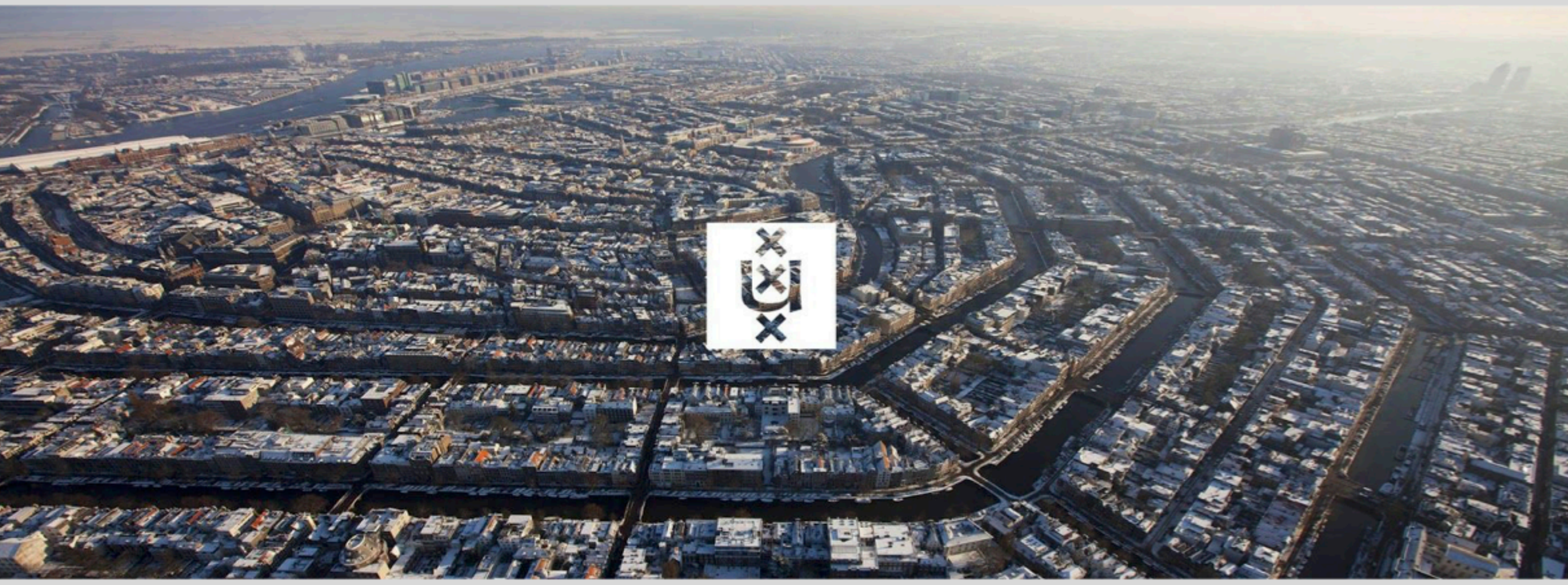

Institute for Logic, Language and Computation

Dynamic Epistemic Model Checking with Yices

Shuai Wang

UID: 11108339

ILLC, UvA

Contents

ABSTRACT

This project aims at compare effeciency of SMT/QBF with BDD as integrated solver for model checking of dynamic epistemic logic (DEL) problems. More specifically, this project takes advantage of the implementation of the DEL model checker, namely SMCDEL.

1 DYNAMIC EPISTEMIC LOGIC

Epistemic Logic is a subfield of modal logic. It is a formal study of expressions like 'knows that' or 'believes that'. A Multi-agent perspective of Epistemic Logic is about the knowledge of agent(s). The basic modal operator of epistemic logic is usually written K. For an agent i, $K_i\varphi$ is read as "the agent i knows φ". Different from Epistemic Logic where agents' knowledge don't change, Dynamic epistemic logic studies what kinds of events are responsible for change of knowledge [6] where agents can make public announcement about truth. After the announcement, there are certain changes in the agents' knowledge. The semantics of DEL is defined on Kripke models. A Dynamic Epistemic Model Checking problem is about evaluation of formula on such Kripke models.

This report follows the notation of [5].

Definition 1.1. A dynamic eistemic logic (DEL) formula for the set of propositions P and the set of agents A is given by:

$$\varphi ::= p|\neg\varphi|\varphi \wedge \varphi|\, K_i\varphi \,|\, C_\Delta\varphi \,|\, [\varphi]\varphi \,|\, [\varphi]_\Delta\varphi$$

where $p \in P$ and $i \in A$

The formula $K_i\varphi$ means "agent i knows φ". A common knowledge φ among a group of agents Δ is denoted as $C_\Delta\varphi$. $[\psi]\varphi$ indicates that a public announcement ψ makes φ true. Similarly, $[\varphi]_\Delta\varphi$ is a public announcement for the group Δ.

2 SMCDEL

SMCDEL [1] takes advantage of BDD solvers for model checking [5]. Instead of explicit model checking like DEMO [7], SMCDEL introduced *knowledge structure* by converting public announcements to quantified boolean formulas. SMCDEL used HasCacBDD [2] as a binder for CacBDD [3] to perform model checking.

2.1 LANGUAGE OF SMCDEL

SMCDEL [5] extends the standard DEL with group announcement and group knowledge.

```
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 |
```

[1] `https://github.com/jrclogic/SMCDEL`
[2] `https://github.com/m4lvin/HasCacBDD`
[3] `http://www.kailesu.net/CacBDD/CacBDD1.01.zip`

```
  Announce [Agent] Form Form | AnnounceW [Agent] Form Form
  deriving (Eq,Ord,Show)
```

2.2 KNOWLEDGE STRUCTURE

Definition 2.1. A knowledge structure is a tuple $F = (V, \theta, O_1, O_2, ..., O_n,)$.

where V is a finite set of propositional variables and θ a boolean formula over V. For each agent i, the propositions she can observe forms a set O_i.

The equivalence of a knowledge structure and explicit representation was proved in [5].

2.3 MODEL CHECKING WITH BDD

Binary Decision Diagrams (BDDs) are representations of decision process of formulas

Due to its effeciency for checking the satisfiability of a boolean formula for a specific interpretation, BDDs have been widely used in model checking. In SMCDEL, BDDs are employed to for several reasons:

Compact Representation

Compact representation of public announcements. Although computing BDDs takes time, once constructed. BDDs takes linear space.

Satisfiability Checking

Model checking with BDDs is fast. It takes linear time and has been used in many temporal logic solvers.

Third

BDDs are easy to use and has a wide range of choice in terms of implementation ...

This project aim at evaluating the time effeciency of BDDs and SMT regarding Quantified Boolean Formulas (QBFs). The SMT solver employed is Yices.

3 YICES

Yices[4] is a SMT (Satisfiability Modulo Theories) solver. In this project I used an older version[4, 2] of Yices (1.0.40). This is because the only working Haskell binding [4] is not working with any other version of Yices and due to the limit of time, this older version of Yices was used.

[4] `https://github.com/airobert/yices_haskell`

3.1 TYPES AND DECLARATIONS

There is no pre-defined type in Yices. Here I give a brief example of the definition of boolean variables. For example, to declare a boolean variable l_1, I need the following code to introduce boolean type and declare l_1 with reference to it. Note that to define a variable, it is required to declare the variable (its type) first. The following is just a declaration of l_1.

```
bool = VarT "bool"
l1 = DEFINE ("l1",bool)
```

3.2 EXPRESSIONS

```
data ExpY
  = VarE String
  | LitB Bool
  | LitI Integer
  | LitR Rational
  | AND [ExpY]
  | OR [ExpY]
  | NOT ExpY
  | ExpY :=> ExpY
  | ExpY := ExpY
  | ExpY :/= ExpY
  | ExpY :< ExpY
  | ExpY :<= ExpY
  | ExpY :> ExpY
  | ExpY :>= ExpY
  | ExpY :+: ExpY
  | ExpY :-: ExpY
  | ExpY :*: ExpY
  | ExpY :/: ExpY
  | DIV ExpY ExpY
  | MOD ExpY ExpY
  | IF ExpY ExpY ExpY
  | ITE ExpY ExpY ExpY
  | LET [((String,Maybe TypY),ExpY)] ExpY
  -- quantifires
  | FORALL [(String,TypY)] ExpY
  | EXISTS [(String,TypY)] ExpY
  -- functions
  | APP ExpY [ExpY]
  | UPDATE_F ExpY [ExpY] ExpY
  | LAMBDA [(String,TypY)] ExpY
  -- tuples
  | MKTUP [ExpY]
  | SELECT_T ExpY Integer
  | UPDATE_T ExpY Integer ExpY
  -- records
  | MKREC [(String,ExpY)]
  | SELECT_R ExpY String
  | UPDATE_R ExpY String ExpY
  -- bitvectors -- TODO
```

We can now define boolean constants.

```
true = LitB True
false = LitB False

l1 = VarE "l1"
l2 = VarE "l2"
l3 = VarE "l3"
```

To obtain an expression about $\forall l1, l2.(l1 \wedge l2) \vee l3$

```
ex1 = ASSERT (FORALL [("l1",bool), ("l2", bool)] (OR[AND[l1, l2], l3]))
```

EXERCISE 1

To translate the propositional sentences defined in DELLANG to QBF Yices expressions and test the satisfiability. We define the following function:

```
  boolSMTOf :: Form -> ExpY
  boolSMTOf Top            = true
  boolSMTOf Bot            = false
  boolSMTOf (PrpF (P n))   = VarE (show n)
  boolSMTOf (Neg forms)     = NOT (boolSMTOf forms)
  boolSMTOf (Conj forms)   = AND (map boolSMTOf forms)
  boolSMTOf (Disj forms)   = OR (map boolSMTOf forms)
  boolSMTOf (Xor [])       = false
  boolSMTOf (Xor l)        =
    let (b:bs) = map boolSMTOf l in
    foldl myxor b bs
    where
      myxor :: ExpY -> ExpY -> ExpY
      myxor s t  = (AND[OR[s, t], NOT (AND [s, t])])
 -- can we translate this as not equal ?
  boolSMTOf (Impl p1 p2)     = (boolSMTOf p1) :=> (boolSMTOf p2)
  boolSMTOf (Equi p1 p2)     = (boolSMTOf p1) :=  (boolSMTOf p2)
  boolSMTOf (Forall ps f) =
    let ps' = map (\(P x) -> (show x, bool)) ps in
    FORALL ps' (boolSMTOf f)
  boolSMTOf (Exists ps f) =
    let ps' = map (\(P x) -> (show x, bool)) ps in
    FORALL ps' (boolSMTOf f)
```

```
boolSMTOf _                = error "boolSMTOf␣failed:␣Not␣a␣boolean␣formula."
```

To test if the translated function is satisfiable with a assignment:

```
boolEval :: [Prp] -> Form -> IO Bool
boolEval truths form =
  do yp@(Just hin, Just hout, Nothing, p) <- createYicesPipe yicesPath []
    -- first, construct all the truth values as clauses
      let values = vars truths
      let values_declare = map ASSERT values
      -- also need to test if values are in def
      let props = propsInForm form
      let def = defs props
      let form_assert = [ASSERT (boolSMTOf form)]
      --yp@(Just hin, Just hout, Nothing, p) <- createYicesPipe yicesPath []
      runCmdsY yp (def ++ values_declare ++ form_assert) --
      check <- checkY yp
      print "-----def--------"
      print def
      print "------values_declare-----"
      print values_declare
      print "-----form--------"
      print form_assert
      return $
        case check of
          Sat ss -> True
          UnSat _ -> False
          Unknown _ -> error "SMT␣gives␣an␣unknown␣as␣reply"
          otherwise -> error "SMT␣gives␣other␣replies"
```

EXERCISE 2

With the following function, we can then test if $((p_1 \wedge p_2) \rightarrow p_2)$ is truth when p_1 and p_2 are both assigned to be true.

```
test_1 =
  let p1 = PrpF (P 1) in
  let p2 = PrpF (P 2) in
  let p = Conj [p1, p2] in
  let q = Impl p p2 in
  q

test_truth = do
  result <- (boolEval [P 1, P 2] test_1)
  if result then print "yes!"
    else print "oh␣no!"
```

It worth noticing that in SMCDEL, propositions are indexed by integer but in Yices (and many other SMT solver), propositions are ”indexed” by a string.

3.3 TRANSLATION OF KNOWLEDGE STRUCTURES TO QBF IN YICES

This project follows from SMCDEL and the semantics is as presented on the paper (Definition 5 and 6) [5]. Instead of using BDDs, we use a list of boolean formulas. The public announcement would then become a list of boolean formulas and the model checking problem is then the satisfiability problem. This project translate a restricted syntax by omitting group knowledge and group announcement.

EXERCISE 3

To update the data structure for knowledge structure, we first present the data structure in SMCDEL:

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

When replacing BDD with SMT clauses, we keep record of the declarations, the variables and the clauses. Thus, we introduce a new type, and define the new data structure for knowledge structures.

```
type BddY = ([Prp], [CmdY])

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

Moreover, we need the following two functions to update the knowledge structure for public annoucement.

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

pubAnnounceOnScn :: ScenarioY -> Form -> IO ScenarioY
pubAnnounceOnScn (kns,s) psi =
  do
    print "public␣announcement␣test"
    noconflict <- eval (kns,s) psi
    if noconflict then
      return (pubAnnounceY kns psi,s)
      else error "Liar!"
```

EXERCISE 4

Here we translate formulas of limited syntax. We removed public annoucement and knowledge for groups of agents. We will not translate ”wheather an agent or a group of agents know something”. With this restriction on language, we have the following translation (to keep the correspondence, the function name was kept the same as SMCDEL):

```
bddOf :: KnowStructY -> Form -> ExpY
bddOf _   Top              = true
bddOf _   Bot              = false
bddOf _   p@(PrpF (P n))   = VarE (name n)
bddOf kns p@(Neg forms)     = NOT (bddOf kns forms)
bddOf kns p@(Conj forms)   = AND (map (bddOf kns) forms)
bddOf kns p@(Disj forms)   = OR (map (bddOf kns) forms)
bddOf kns p@(Xor  [])     = false
bddOf kns p@(Xor  forms)   =
  let (b:bs) = map (bddOf kns) forms in
  foldl myxor b bs
  where
    myxor :: ExpY -> ExpY -> ExpY
    myxor s t  = (AND[OR[s, t], NOT (AND [s, t])])
bddOf kns p@(Impl p1 p2)    = (bddOf kns p1) :=> (bddOf kns p2)
bddOf kns p@(Equi p1 p2)    = (bddOf kns p1) :=  (bddOf kns p2)
bddOf kns p@(Forall ps f) =
  let ps' = map (\(P x) -> (show x, bool)) ps in
  FORALL ps' (bddOf kns f)
bddOf kns p@(Exists ps f) =
  let ps' = map (\(P x) -> (show x, bool)) ps in
  FORALL ps' (bddOf kns f)
bddOf kns@(KnS allprops lawbdd obs) (K i form) =
  FORALL otherps $ (AND ((map (bddOf kns) lawbdd))) :=> (bddOf kns form) where
    otherps = map (\(P n) -> (name n, bool)) $ allprops \\ apply obs i
bddOf kns (PubAnnounce form1 form2) = bddOf (pubAnnounceY kns form1) form2
```

EXERCISE 5

The evaluatin function takes a knowledge structure, a state and a formula. We first encode the knowledge structure, more specifically, we declare the variables. The encoding takes advantage of the translation function above. The results from Yices will then be analysed. The functin is as follows:

```
smtEval :: KnowStructY -> KnState -> Form -> IO Bool
smtEval kns@(KnS allprops lawbdd obs) state form =
  do yp@(Just hin, Just hout, Nothing, p) <- createYicesPipe yicesPath []
      let def = defs allprops
      let theta_assert = map (\x -> ASSERT (bddOf kns x)) lawbdd
```

```
    let state_assert = map ASSERT (vars state)
    let ex = ASSERT $ bddOf kns form
    --yp@(Just hin, Just hout, Nothing, p) <- createYicesPipe yicesPath []
    runCmdsY yp (def ++ theta_assert ++ state_assert) --
    print "--------def-----------"
    print def
    print "----------theta_assert------"
    print theta_assert
    print "--------state_assert----------"
    print state_assert
    Sat ss <- checkY yp
    print "--------has␣to␣be␣true␣here␣(the␣following␣is␣a␣model)---"
    print ss
    runCmdsY yp [ex]
    print "------now␣add␣the␣ex␣you␣want␣to␣test"
    print ex
    check <- checkY yp
    return $
      case check of
        Sat ss -> True
        UnSat _ -> False
        Unknown _ -> error "SMT␣gives␣an␣unknown␣as␣reply"
        otherwise -> error "SMT␣gives␣other␣replies"
```

Taking advantage of both functions. We can now perform model checking for certain knowledge structure and formulas. Note that we only consider valid public announcement.

```
eval :: ScenarioY -> Form -> IO Bool
eval (kns@(KnS allprops lawbdd obs),s) Top           = return True
eval (kns@(KnS allprops lawbdd obs),s) Bot           = return False
eval (kns@(KnS allprops lawbdd obs),s) (PrpF p)     = return $ p `elem` s
eval (kns@(KnS allprops lawbdd obs),s) p@(Neg form)    = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Conj forms)  = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Disj forms)  = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Xor  forms)  = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Impl f g)    = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Equi f g)    = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Forall ps f) = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(Exists ps f) = smtEval kns s p
eval (kns@(KnS allprops lawbdd obs),s) p@(K i form)    = smtEval kns s p
eval (kns,s) (PubAnnounce form1 form2) = do
  -- Do two evaluations
  print "**************<␣form1␣test␣>***************"
  anno <-(eval (kns, s) form1)
  if anno then
    do
      print "**************<␣form␣2␣test␣>************"
```

```
        anno2 <- (eval (pubAnnounceY kns form1, s) form2)
        if anno2 then
          return True
          else return False
      else error "Announcing␣something␣false"
```

4 EVALUATION

4.1 MUDDY CHILDREN

A muddy children game is a situation that some children are muddy after playing and a teacher come and announce some propositional or epistemic facts. A child then guess if he/she is muddy without looking at the mirror. In addition, what the teacher announce must be correct.

Similar as SMCDEL, we first define some side functions to help define the setting of the Muddy Children game. The first formula in θ is set to be truth by default. Additional announcements will be "stacked" here.

```
mudScnInitY :: Int  -> ScenarioY
mudScnInitY n = ( KnS mudProps [Top]
      [ (show i, delete (P i) mudProps) | i <- [1..n] ], [P 1 .. P n])
      where mudProps = [P 1 .. P n]

knows :: Int -> Form
knows i = Disj [K (show i) (PrpF $ P i), K (show i) (Neg (PrpF $ P i))]
```

4.2 TESTING AND RESULTS

In this section, we perform a series of tests on the structure and translation of this new approach.

EXERCISE 6

Muddy Children 1 should know truth.

```
myMud3 :: ScenarioY
myMud3 = mudScnInitY 3

-- test 1, Agent 1 know true : Success
test_1 = eval myMud3 (K "1" Top)
```

EXERCISE 7

Next we test if agent 1 knows if she is muddy.

```
test_2 = eval myMud3 (knows 1)
```

However Yices gives ”unknown”. This could be due to the implementation or the translation. I printed the encodidng and checked by hand. It was unlikely to be wrong in encoding.

```
*** Exception: SMT gives an unknown as reply
```

EXERCISE 8

We introduce the function to check nobody knows their color in a group.

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

To be delighted, these tests passed.

```
no3 = nobodyknows 3
test_5 = bddOf (fst myMud3) no3
test_6 = eval myMud3 no3
```

EXERCISE 9

Next we define a public announcement to tell all agents that at least one of the agent is muddy.

```
at_least :: Int -> Form
at_least n = Disj (map PrpF [P 1 .. P n])
```

Test 8 passed the test but Yices could not work out Test 9. This is another evidence that this version of Yices is not consistent enough.

```
test_8 =
  do
    mys <- pubAnnounceOnScn myMud3 (at_least 3)
    result <- eval mys (nobodyknows 3)
    if result then print "test␣passed" else print "test␣failed"

test_9 =
  do
    mys <- pubAnnounceOnScn myMud3 (at_least 3)
    result <- eval mys (PubAnnounce (PrpF(P 1)) (nobodyknows 3))
    if (not result) then print "test␣passed" else print "test␣failed"
```

5 CONCLUSION AND FUTURE WORK

This project aimed at comparing BDD and SMT for model checking of epistemic logic problems. In this project, I used an older version of Yices. I noticed that it was hard to encode group announcement and group knowledge without paying the price of iterating and computation. The reason is that to compute the greatest fix point of a BDD, we need such iterations and this can be computationally expensive. Moreover, Yices sometimes could not work out the satisfiability of a set of formulas. However, Yices (2.2) [3] is now more powerful and more stable after years of development. It would be necessary to update the Haskell binding for evaluation for better effeciency and accuracy. Another way to solve the problem is to use C++ and try the SMT solver Z3. Similar attempts have been proved to be successful, especially ECTL[8] and ENCoVer (using Z3)[1] [5].

[5] `http://www.nada.kth.se/~musard/encover.html`

ACKNOWLEDGEMENT

I would like to thank Mr. Malvin Gattinger and Dr. Jan van Eijck for their kind help during this project. I also appreciate help from classmates in the Functional Specification and Algorithms course while learning Haskell.

References

[1] Musard Balliu, Mads Dam, and Gurvan Le Guernic. ENCoVer: Symbolic Exploration for Information Flow Security. In *Proceedings of the IEEE Computer Security Foundations Symposium*, June 2012.

[2] Leonardo de Moura and Bruno Dutertre. Yices 1.0: An efficient smt solver. *The Satisfiability Modulo Theories Competition (SMT-COMP)*, page 10, 2006.

[3] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer, 2014.

[4] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[5] J van Benthem, J van Eijck, M Gattinger, and K Su. Symbolic model checking for dynamic epistemic logic.

[6] HP van Ditmarsch, J Ruan, and W van der Hoek. Model checking dynamic epistemics in branching time. *Formal Approaches to Multiagent Systems*, page 107.

[7] J van Eijck. Demo-s5, 2014.

[8] Agnieszka M Zbrzezny, Bozena Wozna-Szczesniak, and Andrzej Zbrzezny. Smt-based bounded model checking for weighted interpreted systems and for weighted epistemic ectl. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1671–1672. International Foundation for Autonomous Agents and Multiagent Systems, 2015.