Symbolic Gossip

Joris Galema
Madeleine Gignoux
Djanira Gomes
Wouter Smit

Friday 31st May, 2024

Abstract

We extend the interpretability of output from SMCDEL's Knowledge Scenes for The Gossip Problem, implement the Transparent Gossip Problem using SMCDEL's existing Knowledge Transformer, and write a Simple Knowledge Transformer for computing the Synchronous Gossip Problem efficiently.

Contents

1 Introduction

The Gossip Problem or *Gossip* is the problem of sharing information in a network. Many variants of Gossip exist, each with their own computational challenges. Most notably, a distinction is made between the *Transparent* Gossip Problem - the situation where all agents know which agents exchange information at any update - and the *Synchronous* Gossip Problem, where agents know when an update occurs but not which agents exchange information during that update.

For modelling Gossip, an explicit model checker for Gossip called *GoMoChe* exists [Gat23]. Explicit model checkers are generally less efficient than symbolic ones, which aim to cut down on computation time. GoMoChe too is therefore computationally limited to small examples. On the other hand, a symbolic model checker for dynamic epistemic logic (DEL) called SMCDEL exists, which is much more general than *GoMoChe*. SMCDEL is implemented for both K and $S5$ and contains symbolic representations for various logic problems, including Gossip [Gat18]. However, in terms of Gossip, SMCDEL only covers an encoding of the Synchronous Gossip Problem (in standard S5 DEL), and the implementation of its update function causes the model to blow up in terms of complexity.

A solution to this exponential blowup was proposed in the unpublished master's thesis by [Rei23], in the shape of a *Simple Knowledge Transformer* that should replace the *Classic Knowledge Transformer* from SMCDEL. An existing implementation by [Yuk23] extends SMCDEL to incorporate updates with Simple Transformers, but an instance of this transformer tailored to the Gossip problem wasn't included.

This project expands on SMCDEL's functionality. Section 2 contains a description of the Classic Knowledge Transformer in SMCDEL, and specifically how it is used to model updates to the state in [Gat18]. Next, Section 3 contains a number of functions that provide an interpretation of the current state, which makes the Synchronous Gossip Problem already provided in SMCDEL more user-friendly. Next we create a variant of the Classic Transformer for the transparent variant of the Gossip Problem in Section 4. To conclude our work, Section 5.2 describes our implementation of the Simple Transformer, which cuts down on the complexity of computing the Synchronous Gossip Problem, with the tradeoff of losing higher-order knowledge. Finally, the code of Section 3, 4, and 5.2 is tested in Subsections 6.1, 6.2, and 6.3 respectively.

2 Background

For the language and syntax of Gossip, please refer to [Gat18] (Section 6.6). We discuss how the Gossip Problem is approached in SMCDEL using [Gat18] (in particular, Section 6.6.5 on Symbolic Gossip). For an in-depth explanation, please refer to the aforementioned source.

The Gossip Problem models the flow of information called secrets. At the initial state of the problem, no information has been shared and each agent knows only their own secret. The goal is for the agents to exchange all secrets, which happens through *updates* on the model, which is called a *Knowledge Structure*. The Knowledge Structure and the actual state are described by the *vocabulary* (V), *state law* (θ), and *observations* (O_i for each agent i). The vocabulary V expresses all existing atomic propositions of the form S_ij, where S_ij denotes agent i knowing agent j's secret. Next, the state law θ describes the possible worlds in the current model. Following the conceptual assumption that all agents are aware of the model they reside in, θ is common knowledge among the agents. Initially, θ states that nobody knows anyone else's secret. Finally, the observations O_i describe which propositional variables agent i observes; following [Gat18], the observations are initially empty for all agents. Throughout the run of the model, propositions are added to the observables, which encode which calls each agent can observe.

For the sake of simplicity, the notions of knowing one's own secret are completely removed. Equation 1 (from [Gat18]) shows the tuple describing the initial Knowledge Structure.

$$F_{\text{init}} = (V = \{S_i j \mid i, j \text{ Agents }, i \neq j\}, \theta = \bigwedge_{i \neq j} \neg S_i j, O_i = \emptyset) \tag{1}$$

In order to transform the model after a call happens, we use a Knowledge Transformer. The crux of this paper involves changing the Knowledge Transformer for the Synchronous Gossip Problem provided in SMCDEL to fit our needs.

The Knowledge Transformer explains how the state should change after an update, in this case an arbitrary call. The vocabulary is extended with propositional variables q_{ij}, which express that agent i called agent j. Recalling that we are dealing with the Synchronous Gossip Problem, where agents only know a call occurred, but not which two agents called, we encode this into two laws θ^+ and θ_-, where θ^+ (also: *preconditions* for a call) expresses that exactly one call happens, and θ_- (also: *postconditions* of a call) expresses the conditions under which agent i can learn agent j's secret. Finally, each agent i observes only calls they participate in, which we describe in O_i^+.

In short, the Knowledge Transformer for The Synchronous Gossip Problem is the quintuple $\chi_{\text{call}} = (V^+, \theta^+, V_-, \theta_-, O^+)$ (see [Gat18], page 195 for the exact encoding).

The design of the Knowledge Transformer allows it to encode and check higher-order knowledge, but it also poses a problem in the form of exponential blowup. The state law (θ) keeps track of the updates in the model and is itself updated using θ^+ and θ_-. Essentially, the state law after the final update forms a conjunction of the original state law with event laws (θ^+) for each update and `changelaws` (θ_-), such that the validity of a logical formula on a given Knowledge Structure can be evaluated by solely checking if it's implied by the state law.

However, it is possible for an update to create states that previously were excluded by the state law. In order to allow this type of flexibility, each update causes all propositional variables of the form $S_i j$ to be copied and labelled in the state law. For example, suppose Alice learns Bob's secret during update n. Any occurrence of the corresponding proposition $S_a b$ in the state law need to be flagged in update $n + 1$, just in case Alice would *forget* Bob's secret in some future update. A copy of $S_a b$ is added and now exists alongside the flagged version (denoted by $(S_a b)^o$ to indicate that it is an "old" proposition). Even for a small number of agents and calls (say, 4 agents and 3 calls), the blowup of the state law is as such that it's unfeasible to print an example of the representation of the resulting Knowledge Structure.

The possibility of the truth value $S_a b$ to change back to false is an unrealistic hypothetical in Gossip, as in this situation agents aren't modelled to forget any secrets. However, SMCDEL is implemented for a wide range of logical problems, which prevents it from making such assumptions.

The existing implementation ([Gat18]) includes optimization functions that discard the redundant propositions (by checking which propositional variables are equivalent), but this optimization is only implemented to be run after running the model and is therefore not optimal.

With this background on how to model Gossip symbolically, we write our own transformer for modelling the transparant variant of The Gossip Problem, implement an adapted optimization that runs in between updates, and a simple transformer based on Daniel Reifsteck's master's thesis.

3 Gossip Scene Investigation

This section explains the functions that we created to make sense of the current state of a given gossip problem, i.e. gossip scene investigation. The functions only work on the unoptimized, Classic Transformer, since the code relies on the exact vocabulary being copied. First of all, the code makes use of the following imports:

```
module Explain where

import SMCDEL.Symbolic.S5
import SMCDEL.Language
import SMCDEL.Other.BDD2Form
import Data.Maybe

-- import Debug.Trace
```

One remarkable property of the SMCDEL implementation [Gat18] is how the transformer updates the vocabulary by copying all of the secret propositions. This means that in any given transformation, there will be a propositional variable representing a secret S_ij, as well as a copy of said variable $(S_ij)^o$. Moreover, we have propositions for calls q_{ij}. In order to prevent overlap between the several groups of variables, a unique value is computed for each propositional variable. A propositional variable is of the form Pi, where i is generated using one of the following functions ([Gat18]):

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

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

In order to make the description of a Knowledge Structure human-readable, we defined the following functions to translate the encoded propositions: `prpLibrary` checks whether a proposition denotes a secret, call proposition, or copy of a secret. The function takes the vocabulary as input, as well as the number of agents, and returns the library from which we can decipher propositions in our gossip scene investigation.

```
-- decode secrets
secretDecoder ::  [Prp] -> Int -> [String]
secretDecoder [] _ = []
secretDecoder ((P p):ps) n = ("s"++ show i ++ show j) : secretDecoder ps n
```

```
      where (i, j) = (p `quot` n, p `rem` n)

prpLibrary :: [Prp] -> Int  -> [(Prp,String)]
prpLibrary prps n = zip prps (prpLibraryHelper prps)
   where
      -- assign the propositions to secrets, calls, and copies of secrets
      -- and decode each with the appropriate decoder
      prpLibraryHelper :: [Prp]  -> [String]
      prpLibraryHelper [] = []
      prpLibraryHelper prps' = a ++ copyDecoder (drop (div (3*n*(n-1)) 2) prps') a "'"
         where
            a = secretDecoder (take (n*(n-1)) prps') n ++ callDecoder 0 (take (div (n*(n-1)
                ) 2) (drop (n*(n-1)) prps'))

      -- decode calls
      callDecoder :: Int -> [Prp] -> [String]
      callDecoder k calls | k >= div (n*(n-1)) 2 = []
                          | null calls = []
                          | otherwise = ("q" ++ show i ++ show j) : callDecoder (k + 1)
                            calls
         where
            (i, j) = getCNums k 0
            getCNums :: Int -> Int -> (Int,Int)
            getCNums k' r'' | (k'+1) < n = (r'',k'+1)
                            | otherwise = getCNums (k'-n+2+r'') (r''+1)

      -- decode copies
      copyDecoder :: [Prp] -> [String] -> String -> [String]
      copyDecoder [] _ _ = []
      copyDecoder props lib r = map (++r) lib ++ copyDecoder (drop (length lib) props) lib
          (r++"'")


prpLibraryTr :: [Prp] -> Int -> [(Int, Int)] -> [(Prp, String)]
prpLibraryTr prps n calls = zip prps (decSec ++ callsNcopies (drop nS prps) calls "'")
   where
      nS = (n-1)*n
      decSec = secretDecoder (take nS prps) n
      -- decode calls and append a decoded Secrets primed (copies)
      callsNcopies :: [Prp] -> [(Int, Int)] -> String -> [String]
      callsNcopies [] _ _ = []
      callsNcopies _ [] s = map (++s) decSec
      callsNcopies (_:ps) ((a,b):c) s = ["q"++show a++show b++tail s] ++ map (++s) decSec
          ++ callsNcopies (drop nS ps) c (s++"'")
```

Additionally, we wrote the (unsafe) function `explainPrp`, which takes in a proposition as well as the library, to return its meaning (as String).

```
explainPrp :: Prp -> [(Prp,String)] -> String
explainPrp (P x) prpLib = fromJust (lookup (P x) prpLib)
```

We follow this up with `gsi`, our gossip scene investigation, which takes in a knowledge scene and the number of agents, and uses `explainPrp` to make sense of the vocabulary and observations.

```
-- Gossip Scene Investigation: GSI. ...like the tv show but with less crime and more gossip
    .
--

gsiVoc :: KnowScene -> IO()
gsiVoc kns@(KnS voc _ _, _) = do
   putStrLn "Vocabulary: "
   mapM_ (putStrLn . (++) " --  " . \p -> explainPrp p lib) voc
      where
      lib = prpLibrary voc (length $ agentsOf kns)

gsiStLaw :: KnowScene -> IO()
gsiStLaw kns@(KnS voc stl _ ,_) = do
   putStrLn "State Law: "
   print (ppFormWith (`explainPrp` lib) (formOf stl))
      where
      lib = prpLibrary voc (length $ agentsOf kns)
```

```
gsiObs :: KnowScene -> IO()
gsiObs kns@(KnS voc _ obs ,_) = do
   putStrLn "Observables: "
   mapM_ (putStrLn . (++) " --   " . (\ x -> fst x ++ ":  " ++ show (map (`explainPrp` lib)
       (snd x)) )) obs
      where
      lib = prpLibrary voc (length $ agentsOf kns)

gsiState :: KnowScene -> IO()
gsiState kns@(KnS voc _ _,s) = do
   putStrLn "Actual state: "
   if null s then putStrLn " --  Nobody knows about any other secret"
      else
      mapM_ (putStrLn . (++) " --   " . \p -> explainPrp p lib) s
         where
      lib = prpLibrary voc (length $ agentsOf kns)


gsi :: KnowScene -> Maybe [(Int, Int)] -> IO ()
gsi kns@(KnS voc stl obs, s) calls = do
    putStrLn "Vocabulary: "
    mapM_ (putStrLn . (++) " --   " . \p -> explainPrp p lib) voc
    putStrLn "State Law: "
    print (ppFormWith (`explainPrp` lib) (formOf stl))
    putStrLn "Observables: "
    mapM_ (putStrLn . (++) " --   " . (\ x -> fst x ++ ":  " ++ show (map (`explainPrp` lib)
        (snd x)) )) obs
    putStrLn "Actual state: "
    if null s then putStrLn " --  Nobody knows about any other secret"
      else
      mapM_ (putStrLn . (++) " --   " . \p -> explainPrp p lib) s
   where
      lib | isNothing calls = prpLibrary voc (length $ agentsOf kns)
          | otherwise = prpLibraryTr voc (length $ agentsOf kns) (fromJust calls)
```

We can then run the following:

```
import SMCDEL.Examples.GossipS5
ghci> gsi $ gossipInit 3
Vocabulary:
 --  s01
 --  s02
 --  s10
 --  s12
 --  s20
 --  s21
State Law:
"(~s01 & ~s02 & ~s10 & ~s12 & ~s20 & ~s21)"
Observables:
 --  0:  []
 --  1:  []
 --  2:  []
Actual state:
 --  Nobody knows about any other secret

ghci> gsi $ doCall (gossipInit 3) (0,1)
Vocabulary:
 --  s01
 --  s02
```

```
 --  s10
 --  s12
 --  s20
 --  s21
 --  q01
 --  q02
 --  q12
 --  s01'
 --  s02'
 --  s10'
 --  s12'
 --  s20'
 --  s21'
State Law:
"((s01 & ~s02 & s10 & ~s12 & ~s20 & ~s21 & q01 & ~q02
      & ~q12 & ~s01' & ~s02' & ~s10' & ~s12' & ~s20' & ~s21')
   | (~s01 & ((s02 & ~s10 & ~s12 & s20 & ~s21 & ~q01
      & q02 & ~q12 & ~s01' & ~s02' & ~s10' & ~s12' & ~s20' & ~s21')
   | (~s02 & ~s10 & s12 & ~s20 & s21 & ~q01 & ~q02
      & q12 & ~s01' & ~s02' & ~s10' & ~s12' & ~s20' & ~s21'))))"
Observables:
 --  0:  ["q01","q02"]
 --  1:  ["q01","q12"]
 --  2:  ["q02","q12"]
Actual state:
 --  s01
 --  s10
 --  q01
```

In the future, we hope to also show the law as its BDD (Binary Decision Diagram [1]) using the tool graphviz.

4 Transparent Transformer

This section describes a variant of the Classic Knowledge Transformer that is implemented for the Transparent Gossip Problem. This transformer is tailored to the actual call that happens, which makes sure that whenever a call happens, all agents know this and also know which agents participate.

```haskell
module Transparent where

import SMCDEL.Examples.GossipS5
import SMCDEL.Language
import SMCDEL.Symbolic.S5
```

We chose to adapt the existing function `callTrf` from GossipS5, which is the call transformer for the Synchronous Gossip Problem. Instead of `Int -> KnowTransformer`, the function is

[1] A Binary Decision Diagram provides a concise representation of a Boolean formula. SMCDEL uses BDDs for the symbolic evaluation of logic problems.

now `Int -> Int -> Int -> KnowTransformer`, so that agents a and b are arguments for the transformer for call ab. As in Section 2, we redefine how to update the vocabulary, law, and observations of each agent.

First, the vocabulary V^+ (the `eventprops`), now simply consists of the call between agents a and b. As opposed to the synchronous case, we don't need extra vocabulary to describe all possible calls that could be happening: all agents know exactly which call happens.

The `eventlaw`, θ^+ (which originally stated that only one call happens at a time), is simplified to describe that only the specified call between a and b happens. The `changelaws`, θ_-, are quite different from those in the Classic Transformer: the conditions for the proposition S_ij to be true after *some* call happens, are simplified to the conditions S_ij to be true after the *actual* call ab happens.

For instance, if i is agent a, then i knows j's secret after call ab if either

1. i knew it already, or
2. j equals b, or
3. b told i the secret of j during their call.

Finally, the `eventobs`, O_k^+ for each agent k, are also simplified to call ab, since there is only one possible event happening and every agent observes it.

```
callTrfTransparent :: Int -> Int -> Int -> KnowTransformer
callTrfTransparent n a b = KnTrf eventprops eventlaw changelaws eventobs where
  thisCallHappens = thisCallProp (a,b)
  -- the only event proposition is the current call
  eventprops = [thisCallHappens]

  -- call ab takes place and no other calls happen
  eventlaw = Conj [PrpF thisCallHappens,
                Conj [Neg (PrpF $ thisCallProp (i,j)) | i <- gossipers n
                                                      , j <- gossipers n
                                                      , not ((i == a && j == b) || (i == b
                                                          && j == a))
                                                      , i < j ]]
  changelaws =
  -- i has secret of j
      -- case: i is not a and i is not b: then i can not have learned the secret unless it
          already knew it (has n i j)
    [(hasSof n i j, boolBddOf $ has n i j) | i <- gossipers n, j <- gossipers n, i /= j, i
        /= a, i /= b] ++
      -- case: i is a, j is not b: then i learned the secret if it already knew it, or b
          knew the secret of j
    [(hasSof n a j, boolBddOf $ Disj [ has n a j , has n b j ]) | j <- gossipers n, a /= j,
        b /= j ] ++
      -- case: i is a, j is b: then Top (also: i is b, j is a)
    [(hasSof n a b, boolBddOf Top)] ++ [(hasSof n b a, boolBddOf Top)] ++
      -- case i is b, j is not a: synonymous to above
    [(hasSof n b j, boolBddOf $ Disj [ has n b j , has n a j ]) | j <- gossipers n, a /= j,
        b /= j ]

  eventobs = [(show k, [thisCallHappens]) | k <- gossipers n]
```

Since the transparent transformer has the same type as the synchronous variant, we inherited its update function. The following functions were adapted from the original implementation to perform the transparent update:

```
callTransparent :: Int -> (Int,Int) -> Event
callTransparent n (a,b) = (callTrfTransparent n a b, [thisCallProp (a,b)])

doCallTransparent :: KnowScene -> (Int,Int) -> KnowScene
```

```
doCallTransparent start (a,b) = start `update` callTransparent (length $ agentsOf start) (a
    ,b)

afterTransparent :: Int -> [(Int,Int)] -> KnowScene
afterTransparent n = foldl doCallTransparent (gossipInit n)

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

5 Optimization

We will now look at improving the runtime of the synchronous case.

As mentioned before, the classical transformer defined in [Gat18] involves a considerable amount of inserted propositions and extensions of the state law.

This is further worsened by the nature of the gossip problem: both the event propositions (the calls) and the secret atoms grow rapidly as the number of agents increases. Moreover, the statelaw encodes all of the secret atoms.

The route of optimization therefore seems to be in limiting the amount of proposition insertions. We show two methods do that: using a optimization function to trim redundant propositions, and a different notion of transformer that avoids copying alltogether.

5.1 Using the `optimize` function

The SMCDEL library contains an `optimize` function which aims to minimize the size of the knowledge structure by removing redundant propositions. Usually this is run at the end of a sequence of calls, but we will now define a few wrappers to interleave the optimisation step between each individual call.

Simply trimming redundant propositions that were added by the classical transformer could potentially already provide a reasonable speed improvement.

```
doCallOpt :: [Prp] -> KnowScene -> (Int, Int) -> KnowScene
doCallOpt vocab start (a,b) = optimize vocab $ start `update` call (length $ agentsOf start
    ) (a,b)

afterOpt :: Int -> [(Int,Int)] -> KnowScene
afterOpt n = foldl (doCallOpt $ vocabOf (gossipInit n)) (gossipInit n)
```

5.2 Simple Transformer

As we have seen, the (classical) implementation of transformer inserts propositions into the vocabulary and modfies the state law, which is suspected to be the main source of computation. We therefore consider a different notion of transformers called *Simple Transformers*, which was introduced by [Rei23]. There is a notion of such transformers with and without factual change. For gossip we will only use the simple transformers with factual change.

The simple transformer is a less expressive transformer that is ‘simple’ in the sense that it disregards complexities in the knowledge semantics. The benefit of this simplification is that it does not modify the vocabulary or state law, which are precisely the parts of the knowledge structure that grow uncontrollably in the classical case.

The transformer still uses the event propositions in V^+ and the change laws θ_- to determine the factual change V_-, but instead uses that result to modify the state rather than the knowledge structure.

Meanwhile, observables can be mutated similarly to the classical case, with the addition of the notion to remove observables from agents too. However, for Gossip we will not use the observable management and instead only rely on the transformer to compute the factual change.

5.2.1 Simple Initial Knowledge Scene

Due to the limitations in changing the knowledge structure with every update, we must make minimal changes to the initial knowledge scene.

The model is initialized by the `gossipInitSimple` function, which is a modification of the `gossipInit` function part of the Gossip implemenation in SMCDEL `GossipS5` file.

```
-- Initialize a gossip scene for the simple transformer
gossipInitSimple :: Int -> KnowScene
gossipInitSimple n = (KnS vocab law obs, actual) where
    vocab  = [ hasSof n i j | i <- gossipers n, j <- gossipers n, i /= j ]
    law    = boolBddOf Top                                              -- Top
    obs    = [ (show i, allSecretsOf n i) | i <- gossipers n ]          -- observe all
        own secret knowledge
    actual = [ ]

-- Retrieve for some agent x all secret atoms of the form $S_xi$
allSecretsOf :: Int -> Int -> [Prp]
allSecretsOf n x = [ hasSof n x j | j <- gossipers n, j /= x ]
```

The vocabulary `vocab` stays the same and contains all secret atoms from the language, the state law and observables however are modified.

Whereas the state law in `gossipInit` describes the situation in which agents only know their own secrets, this definition is too restrictive for the simple implementation: it prevents the learning of secrets. In order not to exclude any possible later states, we chose a simple state law of $\theta = \top$.

The observables `obs` are slightly different too. While in the classical case these were empty, in the simple transformer we want them to reflect each agent's own secret-knowledge atoms. That is, each agent x can observe the set $\{S_x j \mid j \in Ag \wedge x \neq j\}$ where Ag is the set of all agents.

Conceptually, these observables make sense to be true from the very start: an agent should be aware of what secrets they know themselves.

Analogous to the classic implementation, the state `actual` is initially empty as it describes all true propositions of the form "i knows the secret of agent j". Note specifically that we again ignore the atoms $S_a a$: while agents do know their own secrets, these are not encoded by propositions and therefore not mentioned in the state.

5.2.2 The Simple Transformer for Gossip

We wil now define the transformer itself. The function `CallTrfSimple` is the simple analogue of the classic transformer `callTrf` from [Gat18]. Note that we have a single transformer to execute any of the calls, in order for the semantics to be synchronous.

The event vocabulary V^+ contains again all fresh variables needed to describe the transformation, just like in the classical transformer.

The state law θ_- (`changelaws`) is similarly defined as in the classic transformer, allowing the update to compute the factual change V_- and modify the state

The transformation observables in this transformer are empty, as we will show that the specific update function will only need the observables in the original knowledge structure.

```
callTrfSimple :: Int -> SimpleTransformerWithFactual
callTrfSimple n = SimTrfWithF eventprops changelaws changeobs where
    -- helper functions to construct the required formulae
    thisCallHappens (i,j) = PrpF $ thisCallProp (i,j)
    isInCallForm k = Disj $ [ thisCallHappens (i,k) | i <- gossipers n \\ [k], i < k ]
                        ++ [ thisCallHappens (k,j) | j <- gossipers n \\ [k], k < j ]
    allCalls = [ (i,j) | i <- gossipers n, j <- gossipers n, i < j ]

    -- V+ event props stay the same as classic transformer
    eventprops = map thisCallProp allCalls

    -- theta- change law stays same as classic transformer
    changelaws =
      [(hasSof n i j, boolBddOf $               -- after a call, i has the secret of j iff
          Disj [ has n i j                      -- i already knew j, or
               , Conj (map isInCallForm [i,j]) -- i and j are both in the call or
               , Conj [ isInCallForm i          -- i is in the call and there is some k in
                       , Disj [ Conj [ isInCallForm k, has n k j ] -- the call who knew j
                             | k <- gossipers n \\ [j] ] ]
               ])
      | i <- gossipers n, j <- gossipers n, i /= j ]

    -- Change observables are empty as they are not used
    changeobs    = [ (show i, ([],[])) | i <- gossipers n ]
```

The following functions are analogues of those in originally defined in SMCDEL `GossipS5.hs` and instead use the simple transformer.

```
-- construct a a single call event with a simple transformer
simpleCall :: Int -> (Int,Int) -> StwfEvent
simpleCall n (a,b) = (callTrfSimple n, [thisCallProp (a,b)])

-- execute a simple call event
doCallSimple :: KnowScene -> (Int,Int) -> KnowScene
doCallSimple start (a,b) = start `update` simpleCall (length $ agentsOf start) (a,b)

-- execute repeated calls using the simple transformer
afterSimple :: Int -> [(Int, Int)] -> KnowScene
afterSimple n = foldl doCallSimple (gossipInitSimple n)

-- evaluate if the sequence cs is successful for n agents
isSuccessSimple :: Int -> [(Int,Int)] -> Bool
isSuccessSimple n cs = evalViaBdd (afterSimple n cs) (allExperts n)
```

5.2.3 Updates using the Simple Transformer

While the original definition of the Simple Transformer in [Rei23] specifies how the new knowledge scene is constructed, we have to modify it for our Gossip-specific observable management to work as desired.

The following code extends the SMCDEL library, specifically the S5-specific symbolic implementation `SMCDEL.Symbolic.S5` with Simple Transformers with Factual Change.

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

```haskell
module SmpTrfS5 where

{-
- This file is a partial copy Symbolic.S5 of Haitian's fork of SMCDEL
  the file includes definitions for Simple Transformers (SmpTrf).
- The update function for SmpTrf with factual change is specific to Gossip
- NOTE: Due other changes in Haitian's fork and the SMCDEL main repo,
  dynamic operators in formulae do not work.
  Instead update the knowledge structure
-}
```

```haskell
{-
    Simple transformer with factual change
-}
data SimpleTransformerWithFactual = SimTrfWithF
  [Prp]                        -- V+ is a set of new variables encoding of a set of events
  [(Prp,Bdd)]                  -- Theta- assigns a formula to each modified variable.
  [(Agent,([Prp],[Prp]))]      -- O+ and O- for each agent
  deriving (Eq,Show)

instance Pointed SimpleTransformerWithFactual State
type StwfEvent = (SimpleTransformerWithFactual,State)

instance HasPrecondition StwfEvent where
  preOf _ = Top

-- The following instance is modified from Haitian's implementation of the
-- general simple transformer definition
-- It is *only* applicable to synchronous Gossip calls
instance Update KnowScene StwfEvent where
  checks = [haveSameAgents]
  unsafeUpdate kns@(KnS v th obs,s) (SimTrfWithF _ thetaminus _,x) = (newkns, newstate)
      where
    -- gossip helper functions to be able to find the current two agents in the call
    thisCallProp :: (Int,Int) -> Prp
    thisCallProp (i,j) | i < j     = P (100 + 10*i + j)
                       | otherwise = error $ "wrong call: " ++ show (i,j)

    n = length $ agentsOf kns
    gossipers :: [Int]
    gossipers = [0..(n-1)]

    allCalls = [ (i,j) | i <- gossipers, j <- gossipers, i < j ]
    allCallprops = map thisCallProp allCalls
    callPropResolver = zip allCallprops allCalls

    -- the transformation state x contains only 1 call proposition
    inThisCall :: (Int, Int)
    inThisCall = callPropResolver ! head x

    -- Compute special observable management for Gossip
    -- Calling agents get their own original observables O_i plus
    -- the intersection of the other agent's observables with the state (their known true
       secrets)
    -- Note that the transformer observables are ignored fully.
    newobs = [ (show i,obs ! show i) | i <- gossipers, i < fst inThisCall ] ++
             [ (show (fst inThisCall) , obs ! show (fst inThisCall) ++ intersect newstate (
                 obs ! show (snd inThisCall))) ] ++
             [ (show i,obs ! show i) | i <- gossipers, i > fst inThisCall, i < snd
                 inThisCall ] ++
             [ (show (snd inThisCall) , obs ! show (snd inThisCall) ++ intersect newstate (
                 obs ! show (fst inThisCall))) ] ++
             [ (show i,obs ! show i) | i <- gossipers, i > snd inThisCall  ]


    newkns = KnS v th newobs -- keep V and Theta but changes obs
    newstate = sort ((s \\ map fst thetaminus) ++ filter (\ p -> bddEval (s ++ x) (
       thetaminus ! p)) (map fst thetaminus))
```

6 Testing

6.1 Gossip Scene Investigation

```
module ExplainTestsSpec where

import Explain
import SMCDEL.Examples.GossipS5
import SMCDEL.Language
-- import SMCDEL.Symbolic.S5
-- import Test.QuickCheck

import Test.Hspec hiding (after)
```

Tests:

- Secret propositions are translated correctly
- The vocabulary has correct length
- After a call the state is updated correctly

```
spec :: Spec
spec = do
    describe "secret translation:" $ do
        it "init " $ do
           prpLibrary (hasSofs 1) 1 `shouldBe` []
           prpLibrary (hasSofs 2) 2 `shouldBe` [(P 1, "s01"), (P 2, "s10")]
           prpLibrary (hasSofs 5) 5 `shouldBe` zip (hasSofs 5) (enumS 5)
           prpLibrary (hasSofs 10) 10 `shouldBe` zip (hasSofs 10) (enumS 10)
        --it "after calls" $ do
        --   prpLibrary (callsVoc 3 [(0,1)]) 3 `shouldBe` [] --- How to test this?
        --it "length" $ do
           where
            hasSofs :: Int -> [Prp]
            hasSofs n = [ hasSof n i j | i <- gossipers n, j <- gossipers n, i /= j ]
            enumS :: Int -> [String]
            enumS n = ["s"++ show i ++ show j | i <- gossipers n, j <- gossipers n, i /= j
                ]
            --callsVoc :: Int -> [(Int, Int)] -> [Prp]
            --callsVoc n sequ = v
             --   where
               --     (KnS v _ _, _) = after n sequ
```

6.2 Transparent Transformer

We execute the following tests on the transparent variant of the Classic Transformer. The simple checks also apply to the Classic Transformer and encode the basic requirements of a transformer for a Gossip problem. However, some of the higher-order knowledge (for instance, after a call ab, agent c should know that a knows b's secret, since c knows which call happened) is specific to the transparent implementation. Finally, we include a number of higher-order knowledge tests that are not specific to the transparent variant.

```
module TransparentTransformerSpec where

import Test.Hspec hiding ( after )
import SMCDEL.Examples.GossipS5
import SMCDEL.Language
import SMCDEL.Symbolic.S5
import Transparent (afterTransparent, isSuccessTransparent)
```

We use the following functions (previously defined in [Gat18]) concerning experts[2], which define the formulas "agent a is an expert" and "all agents are experts":

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

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

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

We run the following tests, in this order:

1. For agents a, b: in the initial model, a knows that b doesn't know a's secret
2. For agents a, b: after call ab, a knows b's secret
3. For agents a, b, c: after call sequence $[ab, bc]$, c knows a's secret
4. For agents a, b, c: after one call, there should be no experts
5. For agents a, b, c: after call sequence $[ab, bc, ca]$, everyone should be an expert
6. For agents a, b, c: after call ab, c knows that a knows b's secret
7. For agents a, b, c, d: after call sequence $[ab, bc]$, d knows that c knows a's secret
8. For agents a, b, c: after call sequence $[ab, bc, ca]$, everyone should know that everyone's an expert
9. For agents a, b: after call ab, b knows that a knows b's secret
10. For agents a, b, c, d: after call sequence $[ab, bc, cd, ca]$, a knows that d knows a's secret and that d knows that c knows a's secret

```
spec :: Spec
spec = do
        -- simple tests
        it "trsTrf 1: knowledge of initial state" $ do
            eval (gossipInit 2) (K "0" (Neg (has 2 1 0))) `shouldBe` True
        it "trsTrf 2: call shares secrets between agents" $ do
            eval (afterTransparent 2 [(0,1)]) (Conj [has 2 1 0, has 2 0 1]) `shouldBe` True
        it "trsTrf 3: call sequence shares secrets between agents" $ do
            eval (afterTransparent 3 [(0,1),(1,2)]) (has 3 2 0) `shouldBe` True
        it "trsTrf 4: no faulty experts" $ do
            eval (afterTransparent 3 [(0,1)]) (Disj [expert 3 i | i <- [0..2]]) `shouldBe`
                False
        it "trsTrf 5: all are experts after the correct call sequence" $ do
            isSuccessTransparent 3 [(0,1),(1,2),(0,2)] `shouldBe` True

        -- transparent-specific tests
        it "trsTrf 6: call is observed by other agents" $ do
            eval (afterTransparent 3 [(0,1)]) (K "2" (has 3 0 1)) `shouldBe` True
        it "trsTrf 7: call sequence is observed by other agents" $ do
            eval (afterTransparent 4 [(0,1),(1,2)]) (K "3" (has 3 2 0)) `shouldBe` True
```

[2] An expert is an agent who knows all secrets, that is, `expert n a` is defined as $\bigwedge\{S_a b \mid b \in [1, ..., n]\}$

```
        it "trsTrf 8: all agents know that all are experts after the correct call sequence"
            $ do
            eval (afterTransparent 3 [(0,1),(1,2),(0,2)]) (Conj [ K (show i) (allExperts 3)
                                                                  | i <- [(0::Int)..2] ]) `
                                                                    shouldBe` True
        -- general higher-order knowledge tests
        it "trsTrf 9: higher-order knowledge after one call" $ do
            eval (afterTransparent 3 [(0,1)]) (K "1" (has 3 0 1)) `shouldBe` True
        --it "trsTrf 10: higher-order knowledge after call sequence" $ do
        --      eval (afterTransparent 3 [(0,1),(1,2),(2,3),(0,2)]) (K "0" $ Conj [has 3 3
            0, K "3" (has 3 2 0)]) `shouldBe` True
```

6.3 Simple Transformer

The Simple Transformer does not satisfy the same formulas as the Classic Transformer (and the transparent variant): some instances of higher-order knowledge fail. The following tests show how the Simple Transformer differs from the other two.

```
module SimpleTransformerSpec where

import SimpleTransformer

import Test.Hspec hiding ( after )
import SMCDEL.Examples.GossipS5
import SMCDEL.Symbolic.S5
import SMCDEL.Language
```

We test the implementation of the Simple Transformer with the following tests. The first four tests (explained below) describe instances of higher-order knowledge and aren't all satisfied by the Simple Transformer, even though they should be. The other tests are identical instances from the transparent test. from As with the transparent variant, tests 5-9 encode the basic requirements of a transformer for a Gossip problem and 10-11 encode general instances of higher-order knowledge.

New tests:

1. For agents a, b, c, d: after call sequence $[ab, bc]$, c knows that a knows that b knows a's secret
2. For agents a, b, c: after call ab, c can infer that a knows b's secret (since there was only one possible call)
3. For agents a, b, c: after call sequence $[ab]$, a knows that c doesn't know b's secret
4. For agents a, b, c: after call sequence $[ab, bc, ca, ab, bc, ca]$, everyone should know that everyone's an expert

```
spec :: Spec
spec = do
        -- simple-trf-specific tests: these might fail but we'd like them to be true --
            CHECK THIS
        it "simpTrf 1: agents can reason about other agents' knowledge 1" $ do
            eval (afterSimple 4 [(0,1),(1,2)]) (K "2" (has 4 1 0)) `shouldBe` True
        it "simpTrf 2: three agents non-involed knows what call happened" $ do
            eval (afterSimple 3 [(0,1)]) (K "2" (has 3 0 1)) `shouldBe` False --
                LIMITATIONS of SimpleTrf
        it "simpTrf 3: agents can reason about the limits of other agents' knowledge" $ do
            eval (afterSimple 3 [(0,1)]) (K "0" (Neg (has 3 2 1))) `shouldBe` False --
                LIMITATIONS of SimpleTrf
```

```
    it "simpTrf 4: all agents know that all are experts after the correct call sequence
      " $ do
      eval (afterSimple 3 [(0,1),(1,2),(0,2),(0,1),(1,2),(0,2)]) (Conj [ K (show i) (
          allExperts 3)
                                                                 | i <- [(0::Int)..2] ]) `
                                                                     shouldBe` True
    -- simple tests (same tests as those for the transparent implementation)
    it "simpTrf 5: knowledge of initial state" $ do
      eval (simpleGossipInit 2) (K "0" (Neg (has 2 1 0))) `shouldBe` True
    it "simpTrf 6: call shares secrets between agents" $ do
      eval (afterSimple 2 [(0,1)]) (Conj [has 2 1 0, has 2 0 1]) `shouldBe` True
    it "simpTrf 7: call sequence shares secrets between agents" $ do
      eval (afterSimple 3 [(0,1),(1,2)]) (has 3 2 0) `shouldBe` True
    it "simpTrf 8: no faulty experts" $ do
      eval (afterSimple 3 [(0,1)]) (Disj [expert 3 i | i <- [0..2]]) `shouldBe` False
    it "simpTrf 9: all are experts after the correct call sequence" $ do
      isSuccessSimple 3 [(0,1),(1,2),(0,2)] `shouldBe` True

    -- other general higher-order knowledge tests (same tests as those for the
      transparent implementation)
    it "simpTrf 10: higher-order knowledge after one call" $ do
      eval (afterSimple 3 [(0,1)]) (K "1" (has 3 0 1)) `shouldBe` True
    -- it "simpTrf 11: higher-order knowledge after call sequence" $ do
    --     eval (afterSimple 3 [(0,1),(1,2),(2,3),(0,2)]) (K "0" $ Conj [has 3 3 0, K
      "3" (has 3 2 0)]) `shouldBe` True
```

7 Benchmarks

The primary motivation for using symbolic model checking is to provide faster computation, as explicit model checking in DEL is generally slow even for small examples [Gat18].

We therefore benchmark the runtime of the various implementations and compare them. Comparing the resuls, we can find what parts of the knowledge structure or updates on it cause the slowdown.

We execute three different call sequences, dependent on the number of agents: with a higher number of agents, we use call sequences in which more agents participate. This prevents situations in which a model containing five agents is only tested on a call sequence that concerns only a small subset of those agents, which could skew the results of the tests for models with a large number of agents.

```
module Main where

import Criterion.Main
import SimpleTransformer
import OptimizedTransformer
import Transparent
import SMCDEL.Symbolic.S5
import SMCDEL.Examples.GossipS5
import SMCDEL.Language

{-
    This module benchmarks the various transformers.
    Currently we compare
    - the SimpleTransformer (SmpTrf)
    - the ClassicTransformer in Transparent setting (TnsTrf)
    - the ClassicTransformer using the SMCDEL optimization function (OptTrf)
    - the ClasicTransformer (ClsTrf)
    The program runs updates in various settings (3,4,5 agents and 1,2,3 calls)

    * Running the Benchmark
    To run the benchmark, execute `stack bench` from the root of the project
-}
```

```haskell
-- The call sequences we apply
callsequence :: Int -> [(Int, Int)]
callsequence 3 = [(0,1),(1,2),(1,2),(0,2),(1,2)]
callsequence 4 = [(0,1),(1,2),(0,2),(2,3),(1,3)]
callsequence 5 = [(0,1),(1,2),(0,2),(3,4),(1,4)]
callsequence _ = []

-- The function we're benchmarking.
-- Simple Transformer
benchSmpTrf :: Int -> Int -> Bool
benchSmpTrf a c = evalViaBdd (afterSimple a $ take c $ callsequence a) (K "0" $ allExperts
    a)

-- Classic Transformer
benchClsTrf :: Int -> Int -> Bool
benchClsTrf a c = evalViaBdd (after a $ take c $ callsequence a) (K "0" $ allExperts a)

-- Optimized Transformer
benchOptTrf :: Int -> Int -> Bool
benchOptTrf a c = evalViaBdd (afterOpt a $ take c $ callsequence a) (K "0" $ allExperts a)

-- Transparent Transformer
benchTnsTrf :: Int -> Int -> Bool
benchTnsTrf a c = evalViaBdd (afterTransparent a $ take c $ callsequence a) (K "0" $
    allExperts a)

-- Our benchmark harness.
main :: IO ()
main = defaultMain [
  bgroup "SmpTrf - 3 agents"    [ bench "1 call"   $ whnf (benchSmpTrf 3) 1
                                , bench "3 calls"  $ whnf (benchSmpTrf 3) 3
                                , bench "5 calls"  $ whnf (benchSmpTrf 3) 5
                                ],
  bgroup "SmpTrf - 4 agents"    [ bench "1 call"   $ whnf (benchSmpTrf 4) 1
                                , bench "3 calls"  $ whnf (benchSmpTrf 4) 3
                                , bench "5 calls"  $ whnf (benchSmpTrf 4) 5
                                ],
  bgroup "SmpTrf - 5 agents"    [ bench "1 call"   $ whnf (benchSmpTrf 5) 1
                                , bench "3 calls"  $ whnf (benchSmpTrf 5) 3
                                , bench "5 calls"  $ whnf (benchSmpTrf 5) 5
                                ],
  bgroup "TnsTrf - 3 agents"    [ bench "1 call"   $ whnf (benchTnsTrf 3) 1
                                , bench "3 calls"  $ whnf (benchTnsTrf 3) 3
                                , bench "5 calls"  $ whnf (benchTnsTrf 3) 5
                                ],
  bgroup "TnsTrf - 4 agents"    [ bench "1 call"   $ whnf (benchTnsTrf 4) 1
                                , bench "3 calls"  $ whnf (benchTnsTrf 4) 3
                                , bench "5 calls"  $ whnf (benchTnsTrf 4) 5
                                ],
  bgroup "TnsTrf - 5 agents"    [ bench "1 call"   $ whnf (benchTnsTrf 5) 1
                                , bench "3 calls"  $ whnf (benchTnsTrf 5) 3
                                , bench "5 calls"  $ whnf (benchTnsTrf 5) 5
                                ],
  bgroup "OptTrf - 3 agents"    [ bench "1 call"   $ whnf (benchOptTrf 3) 1
                                , bench "3 calls"  $ whnf (benchOptTrf 3) 3
                                , bench "5 calls"  $ whnf (benchOptTrf 3) 5
                                ],
  bgroup "OptTrf - 4 agents"    [ bench "1 call"   $ whnf (benchOptTrf 4) 1
                                , bench "3 calls"  $ whnf (benchOptTrf 4) 3
                                , bench "5 calls"  $ whnf (benchOptTrf 4) 5
                                ],
  bgroup "OptTrf - 5 agents"    [ bench "1 call"   $ whnf (benchOptTrf 5) 1
                                , bench "3 calls"  $ whnf (benchOptTrf 5) 3
                                , bench "5 calls"  $ whnf (benchOptTrf 5) 5
                                ],
  bgroup "ClsTrf - 3 agents"    [ bench "1 call"   $ whnf (benchClsTrf 3) 1
                                , bench "3 calls"  $ whnf (benchClsTrf 3) 3
                                , bench "5 calls"  $ whnf (benchClsTrf 3) 5
                                ],
  bgroup "ClsTrf - 4 agents"    [ bench "1 call"   $ whnf (benchClsTrf 4) 1
                                , bench "3 calls"  $ whnf (benchClsTrf 4) 3
                                , bench "5 calls"  $ whnf (benchClsTrf 4) 5
                                ],
```

```
  bgroup "ClsTrf - 5 agents"     [ bench "1 call"   $ whnf (benchClsTrf 5) 1
                                 , bench "3 calls"  $ whnf (benchClsTrf 5) 3
                                 , bench "5 calls"  $ whnf (benchClsTrf 5) 5
                                 ]
  ]
```

7.1 Benchmarking Results

We compared the performance of the Classic Transformer, the Optimized Classic Transformer, the Transparent variant and the Simple Transformer. The Optimized Transformer timed out at all runs and was therefore not included in the results. The relevant results of the other tests are discussed below.

The benchmarks evaluate the average running time needed to execute call sequences of different lengths, on models containing respectively three, four, and five agents. Below we highlight the results for three and five agents; for a complete documentation, we refer to the Appendix.

Table 1 compares the results on models containing three agents.

Table 1: Call sequences on models containing three agents

Nr. of calls	Classic	Transparent	Simple
1	388.4 μs	146.0 μs	88.29 μs
3	1.308 ms	331.4 μs	99.34 μs
5	1.876 ms	491.2 μs	486.2 μs

To illustrate the differences between the models on larger problems, the following table compares the results on models containing five agents.

Table 2: Call sequences on models containing five agents

Nr. of calls	Classic	Transparent	Simple
1	18.33 ms	477.0 μs	638.0 μs
3	63.01 ms	1.323 ms	2.153 ms
5	12.27 s	1.831 ms	2.096 ms

We see that the differences in performance grow with the number of agents and the length of call sequences: on larger models, the Transparent and Simple implementation are significantly faster than the Classic implementation. This is most apparent in the results for five calls between five agents (see table 2).

8 Conclusion

This project looked into how we can use the SMCDEL library to better understand and model Gossip. To the first point, we wrote `gsi`, our *Gossip Scene Investigation* function to better understand SMCDEL's Knowledge Scenes as they pertain to The Gossip Problem. To the latter point, we used the existing notion of a Knowledge Transformer in SMCDEL to write a Knowledge Transformer for the Transparent Gossip Problem. Both of these aforementioned processes helped us build a strong understanding of how SMCDEL approaches Gossip, and specifically what makes it so computationally intensive. With this in mind, we tried using a pre-existing optimize function within SMCDEL's library to reduce complexity, as well as writing our own Simple Transformer, to target the blow up in vocabulary that the Classic Transformer implemented in SMCDEL encrues.

There are two ways we can analyze our work; by considering how fast it is and how correct it is. In Section X (add Benchmarks), we explore the first point. As we saw, the Transparent Transformer and the Simple Transformer both had large improvements on computation time, specifically as the number of calls increased.

On the other hand, the correctness of our code still has room for improvement. We believe knowing the true differences between what information these transformers codify requires mathematics outside the scope of a programming project. However, it is our belief that the Simple Transformer makes *less* propositions true than the Classic Transformer, and therefore does not tell false truths.

In terms of further work, readability and correctness is a big focus. The Gossip Problem is a specific example within the area of Dynamic Epistemic Logic (DEL), and is therefore pretty hard to work with from afar. This is part of the reason we wrote the `gsi` function, and we advise the reader to test its usability by running `main`. However, although we decode the propositional variables and observations, the state law is still a large BDD, and uninterpretable by the user. Future work could be done to make this more user friendly, perhaps by way of `graphviz` (Cite graphviz?).

References

[Gat18] Malvin Gattinger. *New Directions in Model Checking Dynamic Epistemic Logic*. PhD thesis, University of Amsterdam, 2018.

[Gat23] Malvin Gattinger. GoMoChe-gossip model checking. *Branch async from*, 1, 2023.

[Rei23] Daniel Reifsteck. Comparing state representations for del planning (not public). Master's thesis, University of Freiburg, April 2023.

[Yuk23] Haitian Yuki. SMCDEL-Hanabi. unpublished code / private git repository, 2023.