Examples of Applications of Higher Order Theorem Proving

Content

- Applications of Higher Order Theorem Proving :
  - Verification of distributed algorithms
  - Verification of cryptographic protocols

- Notes:
  - In both the approach is by *embedding* suitable logics in HOL
  - We can handle infinite state space ☺
  - Expect lots of manual proofs.
  - But we can still program heuristics to eliminate trivialities and frequently occurring subgoals.
- Challenges:
  - How to represent in HOL ?
  - How to automate ?

Embedding a Logic for Distributed Systems

UNITY

- Based on UNITY, proposed by Chandy and Misra, 1988, in *Parallel Program Design: a Foundation*.

  Later, 2001, becomes Seuss, with a bit OO-flavour in: *A Dicipline of Multiprogramming: Programming Theory for Distributed Applications*

- Unlike LTL, UNITY defines its logic Axiomatically:
  - more abstract (so easier to understand).
  - more suitable for deductive style of proving
  - with HOL support good for verifying (high level) algorithms
  - not very good to handle models at e.g. Promela level.

UNITY Program & Execution

- A program P is (simplified) a pair *(Init,A)*

  Init : a predicate specifying allowed initial states
  A : a set of concurrent (atomic and guarded) actions

- Execution model :

  - Each action α is executed atomically. Only when its guard is enabled (true), α can be selected for execution.

  - A run of P is *infinite*. At each step an enabled action is *non-deterministically* selected for execution. The run has to be *weakly fair:* when an action is continuously enabled, it will eventually be selected. When no action is enabled, the system stutters (does a skip).

However the logic is axiomatic. It will not actually construct the runs. → next slides.

Parallel composition

- Can be expressed straight forwardly :

$$(Init_1, A_1) \;\; [] \;\; (Init_2, A_2) \quad = \quad (Init_1 \wedge Init_1 \;,\; A_1 \cup A_2)$$

Temporal properties

- Safety is expressed by this operator:

$$(Init,A) \;\; |\!- \;\; p \;\; \underline{unless} \;\; q \;\; = \;\; \forall \alpha \in A. \; \{ \, p \wedge \neg q \, \} \;\; \alpha \;\; \{ \, p \vee q \, \}$$

Whenever p holds the program will either stay in p, or go over to q.

- Embedding this in HOL is straight forward:

Recall that we have chosen to represent predicates with functions State→bool. So e.g. boolean /\ can't be used to conjuct them.

```
Define  `unless  (Init,A)  p  q
          =
          ∀α.  α∈A  ⇒  HOARE  (p AND NOT q)  α  (p OR q)`
```

- NOT, AND, OR → lifting ~, /\, \/ to function space, e.g.

```
Define  `p  AND  q   =   ( λs.  p  s  /\ q  s )`
```

Progress-1

- A predicate *p* is transient in *P=(Init,A)* if there is an action in *A* that can make it false.

$$(I,A) \ \vdash \ \underline{transient}\ p \ = \ \exists \alpha \in A.\ \{ p \}\ \alpha\ \{ \neg p \}$$

- Now define:

$$\begin{aligned}(I,A) \ \vdash \ p\ \underline{ensures}\ q \ = \ & (I,A) \ \vdash \ p\ \underline{unless}\ q \\ & \text{and } (I,A) \ \vdash \ \underline{transient}\ (p \wedge \neg q)\end{aligned}$$

  The *weak fairness* assumption now forces *P* to progress from *p* to *q*. (implying $[](p \rightarrow \langle\rangle q)$)

- Also straight forward to embed in HOL, e.g. :

```
Define `transient (Init,A) = ∃α. α∈A /\ HOARE p α (NOT p)`
```

Progress-Gen

- "ensures" only captures progress driven by a single action. More general progress is expressed by $|\!\rightarrow$ (leads-to).

  It is defined as the *<u>smallest relation</u>* satisfying:

$$\frac{p \ \underline{\text{ensures}} \ q}{p \ |\!\rightarrow \ q}$$

// ensures lifting

$$\frac{p \ |\!\rightarrow \ q \quad , \quad q \ |\!\rightarrow \ r}{p \ |\!\rightarrow \ r}$$

// transitivity

$$\frac{p_1 \ |\!\rightarrow \ q \quad , \quad p_2 \ |\!\rightarrow \ q}{p_1 \vee p_2 \ |\!\rightarrow \ q}$$

// disjunctivity

Acting as the basic rules to infer general progress

But … how to define this in HOL?

Embedding “leads-to” in HOL

- Define $\textbf{Elift}\ \ P\ Rel\ \ =\ \ \forall p\ q.\ \ \underline{\text{ensures}}\ P\ p\ q \Rightarrow Rel\ p\ q$
- Define $\textbf{Trans}\ \ Rel\ \ =\ \ \forall p\ q\ r.\ \ Rel\ p\ q\ \ \wedge\ \ Rel\ q\ r\ \Rightarrow Rel\ p\ r$

Specifying all relations which are ens-lifting, transitive, and disjunctive.

- Define $\textbf{Disj}\ \ Rel\ \ =\ \dots$
- Define $\textbf{LeadstoLike}\ \ P\ \ Rel$
  $=$
  $\underline{\text{Elift}}\ \ P\ Rel\ \ \wedge\ \ \underline{\text{Trans}}\ \ Rel\ \ \wedge\ \ \underline{\text{Disj}}\ \ Rel$

A bit indirectly this says that leadsto is the smallest Leadsto-like relation.

- Define $\textbf{leadsto}\ \ P\ p\ \ q$
  $=$
  $\forall Rel.\ \ \underline{\text{LeadstoLike}}\ \ P\ \ Rel\ \ \Rightarrow\ \ Rel\ p\ q$

, this definition does not directly give you Elift, Trans, and Disj for leadsto. <u>But</u> you can prove that leadsto also LeadstoLike, hence you can recover Elift, Trans, and Disj!

Some other (derived) laws

- $|\!\rightarrow$ itself is relf, trans, and disj.
- Progress – Safety

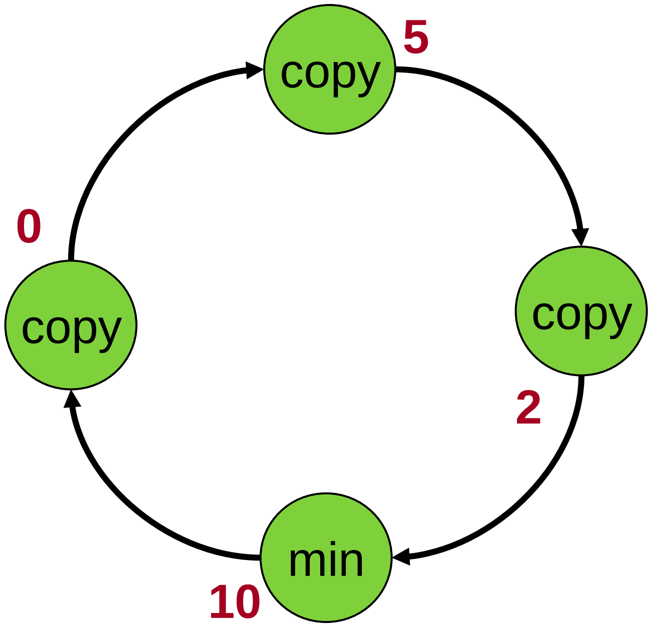
$$\frac{p \ |\!\rightarrow\ q \quad , \quad a \ \underline{\text{unless}}\ b}{p \wedge a \ \ |\!\rightarrow\ \ (q \wedge a) \vee\ b}$$

- Bounded progress

$$\frac{p \wedge m{=}C \ \ |\!\rightarrow\ \ (p \wedge m{<}C) \ \vee\ q}{p \ |\!\rightarrow\ q}$$

- $0 < m$ *holds* innitially
- $0 < m$ **unless** false

Example



Self-stabilizing leader election in a ring.

Problem-1: Leader Election (LE).

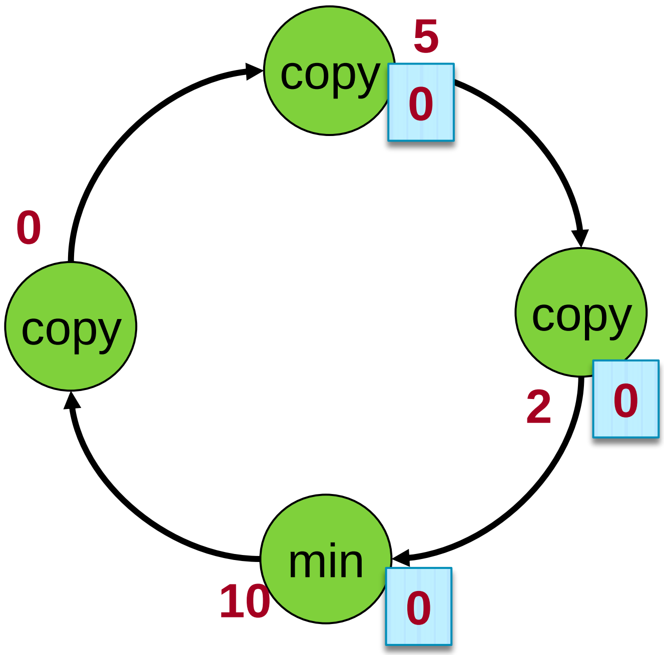
for time to time a ring of processes need to appoint a 'leader'. A centralized decision is undesired.

Problem-2: Self-Stabilizing (SS).

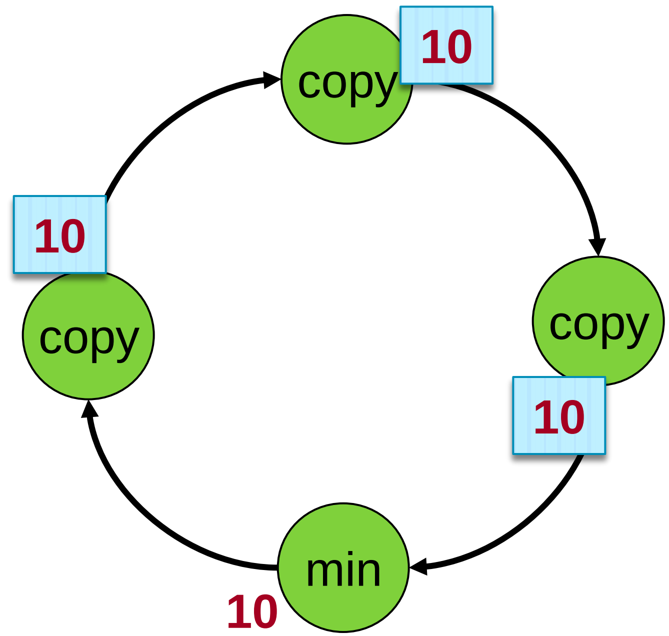
It can start from any state.

A solution: (see left) → relies on the non-determinism of concurrency.

The selection is non-deterministic



(0 mod 4) is selected as the leader.



But we can also end up with (10 mod 4) as the leader.

True, the nodes are not identical. If all nodes are identical (they have identical ID and do the same thing) LE is unfortunately impossible.

Encoding it in HOL

- $\boxed{\textbf{node}\ \ i\ \ =\ \ x[i\text{-}1] \neq x[i]\ \rightarrow\ x[i] := x[i\text{-}1]}$

  Guarded assignment. Model in HOL:

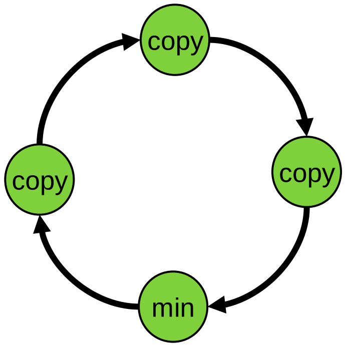
  Define GUARDED *g a* = IF g a SKIP

  "enabledness" is not explicitly modeled → a disabled action cannot be selected; so its effect is skip. So, we modeled as above. This is ok because we don't have an explicit concept of executions anyway.

- $\boxed{\textbf{node0}\ \ N\ \ =\ \ \textbf{if}\ \ x[N\text{-}1] < x[0]\ \ \textbf{then}\ x[0] := x[N\text{-}1]}$

- Define **ring** N = $\{\ \text{node0}\ N\ \} \cup \{\ \text{node}\ i\ \ |\ \ 0<i\ \ \wedge\ \ i<N\ \}$

Specification



Define `**done** k = (\x. (∀i. 0≤i<k ⇒ (x i = x 0))`

leadsto (ring N) (\x. T) (done N)

unless (ring N) (done N) (\x. F)

Verification

- In SPIN you would verify this for N = 1,2,3 and then argue that other N is just analogous to N=3.

- In HOL you can prove the correctness for <u>all</u> N.

  In the proof you will need to come up with a “progress metric” m. Then show this :

  $$m{=}C \quad |\!\!\rightarrow \quad m < C \quad \vee \quad \text{done } N$$

  where “<“ is some <u>well-founded</u> relation over finite domain D. Well-founded means that every subset of D has a minimum element wrt <.

- In HOL you can also prove general theories about e.g. self-stabilization, classes of distributed algorithms.

Verification of Cryptographic Protocols with HOL

Reference

- *Proving Properties of Security Protocols by Induction*, tech. rep. by Paulson.

Cryptographic Protocol

- Having a strong encryption method like RSA is not sufficient in order to secure our electronic “transactions”.
- We furthermore need to implement a certain *protocol*; but this protocol is often very *error prone*.
- Most cryptographic protocols are simple, but surprisingly very difficult to verify, due to complex ways a “spy” may interfere.
- Additional aspects may further add complexity:
  - people may accidentally lose old keys
  - authenticity
  - sometimes non-tracability is required

Notation

- A,B,C : *agents*, parties involved in the protocols.

  Agents can send messages to each other.

- $\{| M |\}$: a message M
  $\{| M, N |\}$: a message containing the tuple M , N

- $\{| M |\}_K$: message M, <u>encrypted</u> with the key K

Notation

- K : *key*
  K_A : A's private keys
  $pubK_A$: A's public keys

- If K is a shared key, then an agent can decrypt $\{|M|\}_K$ only if he also has K

- If K is a public key (in private-public key scheme), then $\{|M|\}_K$ can only be decrypted with the corresponding private key.

A simple protocol P_0

- *A* and *B* want to chat securely. They first exchange a session key. This is a shared key that will be used to encrypt the rest of the communication.

- $A \rightarrow B \quad : \quad \{|\ pubK_A\ |\}$ // here is my pub-key

  $B \rightarrow A \quad : \quad \{|\ k\ |\}_{pubKA}$ // ok , here is a session key

  From this point on *A* and *B* exchanges messages encrypted with the shared key *k*.

Man-in-the-Middle Attack

- $A \rightarrow B \quad : \quad \{| \, pubK_A \, |\}$

// intercepted by Spy !

Spy $\rightarrow B \quad : \quad \{| \, pubK_{\text{Spy}} \, |\}$

$B \rightarrow$ **Spy** $\quad : \quad \{| \, k \, |\}_{pubKspy}$

Spy $\rightarrow A \quad : \quad \{| \, k \, |\}_{pubKA}$

A and *B* now communicate using the session key *k*, unaware that **Spy** also knows *k*.

Now *A* and *B* try to use a KeyServer

- There is now a *trusted* server **S** : it also knows the private keys of *A* and *B*. When *A* want to communicate with *B*, it first requests a session key to *S*. This key has to be securely distributed to *A* and *B*.

- A possible way to do it:

$A \rightarrow \mathbf{S}$: $\{| A, B |\}$

A prompts S that it wants to start a session with B.

$\mathbf{S} \rightarrow A$: $\{| B, k, \{| k, A |\}_{KB} |\}_{KA}$

S generates a seesion key k, send it back encrypted to A. It also prepare a copy of the key for B, encrypted privately for B.

$A \rightarrow B$: $\{| k, A |\}_{KB}$

A pass on the encrypted copy of k to B

However people/application may accidentally lose old session keys. If Spy somehow gets an older packg in step 2, and the corresponding session key k, it can resend that old pckg to A, when A requests S for a new session key. But now k is compromised. Called *replay* attack.

Needham-Schroeder Protocol

- Idea: use fresh numbers, so-called *nonces*, to identify each session. So now you can't replay.

  Protocol:

  $A \rightarrow \mathbf{S}$: {| A, B, η_A |} , η is a nonce

  $\mathbf{S} \rightarrow A$: {| $\eta_A, B, k,$ {| k, A |}$_{KB}$ |}$_{KA}$

  $A \rightarrow B$: {| k, A |}$_{KB}$

  $B \rightarrow A$: {| η_B |}$_k$

  $A \rightarrow B$: {| $\eta_B - 1$ |}$_k$

Unfortunately... this is not really right yet, This part is still vulnerable to replay attack.

Some formal approaches

- Model checking. Model the protocol (and Spy) as automatons, then check that every state is safe.

  \+ Find attacks quickly.
  \- State explosion (forcing simplifying assumptions)

- Belief logic, e.g. Burrows-Abadi-Needham (BAN logic).

  \+ Short, abstract proofs.
  \- Some variants are complicated & ill-motivated

- Inductive approach → Paulson. Mechanized in Isabelle/HOL.

Inductive Approach

- Features
  - Seems to be feasible
  - Based on a clear logical framework

- Statistics:
  - 200 theorems about 10 protocol variants
    (3 × Otway-Rees, 2 × Yahalom, Needham-Schroeder, . . .)
  - 110 laws proved concerning messages
  - 2–9 minutes CPU time per protocol
  - few hours or days human time per protocol
  - over 1200 proof commands in all

Representing Messages

$$data \quad Agent \quad = \quad Server \;\mid\; Friend\ int \;\mid\; Spy$$

Use A,B,C … to denote agents.

$$\begin{aligned} data \quad Msg \quad = \quad & Agent \quad A \\ \mid \quad & Nonce \quad N \\ \mid \quad & Key \quad K \\ \mid \quad & \{\mid X , Y \mid\} \\ \mid \quad & Hash \quad X \\ \mid \quad & Crypt \quad K \quad X \qquad // \ \{\mid X \mid\}_K \end{aligned}$$

Can be easily translated to HOL

Use X,Y, … to denote message

Representing Events

- Protocol steps are represented by *events*:

$$data \quad Event \quad = \quad Say \quad Agent \quad Agent \quad Msg$$

Example:

$$A \rightarrow B \quad : \quad \{|\, k, A \,|\}_{KB}$$

is represented by

$$\text{Say A B (Crypt } K_B \text{ \{| k , A |\})}$$

Model

- We maintain a _history_ **evs**, which is a set of all communication *events* so far.

- Agents are assumed to monitor evs. When an agent B sees an event "Say A B X" in evs it knows that there is a message X from him and can act accordingly.

  (However B does not actually know who sends it (it could be Spy). So B can only infer "Say ? B X" from evs.)

- Spy also has access to evs.

Representing Protocol Steps

- Every *step* σ of the protocol is a function of type:

$$\boxed{\sigma \ : \ \mathit{Event\ set} \rightarrow (\mathit{Event\ set}\,)\ \mathit{set}}$$

such that $evs_2 \in \sigma\, evs_1$ means that evs_2 is a possible history after executing σ on the history evs_1.

(So, σ can be non-deterministic)

- Add a SPY-step (same type as above).
- A protocol can be defined by a transition function

$$\boxed{\mathit{Protocol} \ : \ \mathit{Event\ set} \rightarrow (\mathit{Event\ set})\ \ \mathit{set}}$$

such that $evs_2 \in \mathit{Protocol}\ evs_1$ iff this is allowed by one of the protocol steps or SPY-step.

Representing the Protocol Steps

$A \rightarrow \mathbf{S}: \{| A, B |\}$

$\mathbf{S} \rightarrow A: \{| B, k, \{| k, A |\}_{KB} |\}_{KA}$

$A \rightarrow B: \{| k, A |\}_{KB}$

Step-1 can be modeled by a function σ_1

$$\sigma_1 \; H = H \cup \{ \text{Say } A \; S \; \{| A, B |\} \}$$

Representing the Protocol Steps

$A \rightarrow \mathbf{S}: \{| A, B |\}$

$\mathbf{S} \rightarrow A: \{| B, k, \{| k, A |\}_{KB} |\}_{KA}$

$A \rightarrow B: \{| k, A |\}_{KB}$

$\sigma_2\ H =$

<u>if</u> Say X **S** $\{| Y, Z |\}$ $\in$ *evs*, for <u>some</u> X, Y, Z
<u>then</u>
$H \cup \{$ Say **S** Y (**Crypt** K_Y $\{| Z, k,$ **Crypt** K_Z $\{| k, X |\}$ $|\}$) $\}$

<u>else</u> H

Some concepts

- Let *H* be a set of messages.

- **parts** *H* : all parts of the messages in *H*, applying decryption when necessary.

  What God can infer from *H* ☺

- **analz** *H* : all parts of messages in *H*, applying decryption with keys exposed in *H*.

  What Spy can infer from *H*.

- **synth** *H* : all spoof messages Spy can construct from *H*. In particular, synth (analz *H*) is interesting.

Inductive Def. of parts

Decryption

$$\frac{X \in H}{X \in \mathsf{parts}\, H} \qquad \frac{\mathsf{Crypt}\, K\, X \in \mathsf{parts}\, H}{X \in \mathsf{parts}\, H}$$

$$\frac{\{\!|X, Y|\!\} \in \mathsf{parts}\, H}{X \in \mathsf{parts}\, H} \qquad \frac{\{\!|X, Y|\!\} \in \mathsf{parts}\, H}{Y \in \mathsf{parts}\, H}$$

More precisely, parts is the smallest predicate satisfying the above rules. We can define this in HOL indirectly as we did with the "leadsto" relation in UNITY.

Analz

If Spy can infer the key, then it can decrypt.

$$\frac{X \in H}{X \in \text{analz}\, H}$$

$$\frac{\text{Crypt}\, K\, X \in \text{analz}\, H \qquad K^{-1} \in \text{analz}\, H}{X \in \text{analz}\, H}$$

$$\frac{\{\!|X, Y|\!\} \in \text{analz}\, H}{X \in \text{analz}\, H}$$

$$\frac{\{\!|X, Y|\!\} \in \text{analz}\, H}{Y \in \text{analz}\, H}$$

Synth

Agents' names are assumed to be public.

$$\frac{X \in H}{X \in \text{synth } H}$$

$$\text{Agent } A \in \text{synth } H$$

$$\frac{X \in H}{\text{Hash } X \in \text{synth } H}$$

$$\frac{X \in \text{synth } H \qquad Y \in \text{synth } H}{\{\!|X, Y|\!\} \in \text{synth } H}$$

$$\frac{X \in \text{synth } H \qquad K \in H}{\text{Crypt } K\, X \in \text{synth } H}$$

Spy's steps

- Spy can *extend* *H* with $\textit{Say}\ \mathbf{Spy}\ B\ X$

  where *B* is any agent (other than Spy) and *X* is any spoof message drawn from:

  $$\mathbf{synth}\ (\ \mathbf{analz}\ (\ H \cup Ini\ ))$$

  where

  - *Ini* is Spy's initial knowledge, e.g. the "names/id" of some agents.

Oops rule

- You may want to model schemes where some agents are sometimes careless and lose their _*past*_ session keys.

- This can be modeled by the following "oops" rule.

  If a past H contains an event where Server distributes a session key k to A, marked with some nonces e.g. η_A and η_B, and that this nonces belong to past sessions, then add this to current H :

$$\boxed{\mathit{Say}\quad A\quad \mathbf{Spy}\quad \{|\, k\,,\ \eta_A\,,\ \eta_B\ |\}}$$

Protocol Run

- The "run" can be defined inductively :

$$run : num \rightarrow (Events\ set)\ set$$

run 0 = possible initial histories, e.g. just { [] }

run n = the set of possible histories after n-steps of the protocol (+spy).

- Security property *safe* can be defined over run, in the form:

$$\forall n.\quad \forall H.\quad H \in run\ n \quad \Rightarrow \quad safe\ H$$

Can be proven with induction.

Example of specification

- First extend the protocol with

  $B \rightarrow A : \{| \, o \, |\}_k$

  as the last step, to model the sending of a data message encrypted using the exchanged session key.

  $\boxed{\text{Spec:} \quad o \notin H}$

- In KeyServer, also allow the oops rule.