

Lecture 2, part 3: GoMoChe

Knowledge and Gossip — ESSLLI 2025

Malvin Gatteringer (ILLC, Amsterdam)

2025-07-28, Bochum

<https://malv.in/2025/esslli-gossip/>

Preface: 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

Execution Trees

Examples

Motivation



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 $\mathcal{M} \models \varphi$ or not?

Model Checking

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

In our case:

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

Model Checking

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

In our case:

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

And φ will be something that says:

- “x knows the secret of y”
- “Everyone is an expert.”
- “The protocol XYZ does not allow any more calls.”
- etc.

Theory

The *language* we use:

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

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

The *language* we use:

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

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

Definition

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

The *language* we use:

$$\varphi ::= \top \mid N_{ii} \mid S_{ii} \mid C_{ii} \mid i = i \mid \neg\varphi \mid \varphi \wedge \varphi \mid K_i^P \varphi \mid [\pi]\varphi$$

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

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 $LNS_{ab} := \neg S_a b$.

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 .

$$G, \sigma \models N_x y \quad :\Leftrightarrow \quad (x, y) \in N^\sigma$$

$$G, \sigma \models S_x y \quad :\Leftrightarrow \quad (x, y) \in S^\sigma$$

$$G, \sigma \models C_x y \quad :\Leftrightarrow \quad xy \in \sigma \text{ or } yx \in \sigma$$

$$G, \sigma \models x = y \quad :\Leftrightarrow \quad x = y$$

$$G, \sigma \models K_a^P \varphi \quad \text{iff} \quad G, \sigma' \models \varphi \text{ for all } (G, \sigma') \sim_a^P (G, \sigma)$$

$$G, \sigma \models [\pi] \varphi \quad \text{iff} \quad G, \sigma' \models \varphi \text{ for all } (G, \sigma') \in \llbracket \pi \rrbracket (G, \sigma)$$

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 .

$$G, \sigma \models N_x y \quad :\Leftrightarrow \quad (x, y) \in N^\sigma$$

$$G, \sigma \models S_x y \quad :\Leftrightarrow \quad (x, y) \in S^\sigma$$

$$G, \sigma \models C_x y \quad :\Leftrightarrow \quad xy \in \sigma \text{ or } yx \in \sigma$$

$$G, \sigma \models x = y \quad :\Leftrightarrow \quad x = y$$

$$G, \sigma \models K_a^P \varphi \quad \text{iff} \quad G, \sigma' \models \varphi \text{ for all } (G, \sigma') \sim_a^P (G, \sigma)$$

$$G, \sigma \models [\pi] \varphi \quad \text{iff} \quad G, \sigma' \models \varphi \text{ for all } (G, \sigma') \in \llbracket \pi \rrbracket (G, \sigma)$$

$$\llbracket ?\varphi \rrbracket (G, \sigma) \quad := \quad \{(G, \sigma) \mid G, \sigma \models \varphi\}$$

$$\llbracket ab \rrbracket (G, \sigma) \quad := \quad \{(G, (\sigma; ab)) \mid G, \sigma \models N_a b\}$$

$$\llbracket \pi; \pi' \rrbracket (G, \sigma) \quad := \quad \bigcup \{ \llbracket \pi' \rrbracket (G, \sigma') \mid (G, \sigma') \in \llbracket \pi \rrbracket (G, \sigma) \}$$

$$\llbracket \pi \cup \pi' \rrbracket (G, \sigma) \quad := \quad \llbracket \pi \rrbracket (G, \sigma) \cup \llbracket \pi' \rrbracket (G, \sigma)$$

$$\llbracket \pi^* \rrbracket (G, \sigma) \quad := \quad \bigcup \{ \llbracket \pi^n \rrbracket (G, \sigma) \mid n \in \mathbb{N} \}$$

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:

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

(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

```
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

```
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

```
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`

Use total graph and parse sequence: `after`

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

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

List of useful functions

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

Use total graph and parse sequence: `after`

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

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

Hakell hints:

- Use TAB completion in the terminal!
- Use `:t whatever` to see the type of things.
- Get more information about things with `:i whatever`

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) =
```

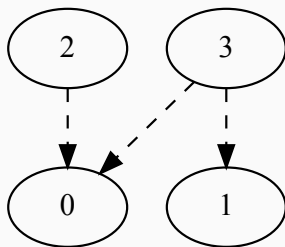
```
  (length succSequ, length sequ - length succSequ) where
```

```
    sequ = sequences proto (g,sigma) \\ [[]]
```

```
    succSequ = filter (isSuccSequence (g,sigma)) sequ
```

Execution Trees

Running Example



nExample

Execution Trees in GoMoChe

```
GoMoChe> :i tree
```

```
tree :: Protocol -> State -> ExecutionTree
```

Execution Trees in GoMoChe

```
GoMoChe> :i tree
```

```
tree :: Protocol -> State -> ExecutionTree
```

```
GoMoChe> tree lns (nExample,[])
```

```
0-1-02-013 I4
```

```
(2,0): 02-1-02-013 02-1-02-3
```

```
(2,0)(3,0): 0123-1-02-0123 023-1-02-023
```

```
(2,0)(3,0)(0,1): 0123-0123-02-0123 0123-0123-02-023
```

```
(2,0)(3,0)(0,1)(3,1): 0123-0123-02-0123 0123-0123-02-0123
```

```
(2,0)(3,0)(3,1): 0123-0123-02-0123 023-0123-02-0123
```

```
(2,0)(3,0)(3,1)(0,1): 0123-0123-02-0123 0123-0123-02-0123
```

```
(2,0)(3,1): 02-013-02-013 02-13-02-13
```

```
...
```

Show me the tree!

Using the wonderful GraphViz library (<https://graphviz.org/>) we can also actually draw the tree.

```
pdf $ tree lns (nExample, [])
```

(Not fitting on slide; run it yourself and zoom in to actually see something the resulting picture!)

Execution Trees with epistemic edges

We can further annotate the tree with indistinguishability relation(s).

```
GoMoChe> :i pdfTreeWith
```

```
pdfTreeWith :: [Agent] -> Int -> Int -> Protocol -> ExecutionTree -> IO ()
```

(The first Int limits the depths of the tree, the second controls the arrangement of nodes.)

Execution Trees with epistemic edges

We can further annotate the tree with indistinguishability relation(s).

```
GoMoChe> :i pdfTreeWith
```

```
pdfTreeWith :: [Agent] -> Int -> Int -> Protocol -> ExecutionTree -> IO ()
```

(The first Int limits the depths of the tree, the second controls the arrangement of nodes.)

```
GoMoChe> pdfTreeWith [2] 2 1 lns (tree lns (nExample,[]))
```


Execution Trees with epistemic edges

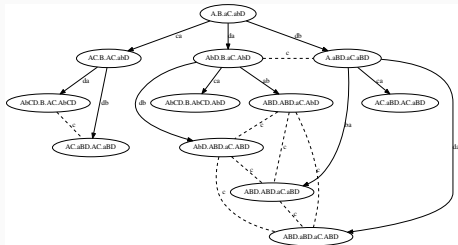
We can further annotate the tree with indistinguishability relation(s).

```
GoMoChe> :i pdfTreeWith
```

```
pdfTreeWith :: [Agent] -> Int -> Int -> Protocol -> ExecutionTree -> IO ()
```

(The first Int limits the depths of the tree, the second controls the arrangement of nodes.)

```
GoMoChe> pdfTreeWith [2] 2 1 lns (tree lns (nExample,[]))
```




Cheap optimization tricks for search problems

- W.l.o.g. fix the first call to be ab .

Cheap optimization tricks for search problems

- W.l.o.g. fix the first call to be ab .
- Fix the first *two* calls to be either $ab; bc$ or to be $ab; cd$.

Wait, why is this w.l.o.g.? 

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> after "ab;bc;ac"  |=  Kn a anyCall (S b c)
```

```
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> after "ab;cd;ac;bd" |= allExperts
```

```
True
```

```
GoMoChe> after "ab;cd;ac;bd" |= allSuperExperts lns
```

```
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 $ab; bc; cd; bd \models K_a[ab]S_a d$ 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 $ab; bc; cd; bd \models K_a[ab]S_a d$ or not?

```
GoMoChe> after "ab;bc;cd;bd" |= K a anyCall (Box (Call a b) (S a d))
```

```
False
```

```
GoMoChe> after "ab;bc;cd;bd" |= K a lns (Box (Call a b) (S a d))
```

```
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> after "ab;bc;cd;bd" |= K a anyCall (Box (Call a b) (S a d))
```

```
False
```

```
GoMoChe> after "ab;bc;cd;bd" |= K a lns (Box (Call a b) (S a d))
```

```
True
```

Answer: it depends!

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

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
da	abcd A D	abc	abcd	CD
ab	abcd ABCD	abcd AB	abcd	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
da	abcd	A D	abc	abcd
ab	abcd	ABCD	abcd	AB

```
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
da	abcd	_ _	abc	abcd
ab	abcd	----	abcd	ABCD

- *GoMoChe: Gossip Model Checking*, extended abstract LAMAS&SR 2022, Rennes.
<https://malv.in/2022/LAMASSR-GoMoChe.pdf>
- Appendix C of *Everyone knows that everyone knows: gossip protocols for super experts*.
<https://arxiv.org/pdf/2011.13203.pdf#page=37>
- Further examples: `test/results.hs`, (run them with `stack test`).