Lecture 2, part 2: GoMoChe

Knowledge and Gossip — ESSLLI 2022

Malvin Gattinger (ILLC, Amsterdam)

2022-08-09, Galway

https://malv.in/2022/gossip/

Preface / slide for the break: How to get and run GoMoChe

Option A:

- GitHub account required! 😞
- Open https://github.com/m4lvin/GoMoChe, click on the *GitPod* link, log in and **wait**.

Option B:

- Haskell and `stack` required! 😺
- Do this:

```
git clone https://github.com/m4lvin/GoMoChe.git
cd GoMoChe
stack ghci
```

Motivation

Theory

Implementation

Examples

Motivation

Gomoche

Paulo Grobel: L'Ermitage de Gomoche

Motivation

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?

Motivation

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?
- Is the call sequence $ab; cd; ac; bd$ successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?

Motivation

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?
- Is the call sequence $ab; cd; ac; bd$ successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?
- Given the gossip graph below, how many LNS sequences are (un)successful?

Motivation

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?
- Is the call sequence $ab; cd; ac; bd$ successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?
- Given the gossip graph below, how many LNS sequences are (un)successful?

- After the sequence $ab; bc; cd; bd$, does agent a know that if they call agent b then b will tell a the secret d?

Motivation

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?
- Is the call sequence $ab; cd; ac; bd$ successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?
- Given the gossip graph below, how many LNS sequences are (un)successful?

- After the sequence $ab; bc; cd; bd$, does agent a know that if they call agent b then b will tell a the secret d?

Model Checking

Given a model $\mathcal{M}$ and a formula φ, do we have $\mathcal{M} \vDash \varphi$ or not?

Model Checking

Given a model $\mathcal{M}$ and a formula φ, do we have $\mathcal{M} \vDash \varphi$ or not?

In our case:

Given an initial gossip graph G and a call sequence σ, do we have $G, \sigma \vDash \varphi$ or not?

Theory

Syntax

The *language* of protocol-dependent knowledge:

$$\varphi ::= \top \mid N_i i \mid S_i i \mid C_i i \mid i = i \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^P \varphi \mid [\pi]\varphi$$

$$\pi ::= ?\varphi \mid ii \mid \pi;\pi \mid \pi \cup \pi \mid \pi^*$$

Syntax

The *language* of protocol-dependent knowledge:

$$\varphi ::= \top \mid N_i i \mid S_i i \mid C_i i \mid i = i \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^P \varphi \mid [\pi]\varphi$$

$$\pi ::= ?\varphi \mid ii \mid \pi;\pi \mid \pi \cup \pi \mid \pi^*$$

Definition

A *protocol* is a function P mapping any agent pair ab to a formula P_{ab} called the *protocol condition*.

Syntax

The *language* of protocol-dependent knowledge:

$$\varphi ::= \top \mid N_i i \mid S_i i \mid C_i i \mid i = i \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^P \varphi \mid [\pi]\varphi$$

$$\pi ::= ?\varphi \mid ii \mid \pi;\pi \mid \pi \cup \pi \mid \pi^*$$

Definition

A *protocol* is a function P mapping any agent pair ab to a formula P_{ab} called the *protocol condition*.

Example

The *Learn New Secrets* (LNS) protocol is $LNS_{ab} := \neg S_a b$.

Syntax

The *language* of protocol-dependent knowledge:

$$\varphi ::= \top \mid N_i i \mid S_i i \mid C_i i \mid i = i \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^P \varphi \mid [\pi]\varphi$$

$$\pi ::= ?\varphi \mid ii \mid \pi;\pi \mid \pi \cup \pi \mid \pi^*$$

Definition

A *protocol* is a function P mapping any agent pair ab to a formula P_{ab} called the *protocol condition*.

Example

The *Learn New Secrets* (LNS) protocol is $LNS_{ab} := \neg S_a b$. The *soft look-ahead strengthening* of LNS is $LNS^{\blacklozenge} := LNS_{ab} \wedge \hat{K}_a^{LNS}[ab]\langle P\rangle \bigwedge_{x,y} S_x y$. (See Lecture 4.)

Semantics

A *state* is a tuple (G, σ) where $G = (A, N, S)$ is an initial graph and σ a call sequence.

Let N^σ and S^σ be the resulting relations after executing σ.

Semantics

A *state* is a tuple (G, σ) where $G = (A, N, S)$ is an initial graph and σ a call sequence.

Let N^σ and S^σ be the resulting relations after executing σ.

$$
\begin{array}{lll}
G, \sigma \vDash N_x y & :\Leftrightarrow & (x, y) \in N^\sigma \\
G, \sigma \vDash S_x y & :\Leftrightarrow & (x, y) \in S^\sigma \\
G, \sigma \vDash C_x y & :\Leftrightarrow & xy \in \sigma \text{ or } yx \in \sigma \\
G, \sigma \vDash x = y & :\Leftrightarrow & x = y \\
G, \sigma \vDash K_a^P \varphi & \text{iff} & G, \sigma' \vDash \varphi \text{ for all } (G, \sigma') \sim_a^P (G, \sigma) \\
G, \sigma \vDash [\pi]\varphi & \text{iff} & G, \sigma' \vDash \varphi \text{ for all } (G, \sigma') \in [\![\pi]\!](G, \sigma)
\end{array}
$$

Semantics

A *state* is a tuple (G, σ) where $G = (A, N, S)$ is an initial graph and σ a call sequence.

Let N^σ and S^σ be the resulting relations after executing σ.

$$
\begin{array}{lll}
G, \sigma \vDash N_x y & :\Leftrightarrow & (x, y) \in N^\sigma \\
G, \sigma \vDash S_x y & :\Leftrightarrow & (x, y) \in S^\sigma \\
G, \sigma \vDash C_x y & :\Leftrightarrow & xy \in \sigma \text{ or } yx \in \sigma \\
G, \sigma \vDash x = y & :\Leftrightarrow & x = y \\
G, \sigma \vDash K_a^P \varphi & \text{iff} & G, \sigma' \vDash \varphi \text{ for all } (G, \sigma') \sim_a^P (G, \sigma) \\
G, \sigma \vDash [\pi]\varphi & \text{iff} & G, \sigma' \vDash \varphi \text{ for all } (G, \sigma') \in [\![\pi]\!](G, \sigma)
\end{array}
$$

$$
\begin{array}{rcl}
[\![?\varphi]\!](G, \sigma) & := & \{(G, \sigma) \mid G, \sigma \vDash \varphi\} \\
[\![ab]\!](G, \sigma) & := & \{(G, (\sigma; ab)) \mid G, \sigma \vDash N_a b\} \\
[\![\pi; \pi']\!](G, \sigma) & := & \bigcup \{[\![\pi']\!](G, \sigma') \mid (G, \sigma') \in [\![\pi]\!](G, \sigma)\} \\
[\![\pi \cup \pi']\!](G, \sigma) & := & [\![\pi]\!](G, \sigma) \cup [\![\pi']\!](G, \sigma) \\
[\![\pi^*]\!](G, \sigma) & := & \bigcup \{[\![\pi^n]\!](G, \sigma) \mid n \in \mathbb{N}\}
\end{array}
$$

Protocol-dependent Epistemic Alternatives

Definition

For any agent a and protocol P let $\sim_a^P$ be the smallest relation such that:

- $(G, \epsilon) \sim_a^P (G, \epsilon)$;
- if $(G, \sigma) \sim_a^P (G, \tau)$, $N_b^\sigma = N_b^\tau$, $S_b^\sigma = S_b^\tau$, and $G, \sigma \models P_{ab}$ and $G, \tau \models P_{ab}$, then $(G, \sigma; ab) \sim_a^P (G, \tau; ab)$;
  if $(G, \sigma) \sim_a^P (G, \tau)$, $N_b^\sigma = N_b^\tau$, $S_b^\sigma = S_b^\tau$, and $G, \sigma \models P_{ba}$ and at $G, \tau \models P_{ab}$, then $(G, \sigma; ba) \sim_a^P (G, \tau; ba)$;
- if $(G, \sigma) \sim_a^P (G, \tau)$ and $a \notin \{c, d, e, f\}$ such that $G, \sigma \models P_{cd}$ and $G, \tau \models P_{ef}$, then $(G, \sigma; cd) \sim_a^P (G, \tau; ef)$.

Protocol-dependent Epistemic Alternatives

Definition

For any agent a and protocol P let $\sim_a^P$ be the smallest relation such that:

- $(G, \epsilon) \sim_a^P (G, \epsilon)$;
- if $(G, \sigma) \sim_a^P (G, \tau)$, $N_b^\sigma = N_b^\tau$, $S_b^\sigma = S_b^\tau$, and $G, \sigma \models P_{ab}$ and $G, \tau \models P_{ab}$, then $(G, \sigma; ab) \sim_a^P (G, \tau; ab)$;
  if $(G, \sigma) \sim_a^P (G, \tau)$, $N_b^\sigma = N_b^\tau$, $S_b^\sigma = S_b^\tau$, and $G, \sigma \models P_{ba}$ and at $G, \tau \models P_{ab}$, then $(G, \sigma; ba) \sim_a^P (G, \tau; ba)$;
- if $(G, \sigma) \sim_a^P (G, \tau)$ and $a \notin \{c, d, e, f\}$ such that $G, \sigma \models P_{cd}$ and $G, \tau \models P_{ef}$, then $(G, \sigma; cd) \sim_a^P (G, \tau; ef)$.

Note: This is synchronous!

Avoiding Russel's Protocol

Protocol(condition)s may not refer to themselves!

That is, we do *not* allow this:

$$P_{ab} := \ldots K_a^P \ldots$$

(see exercise)

Implementation

Learning Haskell in one slide

A function `f` with the **type** `a -> b`:

```
f :: a -> b
f x = x + x
```

Learning Haskell in one slide

A function `f` with the **type** `a -> b`:

```
f :: a -> b
f x = x + x
```

Note: no parentheses for function application!

```
GHCI> f 10 + 3
23
GHCI> map f [1,2,3]
[2,4,6]
```

Learning Haskell in one slide

A function `f` with the **type** `a -> b`:

```
f :: a -> b
f x = x + x
```

Note: no parentheses for function application!

```
GHCI> f 10 + 3
23
GHCI> map f [1,2,3]
[2,4,6]
```

A definition of Boolean Formulas and tuples thereof:

```
data BForm = Atom Int | Not BForm | And BForm BForm | Or BForm BForm
type MyPair = (BForm, BForm)
```

Syntax in Haskell

```
data Form
  = N Agent Agent
  | S Agent Agent
  | C Agent Agent
  | Same Agent Agent
  | Top
  | Neg Form
  | Conj [Form]
  | Disj [Form]
  | K Agent Protocol Form
  | HatK Agent Protocol Form
  | Box Prog Form
  | Dia Prog Form
  | ForallAg FormWithAgentVar
  | ExistsAg FormWithAgentVar
```

Syntax in Haskell

```
data Form
  = N Agent Agent
  | S Agent Agent
  | C Agent Agent
  | Same Agent Agent
  | Top
  | Neg Form
  | Conj [Form]
  | Disj [Form]
  | K Agent Protocol Form
  | HatK Agent Protocol Form
  | Box Prog Form
  | Dia Prog Form
  | ForallAg FormWithAgentVar
  | ExistsAg FormWithAgentVar
```

```
data Prog = Test Form
          | Call Agent Agent
          | Seq [Prog]
          | Cup [Prog]
          | CupAg ProgWithAgentVar
          | Star Prog
```

Graphs in Haskell

```haskell
type Agent = Int

type Relation = IntMap IntSet

type Graph = (Relation, Relation)

type Call = (Agent,Agent)
type Sequence = [Call]

type State = (Graph,Sequence)
```

Making Graphs

```
GoMoChe> totalInit 3
(fromList [(0,fromList [0,1,2]),(1,fromList [0,1,2]),(2,fromList [0,1,2])]
,fromList [(0,fromList [0]),(1,fromList [1]),(2,fromList [2])])
```

Making Graphs

```
GoMoChe> totalInit 3
(fromList [(0,fromList [0,1,2]),(1,fromList [0,1,2]),(2,fromList [0,1,2])]
,fromList [(0,fromList [0]),(1,fromList [1]),(2,fromList [2])])

GoMoChe> ppGraphShort (totalInit 3)
"Abc.aBc.abC"
GoMoChe> ppGraphShort (totalInit 4)
"Abcd.aBcd.abCd.abcD"
```

Making Graphs

```
GoMoChe> totalInit 3
(fromList [(0,fromList [0,1,2]),(1,fromList [0,1,2]),(2,fromList [0,1,2])]
,fromList [(0,fromList [0]),(1,fromList [1]),(2,fromList [2])])

GoMoChe> ppGraphShort (totalInit 3)
"Abc.aBc.abC"
GoMoChe> ppGraphShort (totalInit 4)
"Abcd.aBcd.abCd.abcD"
```

More generally:

```
GoMoChe> :t parseGraph
parseGraph :: String -> Graph
GoMoChe> ppGraphShort (parseGraph "01-12-231-3 I4")
"Ab.Bc.bCd.D"
```

Semantics in Haskell

The following main model checking function implements $\models$.

```
eval :: State -> Form -> Bool
eval state (N a b)      = b `IntSet.member` (fst (uncurry calls state) `at` a)
eval state (S a b)      = b `IntSet.member` (snd (uncurry calls state) `at` a)
eval state (C a b)      = (a,b) `elem` snd state
eval _     (Same a b)   = a == b
eval _     Top          = True
eval state (Neg f)      = not $ eval state f
eval state (Conj fs)    = all (eval state) fs
eval state (Disj fs)    = any (eval state) fs
eval state (K    a p f) = all (`eval` f) (epistAlt a p state)
eval state (HatK a p f) = any (`eval` f) (epistAlt a p state)
eval state (Box p f)    = all (`eval` f) (Set.toList $ runs state p)
eval state (Dia p f)    = any (`eval` f) (Set.toList $ runs state p)
eval state (ForallAg f) = all (eval state . f) (agentsOf $ fst state)
eval state (ExistsAg f) = any (eval state . f) (agentsOf $ fst state)
```

Easy Example

We also define `|=` as an infix alias of `eval` looking more like $\models$.

```
GoMoChe> (totalInit 4, [])  |=  S 1 1
True

GoMoChe> (totalInit 4, [])  |=  S 1 2
False

GoMoChe> (totalInit 4, [(1,2)])  |=  S 1 2
True
```

Epistemic equivalences in Haskell

```haskell
epistAlt :: Agent -> Protocol -> State -> [State]
epistAlt _ _      (g, []     ) = [ (g, []) ] -- initial graph is common knowledge!
epistAlt a proto (g, history) =
  let (prev, lastevent) = (init history, last history)
      lastcall@(x,y)    = lastevent
  in sort $
    if a `isin` lastevent
       then [ (g',althist ++ [lastcall]) -- alternative histories and same last call
            | (g',althist) <- epistAlt a proto (g,prev)
            , eval (g',prev) (proto lastcall)
            , eval (g',althist) (proto lastcall)
            , localSameFor x (calls g' althist) (calls g prev) -- inspect-then-merge!
            , localSameFor y (calls g' althist) (calls g prev) ]
       else [ (g',cs'++[altevent]) -- alternative histories and alternative last calls
            | (g',cs') <- epistAlt a proto (g,prev)
            , altevent <- allowedCalls proto (g',cs')
            , not $ a `isin` altevent ]
```

Epistemic equivalences in Haskell

```haskell
epistAlt :: Agent -> Protocol -> State -> [State]
epistAlt _ _      (g, []     ) = [ (g, []) ] -- initial graph is common knowledge!
epistAlt a proto (g, history) =
  let (prev, lastevent) = (init history, last history)
      lastcall@(x,y)    = lastevent
  in sort $
    if a `isin` lastevent
       then [ (g',althist ++ [lastcall]) -- alternative histories and same last call
            | (g',althist) <- epistAlt a proto (g,prev)
            , eval (g',prev) (proto lastcall)
            , eval (g',althist) (proto lastcall)
            , localSameFor x (calls g' althist) (calls g prev) -- inspect-then-merge!
            , localSameFor y (calls g' althist) (calls g prev) ]
       else [ (g',cs'++[altevent]) -- alternative histories and alternative last calls
            | (g',cs') <- epistAlt a proto (g,prev)
            , altevent <- allowedCalls proto (g',cs')
            , not $ a `isin` altevent ]
```

The above implements the synchronous $\sim_i$ as above. See the `async` branch if you are curious!

List of useful functions

Making and showing graphs: `totalInit`, `parseGraph`, `ppGraph`

Formula abbreviations: `con`, `dis`, `expert`, `allExperts`, `superExperts`

Predefined protocols: `anyCall`, `lns`, `cmo`, `pig`

List of useful functions

Making and showing graphs: `totalInit`, `parseGraph`, `ppGraph`

Formula abbreviations: `con`, `dis`, `expert`, `allExperts`, `superExperts`

Predefined protocols: `anyCall`, `lns`, `cmo`, `pig`

Hint: Look up things with `:i whatever` and use TAB completion in the terminal!

Life should be easy

Additional convenience functions

```
isSuperSuccSequence :: Protocol -> State -> Sequence -> Bool
isSuperSuccSequence proto (g,sigma) cs =
  (g, sigma ++ cs)  |=  ForallAg (`superExpert` proto)
```

Life should be easy

Additional convenience functions

```
isSuperSuccSequence :: Protocol -> State -> Sequence -> Bool
isSuperSuccSequence proto (g,sigma) cs =
  (g, sigma ++ cs)  |=  ForallAg (`superExpert` proto)

statistics :: Protocol -> State -> (Int,Int)
statistics proto (g,sigma) =
  (length succSequ, length sequ - length succSequ) where
    sequ = sequences proto (g,sigma) \\ [[]]
    succSequ = filter (isSuccSequence (g,sigma)) sequ
```

Examples

Answering “Motivation” question 1

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?

Answering “Motivation” question 1

- After the call sequence $ab; bc; ac$, does agent a know that agent b knows the secret of agent c?

```
GoMoChe> eval (totalInit 3, parseSequence "ab;bc;ac") (K 0 anyCall (S 1 2))
True
```

Answering “Motivation” question 2

- Is the call sequence $ab; cd; ac; bd$ successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?

Answering “Motivation” question 2

- Is the call sequence *ab*; *cd*; *ac*; *bd* successful, i.e. do all agents know all secrets afterwards? Moreover, is it super successful, i.e. do all agents know that all agents know all secrets?

```
GoMoChe> isSuccSequence (totalInit 4,[]) (parseSequence "ab;cd;ac;bd")
True
GoMoChe> isSuperSuccSequence lns (totalInit 4,[]) (parseSequence "ab;cd;ac;bd"
False
```

Answering “Motivation” question 3

- Given the gossip graph below, how many LNS sequences are (un)successful?

Answering “Motivation” question 3

- Given the gossip graph below, how many LNS sequences are (un)successful?


```
GoMoChe> statistics lns (parseGraph "01-12-231-3 I4",[])
(57,20)
```

Answering “Motivation” question 4

- After the sequence $ab; bc; cd; bd$, does agent a know that if they call agent b then b will tell a the secret d? That is, do we have $ab; bc; cd; bd \vDash K_a[ab]S_a d$ or not?

Answering “Motivation” question 4

- After the sequence $ab; bc; cd; bd$, does agent a know that if they call agent b then b will tell a the secret d? That is, do we have $ab; bc; cd; bd \vDash K_a[ab]S_a d$ or not?

```
GoMoChe> eval (totalInit 4, parseSequence "ab;bc;cd;bd") (K 0 anyCall (Box (Call 0 1) (S 0 3)))
False
GoMoChe> eval (totalInit 4, parseSequence "ab;bc;cd;bd") (K 0 lns (Box (Call 0 1) (S 0 3)))
True
```

Answering “Motivation” question 4

- After the sequence $ab; bc; cd; bd$, does agent a know that if they call agent b then b will tell a the secret d? That is, do we have $ab; bc; cd; bd \models K_a[ab]S_a d$ or not?

```
GoMoChe> eval (totalInit 4, parseSequence "ab;bc;cd;bd") (K 0 anyCall (Box (Call 0 1) (S 0 3)))
False
GoMoChe> eval (totalInit 4, parseSequence "ab;bc;cd;bd") (K 0 lns (Box (Call 0 1) (S 0 3)))
True
```

Answer: it depends!

We have $ab; bc; cd; bd \not\models K_a^{\mathsf{ANY}}[ab]S_a d$ but $ab; bc; cd; bd \models K_a^{\mathsf{LNS}}[ab]S_a d$.

Complex Example: Knowledge Overviews

```
knowledgeOverview :: State -> Protocol -> IO ()
```

This generates tables such as these:

```
GoMoChe> knowledgeOverview (totalInit 4, parseSequence "ab;bc;cd;da;ab") anyCall
    a           b           c           d
ab  ab          ab          c           d
bc  ab          abc         abc         d
cd  ab          abc         abcd   CD   abcd   CD
da  abcd A  D   abc         abcd   CD   abcd A CD
ab  abcd ABCD   abcd AB     abcd   CD   abcd A CD
```

Complex Example: Knowledge Overviews

```
knowledgeOverview :: State -> Protocol -> IO ()
```

This generates tables such as these:

```
GoMoChe> knowledgeOverview (totalInit 4, parseSequence "ab;bc;cd;da;ab") anyCall
    a           b           c           d
ab  ab          ab          c           d
bc  ab          abc         abc         d
cd  ab          abc         abcd   CD   abcd   CD
da  abcd A  D   abc         abcd   CD   abcd A CD
ab  abcd ABCD   abcd AB     abcd   CD   abcd A CD
```

```
GoMoChe> knowledgeOverview (totalInit 4, parseSequence "ab;bc;cd;da;ab") lns
    a           b           c           d
ab  ab          ab          c           d
bc  ab          abc         abc         d
cd  ab          abc         abcd   CD   abcd   CD
da  abcd _ __   abc         abcd   CD   abcd _ __
ab  abcd ____   abcd ____   abcd ABCD   abcd ____
```

Reference, Links, Exercises

- *GoMoChe: Gossip Model Checking*, extended abstract, to be presented at LAMAS&SR 2022, Rennes. https://malv.in/2022/LAMASSR-GoMoChe.pdf
- Appendix C of *Everyone knows that everyone knows: gossip protocols for super experts*, submitted. https://arxiv.org/pdf/2011.13203.pdf#page=37

- See course website for exercises!
- Further examples: test/results.hs, (run them with `stack test`).

- Please report bugs and provide feedback — click here for a form.


Críoch
END