBs, Ck [alice,bob] aKnowsBs, Ck rcPlayers cIgnorant ] ) ]

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 (pickHands (possibleHandsN n na)) [5,6,7] -- FIXME how to
    adapt the number of hands for larger n?
```

Note that we still use the same pickHands. 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 ]

checkSetFor :: RusCardProblem -> [[Int]] -> Bool
checkSetFor (na,nb,nc) set = plan `succeedsOn` 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, Conj [ bKnowsAsN n, Ck [alice,bob] (bKnowsAsN n), Ck [alice,bob,carol] (
       cIgnorantN n) ] )
```

```
   , (bobSays   , Conj [ Ck [alice,bob] $ Conj [aKnowsBsN n, bKnowsAsN n], Ck rcPlayers (
       cIgnorantN n) ] )
   ]

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 ]
```

14.13 Russian Cards on Belief Structures with Less Atoms

```
dontChange :: [Form] -> K.RelBDD
dontChange fs = conSet <$> sequence [ equ <$> K.mvBdd b <*> K.cpBdd b | b <- map boolBddOf
    fs ]

noDoubles :: Int -> Form
noDoubles n = Conj [ notDOuble k | k <- nCards n ] where
  notDOuble k = Neg $ 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"
```

14.14 The Sally-Anne false belief task

```
module SMCDEL.Examples.SallyAnne where

import Data.Map.Strict (fromList)

import SMCDEL.Language
import SMCDEL.Symbolic.K
import SMCDEL.Symbolic.K.Change
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 $(\mathcal{F}_0, s_0) = ((\{p, t\}, (p \wedge \neg t), \top, \top), \{p\})$ where the last two components are Ω_S and Ω_A.

```
pp, qq, tt :: Prp
pp = P 0
tt = P 1
```

```
qq = P 7 -- this number should 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]
```

$$\left(\left(\{p,p_1\},\ \begin{array}{c}0\\ \big/\ \backslash\\ 0\quad 1\end{array}\ ,\Omega_{\text{Anne}}=\ \boxed{1}\ ,\Omega_{\text{Sally}}=\ \boxed{1}\ \right),\{p\}\right)$$

The sequence of events is:
Sally puts the marble in the basket: $(\mathcal{X}_1=(\varnothing,\top,\{t\},\theta_-(t)=\top,\top,\top),\varnothing)$,

```
sallyPutsMarbleInBasket :: Event
sallyPutsMarbleInBasket = (Trf [] Top [tt]
  (fromList [ (tt,boolBddOf Top) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate1 :: BelScene
sallyIntermediate1 = sallyInit `transform` sallyPutsMarbleInBasket
```

$$\left(\left(\varnothing,\top,\{p_1\},p_1:=\top,\Omega^+_{\text{Anne}}=\ \boxed{1}\ ,\Omega^+_{\text{Sally}}=\ \boxed{1}\ \right),\varnothing\right)$$

$$\left(\left(\{p,p_1,p_2\},\ \begin{array}{c}0\\ \big/\ \backslash\\ 1\quad 0\\ \big/\ \backslash\\ 1\quad 0\end{array}\ ,\Omega_{\text{Anne}}=\ \boxed{1}\ ,\Omega_{\text{Sally}}=\ \boxed{1}\ \right),\{p,p_1\}\right)$$

Sally leaves: $(\mathcal{X}_2=(\varnothing,\top,\{p\},\theta_-(p)=\bot,\top,\top),\varnothing)$.

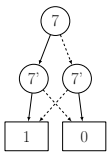
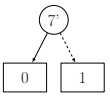
```
sallyLeaves :: Event
sallyLeaves = (Trf [] Top [pp]
  (fromList [ (pp,boolBddOf Bot) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate2 :: BelScene
sallyIntermediate2 = sallyIntermediate1 `transform` sallyLeaves
```

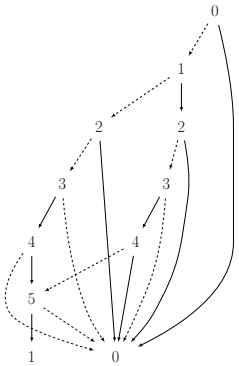
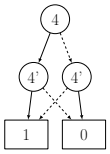
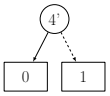
$$\left(\left(\varnothing,\top,\{p\},p:=\bot,\Omega^+_{\text{Anne}}=\ \boxed{1}\ ,\Omega^+_{\text{Sally}}=\ \boxed{1}\ \right),\varnothing\right)$$

$$\left(\left(\{p,p_1,p_2,p_3\},\ \begin{array}{c}0\\ \big/\ \backslash\\ 1\quad \\ \big/\ \backslash\\ 2\quad \\ \big/\ \backslash\\ 0\quad 1\end{array}\ ,\Omega_{\text{Anne}}=\ \boxed{1}\ ,\Omega_{\text{Sally}}=\ \boxed{1}\ \right),\{p_1,p_3\}\right)$$

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 [tt]
  (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 `transform` annePutsMarbleInBox
```

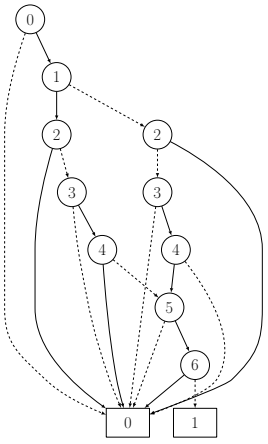
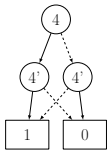
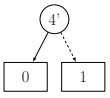
$$\left(\left(\{p_7\}, \top, \{p_1\}, p_1 := (p_1 \wedge \neg p_7), \Omega^+_{\text{Anne}} = \right.\right.$$  $$\left.\left., \Omega^+_{\text{Sally}} = \right.\right.$$  $$\left.\right), \{p_7\}\right)$$

$$\left(\left(\{p, p_1, p_2, p_3, p_4, p_5\}, \right.\right.$$  $$, \Omega_{\text{Anne}} =$$  $$, \Omega_{\text{Sally}} =$$  $$\left.\left.\right), \{p_3, p_4, p_5\}\right)$$

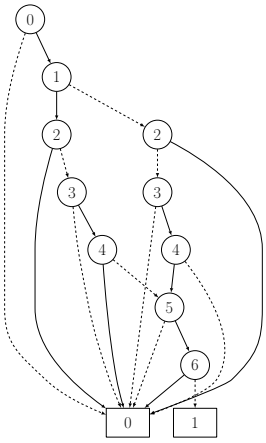
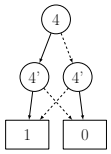
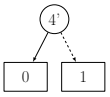
Sally comes back: $(\mathcal{X}_4 = (\varnothing, \top, \{p\}, \theta_-(p) = \top, \top, \top), \varnothing)$.

```
sallyComesBack :: Event
sallyComesBack = (Trf [] Top [pp]
  (fromList [ (pp,boolBddOf Top) ])
  (fromList [ (i,totalRelBdd) | i <- ["Anne","Sally"] ]), [])

sallyIntermediate4 :: BelScene
sallyIntermediate4 = sallyIntermediate3 `transform` sallyComesBack
```

$$\left(\left(\varnothing, \top, \{p\}, p := \top, \Omega^+_{\text{Anne}} = \boxed{1}, \Omega^+_{\text{Sally}} = \boxed{1}\right), \varnothing\right)$$

$$\left(\left(\{p, p_1, p_2, p_3, p_4, p_5, p_6\}, \right.\right.$$  $$, \Omega_{\text{Anne}} =$$  $$, \Omega_{\text{Sally}} =$$  $$\left.\left.\right), \{p, p_3, p_4, p_5\}\right)$$

```
sallyFinal :: BelScene
sallyFinal = sallyInit
               `transform` sallyPutsMarbleInBasket
               `transform` sallyLeaves
               `transform` annePutsMarbleInBox
               `transform` sallyComesBack

sallyFinalCheck :: (Bool,Bool)
sallyFinalCheck =
  ( SMCDEL.Symbolic.K.evalViaBdd sallyFinal (K "Sally" (PrpF tt))
  , sallyIntermediate4 == sallyFinal )
```

$$\left(\left(\{p, p_1, p_2, p_3, p_4, p_5, p_6\},\ \text{  },\ \Omega_{\text{Anne}} = \text{  },\ \Omega_{\text{Sally}} = \text{  }\right), \{p, p_3, p_4, p_5\}\right)$$

```
>>> SMCDEL.Examples.SallyAnne.sallyFinalCheck
(True,True)
0.12 seconds
```

We check that in the last scene Sally believes the marble is in the basket:

$\{p, q\} \vDash \Box_{\text{S}} t$

iff $\{p, q\} \vDash \forall V'(\theta' \to (\Omega_{\text{S}} \to t'))$

iff $\{p, q\} \vDash \forall \{p', t', q'\}((t' \leftrightarrow \neg q') \wedge p' \to (\neg q' \to t'))$

iff $\{p, q\} \vDash \top$

14.15 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.

```
-- 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` $= \bigwedge\{p_1, p_2, p_3, p_4, p_6, \neg p_5, \neg p_7\}$

```
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.

```
sapSolutions :: [[Prp]]
sapSolutions = SMCDEL.Symbolic.S5.whereViaBdd sapKnStruct sapProtocol
```

```
>>> SMCDEL.Examples.SumAndProduct.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]]

1.53 seconds
```

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

```
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 SMCDEL.Examples.SumAndProduct.sapExplainState
SMCDEL.Examples.SumAndProduct.sapSolutions

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

1.52 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 SMCDEL.Examples.SumAndProduct.sapKnStruct (Impl (Conj
[SMCDEL.Examples.SumAndProduct.xIs 4, SMCDEL.Examples.SumAndProduct.yIs 13])
SMCDEL.Examples.SumAndProduct.sapProtocol)

  True

1.52 seconds
```

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

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

  True

1.54 seconds
```

Our implementation is faster than the one in [Luo+08].

14.16 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 [VR07] 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:

```
pubAnnounceSeq :: [Form] -> KnowStruct -> KnowStruct
pubAnnounceSeq [] = id
pubAnnounceSeq (f:fs) = \kns -> pubAnnounceSeq fs (pubAnnounce kns f)

wsResult :: KnowStruct
wsResult =
  pubAnnounceSeq
    [ Neg wsKnowSelfA, Neg wsKnowSelfB, Neg wsKnowSelfC ]
    wsKnStruct

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 [VR07]	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 [VR07, 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.

15 Benchmarks

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

15.1 Muddy Children

In this section we compare the performance of different model checking functions using the Muddy Children example from Section 14.7.

- 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 117.

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

```
module Main where
import Criterion.Main
import Data.Function
import Data.List
import Data.Ord (comparing)
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
import qualified SMCDEL.Explicit.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.

```
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) $
      sortBy (comparing (setForAt i)) ws
    setForAt i s = delete (P i) $ setAt s
    setAt s = map fst $ filter snd (apply val s)
  val         = zip ws table
  ((cur,_):_) = filter (\(_,ass)-> sort (map fst $ filter snd ass) == [P 1..P m]) val
  table = foldl buildTable [[]] [P k | k<- [1..n]]
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr<-partrows ]

findNumberNonS5 :: Int -> Int -> Int
findNumberNonS5 = findNumberWith (mudBelScnInit, SMCDEL.Symbolic.K.evalViaBdd)

findNumberNonS5Trans :: Int -> Int -> Int
findNumberNonS5Trans = findNumberWith (start,SMCDEL.Symbolic.K.evalViaBdd) where
  start n m = SMCDEL.Translations.K.kripkeToBls $ SMCDEL.Explicit.K.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.

```
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 117 for the details. Here the formula `nobodyknows` does not depend on the number of agents and therefore the loop function does not have to pass on any variables.

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

findNumberTriangleLoop :: Int -> SMCDEL.Other.MCTRIANGLE.McModel -> Int
findNumberTriangleLoop count curMod =
  if SMCDEL.Other.MCTRIANGLE.eval curMod SMCDEL.Other.MCTRIANGLE.nobodyknows
```

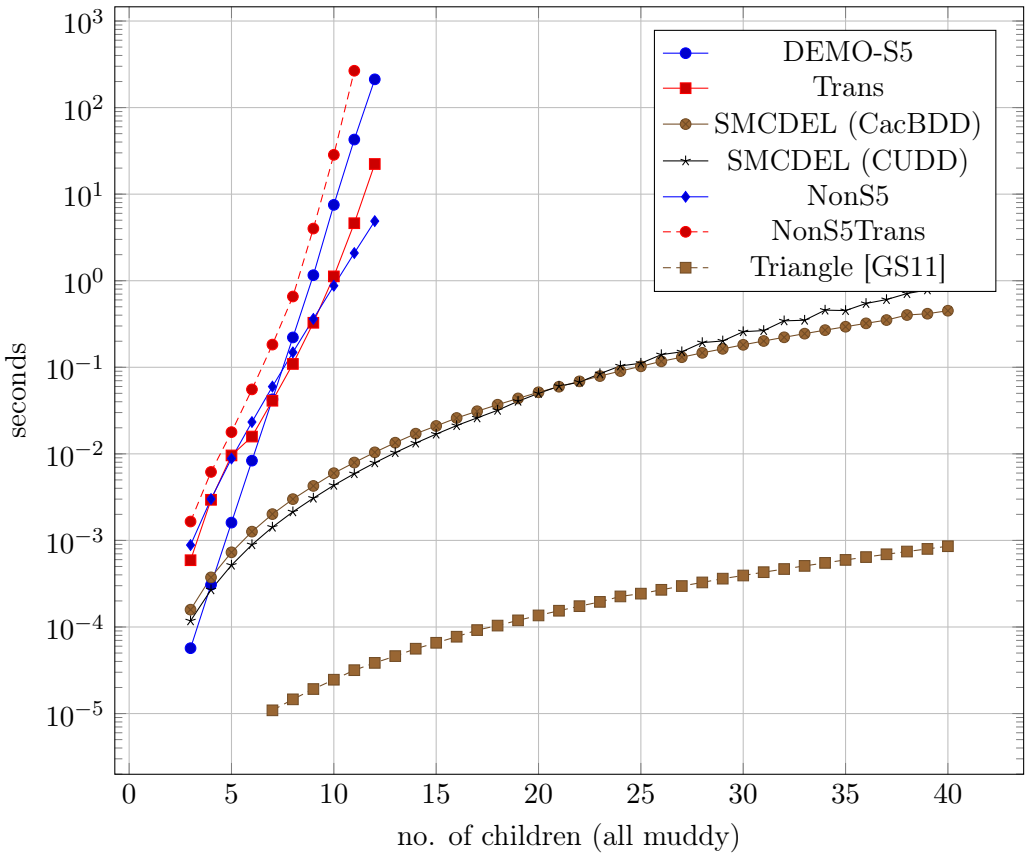


Figure 5: Benchmark Results on a logarithmic scale.

```
    then findNumberTriangleLoop (count+1) (SMCDEL.Other.MCTRIANGLE.update 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] )
  , ("NonS5"     , findNumberNonS5     , [3..12] )
  , ("DEMOS5"    , findNumberDemoS5    , [3..12] )
  , ("Trans"     , findNumberTrans     , [3..12] )
  , ("NonS5Trans", findNumberNonS5Trans, [3..11] ) ])
  where
    mybench (name,f,range) = bgroup name $ map (run f) range
    run f k = bench (show k) $ whnf (\n -> f n n) k
```

As expected we can see in Figure 5 that *SMCDEL* is faster than the explicit model checker DEMO. 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 will be subject to future work if the idea underlying this approach – the identification of agents in the same informational state – can be generalized to other protocols or ideally the full DEL language.

15.2 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 14.3. 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 14.3.

```
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.

15.3 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 14.15.

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 :: Form (Int,Int)

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

16 Executables

16.1 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

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

Is this valid on the given structure? True

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

At which states is this true? $\varnothing$

For more examples, see the `Examples` folder.

```
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
  let (CheckInput vocabInts lawform obs jobs) = parse $ alexScanTokens input
  let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
  when texMode $
    hPutStrLn outHandle $ unlines
      [ "\\section{Given Knowledge Structure}", "\\[ (\\mathcal{F},s) = (" ++ tex ((mykns
          ,[])::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 }
```

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 }

%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]

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

16.2 Web Interface

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

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Prelude
import Control.Monad.IO.Class (liftIO)
import Control.Arrow
import Data.List (intercalate)
import Data.Version (showVersion)
import Paths_smcdel (version)
import Web.Scotty
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.IO as TIO
import SMCDEL.Internal.Lex
import SMCDEL.Internal.Parse
import SMCDEL.Internal.Files
import SMCDEL.Symbolic.S5
import SMCDEL.Internal.TexDisplay
import SMCDEL.Translations.S5
```

```
import SMCDEL.Language
import Data.HasCacBDD.Visuals (svgGraph)
import qualified Language.Javascript.JQuery as JQuery

main :: IO ()
main = do
  putStrLn $ "SMCDEL " ++ showVersion version
  putStrLn "Please open this link: http://localhost:3000/index.html"
  scotty 3000 $ do
    get "" $ redirect "index.html"
    get "/" $ redirect "index.html"
    get "/index.html" . html . T.fromStrict $ embeddedFile "index.html"
    get "/jquery.js" $ liftIO (JQuery.file >>= TIO.readFile) >>= text
    get "/viz-lite.js" . html . T.fromStrict $ embeddedFile "viz-lite.js"
    get "/getExample" $ do
      this <- param "filename"
      html . T.fromStrict $ embeddedFile this
    post "/check" $ do
      smcinput <- param "smcinput"
      let (CheckInput vocabInts lawform obs jobs) = parse $ alexScanTokens smcinput
      let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
      knstring <- liftIO $ showStructure mykns
      let results = concatMap (\j -> "<p>" ++ doJobWeb mykns j ++ "</p>") jobs
      html $ mconcat
        [ T.pack knstring
        , "<hr />\n"
        , T.pack results ]
    post "/knsToKripke" $ do
      smcinput <- param "smcinput"
      let (CheckInput vocabInts lawform obs _) = parse $ alexScanTokens smcinput
      let mykns = KnS (map P vocabInts) (boolBddOf lawform) (map (second (map P)) obs)
      _ <- liftIO $ showStructure mykns -- this moves parse errors to scotty
      if numberOfStates mykns > 32
        then html . T.pack $ "Sorry, I will not draw " ++ show (numberOfStates mykns) ++ "
            states!"
        else do
          let (myKripke, _) = knsToKripke (mykns, head $ statesOf mykns) -- ignore actual
              world
          html $ T.concat
            [ T.pack "<div id='here'></div>"
            , T.pack "<script>document.getElementById('here').innerHTML += Viz('"
            , textDot myKripke
            , T.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>"
```

17 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

```
module SMCDEL.Internal.Help (
  alleq,alleqWith,anydiff,anydiffWith,alldiff,
  apply,applyPartial,(!),(!=),
  powerset,restrict,rtc,tc,Erel,bl,fusion,seteq,subseteq,lfp
  ) where
import Data.List (nub,union,sort,foldl',(\\))

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

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: 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:

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

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

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

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

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

Appendix: DEMO-S5

```
-- Note: This is a slightly modified version of DEMO-S5 by Jan van Eijck
-- which can be found at 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 Prp = P Int | Q Int | R Int | S Int deriving (Eq,Ord)
instance Show Prp where
  show (P 0) = "p"; show (P i) = "p" ++ show i
  show (Q 0) = "q"; show (Q i) = "q" ++ show i
  show (R 0) = "r"; show (R i) = "r" ++ show i
  show (S 0) = "s"; show (S i) = "s" ++ show i

data EpistM state = Mo
             [state]
             [Agent]
             [(state,[Prp])]
             [(Agent,Erel state)]
             [state]  deriving (Eq,Show)

rel :: Show a => Agent -> EpistM a -> Erel a
rel ag (Mo _ _ _ rels _) = apply rels ag

initM :: (Num state, Enum state) =>
         [Agent] -> [Prp] -> 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 Form a = Top
            | Info a
            | Prp Prp
            | Ng (Form a)
            | Conj [Form a]
            | Disj [Form a]
            | Kn Agent (Form a)
          deriving (Eq,Ord,Show)

impl :: Form a -> Form a -> Form a
impl form1 form2 = Disj [Ng form1, form2]

isTrueAt :: Show state => Ord state =>
            EpistM state -> state -> Form state -> Bool
isTrueAt _ _ Top = True
isTrueAt _ w (Info x) = w == x
isTrueAt
  (Mo _ _ val _ _) w (Prp p) = let
   props = apply val w
  in
   elem p props
```

```
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) = let
    r = rel ag m
    b = bl r w
  in
    all (flip (isTrueAt m) f) b

isTrue :: Show a => Ord a => EpistM a -> Form a -> Bool
isTrue m@(Mo _ _ _ _ points) f =
  all (\w -> isTrueAt m w f) points

updPa :: Show state => Ord state =>
          EpistM state -> Form 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 -> [Form state] -> EpistM state
updsPa = foldl updPa

sub :: Show a => [(Prp,Form a)] -> Prp -> Form a
sub subst p =
  if p `elem` map fst subst
    then apply subst p
    else Prp p

updPc :: Show state => Ord state => [Prp] -> EpistM state
                    -> [(Prp,Form 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 => [Prp] -> EpistM state
                     -> [[(Prp,Form state)]] -> EpistM state
updsPc ps = foldl (updPc ps)

updPi :: (state -> state) -> EpistM state -> EpistM state
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 -> Form [Bool]
fatherN n = Ng (Info (replicate n False))

kn :: Int -> Int -> Form [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 -> Form [Bool]
dont n = Conj [Ng (kn n i) | i <- [1..n] ]

knowN :: Int -> Form [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). 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 '98. 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: *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning*. KR'10. 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.

[Dit03] Hans van Ditmarsch. "The Russian Cards problem". In: *Studia Logica* 75.1 (2003), pp. 31–62. DOI: 10.1023/A:1026168632319.

[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+15] Thorsten Engesser, Thomas Bolander, Robert Mattmüller, and Bernhard Nebel. "Cooperative Epistemic Multi-Agent Planning With Implicit Coordination". In: *ICAPS Proceedings of the 3rd Workshop on Distributed and Multi-Agent Planning (DMAP-2015)* (2015), pp. 68–76. URL: https://is.gd/DMAP2015_p68.

[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.

[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'11*. 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.

[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.

[VR07] Hans Van Ditmarsch and Ji Ruan. "Model Checking Logic Puzzles". In: *Annales du Lamsade* 8 (2007). URL: https://hal.archives-ouvertes.fr/hal-00188953.