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 1$^{\text{st}}$ June, 2015

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 [BEGS15].

The report is structured as follows.

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

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

In Section 6 we show on concrete models 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 and Russian Cards. These examples also suggest themselves as benchmarks which we will do in Section 7 to compare the different versions of our model checker to the existing tools DEMO-S5 and MCMAS.

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

In the appendix we provide some installation guidelines, a helper functions module and an implementation of a number triangle analysis of the Muddy Children problem [GS11].

The report is given in literate Haskell style, including all source code and the results of example programs directly in the text. SCMDEL is released as free software under the GPL.

Contents

1   The Language of Dynamic Epistemic Logic

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

```
module DELLANG where
import Data.List (nub,intercalate,(\\))
import Data.Maybe (fromJust)
```

Propositions and Agents are simply represented as integers in Haskell.

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

freshp :: [Prp] -> Prp
freshp [] = P 0
freshp prps = P (maximum (map fromEnum prps) + 1)

type Agent = Int
alice,bob,carol :: Agent
alice   = 0
bob     = 1
carol   = 2

showAgent :: Agent -> String
showAgent 0 = "Alice"
showAgent 1 = "Bob"
showAgent 2 = "Carol"
showAgent n = "Ag " ++ show n
```

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

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

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 $[!\varphi]_\Delta$.*

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 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":*

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

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

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

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

```
data Form =
  Top | Bot | PrpF Prp | Neg Form | Conj [Form] | Disj [Form] | Xor [Form] |
  Impl Form Form | Equi Form Form | Forall [Prp] Form | Exists [Prp] Form |
  K Agent Form | Ck [Agent] Form | Kw Agent Form | Ckw [Agent] Form |
  PubAnnounce Form Form | PubAnnounceW Form Form |
  Announce [Agent] Form Form | AnnounceW [Agent] Form Form
  deriving (Eq,Show)
```

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

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

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

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

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

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

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

```
replPsInF :: [(Prp,Prp)] -> Form -> Form
replPsInF _       Top     = Top
replPsInF _       Bot     = Bot
replPsInF repl (PrpF p)   | p `elem` map fst repl = PrpF (fromJust $ lookup p repl)
                          | otherwise = PrpF p
replPsInF repl (Neg f)    = Neg $ replPsInF repl f
replPsInF repl (Conj fs)  = Conj $ map (replPsInF repl) fs
replPsInF repl (Disj fs)  = Disj $ map (replPsInF repl) fs
replPsInF repl (Xor  fs)  = Xor  $ map (replPsInF repl) fs
replPsInF repl (Impl f g) = Impl (replPsInF repl f) (replPsInF repl g)
replPsInF repl (Equi f g) = Equi (replPsInF repl f) (replPsInF repl g)
```

```
replPsInF repl (Forall ps f) = Forall (map (fromJust . flip lookup repl) ps) (replPsInF
    repl f)
replPsInF repl (Exists ps f) = Exists (map (fromJust . flip lookup repl) ps) (replPsInF
    repl f)
replPsInF repl (K i f)     = K i (replPsInF repl f)
replPsInF repl (Kw i f)    = Kw i (replPsInF repl f)
replPsInF repl (Ck ags f)  = Ck ags (replPsInF repl f)
replPsInF repl (Ckw ags f) = Ckw ags (replPsInF repl f)
replPsInF repl (PubAnnounce f g)   = PubAnnounce   (replPsInF repl f) (replPsInF repl g)
replPsInF repl (PubAnnounceW f g)  = PubAnnounceW  (replPsInF repl f) (replPsInF repl g)
replPsInF repl (Announce ags f g)  = Announce  ags (replPsInF repl f) (replPsInF repl g)
replPsInF repl (AnnounceW ags f g) = AnnounceW ags (replPsInF repl f) (replPsInF repl g)
```

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

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

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

texProp :: Prp -> String
texProp (P 0) = " p "
texProp (P n) = " p_{" ++ show n ++ "} "

texPropSet :: [Prp] -> String
texPropSet [] = " \\varnothing "
texPropSet ps = "\\{" ++ intercalate "," (map texProp 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
                     | otherwise     = Conj (nub $ map simStep (filter (Top /=) fs))
simStep (Disj [])    = Bot
simStep (Disj [f])   = simStep f
simStep (Disj fs)    | Top 'elem' fs = Top
                     | otherwise     = Disj (nub $ map simStep (filter (Bot /=) fs))
simStep (Xor  [])    = Bot
simStep (Xor  [f])   = Neg $ simStep f
```

```
simStep (Xor   fs)      = Xor (map simStep fs)
simStep (Impl Bot _)    = Top
simStep (Impl _ Top)    = Top
simStep (Impl Top f)    = simStep f
simStep (Impl f Bot)    = Neg (simStep f)
simStep (Impl f g)      = 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)      = Equi (simStep f) (simStep g)
simStep (Forall ps f)   = Forall ps (simStep f)
simStep (Exists ps f)   = Exists ps (simStep f)
simStep (K a f)         = K a (simStep f)
simStep (Kw a f)        = Kw a (simStep f)
simStep (Ck ags f)      = Ck ags (simStep f)
simStep (Ckw ags f)     = Ckw ags (simStep f)
simStep (PubAnnounce Top f) = simStep f
simStep (PubAnnounce Bot _) = Top
simStep (PubAnnounce  f g)   = PubAnnounce  (simStep f) (simStep g)
simStep (PubAnnounceW f g)   = PubAnnounceW (simStep f) (simStep g)
simStep (Announce  ags f g) = Announce  ags (simStep f) (simStep g)
simStep (AnnounceW ags f g) = AnnounceW ags (simStep f) (simStep g)
```

We end this module with a small helper function to abbreviate 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 ]
```

2 DEL Semantics on Kripke Models

We start with a quick summary of the standard semantics for DEL on Kripke models. The module of this section provides a very simple explicit state model checker. It is mainly provided as a basis for the translation methods in Section 4 and not meant to be used in practice otherwise. A more advanced and user-friendly explicit state model checker for DEL is DEMO from [Eij14] which we will also use later on.

```
module KRIPKEDEL where
import Data.List (intercalate)
import DELLANG
import KRIPKEVIS
import HELP (alleq,fusion,apply)
```

2.1 Kripke Models

Definition 3. *A* Kripke model *for n agents is a tuple $M = (W, \pi, \mathcal{K}_1, \cdots, \mathcal{K}_n)$, 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 $\mathcal{K}_1, \cdots, \mathcal{K}_n$ are binary accessibility relations on W. By convention, W^M, $\mathcal{K}_i^M$ and π^M are used to refer to the components of M. We omit the superscript M if it is clear from context. Finally, let $\mathcal{C}_\Delta^M$ be the transitive closure of $\bigcup_{i \in \Delta} \mathcal{K}_i^M$.*

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

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

```
type State = Int
type Partition = [[State]]
type Assignment = [(Prp,Bool)]
data KripkeModel = KrM [State] [(Agent,Partition)] [(State,Assignment)] deriving (Show)
type PointedModel = (KripkeModel,State)
```

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

1. *$(M, w) \models p$ iff $\pi^M(w)(p) = \top$.*

2. *$(M, w) \models \neg\varphi$ iff not $(M, w) \models \varphi$*

3. *$(M, w) \models \varphi \wedge \psi$ iff $(M, w) \models \varphi$ and $(M, w) \models \psi$*

4. *$(M, w) \models K_i\varphi$ iff for all $w' \in W$, if $w\mathcal{K}_i^M w'$, then $(M, w') \models \varphi$.*

5. *$(M, w) \models C_\Delta\varphi$ iff for all $w' \in W$, if $w\mathcal{C}_\Delta^M w'$, then $(M, w') \models \varphi$.*

6. *$(M, w) \models [\psi]\varphi$ iff $(M, w) \models \psi$ implies $(M^\psi, w) \models \varphi$ where M^ψ is a new Kripke model defined by the set $W^{M^\psi} := \{w \in W^M \mid (M, w) \models \psi\}$, the relations $\mathcal{K}_i^{M^\psi} := \mathcal{K}_i^M \cap (W^{M^\psi})^2$ and the valuation $\pi^{M^\psi}(w) := \pi^M(w)$.*

7. *$(M, w) \models [\psi]_\Delta\varphi$ iff $(M, w) \models \psi$ implies that also $(M_\psi^\Delta, (1, w)) \models \varphi$ where M_ψ^Δ is defined by*

   (a) *$W^{M_\psi^\Delta} := \{(1, w) \mid w \in W^M \text{ and } (M, w) \models \psi\} \cup \{(0, w) \mid w \in W^M\}$*

   (b) *For (b, w) and (b', w') in $W^{M_\psi^\Delta}$, if $i \in \Delta$, let $(b, w)\mathcal{K}_i^{M_\psi^\Delta}(b', w')$ iff $b = b'$ and $w\mathcal{K}_i^M w'$. If $i \notin \Delta$, then let $(b, w)\mathcal{K}_i^{M_\psi^\Delta}(b', w')$ iff $w\mathcal{K}_i^M w'$.*

   (c) *For each $(b, w) \in W^{M_\psi^\Delta}$, $\pi^{M_\psi^\Delta}((b, w)) := \pi^M(w)$.*

7

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

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

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

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

announce :: PointedModel -> [Agent] -> Form -> PointedModel
announce pm@(m@(KrM sts rel val), cur) ags form =
  if eval pm form then (KrM newsts newrel newval, newcur)
                  else error "announce failed: Liar!"
  where
    tocopy = filter (\s -> eval (m,s) form) sts
    addsts = map (maximum sts +) [1..(length tocopy)]
    copyto = zip tocopy addsts
    copyof = zip addsts tocopy
    mapif  = concatMap (\s -> [apply copyto s | s `elem` tocopy])
    nrel i | i `elem` ags = apply rel i ++ filter ([]/=) (map mapif (apply rel i))
           | otherwise = map (\ec -> ec ++ mapif ec) (apply rel i)
    newsts = sts ++ addsts
    newrel = [ (i, nrel i) | i <- map fst rel ]
    newval = val ++ [ (s,apply val $ apply copyof s)  | s <- addsts ]
    newcur = apply copyto cur
```

With a few lines we can also visualize our models using KRIPKEVIS [Gat14]. To see what the output looks like, see Sections 6.1 and 6.2.

```
showVal :: Assignment -> String
showVal ass = case filter snd ass of
  [] -> ""
  ps -> "$" ++ intercalate "," (map (texProp.fst) ps) ++ "$"

myDispModel :: PointedModel -> IO ()
myDispModel (KrM w r v, cur) = dispModel show showAgent showVal "" (VisModel w r v cur)

myTexModel :: PointedModel -> String -> IO String
myTexModel (KrM w r v, cur) = texModel show showAgent showVal "" (VisModel w r v cur)
```

2.2 Action Models

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

Definition 5. *An* action model *is a tuple* $\mathcal{A} = (A, \mathcal{R}, \mathsf{pre})$ *where A is a set of* action tokens, *$\mathcal{R} = (R_i)_{i \in I}$ is a family of equivalence relations on A and* pre *is a function from A to $\mathcal{L}_{\mathsf{DEL}}$, defining the* precondition $\mathsf{pre}(\alpha)$ *of each $\alpha \in A$.*

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

Definition 6. *The* product update *with an action model* $((A, \mathcal{R}, \mathsf{pre}), \alpha)$ *is a function that maps Kripke models to Kripke models and is defined as follows:*

$$
\begin{aligned}
(W, \mathcal{R}, V) \quad \mapsto \quad &(W', \mathcal{R}', V') \ \textit{where} \\
&W' := \{(w, \alpha) \in W \times A \mid w \vDash \mathsf{pre}(\alpha)\} \\
&(w, \alpha) R_i'(v, \beta) \ \textit{iff} \ w R_i v \ \textit{and} \ \alpha R_i \beta \\
&V'(w, \alpha) := V(w)
\end{aligned}
$$

We write $\mathcal{M}^{\mathcal{A}}$ for the result of updating $\mathcal{M}$ with $\mathcal{A}$.

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

3 DEL Semantics on Knowledge Structures

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

i) *CacBDD* [LSX13], a modern BDD package with dynamic cache management implemented in C++. We use it via the library *HasCacBDD* [Gat15a] which provides Haskell-to-C-to-C++ bindings.

ii) *CUDD* [Som12], probably the best-known BDD library which is used many in other model checkers, including MCMAS [LQR15], MCK [GvdM04] and NuSMV [CCG+02]. It is implemented in C and we use it via the binding library *hBDD* [Gam14].

iii) *robbed* [Rav14], an advanced Haskell library working with reduced and ordered BDDs.

iv) *NooBDD* [Gat15b], a simple and naive Haskell library for non-reduced ordered BDDs.

The corresponding Haskell modules are called `KNSCAC`, `KNSCUDD`, `KNSROB` and `KNSNOO`. For now we focus on the *CacBDD* variant as it can be seen in the beginning of this module.

```
module KNSCAC where
import Data.HasCacBDD hiding (Top,Bot)
import Data.HasCacBDD.Visuals
import Data.List (sort,intercalate,(\\))
import System.IO (hPutStr, hGetContents)
import System.Process (runInteractiveCommand)
import HELP (alleq,apply,rtc)
import DELLANG
```

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 (Impl f g)    = imp (boolBddOf f) (boolBddOf g)
boolBddOf (Equi f g)    = equ (boolBddOf f) (boolBddOf g)
boolBddOf (Forall ps f) = boolBddOf (foldl singleForall f ps) where
  singleForall g p = Conj [ substit p Top g, substit p Bot g ]
boolBddOf (Exists ps f) = boolBddOf (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
boolBddOf _             = error "boolBddOf failed: Not a boolean formula."

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

3.1 Knowledge Structures

Knowledge structures are a compact representation of S5 Kripke models. Their set of states is defined by a boolean formula and instead of epistemic relations we use observational variables. More explanations and proofs that they are indeed equivalent to S5 Kripke models can be found in [BEGS15].

Definition 7. *Fix 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, given as the set of true propositions, 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 $s \cap O_i$.*

Given a knowledge structure $(V, \theta, O_1, \cdots, O_n)$ and a set $\mathcal{V}$ of subsets of V, we use $\mathcal{E}_\mathcal{V}$ to denote a relation between two assignments s, s' on V satisfying θ such that $(s, s') \in \mathcal{E}_\mathcal{V}$ iff there exists a $P \in \mathcal{V}$ with $s \cap P = s' \cap P$. We use $\mathcal{E}_\mathcal{V}^*$ to denote the transitive closure of $\mathcal{E}_\mathcal{V}$. Let $\mathcal{V}_\Delta = \{O_i \mid i \in \Delta\}$. We then have $(s, s') \in \mathcal{E}_{\mathcal{V}_\Delta}$ iff there exists an $i \in \Delta$ with $s \cap O_i = s' \cap O_i$.

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

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

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

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

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

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

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

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

Definition 8. *Semantics for DEL on scenes are defined inductively as follows.*

1. *$(\mathcal{F}, s) \models p$ iff $s \models p$.*

2. *$(\mathcal{F}, s) \models \neg\varphi$ iff not $(\mathcal{F}, s) \models \varphi$*

3. *$(\mathcal{F}, s) \models \varphi \wedge \psi$ iff $(\mathcal{F}, s) \models \varphi$ and $(\mathcal{F}, s) \models \psi$*

4. *$(\mathcal{F}, s) \models K_i\varphi$ iff for all s' of $\mathcal{F}$, if $s \cap O_i = s' \cap O_i$, then $(\mathcal{F}, s') \models \varphi$.*

5. *$(\mathcal{F}, s) \models C_\Delta\varphi$ iff for all s' of $\mathcal{F}$, if $(s, s') \in \mathcal{E}_{\mathcal{V}_\Delta}^*$, then $(\mathcal{F}, s') \models \varphi$.*

6. *$(\mathcal{F}, s) \models [\psi]\varphi$ iff $(\mathcal{F}, s) \models \psi$ implies $(\mathcal{F}^\psi, s) \models \varphi$ where $\|\psi\|_\mathcal{F}$ is given by Definition 9 and*

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

7. *$(\mathcal{F}, s) \models [\psi]_\Delta\varphi$ iff $(\mathcal{F}, s) \models \psi$ implies $(\mathcal{F}_\psi^\Delta, s \cup \{p_\psi\}) \models \varphi$ where p_ψ is a a new propositional variable, $\|\psi\|_\mathcal{F}$ is given by Definition 9 and*

$$\mathcal{F}_\psi^\Delta := (V \cup \{p_\psi\}, \theta \wedge (p_\psi \to \|\psi\|_\mathcal{F}), O_1', \cdots, O_n')$$

where $O_i' := \cup\{p_\psi\}$ if $i \in \Delta$ and $O_i' := O_i$ otherwise.

Whenever $(\mathcal{F}, s) \vDash \varphi$ holds we say that φ is true *at s in $\mathcal{F}$*. If this is the case for all states s of $\mathcal{F}$, then we say φ is valid *on $\mathcal{F}$ and write $\mathcal{F} \vDash \varphi$*.

The following function `eval` implements these semantics. An important warning: This function is not a symbolic algorithm! It is a direct translation of Definition 8. In particular it calls `statesOf` which means that the set of stats is explicitly generated. The symbolic counterpart of `eval` is `evalViaBdd`, see below.

```
eval :: Scenario -> Form -> Bool
eval _         Top          = True
eval _         Bot          = False
eval (_,s)    (PrpF p)      = p 'elem' s
eval (kns,s) (Neg form)     = not $ eval (kns,s) form
eval (kns,s) (Conj forms)   = all (eval (kns,s)) forms
eval (kns,s) (Disj forms)   = any (eval (kns,s)) forms
eval (kns,s) (Xor  forms)   = odd $ length (filter id $ map (eval (kns,s)) forms)
eval scn      (Impl f g)    = not (eval scn f) || eval scn g
eval scn      (Equi f g)    = eval scn f == eval scn g
eval scn      (Forall ps f) = eval scn (foldl singleForall f ps) where
  singleForall g p = Conj [ substit p Top g, substit p Bot g ]
eval scn      (Exists ps f) = eval scn (foldl singleExists f ps) where
  singleExists g p = Disj [ substit p Top g, substit p Bot g ]
eval (kns@(KnS _ _ obs),s) (K i form) = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns@(KnS _ _ obs),s) (Kw i form) = alleq (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> seteq (restrictState s' oi) (restrictState s oi)) (statesOf kns)
  oi = apply obs i
eval (kns,s) (Ck ags form)  = all (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> (sort s, sort s') 'elem' comknow kns ags) (statesOf kns)
eval (kns,s) (Ckw ags form) = alleq (\s' -> eval (kns,s') form) theres where
  theres = filter (\s' -> (sort s, sort s') 'elem' comknow kns ags) (statesOf kns)
eval (kns,s) (PubAnnounce form1 form2) =
  not (eval (kns, s) form1) || eval (pubAnnounce kns form1, s) 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 (kns@(KnS props _ _),s) (Announce ags form1 form2) =
  not (eval (kns, s) form1) || eval (announce kns ags form1, freshp props : s) form2
eval (kns,s) (AnnounceW ags form1 form2) =
  if eval (kns, s) form1
    then eval (announce kns ags form1, s) form2
    else eval (announce kns ags (Neg form1), s) 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 8.

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

pubAnnounceOnScn :: Scenario -> Form -> Scenario
pubAnnounceOnScn (kns,s) psi = if eval (kns,s) psi
                                  then (pubAnnounce kns psi,s)
                                  else 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 (imp (var k) (bddOf kns psi))
  newobs    = [(i, apply obs i ++ [proppsi | i 'elem' ags]) | i <- map fst obs]
```

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 9. *Given any knowledge structure $\mathcal{F} = (V, \theta, O_1, \cdots, O_n)$ and any DEL formula φ, we define a boolean formula $\|\varphi\|_{\mathcal{F}}$.*

1. *For any primitive formula, $\|p\|_{\mathcal{F}} := p$.*

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 $\|C_\Delta\psi\|_{\mathcal{F}} := \mathbf{gfp}\Lambda$ where Λ is the following operator on boolean formulas given and $\mathbf{gfp}\Lambda$ 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 $\|[\psi]_\Delta\xi\|_{\mathcal{F}} := \|\psi\|_{\mathcal{F}} \to (\|\xi\|_{\mathcal{F}^\psi_\Delta})(\frac{p_\psi}{\top})$.*

where $\mathcal{F}^\psi$ and $\mathcal{F}^\psi_\Delta$ are as given by Definition 8.

```
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) newform2 where
    newform2 = 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
```

Given these definitions, a proof by induction on φ gives us the following Theorem.

Theorem 10. *Definition 9 preserves and reflects truth. That is, for any formula φ and any scene $(\mathcal{F}, s)$ we have that $(\mathcal{F}, s) \models \varphi$ iff $s \models \|\varphi\|_{\mathcal{F}}$.*

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

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

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

Theorem 11. *Definition 9 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
```

3.2   Knowledge Transformers

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

Definition 12. *A knowledge transformer for a given vocabulary V is a tuple $\mathcal{X} = (V^+, \theta^+, O_1, \ldots, O_n)$ where V^+ is a set of atomic propositions such that $V \cap V^+ = \varnothing$, θ^+ is a possibly epistemic formula over $V \cup V^+$ and $O_i \subseteq V^+$ for all agents i. An* event *is a knowledge transformer together with a subset $x \subseteq V^+$, written as $(\mathcal{X}, x)$.*

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

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

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

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

Note that θ^+ does not have to be a boolean formula. This is similar to preconditions in action models which can also be arbitrary formulas. Still, applying a knowledge transformer to a knowledge structure again yields a knowledge structure in which the set of states is determined by the state law which has to be a purely boolean formula. Hence, in Definition 12 we not just take the conjunction of θ and θ^+ but instead use the boolean equivalent of θ^+. This formula will be equivalent on the previous, but not necessarily on the new structure. Again this fits to the established action model framework: Truthful announcements can be unsuccessful in the sense that after something is publicly announced it is not true anymore. Famous examples are the so-called Moore sentences of the form "It snows and you don't know it."

Our usage of Definition 9 in Definition 12 is somewhat special: Whenever rewriting an epistemic statement, propositions in V^+ are ignored. For example, to rewrite K_i we quantify over $V \setminus O_i$ and

not over $V \cup V^+ \setminus O_i \cup O_i^+$ as one might first think. The latter would yield boolean equivalents with respect to $\mathcal{F}^{\mathcal{X}}$ but we need a boolean formula that was equivalent to the precondition before the action occurred.

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`.

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

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

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

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

texStructure :: Scenario -> String -> IO String
texStructure (KnS props lawbdd obs, state) filename = do
  lawbddtex <- texBDD lawbdd
  let fullstring = " \\left( \n"
        ++ texPropSet props ++ ", "
        ++ " \\begin{array}{l} \\scalebox{0.4}{"++lawbddtex++"} \\end{array}\n "
        ++ ", \\begin{array}{l}\n"
        ++ intercalate " \\\\\n " (map (\(_,os) -> (texPropSet os)) obs)
        ++ "\\end{array}\n"
        ++ " \\right) , " ++ texPropSet state
  _ <- writeFile ("tmp/" ++ filename ++ ".tex") fullstring
  return ("Structure was TeX'd to"++filename)
```

4 Connecting the two Semantics

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

```
module SYMDEL where
import Control.Arrow (second)
import Data.List (groupBy,sort,(\\))
import DELLANG
import KNSCAC
import KRIPKEDEL
import HELP (apply,powerset)
```

Lemma 13. *Suppose we have a knowledge structure $\mathcal{F} = (V', \theta, O_1, \cdots, O_n)$ and a finite S5 Kripke model $M = (W, \pi, \mathcal{K}_1, \cdots, \mathcal{K}_1)$ with a set of primitive propositions $V \subseteq V'$. 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 $g(w_1) \cap O_i = g(w_2) \cap O_i$ iff $w_1 \mathcal{K}_i w_2$.

*C2 For all $w \in W$ and $v \in V$, we have $v \in g(w)$ iff $\pi(w)(v) = $ **true**.*

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 formula φ over V we have $(\mathcal{F}, g(w)) \models \varphi$ iff $(M, w) \models \varphi$.

4.1 From Knowledge Structures to Kripke Models

Definition 14. *For any knowledge structure $\mathcal{F} = (V, \theta, O_1, \cdots, O_n)$, we define the Kripke model $M(\mathcal{F}) := (W, \pi, \mathcal{K}_1, \cdots, \mathcal{K}_n)$ 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 $w, w' \in W$, let $w \mathcal{K}_i w'$ iff $w \cap O_i = w' \cap O_i$.

Theorem 15. *For any scene $\mathcal{F}, s$ and any formula φ we have $(\mathcal{F}, s) \models \varphi$ iff $(M(\mathcal{F}), s) \models \varphi$.*

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

4.2 From Kripke Models to Knowledge Structures

Definition 16. *For any S5 model $M = (W, \pi, \mathcal{K}_1, \cdots, \mathcal{K}_n)$ we define a knowledge structure $\mathcal{F}(M)$ as follows. For each i, write $\gamma_1, \ldots, \gamma_{k_i}$ 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 primitive 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 $O_{\gamma_1}, \ldots, O_{\gamma_{k_i}}$ and define the function $g_i : W \to \mathcal{P}(O_i)$ by $g_i(w) := O_{\gamma(w)}$ where $\gamma(w)$ is the $\mathcal{K}$-equivalence class of w. Let $V' := V \cup \bigcup_{0 < i \leq n} O_i$ and define $g : W \to \mathcal{P}(V')$ by

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

Let $\overline{V'}$ be the set of atomic propositions and their negations from V'. Now let

$$\theta_M := \bigwedge \left\{ \bigvee Q \mid Q \subseteq \overline{V'} \text{ and } g(w) \models \bigvee Q \text{ for all } w \in W \right\}$$

Finally, let $\mathcal{F}(M) := (V', \theta_M, O_1, \ldots, O_n)$.

Theorem 17. *For any finite S5 pointed Kripke model (M, w) and every formula φ, we have that $(M, w) \models \varphi$ iff $(\mathcal{F}(M), g(w)) \models \varphi$.*

```
kripkeToKns :: PointedModel -> Scenario
kripkeToKns (KrM worlds rel val, cur) = (KnS ps theta obs, curs) where
  v          = map fst $ apply val cur
  ags        = map fst rel
  newstart   = fromEnum $ freshp v -- start counting new propositions here
  amount i   = ceiling (logBase 2 (fromIntegral $ length (apply rel i)) :: Float) -- |O_i|
  newpstep   = maximum [ amount i | i <- ags ]
  newps i    = map (\k -> P (newpstart + (newpstep*i) +k)) [0..(amount i - 1)] -- O_i
  copyrel i  = zip (apply rel i) (powerset (newps i)) -- label equiv.classes with P(O_i)
  gag i w    = snd $ head $ filter (\(ws,_) -> elem w ws) (copyrel i)
  g w        = filter (apply (apply val w)) v ++ concat [ gag i w | i <- ags ]
  fWith set  = [ Disj [ if p `elem` psubset then PrpF p else Neg $ PrpF p | p <- set ] |
       psubset <- powerset set ] -- build a finite state law
  formulas   = concat [ fWith set | set <- powerset ps ]
  glotrue f  = and [ boolEval (g w) f | w<-worlds ]
  ps         = v ++ concat [ newps i | i <- ags ]
  theta      = boolBddOf $ Conj (filter glotrue formulas) -- convert state law to a BDD
  obs        = [ (i,newps i) | i<- ags ]
  curs       = sort $ g cur
```

4.3 From Action Models to Knowledge Transformers

For any S5 action model there is an equivalent knowledge transformer and vice versa. The translations are similar to Definitions 14 and 16 and their soundness also follows from Lemma 13. The implementation below works on pointed models, to simplify tracking the actual world and action.

Definition 18. *The function* Trf *maps an S5 action model* $\mathcal{A} = (A, (\mathcal{K}_i)_{i \in I}, \mathsf{pre})$ *to a transformer as follows. For any two sets of propositions A and B, we abbreviate "of the propositions in B, exactly those in A are true", i.e. let $A \sqsubseteq B := \bigwedge A \wedge \bigwedge \{\neg p \mid p \in B \setminus A\}$. Let P be a finite set of fresh propositions such that there is an injective $g : A \to \mathcal{P}(P)$ and let*

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

Now, for each i: Write $A/\mathcal{K}_i$ for the set of equivalence classes induced by $\mathcal{K}_i$. Let O_i^+ be a finite set of fresh propositions such that there is an injective $g_i : A/\mathcal{K}_i \to \mathcal{P}(O_i^+)$ and let

$$\Phi_i := \left\{ (g_i(\alpha) \sqsubseteq O_i) \to \left(\bigvee_{a \in \alpha} (g(a) \sqsubseteq P) \right) \mid \alpha \in A/\mathcal{K}_i \right\}$$

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

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

where g and $g_{\mathcal{A}}$ are from the construction of $\mathcal{F}(\mathcal{M})$ and $\mathsf{Trf}(\mathcal{A})$, respectively.

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

4.4   From Knowledge Transformers to Action Models

Definition 20. *The function Act maps a given Knowledge Transformer $\mathcal{X} = (V^+, \theta^+, O_1^+, \ldots, O_n^+)$ to an action model 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 $\mathsf{pre}(\alpha) := \theta^+ \left(\frac{\alpha}{\top} \right) \left(\frac{V^+ \backslash \alpha}{\bot} \right)$. Finally, let $\mathsf{Act}(\mathcal{X}) := (A, (R_i)_{i \in I}, \mathsf{pre})$.*

Theorem 21. *For any scene $(\mathcal{F}, s)$, any event $(\mathcal{X}, x)$ and any formula φ over the vocabulary of $\mathcal{F}$ we have:*

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

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

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

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

5 Automated Testing

This module provides automated randomized testing to check our implementations for correctness. 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 TEST where
import System.Random (getStdRandom,randomR)
import DELLANG
import KNSCAC
import KRIPKEDEL
import SYMDEL
import EXAMPLES
```

Some global settings for the tests in this section:

```
mypropsindex, myagsindex, mycomplexity, myconlength :: Int
mypropsindex = 2 -- maximum index of atomic props
myagsindex   = 2 -- maximum index of agents
mycomplexity = 4 -- maximum complexity of formulas
myconlength  = 2 -- maximum number of conjuncts
```

5.1 Generating random formulas

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))

getRandomGroup :: IO [Agent]
getRandomGroup = do
  n <- getRandomInt 2
  case n of 0 -> return [alice]
            1 -> return [bob]
            _ -> return [alice,bob]

getRandomF :: IO Form
getRandomF = do d <- getRandomInt mycomplexity
                getRandomForm d

getRandomForm :: Int -> IO Form
getRandomForm 0 = do
  n <- getRandomInt 4
  case n of 0 -> return Top
            1 -> return Bot
            _ -> do m <- getRandomInt mypropsindex
                    return (PrpF (P m))

getRandomForm d = do
  n <- getRandomInt 9
  case n of
    0 -> do m <- getRandomInt mypropsindex
            return (PrpF (P m))
    1 -> do f <- getRandomForm (d-1)
            return (Neg f)
    2 -> do f <- getRandomForm (d-1)
            g <- getRandomForm (d-1)
            return (Impl f g)
    3 -> do m  <- getRandomInt myconlength
            fs <- getRandomForms (d-1) m
            return (Conj fs)
    4 -> do m  <- getRandomInt myconlength
            fs <- getRandomForms (d-1) m
            return (Disj fs)
    5 -> do i <- getRandomInt myagsindex
            f <- getRandomForm (d-1)
            return (K i f)
    6 -> do i <- getRandomInt myagsindex
```

```
                    f <- getRandomForm (d-1)
                    return (Kw i f)
        7 -> do ags <- getRandomGroup
                    f <- getRandomForm (d-1)
                    return (Ck ags f)
        8 -> do ags <- getRandomGroup
                    f1 <- getRandomForm (d-1)
                    f2 <- getRandomForm (d-1)
                    return (Announce ags f1 f2)
        _ -> do f1 <- getRandomForm (d-1)
                    f2 <- getRandomForm (d-1)
                    return (PubAnnounce f1 f2)

getRandomForms :: Int -> Int -> IO [Form]
getRandomForms _ 0 = return []
getRandomForms d n = do f <- getRandomForm d
                        fs <- getRandomForms d (n-1)
                        return (f:fs)
```

5.2   Testing equivalence of the two semantics

The following creates a Kripke model and a knowledge structure which are equivalent to each other by
Lemma 13. In this model/structure Alice knows everything and the other agents do not know anything.
The function `test` checks for a given number of random formulas whether the implementations of the
different semantics and translation methods agree on whether the formula holds on the model and the
structure.

```
mymodel :: PointedModel
mymodel = (KrM ws rel (zip ws table), 0) where
  ws    = [0..(2^(mypropsindex+1)-1)]
  rel   = (alice, map (:[]) ws) : [ (i,[ws]) | i <- [1..myagsindex] ]
  table = foldl buildTable [[]] [P k | k<- [0..mypropsindex]]
  buildTable partrows p = [ (p,v):pr | v <-[True,False], pr<-partrows ]

myscn :: Scenario
myscn = (KnS ps (boolBddOf Top) ((alice,ps):[(i,[]) | i<-[1..myagsindex]]) , ps)
  where ps = [P 0 .. P mypropsindex]

singleTest :: IO (Bool, Bool)
singleTest = do
  f <- getRandomF
  -- print f -- uncomment this to show formulas while testing.
  singleTestWith f

singleTestWith :: Form -> IO (Bool, Bool)
singleTestWith f = do
  let a = KRIPKEDEL.eval mymodel f              -- evaluate directly on Kripke
  let b = KNSCAC.eval myscn f                   -- evaluate directly on KNS
  let c = KNSCAC.evalViaBdd myscn f             -- evaluate boolean equivalent on KNS
  let d = KRIPKEDEL.eval (knsToKripke myscn) f  -- evaluate on corresponding Kripke
  let e = KNSCAC.eval  (kripkeToKns mymodel) f  -- evaluate on corresponding KNS
  if or [a/=b,b/=c,c/=d,d/=e]
    then do putStr $ "Problem: " ++ show f ++ "\n          " ++ show (a,b,c,d,e) ++"\n\n"
            return (True,a)
    else    return (False,a)

test :: Int -> IO ()
test n = do (problems,truths) <- testLoop 0 0 n
            putStrLn $ show problems ++ " problems among " ++ show n ++ " formulas out of
                  which " ++ show truths ++" were true."

testLoop :: Int -> Int -> Int -> IO (Int,Int)
testLoop p t 0 = return (p,t)
testLoop p t n = do (prob,res) <- singleTest
                    testLoop (if prob then p + 1 else p) (if res then t + 1 else t) (n-1)
```

5.3 Public Announcements

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

```haskell
pubAnnounceTest :: IO Bool
pubAnnounceTest = do
  n <- getRandomInt mypropsindex
  let f = PrpF (P n)
  g <- getRandomF
  print (PubAnnounce f g)
  let a = KRIPKEDEL.eval mymodel (PubAnnounce f g)
  putStr $ show a
  let b = KNSCAC.eval (kripkeToKns mymodel) (PubAnnounce f g)
  putStr $ show b
  let c = KNSCAC.eval (knowTransform (kripkeToKns mymodel) (actionToEvent (
      pubAnnounceAction [0,1] f))) g
  print c
  if a/=b || b/=c
    then do putStr $ "Problem: " ++ show g ++ "\n         "++ show (a,b,c) ++"\n\n"
            return False
    else    return True
```

5.4 Random Action Models

This generates a random action model with four actions. To ensure that it is compatible with all models the actual action token has ⊤ 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.

```haskell
getRandomAction :: IO PointedActionModel
getRandomAction = do
  [f,g,h] <- getRandomForms 2 3
  return (ActM [0..3] [(0,Top),(1,f),(2,g),(3,h)]
    ( (0,[[0],[1],[2],[3]]):[(k,[[0..3]]) | k<-[1..myagsindex] ]), 0)

singleActionTest :: IO Bool
singleActionTest = do
  myact <- getRandomAction
  f <- getRandomForm 3
  let a = KRIPKEDEL.eval (productUpdate mymodel myact) f
  let b = KNSCAC.evalViaBdd (knowTransform (kripkeToKns mymodel) (actionToEvent myact)) f
  if a /= b
    then do putStr $ "Problem: " ++ show myact
                  ++ "\n action: " ++ show (actionToEvent myact)
                  ++ "\n   form: " ++ show f
                  ++ "\n    res: " ++ show (a,b) ++ "\n\n"
            return True
    else return False

actionTest :: Int -> IO ()
actionTest n = do
  problems <- actionTestLoop 0 n
  putStrLn $ show problems ++ " problems among " ++ show n ++ " formula/action pairs."

actionTestLoop :: Int -> Int -> IO Int
actionTestLoop p 0 = return p
actionTestLoop p n = do
  problem <- singleActionTest
  actionTestLoop (if problem then p+1 else p) (n-1)
```

6 Examples

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

```
module EXAMPLES where
import Data.List (delete,intersect,(\\))
import Data.Maybe (fromJust)
import DELLANG
import KNSCAC
import KRIPKEDEL
import SYMDEL
```

6.1 Knowledge and Meta-Knowledge

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

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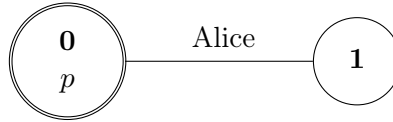

Figure 1: `modelA`

```
>>> map (KRIPKEDEL.eval modelA) [ K bob (PrpF (P 0)), K alice (PrpF (P 0)) ]
[True,False]
0.98  seconds
```

```
>>> KRIPKEDEL.eval modelA (K alice (Kw bob (PrpF (P 0))))
True
0.96  seconds
```

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

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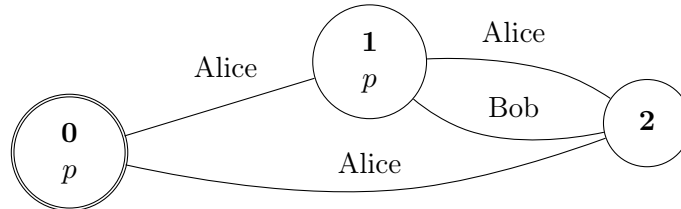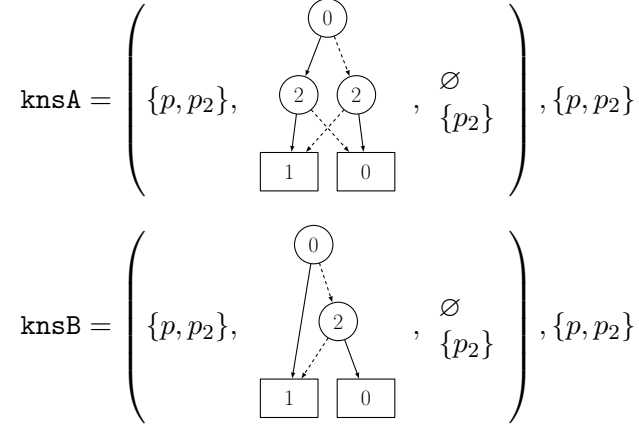

Figure 2: `modelB`

```
>>> KRIPKEDEL.eval modelB (K bob (PrpF (P 0)))
True
0.93  seconds
```

```
>>> KRIPKEDEL.eval modelB (Kw alice (Kw bob (PrpF (P 0))))
False
0.94  seconds
```

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

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

$$\texttt{knsA} = \left(\{p, p_2\}, \quad \begin{array}{c} \text{BDD graph} \end{array} \quad , \quad \begin{array}{c} \varnothing \\ \{p_2\} \end{array} \right), \{p, p_2\}$$

$$\texttt{knsB} = \left(\{p, p_2\}, \quad \begin{array}{c} \text{BDD graph} \end{array} \quad , \quad \begin{array}{c} \varnothing \\ \{p_2\} \end{array} \right), \{p, p_2\}$$

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],[]]
1.11  seconds
```

```
>>> let (structB,foo) = knsB in statesOf structB
[[P 0],[P 0,P 2],[]]
1.08  seconds
```

6.2 Minimization via Translation

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

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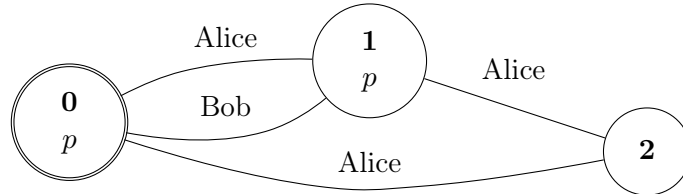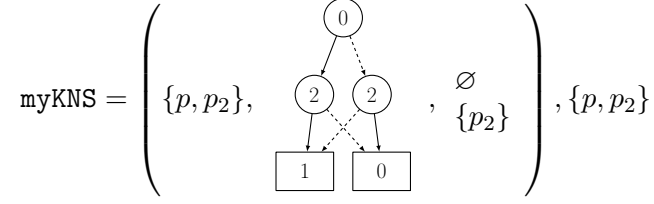

Figure 3: `redundantModel`

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

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

$$
\texttt{myKNS} = \left(\{p, p_2\}, \quad \vcenter{\hbox{\includegraphics}} \quad , \quad \begin{array}{c} \varnothing \\ \{p_2\} \end{array} \right), \{p, p_2\}
$$

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

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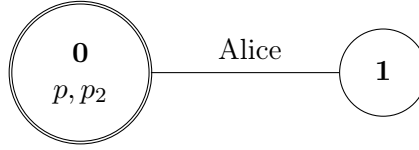

Figure 4: `minimizedModel`

6.3   Different Announcements

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

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

examplePaAction :: PointedActionModel
examplePaAction = pubAnnounceAction [0,1] (PrpF (P 0))
```

```
>>> examplePaAction
(ActM [0] [(0,PrpF (P 0))] [(0,[[0]]),(1,[[0]])],0)
0.95   seconds
```

```
>>> actionToEvent examplePaAction
(KnT [] (PrpF (P 0)) [(0,[]),(1,[])],[])
0.97   seconds
```

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

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

exampleGroupAnnounceAction :: PointedActionModel
exampleGroupAnnounceAction = groupAnnounceAction [0,1] [0] (PrpF (P 0))
```

```
>>> exampleGroupAnnounceAction
(ActM [0,1] [(0,PrpF (P 0)),(1,Top)] [(0,[[0],[1]]),(1,[[0,1]])],0)
0.93  seconds
```

```
>>> actionToEvent exampleGroupAnnounceAction
(KnT [P 1,P 2] (Conj [Impl (PrpF (P 1)) (PrpF (P 0)),Impl (PrpF (P 2)) (PrpF (P
1)),Impl (Neg (PrpF (P 2))) (Neg (PrpF (P 1))),Disj [PrpF (P 1),Neg (PrpF (P 1))]])
[(0,[P 2]),(1,[])],[P 1,P 2])
0.89  seconds
```

But it is not hard to check that this is equivalent to the definition. Consider the θ^+ formula of this transformer, namely $\bigwedge\{p_1 \to p_1, p_2 \to p_1, \neg p_2 \to \neg p_1, p_1 \lor \neg p_1\}$. This is equivalent to $p_1 \leftrightarrow p_2$ and the actual event is given by both p_1 and p_2 being added to the current state, equivalent to the normal announcement. There is no canonical way to avoid such redundancy as long as we use the general two-step process in Definition 18 to translate action models to knowledge transformers.

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

```
>>> eventToAction (actionToEvent exampleGroupAnnounceAction)
(ActM [0,3] [(0,PrpF (P 0)),(3,Top)] [(0,[[3],[0]]),(1,[[0,3]])],0)
0.90  seconds
```

6.4   Muddy Children

We now model the story of the muddy children which is known in many versions. See for example [Lit53], [FHMV95, 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 -> Scenario
mudScnInit n m = ( KnS mudProps (boolBddOf Top) [ (i,delete (P i) mudProps) | i <- [1..n]
    ], [P 1 .. P m] ) where mudProps = [P 1 .. P n]

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

$$\texttt{myMudScnInit} = \left(\{p_1, p_2, p_3\}, \boxed{1}, \begin{array}{c} \{p_2, p_3\} \\ \{p_1, p_3\} \\ \{p_1, p_2\} \end{array} \right), \{p_1, p_2, p_3\}$$

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 i (PrpF $ P i)

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

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

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

25

$$\texttt{mudScn0} = \left(\{p_1,p_2,p_3\}, \quad \begin{array}{c} \text{(graph)} \end{array}, \quad \begin{array}{c} \{p_2,p_3\} \\ \{p_1,p_3\} \\ \{p_1,p_2\} \end{array} \right), \{p_1,p_2,p_3\}$$


```
>>> evalViaBdd mudScn0 (nobodyknows 3)
True
1.07  seconds
```

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

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

$$\texttt{mudScn1} = \left(\{p_1,p_2,p_3\}, \quad \begin{array}{c} \text{(graph)} \end{array}, \quad \begin{array}{c} \{p_2,p_3\} \\ \{p_1,p_3\} \\ \{p_1,p_2\} \end{array} \right), \{p_1,p_2,p_3\}$$


```
>>> evalViaBdd mudScn1 (nobodyknows 3)
True
1.06  seconds
```

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

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

$$\texttt{mudScn2} = \left(\{p_1,p_2,p_3\}, \quad \begin{array}{c} \text{(graph)} \end{array}, \quad \begin{array}{c} \{p_2,p_3\} \\ \{p_1,p_3\} \\ \{p_1,p_2\} \end{array} \right), \{p_1,p_2,p_3\}$$


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

```
>>> KNSCAC.statesOf mudKns2
[[P 1,P 2,P 3]]
1.05  seconds
```

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

6.5 Drinking Logicians

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

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

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

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

myThirstyScene :: Scenario
myThirstyScene = thirstyScene 3
```

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

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:

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

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

thirstyCheck :: Int -> Bool
thirstyCheck n = evalViaBdd (thirstyScene n) (thirstyF n)
```
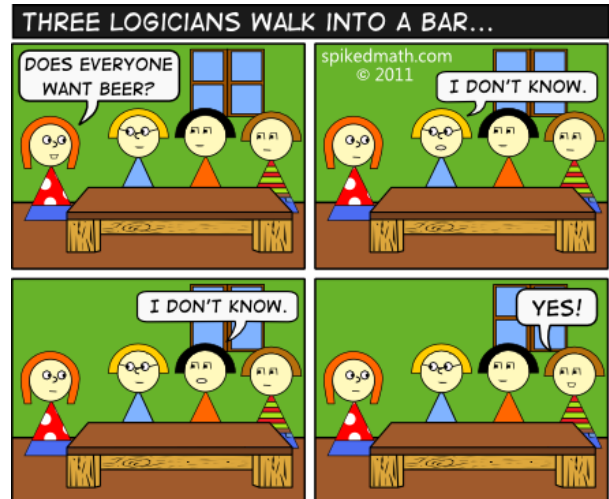
```
>>> thirstyCheck 3
True
1.11  seconds
```

```
>>> thirstyCheck 10
True
1.09  seconds
```

```
>>> thirstyCheck 100
True
1.28  seconds
```

```
>>> thirstyCheck 200
True
1.95  seconds
```

```
>>> thirstyCheck 400
True
5.56  seconds
```


http://spikedmath.com/445.html

27

6.6 Dining Cryptographers

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 use the following protocol: For every pair of cryptographers a coin is flipped in such a way that only those two see the result. Then they announce whether the two coins they saw were different or the same. But, there is an exception: If one of them paid, then this person says the opposite. After these announcements are made, the cryptographers can infer that the NSA paid iff the number of people saying that they saw the same result on both coins is even.

The following function generates a knowledge structure to model this story. Given an index 0, 1, 2, or 3 for who paid and three boolean values for the random coins we get the corresponding scenario.
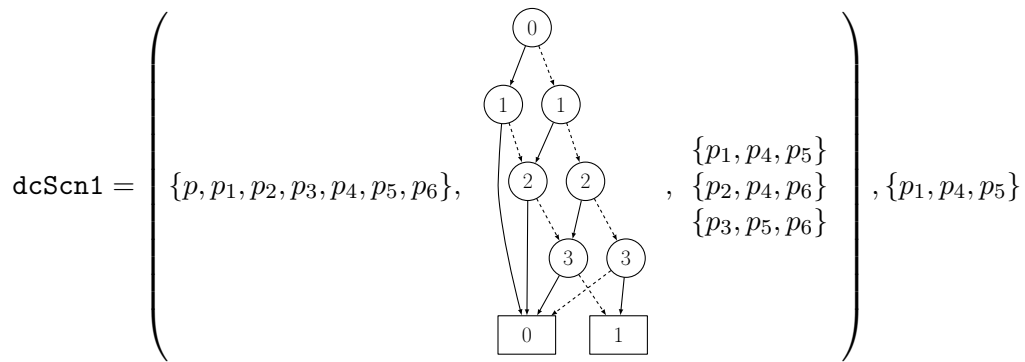
```
dcScnInit :: Int -> (Bool,Bool,Bool) -> Scenario
dcScnInit payer (b1,b2,b3) = ( KnS props law obs , truths ) where
  props = [ P 0    -- The NSA paid
          , P 1    -- Alice paid
          , P 2    -- Bob paid
          , P 3    -- Charlie paid
          , P 4    -- shared bit of Alice and Bob
          , P 5    -- shared bit of Alice and Charlie
          , P 6 ] -- shared bit of Bob and Charlie
  law   = boolBddOf $ Conj [ someonepaid , notwopaid ]
  obs   = [ (1,[P 1, P 4, P 5])
          , (2,[P 2, P 4, P 6])
          , (3,[P 3, P 5, P 6]) ]
  truths = [ P payer ] ++ [ P 4 | b1 ] ++ [ P 5 | b2 ] ++ [ P 6 | b3 ]

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

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

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

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

$$
\text{dcScn1} = \left(\{p, p_1, p_2, p_3, p_4, p_5, p_6\}, \quad \begin{matrix} \text{(graph)} \end{matrix}, \quad \begin{matrix} \{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\} \end{matrix} \right), \{p_1, p_4, p_5\}
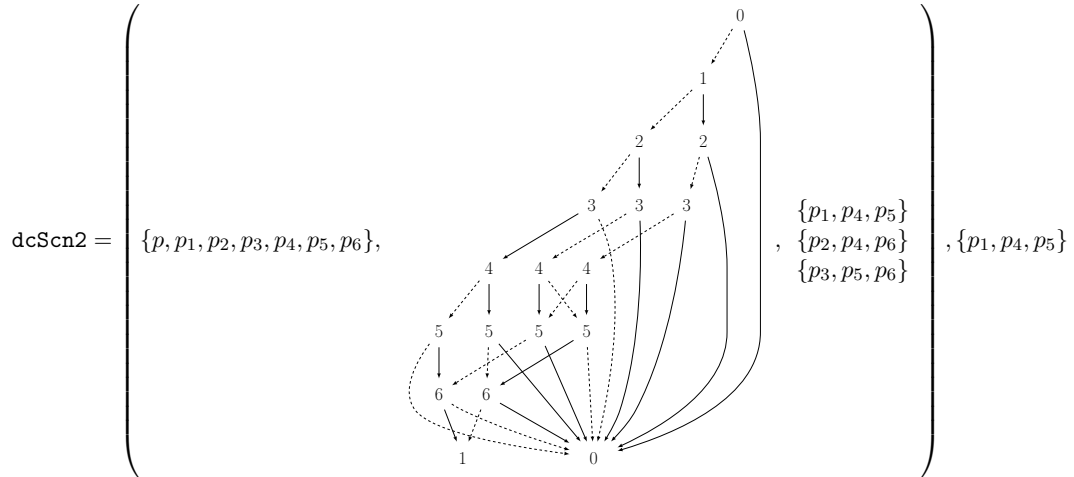$$

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

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

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

Now these three facts are announced:

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


$$\text{dcScn2} = \left(\{p, p_1, p_2, p_3, p_4, p_5, p_6\}, \quad , \begin{matrix} \{p_1, p_4, p_5\} \\ \{p_2, p_4, p_6\} \\ \{p_3, p_5, p_6\} \end{matrix}, \{p_1, p_4, p_5\} \right)$$

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

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

```
>>> evalViaBdd dcScn2 everyoneKnowsWhetherNSApaid
True
1.09  seconds
```

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

```
>>> evalViaBdd dcScn2 (K 1 (PrpF (P 1)))
True
1.07  seconds
```

```
>>> evalViaBdd dcScn2 (K 2 (PrpF (P 1)))
False
1.09  seconds
```

```
>>> evalViaBdd dcScn2 (K 3 (PrpF (P 1)))
False
1.12  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 dcScn1 dcCheckForm
True
1.13  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:
```
>>> dcValid
True
1.13  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 it for a benchmark in Section 7.2. Note that we need $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ many shared bits. This distinguishes the Dining Cryptographers from the Muddy Children and the Drinking Logicians example where the number of propositions needed to model the situation was just the number of agents.

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

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

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

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

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

genNobodyknowsWhoPaid :: Int -> Form
genNobodyknowsWhoPaid n =
  Conj [ Impl (PrpF (P i)) (Conj [Neg $ K k (PrpF $ P i) | k <- delete i [1..n] ]) | i <-
      [1..n] ]

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

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

For example, we can check the protocol for 4 dining cryptographers.
```
>>> genDcValid 4
True
1.07  seconds
```

6.7 Russian Cards

As a second case study we analyze the Russian Cards problem. One of its first logical treatments is [Dit03] and the problem has since gained notable attention as an intuitive example of information-theoretically (in contrast to computationally) secure cryptography [CFDFDST15, DG14].

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

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

```
rcPlayers, rcCards :: [Int]
rcPlayers = [alice,bob,carol]
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 + i))
```

```
>>> carol `hasCard` 5
PrpF (P 17)
0.95  seconds
```

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

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

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

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

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

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

The initial knowledge structure for Russian Cards looks as follows. The BDD describing the state law is generated within less than a second but drawing it is more complicated and the result quite huge:
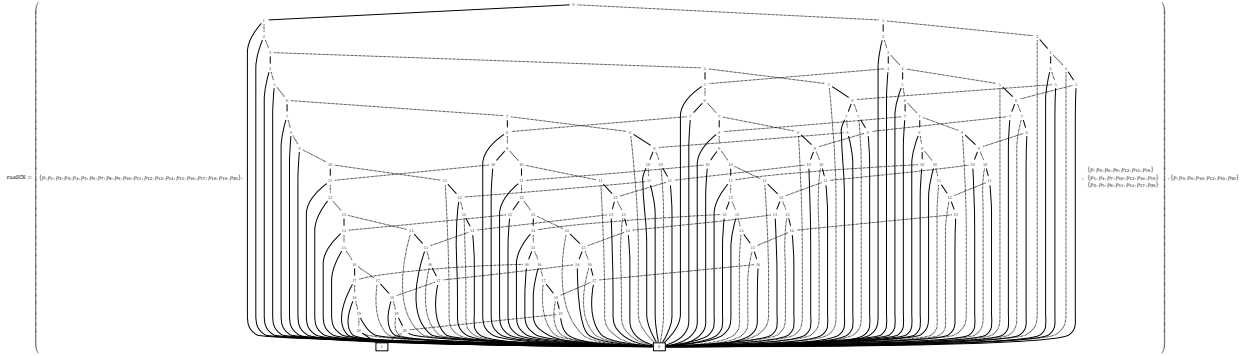
Many different solutions for Russian Cards exist. Here we will focus on so-called five-hands protocols (and their extensions with six or seven hands) which are also used in [DHMR06]: First Alice makes an announcement of the form "My hand is one of these: ...". If her hand is 012 she could for example take the set $\{012, 034, 056, 135, 146, 236\}$. It can be checked that this announcement does not tell Carol anything, independent of which card it has. In contrast, Bob will be able to rule out all but one of the hands in the list because of his own hand. Hence the second and last step of the protocol is that Bob says which card Carol has. For example, if Bob's hand is 345 he would finish the protocol with "Carol has card 6.".

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

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

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

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

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

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

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

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

```
rcCheck 1 = PubAnnounce aAnnounce bKnowsAs
rcCheck 2 = PubAnnounce aAnnounce (Ck [0,1] bKnowsAs)
```

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

```
rcCheck 3 = PubAnnounce aAnnounce (K 1 (PrpF (P 20)))
```

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

```
rcCheck 4 = PubAnnounce aAnnounce (Ck [0,1,2] 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 [0,1] aKnowsBs))
rcCheck 6 = PubAnnounce aAnnounce (PubAnnounce bAnnounce (Ck [0,1] bKnowsAs))
rcCheck _ = PubAnnounce aAnnounce (PubAnnounce bAnnounce (Ck [0,1,2] 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 [DHMR06] needs 4 seconds to check one protocol.

```
>>> EXAMPLES.rcAllChecks
True
1.05  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 [0,1] $ Conj [aKnowsBs, bKnowsAs]))
       , PubAnnounce aliceSays (PubAnnounce bobSays (Ck [0,1,2] 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 allHandLists
1290
1.10  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 checkSet allHandLists)
102
2.60  seconds
```

7 Benchmarks

We now provide two different benchmarks for SMCDEL. All measurements were done under 64-bit Debian GNU/Linux 8.0 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 7.8.3 and g++ 4.9.2.

7.1 Muddy Children

In this section we compare the performance of different model checking approaches to the muddy children example from Section 6.4.

- SMCDEL with different BDD packages: CacBDD, CUDD, ROBBed and NooBdd.

- 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 43.

```
module Main (main) where
import Control.Monad
import Data.List
import Data.Time (getCurrentTime, NominalDiffTime, diffUTCTime)
import System.Environment (getArgs)
import System.IO (stdout, hSetBuffering, BufferMode(NoBuffering))
import DELLANG
import EXAMPLES
import qualified DEMO_S5
import qualified KNSCAC
import qualified KNSCUDD
import qualified KNSROB
import qualified KNSNOO
import qualified MCTRIANGLE
```

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 = loop 0 where
  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,KNSCAC.evalViaBdd) where
  cacMudScnInit n m = ( KNSCAC.KnS (mudPs n) (KNSCAC.boolBddOf Top) [ (i,delete (P i) (
      mudPs n)) | i <- [1..n] ], [P 1 .. P m] )

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

findNumberRobBdd :: Int -> Int -> Int
```

```
findNumberRobBdd = findNumberWith (robMudScnInit,KNSROB.evalViaBdd) where
  robMudScnInit n m = ( KNSROB.KnS (mudPs n) (KNSROB.boolBddOf Top) [ (i,delete (P i) (
      mudPs n)) | i <- [1..n] ], [P 1 .. P m] )

findNumberNooBdd :: Int -> Int -> Int
findNumberNooBdd = findNumberWith (nooMudScnInit,KNSNOO.evalViaBdd) where
  nooMudScnInit n m = ( KNSNOO.KnS (mudPs n) (KNSNOO.boolBddOf Top) [ (i,delete (P i) (
      mudPs n)) | i <- [1..n] ], [P 1 .. P 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]

findNumberDemo :: Int -> Int -> Int
findNumberDemo n m = findNumberDemoLoop n m 0 start where
  start = DEMO_S5.upd_pa (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.upd_pa curMod (DEMO_S5.dont n))
    else count
```

Also the number triangle approach to the Muddy Children puzzle has to be treated separately. See [GS11] and Appendix 1 on page 43 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 = MCTRIANGLE.update (MCTRIANGLE.mcModel (n-m,m)) (MCTRIANGLE.Qf MCTRIANGLE.some)

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

The following functions loop over all the solution methods we defined and generate a table of timing results. The resulting program takes a maximum runtime as a parameter. If a solution method takes longer than this limit then it will not be used for the following instances of the problem with a higher number of agents.

```
timeWith :: Int -> Int -> (Int -> Int -> Int) -> IO NominalDiffTime
timeWith n m function = do
  start <- getCurrentTime
  if function n m == (m - 1)
    then do end <- getCurrentTime
            return (end `diffUTCTime` start)
    else error "Wrong result."

mainLoop :: [(Bool, Int -> Int -> Int)] -> [Int] -> Int -> IO ()
mainLoop _  [] _ = putStrLn ""
mainLoop fs (n:ns) limit = do
  putStr $ show n ++ "\t"
  results <- mapM (\(bit,f) ->
    if bit then do
      result <- timeWith n n f
      putStr $ init (show result) ++ replicate (9 - length (show result)) '0' ++ "\t"
      return result
    else do
```

```
        putStr "nan       \t"
        return 0
    ) fs
  putStrLn ""
  let newfs = map (\((bit,f),t) -> (bit && (t<= fromIntegral limit), f)) (zip fs results)
  when (any fst newfs) $ mainLoop newfs ns limit

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStrLn $ "Initializing CacBDD: 40==" ++ show (findNumberCacBdd 41 41)
  putStrLn $ "Initializing CUDD: 40==" ++ show (findNumberCUDD 41 41)
  putStrLn $ "n\t" ++ concatMap (++ "\t") ["TRIANGLE","KNSCAC  ","KNSCUDD "]-- "KNSROB  ","
      KNSNOO  ","DEMO-S5 "]
  let allfs = [findNumberTriangle, findNumberCacBdd, findNumberCUDD]-- findNumberRobBdd,
      findNumberNooBdd, findNumberDemo]
  args <- getArgs
  case args of
    [aInteger] | [(n,_)] <- reads aInteger ->
      mainLoop (zip (repeat True) allfs) ([3..40]++[50,60,70,80,90,100]) n
    _ -> error "Please give a maximum runtime as an argument."
```
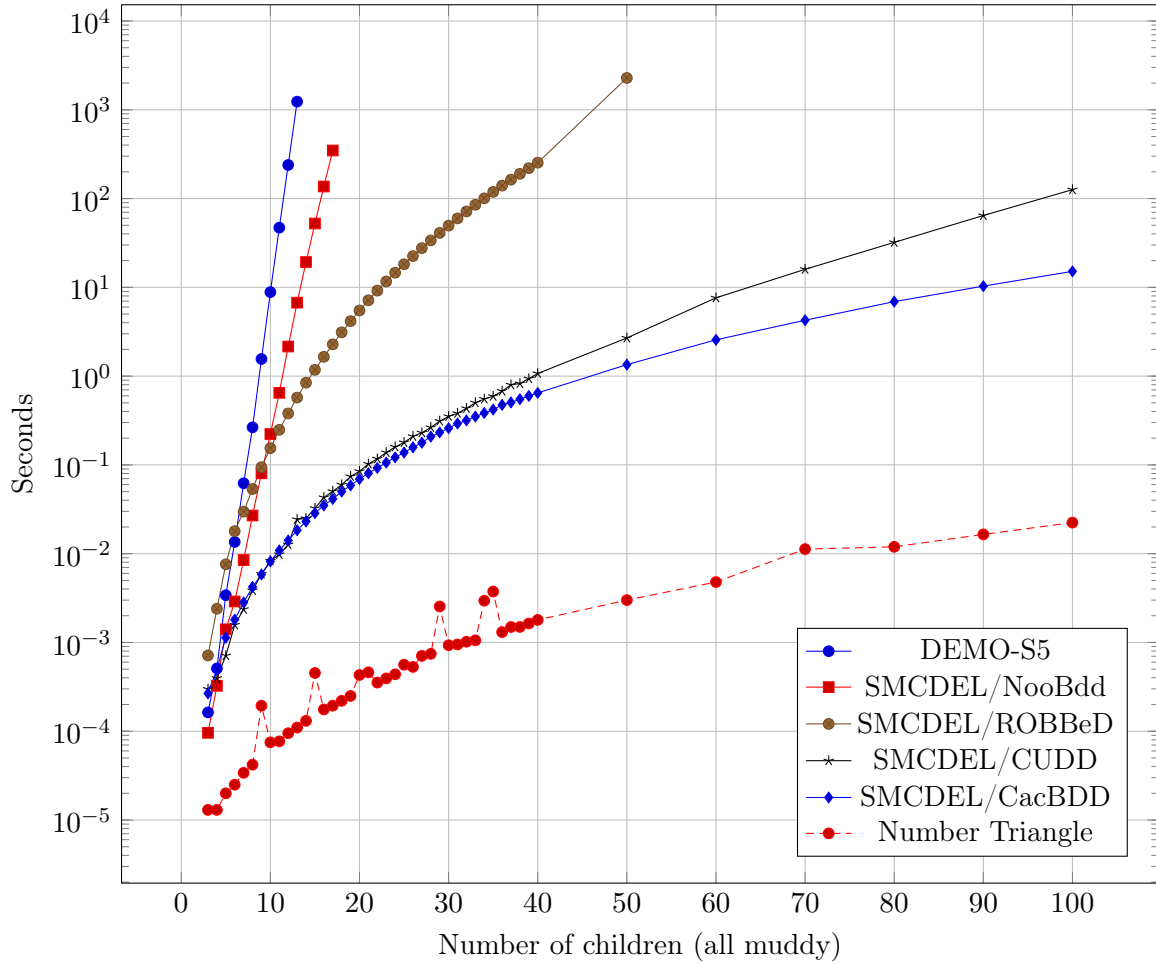

Figure 5: Benchmark Results on a logarithmic scale.

As expected we can see in Figure 5 that *SMCDEL* is faster than the explicit model checker DEMO. We can also see that the choice of the BDD package affects the performance. With the naive package NooBdd (which does not not find and identify isomorphic subtrees etc.) our model checker is almost as slow as DEMO-S5. Somewhat better but still becoming slow above 40 agents is the ROBBeD package written in pure Haskell. The two highly developed packages with elaborate memory management give

us the best performance for *SMCDEL*, with a slightly better performance of CacBDD compared to CUDD. It is important to note that this difference and the performance in general also depends on the binding libraries we use. Especially concerning memory management and garbage collection there should be room for improvement.

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.

7.2 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 [BGHP09, DHR13].

A protocol which fits nicely into both frameworks are the Dining Cryptographers [Cha88] which we implemented in Section 6.6. We will now use it to measure the performance of *SMCDEL* in a way that is more similar to [LQR15].

```
module Main (main) where
import Control.Monad (when)
import Data.Time (diffUTCTime,getCurrentTime,NominalDiffTime)
import System.Environment (getArgs)
import System.IO (hSetBuffering,BufferMode(NoBuffering),stdout)
import DELLANG
import KNSCAC
import EXAMPLES (genDcKnsInit,genDcReveal)
```

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 and conventions from [BGHP09, DHR13] we can formalize it in DEL as

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

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

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

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

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

```
    putStr $ show (length $ show mykns) ++ "\t"
    putStr $ show (length $ show $ genDcCheckForm n) ++ "\t"
    if genDcValid n then do
      end <- getCurrentTime
      return (end `diffUTCTime` start)
    else
      error "Wrong result."

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

main :: IO ()
main = do
  args <- getArgs
  hSetBuffering stdout NoBuffering
  case args of
    [aInteger] | [(n,_)] <- reads aInteger -> do
      putStrLn $ "n" ++ "\tn(prps)"++ "\tsz(KNS)"++ "\tsz(frm)" ++ "\ttime"
      mainLoop (3:(5 : map (10*) [1..])) n
    _ -> error "Please give a maximum runtime as an argument."
```

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

```
n       n(prps)  sz(KNS)  sz(frm)  time
3       7        211      331      0.142654s
5       16       473      633      0.000622s
10      56       1634     1825     0.001898s
20      211      6457     6247     0.009982s
30      466      14512    13357    0.031843s
40      821      25667    23067    0.079768s
50      1276     40750    35929    0.179073s
60      1831     59770    51949    0.334907s
70      2486     82190    70769    0.571617s
80      3241     108010   92389    0.891703s
90      4096     137230   116809   1.3617s
100     5051     169951   144031   1.839588s
110     6106     207036   174071   2.605375s
120     7261     247621   206911   3.328267s
130     8516     291706   242551   4.266749s
140     9871     339291   280991   5.65589s
150     11326    394354   324883   6.653351s
160     12881    453604   372033   8.139113s
170     14536    516654   422183   10.38303s
```

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 case of up to 160 agents in less time.

8 Future Work

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

Non-S5 Models

Currently *SMCDEL* can only work on models where the epistemic accessibility relation is an equivalence relation. This is because only those can be described by sets of observational variables. And in fact not even every S5 relation on distinctly valuated worlds can be modeled with observational variables – this is why our translation procedure in Definition 16 has to use additional atomic propositions.

To overcome this limitation, we will generalize the definition of knowledge structures. Using well-known methods from temporal model checking, arbitrary relations can also be represented as BDDs. 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.

Increase Usability

Concerning the usability of *SMCDEL*, two desiderata come to mind. First, 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.

Second, our model checker currently is only usable as a Haskell module. But ideally, the user should not have to know Haskell and only basic knowledge about DEL should be required to use it. This can be achieved with a stand-alone executable of the model checker that reads a simple text-file containing the description of a model and one or more formulas to be checked. As a first step towards this goal we will write a parser for human-readable DEL formulas, similar to one already used in the online model checker for Epistemic Crypto Logic (`https://is.gd/eclonline`).

SAT Solving

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

Abstraction and Modal Logic

Epistemic and temporal logics have been connected before and also concrete translation methods have been proposed, see [BGHP09, 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 will be to enable symbolic methods for Epistemic Crypto Logic [EG15]. Our methods could then also be used to analyze cryptographic protocols.

Appendix 1: Installation Guidelines

Currently SMCDEL is supported to run on the Haskell Platform 2014.2 under Linux. The following shell commands will install HasCacBDD (including CacBDD) and then SMCDEL.

```
git clone https://github.com/m4lvin/HasCacBDD.git
cd HasCacBDD
make all

git clone https://github.com/m4lvin/SMCDEL.git
cd SMCDEL
make
```

One can then run `ghci EXAMPLES` to explore the examples from Section 6:

```
*EXAMPLES> modelA
(KrM [0,1] [(0,[[0,1]]),(1,[[0],[1]])] [(0,[(P 0,True)]),(1,[(P 0,False)])],0)
*EXAMPLES> modelB
(KrM [0,1,2] [(0,[[0,1,2]]),(1,[[0],[1,2]])] [(0,[(P 0,True)]),(1,[(P 0,True)]),(2,[(P 0,
    False)])],0)
```

To use SMCDEL with other BDD packages these have to be installed first. The following git repositories include some patches and adjusted Makefiles for the other three BDD packages.

- `https://github.com/m4lvin/hBDD`

- `https://github.com/m4lvin/robbed`

- `https://github.com/m4lvin/NooBDD`

The modules KNSCUDD, KNSROB and KNSNOO can then be built with `make otherbdds` in the SMCDEL folder. More information to reproduce the benchmarks and other experimental modules can be found in the Makefile.

Appendix 2: Helper Functions

```
module HELP (alleq,apply,powerset,restrict,rtc,Erel,bl,fusion) where
import Data.List (nub,union,foldl',(\\))

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

type Erel a = [[a]]

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

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

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

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

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

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

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

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

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

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

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

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

fusion :: Ord a => [[a]] -> Erel a
fusion [] = []
fusion (b:bs) = let
   cs = filter (overlap b) bs
   xs = mergeL (b:cs)
   ds = filter (overlap xs) bs
 in
   if cs == ds
     then xs : fusion (bs \\ cs)
     else fusion (xs : bs)
```

Appendix 3: 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 on a Kripke frame with less worlds than needed in the classical analysis, namely $2n + 1$ instead of 2^n for n children.

```
module MCTRIANGLE where
```

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

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

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

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

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

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

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

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

```
type Quantifier = State -> Bool

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

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

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

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

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

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

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

The semantics for our minimal language are implemented as follows.

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

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

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

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

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

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

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

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

References

[BEGS15]   Johan van Benthem, Jan van Eijck, Malvin Gattinger, and Kaile Su. Symbolic Model Checking for Dynamic Epistemic Logic. Forthcoming., 2015.

[BGHP09]   Johan van Benthem, Jelle Gerbrandy, Tomohiro Hoshi, and Eric Pacuit. Merging frameworks for interaction. *Journal of Philosophical Logic*, 38(5):491–526, 2009.

[BMS98]   Alexandru Baltag, Lawrence S. Moss, and Slawomir Solecki. The logic of public announcements, common knowledge, and private suspicions. In I. Bilboa, editor, *TARK'98*, pages 43–56, 1998.

[CCG$^+$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*, pages 359–364. Springer, 2002.

[CFDFDST15] Andrés Cordón-Franco, Hans van Ditmarsch, David Fernández-Duque, and Fernando Soler-Toscano. A geometric protocol for cryptography with cards. *Designs, Codes and Cryptography*, 74(1):113–125, 2015.

[CGP99]   E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, USA, 1999.

[Cha88]   David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[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*, pages 1471–1479. IFAAMAS, 2015.

[DG14]   David Fernández Duque and Valentin Goranko. Secure aggregation of distributed information. *CoRR*, abs/1407.7582, 2014.

[DHK07]   Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic epistemic logic*, volume 1. Springer Heidelberg, 2007.

[DHMR06]   Hans van Ditmarsch, Wiebe van der Hoek, Ron van der Meyden, and Ji Ruan. Model Checking Russian Cards. *Electr. Notes Theor. Comput. Sci.*, 149(2):105–123, 2006.

[DHR13]   Hans van Ditmarsch, Wiebe van der Hoek, and Ji Ruan. Connecting dynamic epistemic and temporal epistemic logics. *Logic Journal of IGPL*, 21(3):380–403, 2013.

[Dit03]   Hans van Ditmarsch. The russian cards problem. *Studia Logica*, 75(1):31–62, 2003.

[EG15]   Jan van Eijck and Malvin Gattinger. Elements of epistemic crypto logic (extended abstract). In *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)*. IFAAMAS, 2015.

[Eij07]   Jan van Eijck. DEMO—a demo of epistemic modelling. In *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London*, volume 1, pages 303–362, 2007.

[Eij14]   Jan van Eijck. DEMO-S5. Technical report, CWI, 2014.

[FHMV95]   Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 1995.

[Gam14]   Peter Gammie. hbdd. `https://github.com/peteg/hBDD`, 2011, updated 2014.

[Gat14]   Malvin Gattinger. Kripkevis, a haskell module to visualize kripke frames. Technical report, 2014.

[Gat15a]   Malvin Gattinger. Hascacbdd. `https://github.com/m4lvin/HasCacBDD`, 2015.

[Gat15b]   Malvin Gattinger. Noobdd. `https://github.com/m4lvin/NooBDD`, 2015.

[GS11]   Nina Gierasimczuk and Jakub Szymanik. A note on a generalization of the Muddy Children puzzle. In Krzysztof R. Apt, editor, *TARK'11*, pages 257–264. ACM, 2011.

[GvdM04]   Peter Gammie and Ron van der Meyden. Mck: Model checking the logic of knowledge. In *Computer Aided Verification*, pages 479–483. Springer, 2004.

[Lit53]   J.E. Littlewood. *A Mathematician's Miscellany.* Methuen, London, 1953.

[LQR15]   Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2015.

[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, pages 229–234, Berlin, Heidelberg, 2013. Springer-Verlag.

[Rav14]   Tristan Ravitch. robbed. `https://github.com/travitch/robbed`, 2011, updated 2014.

[Som12]   Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 2.5.0*, 2012.