INF672
Protocol Safety and Verification

Karthik Bhargavan
Xavier Rival
Thomas Clausen

Course Outline

- Lecture 1   [Today, Sep 15]
  - Introduction, Motivating Examples
- Lectures 2-4  [Sep 22, 29, Oct 6]
  - Network Protocol Verification: Spin
- Lectures 5-8  [Oct 13, 20, 27, Nov 3]
  - Program Verification: Properties, Tools & Techniques
- Lecture 9    [Nov 10]
  - Security Protocol Verification: ProVerif
- Lecture 10   [Nov 17]
  - Exam
Recap

- We learnt to model and verify protocols
  1. Write each participant as a finite state machine communicating with others via buffered channels
  2. Write a model of the network and its topology
  3. Write a model of the application
  4. Write a specification as properties in temporal logic
  5. Enumerate all traces of the model and verify each property

```c
bit sent;
bit received;
proctype application () {
    int src, dest;
    mtype m;
    app_in[0]! 2,req;
    sent = 1;
    app_out[2]? 2,0, req;
    received = 1;
    app_in[2]! 0, resp
    app_out[0]? 2,0, resp
}

ltl deliver
{always
    (sent implies eventually received)};
```
Outline

• Model Checking Techniques
  – Some slides taken from Ed Clarke’s course
    http://www.cs.cmu.edu/~emc/15-820A/reading/
Model Checking

• Automated verification technique
  – We argue that proof construction is unnecessary in the case of finite-state concurrent systems, and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic. - Clarke, Emerson, Sistla [1986]
  – ACM Turing Award 2007: Clarke, Emerson, Sifakis

• Specification language: Temporal Logic
  – CTL, LTL, CTL*, mu calculus, STL

• Modeling language: Concurrent FSMs
  – Amenable to exhaustive search
Model Checking: Pros and Cons

• Push-button verification
  – No (tedious) manual proofs
• Replayable counter-examples
• Handles non-compositional problems
  – Interleaving, shared-memory concurrency
But...
• State space explosion
  – Grows exponentially with no. & depth of processes
  – Many techniques to counter this problem.
Kripke Structure

• Kripke Structure $M = (S, I, T, L)$
  – $S$ is a finite set of states: $S = \{s_0, ..., s_n\}$
  – $I$ is the subset of initial states: $I = \{s_0\}$
  – $T$ is a transition relation between states: $T(s_1, s_2)$
  – $L$ labels states with atomic propositions:
    $L(s_0) = \{a\}, L(s_2) = \{a, b\}, ...$

• Formal notation for state transition graphs
  – Atomic propositions represent observables
    E.g. each bit $b$ of the state could be encoded as a proposition $b = 1$
  – Transitions may be non-deterministic
  – States may be blocked
    (dead-lock = no outgoing transitions)
Traces and Computation Trees

State Transition Graph or Kripke Model

Infinite Computation Tree
Temporal Logic

• Propositional Temporal Logic (PTL, LTL, PLTL)
  – $Xf$ or $\Diamond f$: $f$ holds in “next” state of trace
  – $Gf$ or $\Box f$: $f$ holds “always” in all states of trace
  – $Ff$ or $\Diamond f$: $f$ holds “eventually” in some state of trace
  – $f U g$: $f$ holds “until” $g$ holds in trace

• Computational Tree Logic (CTL, CTL*)
  – $Af$ or $\forall f$: $f$ holds in all traces starting from a state
  – $Ef$ or $\exists f$: $f$ holds in some trace starting from a state
  – CTL: every temporal operator must be prefixed by a path operator: $AGf$, $EFf$, $AXf$, etc.
Temporal Logics

- LTL and CTL are closely related
  - Roughly, LTL formulas must hold on all paths: $A f$
- LTL and CTL are disjoint
  - $GF f$ (always eventually $f$) is not in CTL
  - CTL cannot encode *fairness constraints*
- CTL* is a superset of both LTL and CTL
  - Any interleaving of $A, E$ with $F, G, X, U$

- CTL is more suited to automated verification
  - CTL checking is linear in size of formula (* no of states)
  - LTL checking is exponential in size of formula (* no of states)
Traces and Computation Trees

State Transition Graph or Kripke Model

What LTL/CTL properties are true?

Infinite Computation Tree
Model Checking Problem

• Does a Kripke Structure satisfy a given temporal formula?
  \[ M, s_0 \models f \]

• We can define a semantics for CTL formulas
  – \[ M, s \models AX f \text{ if for all } s', T(s,s') \Rightarrow M, s \models f \]
  – \[ M, s \models EX f \text{ if exists } s' \text{ s.t. } T(s,s') \land M, s \models f \]
  – ...
  – Model checking becomes a inference problem: is there a derivation that proves \[ M, s0 \models f \]?
Enumerating States

- Depth-first search
  - Explore each path in the computation tree
    (Either finite or has a loop)
  - Evaluate formula at each step
  - If false, abandon path
  - If true, success
  - If more obligations, continue along path

- E.g. AF a, EF EX b, XX(a\b)
Representing States

• Explicit state model checking enumerates states
  – Each state represents the values of all variables
  – Typically stored as a hash value to detect cycles
  – State grows with number of states, length of paths

• Depth-first search vs. Breadth-first search
  – BFS requires storing a lot of states in memory
  – DFS can be faster, require less memory
  – BFS produces smaller counterexamples, may reach a counterexample faster in a large system
Symbolic Methods

- Model checking as a fixpoint computation
  - Breadth-first search
  - Symbolic representation of computational tree

- Computing $\mathbf{EF} f$
  - $U_0 = \text{False}$
  - $U_1 = f \lor \mathbf{EX} U_0$
  - $U_2 = f \lor \mathbf{EX} U_1$
  - ...

- $\mathbf{EF} f = \text{lfp } U. f \lor \mathbf{EX} U$
Checking $M,s_0 \models EF \diamond f$

$U_3 = p \lor \textbf{EX} U_2$
Binary Decision Diagrams

- Ordered Binary Decision Tree
  - A boolean function over a set of boolean variables
  - Each path has the same order of variables
  - A representation of a propositional formula
Ordered Binary Decision Diagrams (OBDD)
- Combine isomorphic subtrees
- Eliminate nodes where both subtrees are the same
- Can be a much more compact representation (depends on ordering)
- Given a parameter ordering, OBDD is unique up to isomorphism
Operations on BDDs

• Many logical operations are fast on BDDs
  – Negation: flip 0 and 1 on all leaves
  – Conjunction, Disjunction can use Shannon expansion
    • $f/a$ is the formula $f$ where variable $a$ is set to 1 (true) (replace all “a” nodes by right subtree)
    • $f/-a$ is the formula $f$ where variable $a$ is set to 0 (false) (replace all “a” nodes by left subtree)
    • $f \land g = (a=1 \land f|a \land g|a) \lor (a=0 \land f|-a \land g|-a)$
    • RHS corresponds to the BDD of the composition
    • For efficiency, remember intermediate computations
    • Complexity proportional to $|f| \times |g|$
  – Quantification is similar: exists $a. f = f/a \lor f/-a$
Model Checking with BDDs

- Represent state as assignments to boolean variables
  - \( s1 = (x1 = 1 \land x2 = 0 \land x3 = 1) \)
- Encode transition relation as boolean formulas relating current variables to “next” variables
  - \( T(x1,x2,x3,x1',x2',x3') \)
  - Can be translated to an OBDD
- Encode temporal logic satisfaction query:
  - \( M,(x1,x2,x3) \models EX f \) is encoded as
    \( exists x1',x2',x3'. T(x1,x2,x3,x1',x2',x3') \land M,(x1',x2',x3') \models f \)
  - Translate satisfaction query to an OBDD
  - Truth and Falsity result in trivial OBDDs
- For fixpoint computations (\( EF f \)), generate a sequence of OBDDs (\( U0, U1, U2 \)) until they converge
Model Checking with BDDs

• Advantages of OBDDs
  – Can compactly represent “structured” states
  – Symbolic Model Checking: $10^{20}$ States and Beyond, Burch et al [1990]
  – Efficient logical operations, unique representation
  – Widely applied (e.g. SMV) in software, hardware

• Disadvantages of OBDDs
  – Wrong variable order can exponentially blow up BDD
  – May needs careful manual tuning
  – Sometimes no good ordering exists
Using SAT solvers

• Boolean satisfiability problem (SAT):
  – Given a boolean formula over boolean variables find an assignment to the variables that returns true
  – Classic NP-hard problem
    • Even when restricted to conjunctive normal form (CNF)
    • Even when each clause only has 3 variables (3SAT)
  – Still, many practical breakthroughs and tools
    • Large SAT problems can be solved in minutes
    • Many SAT solvers, competitions: http://www.satcompetition.org/
  – Applicable to more than just boolean functions
    • Satisfiability Modulo Theories (SMT) of integers, arrays etc
    • Industrial SMT solvers: Z3, Alt-Ergo, CVC2, Yices
    • Competitions: http://www.smtcomp.org/
Model Checking with SAT

• Similar approach to BDDs
  – Translate the model checking problem to a boolean satisfiability constraint
  – Rather than translating to an OBDD, directly ask a SAT solver
  – Advantage: No need to order variables
  – Disadvantage: No unique representation, so fixpoint computation is more difficult

• Bounded model checking
  – Fix a bound $k$ for length of traces for which we desire to verify the temporal logic property (traces may have loops)
  – Encode a boolean formula that is satisfiable if and only if the temporal formula holds for traces of length less than $k$
  – Run a SAT solver to find a proof/counterexample of length $k$
  – Increase $k$ and run again
Bounded Model Checking

• Suppose we want to check $EG \ a$
  – There is a path on which variable $a$ is always true
  – E.g. two-bit counter: $EG \ (a_0 = 1 \lor a_1 = 1)$

• Suppose $k = 2$, so each trace has 3 states $s_0, s_1, s_2$

• To be a witness for $EG \ f$:
  – $(s_0, s_1, s_2)$ must be a valid path:
    $T(s_0, s_1) \land T(s_1, s_2)$
  – Each state on the path must satisfy $f$:
    $s_0(a) = 1 \land s_1(a) = 1 \land s_2(a) = 1$
  – $(s_0, s_1, s_2)$ must have a loop:
    $T(s_2, s_0) \lor T(s_2, s_1) \lor T(s_2, s_2)$
  – If there is no loop at $k$, we increase $k$ and try again
Symbolic Methods: Summary

- Translate model checking problem into a fixpoint problem on boolean functions
  - Use OBDDs to efficiently represent the state and to encode the satisfiability problem
  - Use a logical representation and call a SAT solver
- Effectively a breadth-first search
  - Produces short counter-examples
  - Can be used for bounded model checking
- Can rely on clever domain-specific encodings
  - Abstract domains for variables
  - Counter-example guided abstraction refinement
  - Enumerative explicit-state methods better suited to smaller push-button verification requirements
Verifying Concurrency

• In verifying network protocols, the main source of complexity is concurrency (interleaving)
  – 2 processes with $n$ states have $\sim 2^n$ interleavings
• Both symbolic and enumerative model checking need to account for this state space explosion
  – Idea: only account for interleavings when the order of transitions matter
  – When two transitions are commutative (independent), just consider one order
  – Many details: defining independence, generalizing principle to full traces
Partial Order Reduction

• Given a set of state machines, we label each local transition $\alpha$ and compute the global states in which it can be applied:
  – $\alpha(s_1,s_2)$ means that $\alpha$ is enabled after $s_1$ and that it changes the global state to $s_2$
  – $\alpha$ is deterministic if for every $s_1$, there is at most one state $s_2$ s.t. $\alpha(s_1,s_2)$

• $\alpha$ is independent from $\beta$ if:
  – Whenever $\alpha$ is enabled in a state, so is $\beta$
  – Whenever $\alpha$ and $\beta$ are enabled, they commute
  – Independence is symmetric and antireflexive
Partial Order Reduction

• Independence is not enough to eliminate traces
  – The sequence of transitions may be observable to a temporal logic spec
• A transition \( \alpha \) is invisible with respect to a set of variables \( x_1,..x_n \), if whenever \( \alpha(s_1,s_2) \), we have
  \[
  s_1(x)(1) = s_2(x)(1) \land \ldots \land s_1(x)(n) = s_2(x)(n)
  \]
  – Suppose we are trying to prove an LTL formula \( f \)
  – If in a particular state, both \( \alpha \) and \( \beta \) are enabled
  – If \( \alpha \) is independent from \( \beta \) and \( \alpha \) is invisible with respect to the variables in \( f \), then just considering the trace starting \([\alpha,\beta]\) is enough, we don’t have to consider \([\beta,\alpha]\)
• This partial order reduction preserves a so-called “stuttering equivalence” between traces
  – It is a sound abstraction for LTL formulas without “next” \( (X) \)
Model Checking Results

- Using partial-order reduction and other optimizations, model checking has been widely applied and is used systematically in industry:
  - Hardware verification at Intel (SMV)
  - Hardware and software verification at NASA (JPF)
  - Software bug hunting at Microsoft (Boogie/Z3)
  - Hybrid system analysis in Airbus (BMC)
  - Security protocol analysis (FDR, AVISPA)
  - Probabilistic model checking (PRISM)
  - Timed model checking (Uppaal)
Model Checking in Networking

- Routing Protocols
  - Loop Freedom, Convergence
  - Widely applied to ad hoc networking
- Telecommunications Protocols
  - Liveness: connection success, robustness
  - Applied to SIP standard, TCP
- Software Defined Networks
  - Safety, Incremental Installation
  - Tools built for many SDN languages
VERIFYING ROUTING PROTOCOLS
Routing in a network (simplified)

- **Shortest Paths Problem**: Find the cheapest route from S to D
  - $L(i, j) = \text{Cost of direct link } i \rightarrow j$
  - $R(a, b) = \text{Cost of route from } a \text{ to } b$
  - $R(a, b) = \min \{ L(a, k) + R(k, b) \}$
Routing Information Protocol (RIP)

• One of the oldest Internet routing protocols
  – Based on Asynchronous Distributed Bellman-Ford [Bertsekas’91]
• Each node $n$ maintains a routing table
  – $hops_D$: number of hops to $D$ (no weighted edges)
  – $next_D$: next router on the path to $D$

• Global progress:
  – Initially: All nodes know their neighbors ($hops = 1$)
  – Finally: All nodes know distance & successor to all other nodes

• Local processes:
  – Periodically send routing table to all neighbors
  – Locally update $hops_D$ to $1 + \min(\text{received } hops_D)$
  – Use timeouts to detect link breakage and expire routes
Distance-vector routing in RIP

Initially

<table>
<thead>
<tr>
<th></th>
<th>A: 0</th>
<th>A: 1</th>
<th>A: ∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C:</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

After exchange

<table>
<thead>
<tr>
<th></th>
<th>A: 0</th>
<th>A: 1</th>
<th>A: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C:</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Routing Loops in RIP: Count to Infinity

After exchange

A: 0
B: 1
C: 2

A: 1
B: 0
C: 1

A: 2
B: 1
C: 0

C: 2

C: ∞

C: 2 + 1 = 3
C: 3 + 1 = 4
C: 4 + 1 = 5
Poisoned reverse

- Advertise $hops_D = \infty$ to $next_D$
- Prevents loops of two routers (adds more cases for verification)

- Limitation: Doesn’t prevent loops of three or more routers
Infinity = 16

• Since we can’t solve the loop problem
  – Set Infinity to 16
  – RIP is not to be used in a network with more than 15 hops.

• If a routing loop occurs, it will be discovered in at most 15 routing updates
  – That is, the routing loop is transient
  – Until then, packets will be forwarded in a loop

• Concrete protocol design deviates from theory
  – Original proof of asynchronous distributed Bellman-Ford no longer directly applies
  – Many corner cases, race conditions to worry about
Formal Goals for Routing

• Loop Freedom (Safety)
  – In the global state of a routing protocol, there should not be a subset of routing tables that creates a loop on the route to $D$

• Many routing protocols have transient routing loops when links go down or come back up
  – RIP has transient loops that may last 15 updates
  – BGP prevents count-to-infinity by using paths
  – AODV prevents it by using sequence numbers
    • But AODV (v2) still had persistent routing loops
Formal Goals for Routing

• Convergence (Liveness)
  – In the global state of a routing protocol, if all links remain stable, then all routing tables eventually converge
  – Soundness: they should converge to valid routes
  – Optimality: they should converge to minimal routes

• Protocols:
  – RIP converges in at most 15 routing updates
  – AODV (-02) may converge to invalid routes!
  – BGP may not converge!
proctype Update(router ME){
    mesg adv;
    chan in = routerinput[ME];

    do
        :: atomic{in?adv ->
            if
                :: (adv.src == rtable[ME].nextR) &&
                    (adv.net == rtable[ME].nextN) ->
                    if
                        :: adv.hopcount >= INFINITY ->
                            rtable[ME].hops = INFINITY
                        :: adv.hopcount < INFINITY ->
                            rtable[ME].hops = adv.hopcount + 1
                    fi
                :: adv.hopcount + 1 < rtable[ME].hops ->
                    rtable[ME].nextR = adv.src;
                    rtable[ME].nextN = adv.net;
                    rtable[ME].hops = adv.hopcount + 1
        :: else -> skip
    fi}
}
Modeling the Network

• A network process implements the topology
  – Table $\text{neighbors}[i].n[j]$ contains current graph
  – A router $r$ can send $\text{broadcast}!r,msg$
    The network process will forward it to all its neighbors
  – A router $r$ can send $\text{unicast}!r,r',msg$
    The network process will forward it to $r'$
  – The environment can send $\text{downlink}!r,r'$
    The network process will delete $r-r'$ from the graph
  – The environment can send $\text{uplink}!r,r'$
    The network process will add $r-r'$ to the graph
  – A router going down can be simulated by all its links going down
Properties

Definition 1 (Soundness). A protocol state \( s = (\text{hops}, \text{nextN}, \text{nextR}) \) of a universe described by a valid \text{conn} is said to be sound with respect to \( d \) if

1. \( \forall r : \text{'router}.\text{conn} r (\text{nextN}(r)) \land \text{conn} (\text{nextR}(r)) (\text{nextN}(r)) \)
2. \( \forall r : \text{'router}.1 \leq (\text{hops}(r)) \leq 16 \)
3. \( \forall r : \text{'router}.(\text{conn} r d) \Rightarrow (\text{hops}(r) = 1) \land (\text{nextN}(r) = d) \land (\text{nextR}(r) = r) \)
4. \( \forall r : \text{'router}.\neg(\text{conn} r d) \Rightarrow (\text{hops}(r) > 1) \land (\text{nextN}(r) \neq d) \land (\text{nextR}(r) \neq r) \)

Definition 2 (Stability). For \( k \geq 1 \), we say that the universe is \( k \)-stable if both of the following properties hold:

\textbf{(S1)} Every router from the \( k \)-circle has its metric set to the actual distance to \( d \). Moreover, if \( r \) is not connected to \( d \), it has its \text{nextR} set to the first router on some shortest path to \( d \):

\[ \forall r. \ r \in C_k \Rightarrow \text{hops}(r) = D(r) \land (\neg \text{conn} r d \Rightarrow D(\text{nextR}(r)) = D(r) - 1) \]

\textbf{(S2)} Every router outside the \( k \)-circle has its \text{hops} strictly greater than \( k \):

\[ \forall r. \ r \notin C_k \Rightarrow \text{hops}(r) > k. \]
Properties

• Soundness is written as a safety invariant on the state of the rtable variable, written in LTL

• Stability is a liveness property about eventual convergence of the protocol, written in LTL

Lemma 4 (Progress). For any \( k < 15 \), if the universe \( \mathcal{U} \) is \( k \)-stable at some point, then \( \mathcal{U} \) will eventually become and remain \( (k + 1) \)-stable indefinitely.

Theorem 1 (Convergence of RIP). Starting from an arbitrary sound initial state, evolving under an arbitrary fair advertisement trace, the universe \( \mathcal{U} \) eventually becomes and remains 15-stable.
Proofs

- Theorem 1 can be proved automatically for small networks.
  - Starting from an arbitrary routing table, given a stable network topology, the table converges
  - SPIN takes time but succeeds
- For arbitrary networks, model-checking does not scale
  - Need to mix theorem proving and model checking
  - Prove 0-stability, progress, and preservation in SPIN
  - Prove main theorem in Higher Order Logic

<table>
<thead>
<tr>
<th>Task</th>
<th>HOL</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling RIP</td>
<td>495 lines, 19 defs, 20 lemmas</td>
<td>141 lines</td>
</tr>
<tr>
<td>Stability Preservation Once</td>
<td>9 lemmas, 119 cases, 903 steps</td>
<td></td>
</tr>
<tr>
<td>Stability Preservation Again</td>
<td>29 lemmas, 102 cases, 565 steps</td>
<td>207 lines, 439 states</td>
</tr>
<tr>
<td>Stability Progress</td>
<td>Reuse Stability Preservation</td>
<td>285 lines, 7116 states</td>
</tr>
</tbody>
</table>
Ad-hoc On-demand Distance Vector Routing

- Routing protocol for Mobile Ad-hoc Networks
- Temporary networks created by low-range, low-power wireless devices in battlefields or disaster areas
- Routes are computed on-demand to save bandwidth.
AODV Loop Freedom

- AODV is designed to prevent transient loops
  - Avoids bandwidth wastage to unreachable nodes
  - Loops in on-demand routes difficult to get rid of, since they cannot rely on regular route updates
- Each routing table entry contains:
  - $hops_D$, $next_D$ as before
  - $seqno_D$: a number indicating route freshness
  - Only fresher routes can update an existing route
  - Among two routes of equal freshness, smaller hop-count is preferred.
- Property to be formally verified is loop freedom
AODV Loop Freedom

After exchange

A: 0, 0  A: 1, 0  A: 2, 0
B: 1, 0  B: 0, 0  B: 1, 0
C: 2, 0  C: 1, 0  C: 0, 0

C: $\infty$, 1

C: $\infty$, 1
Searching for loops automatically

- We model the 3-node network in SPIN
  - Each node runs an identical Promela process
  - A global link table encodes dynamic topology
- We specify loop freedom as a safety property in Linear Temporal Logic
  - $\text{Always } (!((next_D(A)==B) \land (next_D(B)==A))))$
- We run SPIN, which finds four counterexamples
Looping Counter-examples

- What do we do when routes expire?
Looping Counter-examples

• Let’s keep expired route but set it to infinity?
Looping Counter-examples

- Let’s keep expired route but set it to infinity, increase its sequence number, and delete it later
  - What if the route update gets dropped before the timeout?
Looping Counter-examples

- What if one of the nodes reboots?
  - Its routing table is reset
  - Its neighbours may not detect that it has been rebooted
Sufficient Conditions for Loop Freedom

1. Increase sequence number on every update, even if route expires or breaks
2. Never delete expired routes
   – Can delete them if all other nodes have indicated knowledge of expiry in some way
3. Detect when a neighbor restarts AODV
   – Treated as if all links to neighbors are broken

• Are these conditions enough to guarantee loop freedom for all runs of arbitrary AODV networks?
  – Yes, we can prove a general theorem by combining finitary proofs in SPIN with abstraction proofs in HOL
Verifying RIP and AODV

<table>
<thead>
<tr>
<th>Task</th>
<th>HOL</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling RIP</td>
<td>495 lines, 19 defs, 20 lemmas</td>
<td>141 lines</td>
</tr>
<tr>
<td>Proving Lemma 4.3 Once</td>
<td>9 lemmas, 119 cases, 903 steps</td>
<td></td>
</tr>
<tr>
<td>Proving Lemma 4.3 Again</td>
<td>29 lemmas, 102 cases, 565 steps</td>
<td>207 lines, 439 states</td>
</tr>
<tr>
<td>Proving Lemma 4.4</td>
<td>Reuse Lemma 4.3 Abstractions</td>
<td>285 lines, 7116 states</td>
</tr>
<tr>
<td>Proving Lemma 5.1</td>
<td>Reuse Lemma 4.3 Abstractions</td>
<td>216 lines, 1019 states</td>
</tr>
<tr>
<td>Proving Lemma 5.2</td>
<td>Reuse Lemma 4.3 Abstractions</td>
<td>221 lines, 1139 states</td>
</tr>
<tr>
<td>Proving Lemma 5.4</td>
<td>Reuse Lemma 4.3 Abstractions</td>
<td>342 lines, 21804 states</td>
</tr>
<tr>
<td>Modeling AODV</td>
<td>95 lines, 6 defs</td>
<td>302 lines</td>
</tr>
<tr>
<td>Proving Lemma 6.3</td>
<td></td>
<td>173 lines, 5106 states</td>
</tr>
<tr>
<td>Proving Lemma 6.4</td>
<td></td>
<td>173 lines, 5106 states</td>
</tr>
<tr>
<td>Proving Lemma 6.5</td>
<td></td>
<td>157 lines, 721668 states</td>
</tr>
<tr>
<td>Proving Theorem 6.2</td>
<td>4 lemmas, 2 cases, 5 steps</td>
<td></td>
</tr>
<tr>
<td>Proving Theorem 6.1</td>
<td>4 lemmas, 5 cases, 49 steps</td>
<td></td>
</tr>
</tbody>
</table>

- Detailed models derived from the standards
- Fully formal proofs in HOL + SPIN of
  - AODV safety: loop freedom and route validity
  - RIP liveness: convergence with sharp timing bounds
- Paper:
  - *Formal verification of standards for distance vector routing protocols*, K Bhargavan, D Obradovic, C Gunter, JACM 2002
Summary

• Model-checking can be used to analyze network protocols over arbitrary dynamic topologies
  – With dynamic network misbehavior
  – Both liveness and safety properties

• To scale up to large-sized (or unbounded) networks, we need abstraction techniques
  – Abstractions are generally proved by hand or in a theorem prover, and compose well with model-checking
  – We will see some abstractions for infinite state crypto protocols next week
  – We already saw some abstractions for C programs and general software
TD: Properties in Temporal Logic

• Propositional Temporal Logic (PTL, LTL, PLTL)
  – $X f$: $f$ holds in “next” state of trace
  – $G f$: $f$ holds “always” in all states of trace
  – $F f$: $f$ holds “eventually” in some state of trace
  – $f U g$: $f$ holds “until” $g$ holds in trace

• Computational Tree Logic (CTL, CTL*)
  – $A f$: $f$ holds in all traces starting from a state
  – $E f$: $f$ holds in some trace starting from a state
  – CTL: every temporal operator must be prefixed by a path operator: $AG f$, $EF f$, $AX f$, etc.
LTL Semantics

- Written in terms of traces \((s_1,\ldots,s_n)\)
  - \((s_1,\ldots,s_n) \models p\) iff \(p(s_1)\)
  - \((s_1,\ldots,s_n) \models f \lor g\) iff \((s_1,\ldots,s_n) \models f \lor (s_1,\ldots,s_n) \models f g\)
  - \((s_1,\ldots,s_n) \models f \land g\) iff \((s_1,\ldots,s_n) \models f \land (s_1,\ldots,s_n) \models f g\)
  - \((s_1,\ldots,s_n) \models \neg f\) iff \(\neg (s_1,\ldots,s_n) \models f\)
  - \((s_1,\ldots,s_n) \models X f\) iff \((s_2,\ldots,s_n) \models f\)
  - \((s_1,\ldots,s_n) \models G f\) iff \((s_1,s_2,\ldots,s_n) \models f \land (s_2,\ldots,s_n) \models G f\)
  - \((s_1,\ldots,s_n) \models F f\) iff \((s_1,s_2,\ldots,s_n) \models f \lor (s_2,\ldots,s_n) \models F f\)
  - Can generalize to infinite traces
CTL Semantics

• Written in terms of Kripke Structures
  – \( M = (S,I,T,L) \)
  – \( M,s| = p \iff p(s) \)
  – \( M,s| = f \lor g \iff M,s| = f \lor M,s| = f \land g \)
  – \( M,s| = f \land g \iff M,s| = f \land M,s| = f \lor g \)
  – \( M,s| = \text{not } f \iff \text{not } M,s| = f \)
  – \( M,s| = \text{AX } f \iff \text{for all } (s,s') \text{ in } T, M,s'| = f \)
  – \( M,s| = \text{EX } f \iff \text{exists } (s,s') \text{ in } T, M,s'| = f \)
  – \( M,s| = \text{AG } f \iff M,s| = f \land \text{for all } (s,s') \text{ in } T, M,s'| = G f \)
  – \( M,s| = \text{EG } f \iff M,s| = f \land \text{exists } (s,s') \text{ in } T, M,s'| = G f \)
  – \( M,s| = \text{AF } f \iff M,s| = f \lor \text{for all } (s,s') \text{ in } T, M,s'| = F f \)
  – \( M,s| = \text{EF } f \iff M,s| = f \lor \text{exists } (s,s') \text{ in } T, M,s'| = F f \)
Exercises

• Download:  
  http://prosecco.inria.fr/personal/karthik/teaching/ACN910/Lecture-3-Materials.zip

• TP: See and try to solve handouts on LTL/CTL

• TD: Model the RIP protocol in SPIN  
  or complete the TCP model from last week
Routing Information Protocol (RIP)

- Each node $n$ maintains a routing table
  - $hops_D$: number of hops to $D$ (no weighted edges)
  - $next_D$: next router on the path to $D$

- Global progress:
  - Initially: All nodes know their neighbors ($hops = 1$)
  - Finally: All nodes know distance & successor to all other nodes

- Local processes:
  - Periodically send routing table to all neighbors
  - Locally update $hops_D$ to $1 + \min(\text{received } hops_D)$
  - Optional: Use timeouts to detect link breakage

- Verification goals:
  - Loop freedom: a safety property
  - Convergence: a liveness property
Lecture 7 Materials

http://prosecco.inria.fr/personal/karthik/Lecture-7-Materials.zip