Satisfiability Modulo Theories
Applications to Real-time Fault-Tolerant Systems

SAT/SMT Summer School
Trento, Italy, June 2012

Bruno Dutertre
SRI International

Outline

Fault-tolerant Systems

SMT-Based Model Checking

Three Examples

- Timed Systems
- TTA Startup Protocol
- TTE Clock Synchronization

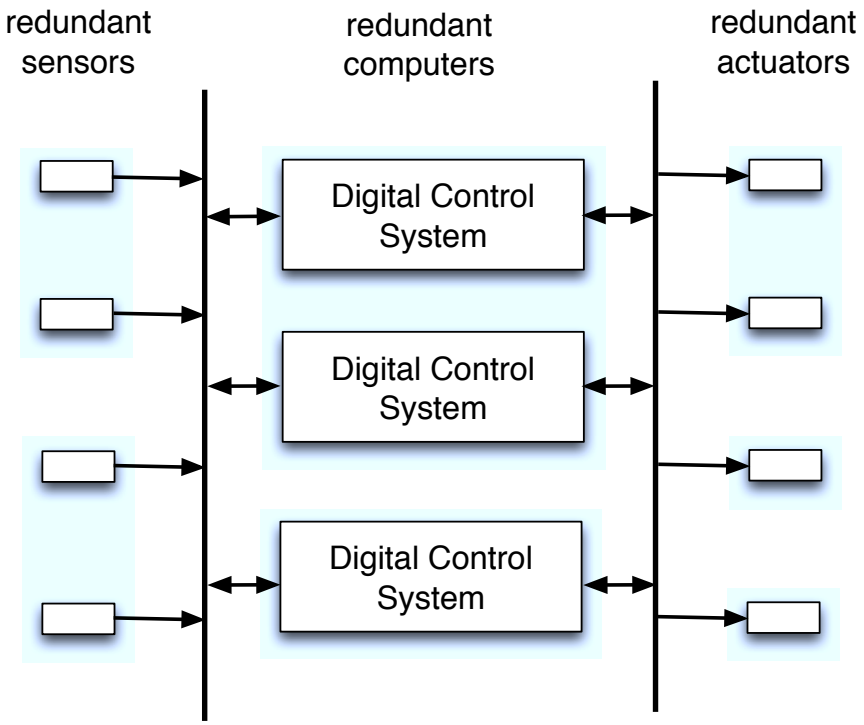
Fault Tolerance

Example: Avionics Control Systems

Flight Control System (Fly-by-Wire)

- Reads pilot input + physical sensors (airspeed, pressure, angle of attack, etc.)
- Computes commands that moves the planes control surfaces
- Must be extremely reliable: the probability of failure must be less than 10^{-9} per flight hour (for civil aircraft)
- Hardware is not reliable enough (estimates are about 10^{-6} to 10^{-7} failure probability per hour for CPU, RAM, etc.)

Highly Reliable Digital Systems



Redundant system of sensors, actuators, computers, communication links

Fault Tolerance Issues

Goal

- The full system must work (possibly in a degraded mode) even if some of its components are faulty

Issues

- Ensure the non-faulty computers agree on the control output (within some margin), under some fault assumptions on the number and types of faults
- Example Fault Types
  - Fail-stop (crash, sends nothing)
  - Inconsistent omissions (send correct data to some component, nothing to others)
  - Symmetric faults (sends same incorrect data to all)
  - Byzantine faults (arbitrary, asymmetric behavior)

Approaches to Fault Tolerance

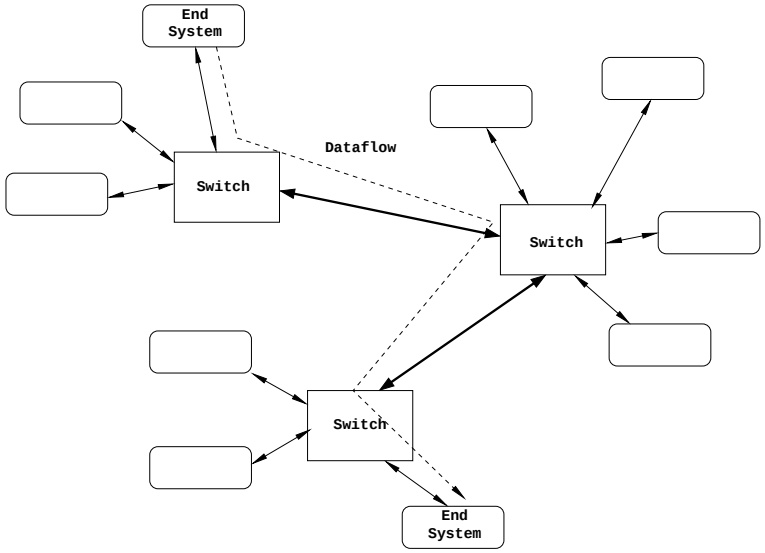
Synchronous Systems

- maintain all the non-faulty components synchronized
- use voting algorithms to ensure that they process the same input data
- all redundant computers are exact replicas of each other: they maintain identical states, process the same input, produce identical output

Asynchronous Systems

- each controller works at its own rate: no synchronization
- lack of synchronization implies: distinct controllers may operate on different input values, so exact agreement on output is impossible
- voting + thresholding + error detection scheme are used to select one control value of out those produced by the redundant controllers

Example Architecture: Timed-Triggered Ethernet (TTE)



Ethernet for fault-tolerant, real-time distributed systems:

- Guarantees for real-time messages: low jitter, predictable latency, no collisions
- All nodes are synchronized (fault-tolerant clock synchronization protocol)
- All communication and computation follow a system-wide, cyclic schedule

Main Fault-Tolerant Protocols in TTE

Startup:

- bring up the network into the synchronized state

Clock Synchronization:

- executed periodically to maintain all clocks within a fixed bound of each other

Clique Detection and Resolution:

- to recover from network-wide transient upsets

Fault Assumptions:

- Single Fault Configuration: at most one faulty component
  - Faulty end system: Byzantine
  - Faulty switch: inconsistent omission
- Dual Fault Configuration: no more than two faulty components
  - Fault type: inconsistent omission

Verification Problems for TTE

Goal

- Show protocol correctness under the stated fault assumption(s)
- Get counterexamples if the protocols are not correct

Issues

- deal with real-time protocol aspects (timers, communication delays, etc.)
- model fault assumptions
- model clocks and clock drift
- make the proofs as automatic as possible

SMT-Based Models + Induction

Symbolic Modeling

State-transition systems

$$\mathcal{M} = \langle X, I(X), T(X, X') \rangle$$

- X set of state variables
- formula $I(X)$ defines the initial states
- formula $T(X, X')$ defines the transition relation

Traces

- Sequences of states $x_0 \to x_1 \to x_2 \ldots$ such that
  - x_0 satisfies $I(X)$
  - for every $t \in \mathbb{N}$, (x_t, x_{t+1}) satisfies $T(X, X')$

Bounded Model Checking

Goal

- Find counterexamples to a property
- Usually the property is an invariant $\Box P$
- The goal is then to find a reachable state that does not satisfy P.

Technique

- Fix a bound k
- Search for a state reachable in k steps that falsifies P
- This is the same as checking the satisfiability of the formula

$$I(x_0) \wedge T(x_0, x_1) \wedge T(x_1, x_2) \wedge \ldots \wedge T(x_{k-1}, x_k) \wedge \neg P(x_k)$$

Induction

Goal

- Prove that P is invariant

Standard Induction

- Show that the following formulas are valid (their negation is not satisfiable)

$$I(x_0) \rightarrow P(x_0)$$

$$P(x_0) \wedge T(x_0, x_1) \rightarrow P(x_1)$$

- If this succeeds then P is an inductive invariant

What if induction fails?

Case 1: $I(x_0) \rightarrow P(x_0)$ is not valid

- some initial state x_0 fails to satisfy P, so P is not invariant

Case 2: $P(x_0) \wedge T(x_0, x_1) \rightarrow P(x_1)$ is not valid

- there are two successive states x_0 and x_1 such that x_0 satisfies P and x_1 does not satisfy P
- if x_0 is reachable, then P is not invariant (but checking whether x_0 is reachable is not easy)
- otherwise, we can't tell whether P is invariant or not
  we can try other things:
  - invariant strengthening
  - use an auxiliary invariant as a lemma
  - use k-induction, a stronger induction rule

Invariant Strengthening

Idea: find an inductive invariant Q that implies P

This amounts to showing that the following formulas are valid

$$I(x_0) \rightarrow Q(x_0)$$

$$Q(x_0) \wedge T(x_0, x_1) \rightarrow Q(x_1)$$

$$Q(x_0) \rightarrow P(x_0)$$

If they are, then P is invariant

Auxiliary Lemma

Assume we know another auxiliary invariant L, we can try to use it as a lemma to prove that P is invariant

Proof Rule: If the following formulas are valid

$$I(x_0) \Rightarrow P(x_0)$$

$$P(x_0) \wedge L(x_0) \wedge T(x_0, x_1) \Rightarrow P(x_1)$$

and L is invariant, then P is invariant (P is inductive relative to L)

k-induction

Generalizes induction to k steps

- Base case:

$$I(x_0) \wedge T(x_0, x_1) \wedge \ldots \wedge T(x_{k-1}, x_k) \Rightarrow P(x_0) \wedge \ldots \wedge P(x_k)$$

- Induction step:

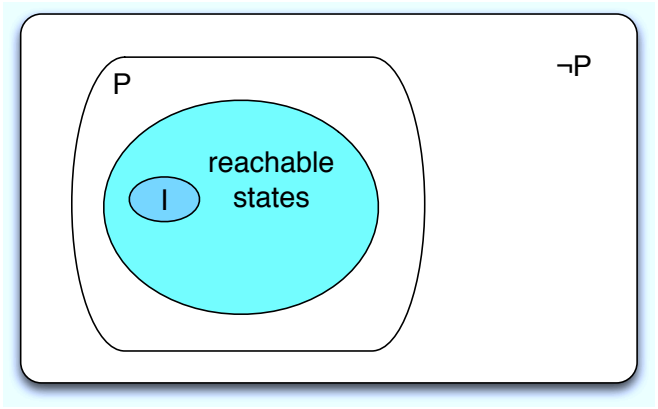
$$T(x_0, x_1) \wedge \ldots \wedge T(x_k, x_{k+1}) \wedge P(x_0) \wedge \ldots \wedge P(x_k) \Rightarrow P(x_{k+1})$$

How good is it?

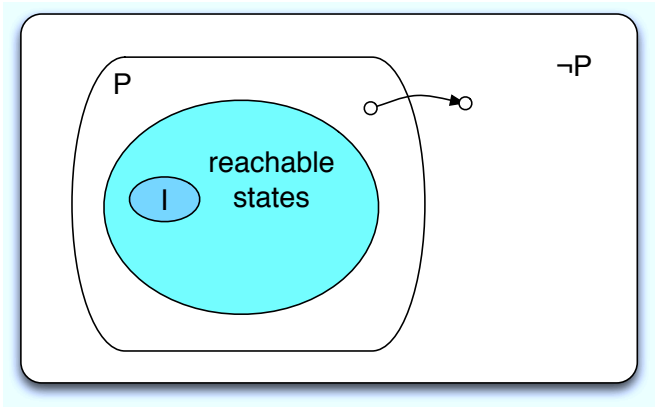
- In most cases, k-induction is stronger than standard induction (when $k \geqslant 2$)

  $\Box P$ is provable by k-induction iff $\Box(P \wedge \circ P \wedge \ldots \wedge \circ^k P)$ is provable by induction, so k-induction can be viewed as a form of invariant strengthening

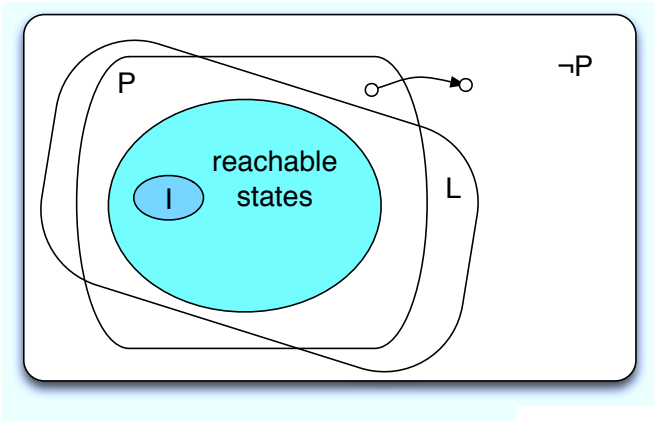
- There are counterexamples: For example, if T is reflexive, then $\Box P$ is provable by k-induction iff $\Box P$ is provable by standard induction.



P invariant



P invariant but not inductive



P inductive relative to L

Timed Systems

Modeling Real-time Systems

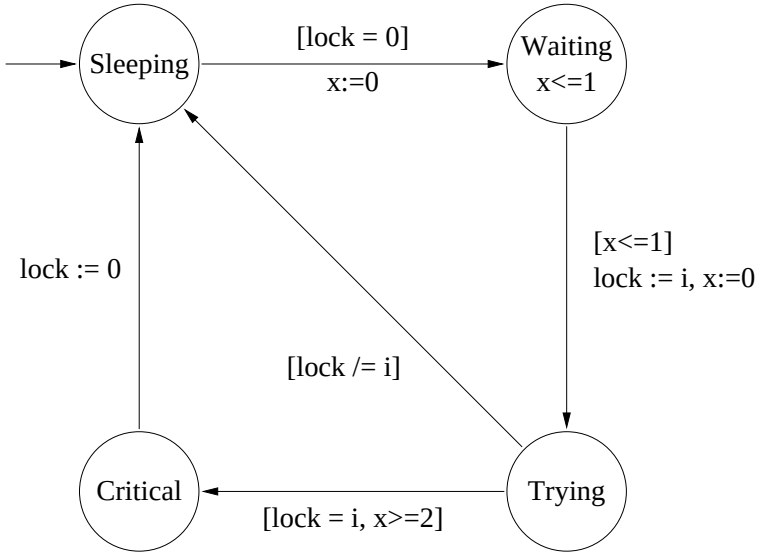
Constraints

- Model timed systems as state-transition systems
- Make the model amenable to analysis using:
  - bounded model checking
  - k-induction

Possible Models

- Implicit time
  - *Timed Automata* (Alur & Dill) and many variants.
  - Many other models (e.g., timed process algebras)
- Explicit time
  - use an explicit `time` variable (e.g., Lamport & Abadi)
  - transition relation encodes time progress: `time' = time + delta`

Timed Automata



- The clock x is a real-valued variable
- It can be reset on discrete transitions
- x increases continuously at a constant rate ($\dot{x} = 1$) between discrete transitions
- Guards specify when transitions can be taken

Timed Automata as State-Transition Systems

Translation

- Discrete transitions

$$\langle x = t_0; \text{sleeping}, lock = 0\rangle \longrightarrow \langle x = 0; \text{waiting}, lock = 0\rangle$$

$$\langle x = t_0; \text{trying}, lock = i\rangle \longrightarrow \langle x = t_0; \text{critical}, lock = i\rangle$$

- Time-progress transitions

$$\langle x = t; \text{waiting}, lock = 0\rangle \xrightarrow{\delta} \langle x = t + \delta; \text{waiting}, lock = 0\rangle \text{ where } \delta \geqslant 0$$

This translation can be used for bounded model checking using SMT solvers (Audemart, et al., 2002)

Issue: not well suited for proofs by k-induction

- Idle transitions: $\langle x = t; \ldots\rangle \xrightarrow{0} \langle x = t; \ldots\rangle$ make k-induction useless
- This remains an issue even if we require $\delta > 0$ in time progress (because δ can be arbitrarily small)

Timeout Automata

Real Time is Really Simple
Leslie Lamport

Basic ideas

- Use an explicit global `time` variable
- Increment `time` by jumps as in Discrete-Event Simulation
  - Jump to the next "interesting" point in time, that is, the time when the next discrete transition must be taken
  - This uses timeout variables to store the times when future discrete transitions are scheduled to occur

- This has lots of similarities with discrete time models

Timeout Automata (continued)

State variables

- global time t and timeouts $\tau_1, \ldots, \tau_n$ (real-valued)
- discrete variables

τ_i stores a time in the future, where a discrete transition is scheduled to happen

$t \leqslant \tau_i$ is an invariant

Discrete Transitions

- Enabled when $t = \tau_i$ for some i
- Do not change t and must update τ_i to a value larger than t

Time-progress transitions

- Enabled when $t < \min(\tau_1, \ldots, \tau_n)$
- Increase t to $\min(\tau_1, \ldots, \tau_n)$
- Time progress is deterministic

Example: Fischer's Mutual Exclusion Protocol

parameters: δ_1 and δ_2 such that $\delta_1 < \delta_2$

shared variable: `lock` initialized to 0

N processes $P(i)$ for $i = 1, \ldots, N$

process $P(i)$
  loop
    wait until `lock` $= 0$
    wait for a delay $d \leqslant \delta_1$
    `lock` $:= i$
    wait for a delay $d \geqslant \delta_2$
    if `lock` $= i$ then
      enter critical section
      `lock` $:= 0$
  endloop

Process in SAL

```
process[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    GLOBAL lock: LOCK_VALUE
    OUTPUT timeout: TIME
    LOCAL pc: PC
  INITIALIZATION
    ...
  TRANSITION
    [ waking_up:
      pc = sleeping AND time = timeout AND lock = 0 -->
           pc' = waiting;
           timeout' IN { x: TIME | time < x AND x <= time + delta1 }
      ...
   [] setting_lock:
      pc = waiting AND time = timeout -->
           pc' = trying;
           lock' = i;
           timeout' IN { x: TIME | time + delta2 <= x }

   [] entering_cs:
      pc = trying AND time = timeout AND lock = i -->
           pc' = critical;
           timeout' IN { x: TIME | time < x }
     ...
```

Clock Module

```
TIMEOUT_ARRAY: TYPE = ARRAY IDENTITY OF TIME;

is_min(x: TIMEOUT_ARRAY, t: TIME): bool =
  (FORALL (i: IDENTITY): t <= x[i]) AND (EXISTS (i: IDENTITY): t = x[i]);



clock: MODULE =
  BEGIN
    INPUT time_out: TIMEOUT_ARRAY
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
    [ time_elapses:
         (FORALL (i: IDENTITY): time < time_out[i]) -->
           time' IN { t: TIME | is_min(time_out, t) }
    ]
  END;
```

Mutual Exclusion Property

A sequence of simple lemmas all proved by 1-induction

```
time_aux1: LEMMA
   system |- G(FORALL (i: IDENTITY): time <= time_out[i]);

time_aux2: LEMMA
   system |- G(FORALL (i: IDENTITY):
                pc[i] = waiting => time_out[i] - time <= delta1);

time_aux3: LEMMA
   system |- G(FORALL (i, j: IDENTITY):
                lock = i AND pc[j] = waiting => time_out[i] > time_out[j]);

logical_aux1: LEMMA
   system |- G(FORALL (i, j: IDENTITY):
                pc[i] = critical => lock = i AND pc[j] /= waiting);
```

Mutual exclusion is implied by `logical_aux1`

```
mutual_exclusion: THEOREM
   system |- G(FORALL (i, j: IDENTITY):
                i /= j AND pc[i] = critical => pc[j] /= critical);
```

TTA Startup

Beyond Timeout-based Models

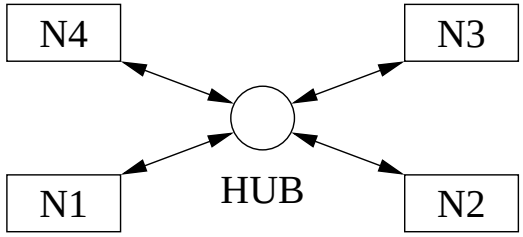
Limitation of Timeout Automata

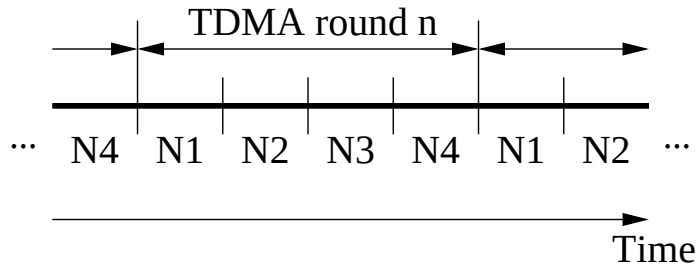
- When processes communicate via messages
- Need to model delays in message transmission

Solution: Calendar Automata

- Each message has an explicit arrival time
- Messages are stored in an event queue (a.k.a. calendar)
- Time progress: jump either to the next timeout or to the smallest arrival time of messages in the calendar
- Sending a message: add events to the calendar with arrival time in the future

The TTA Startup Protocol





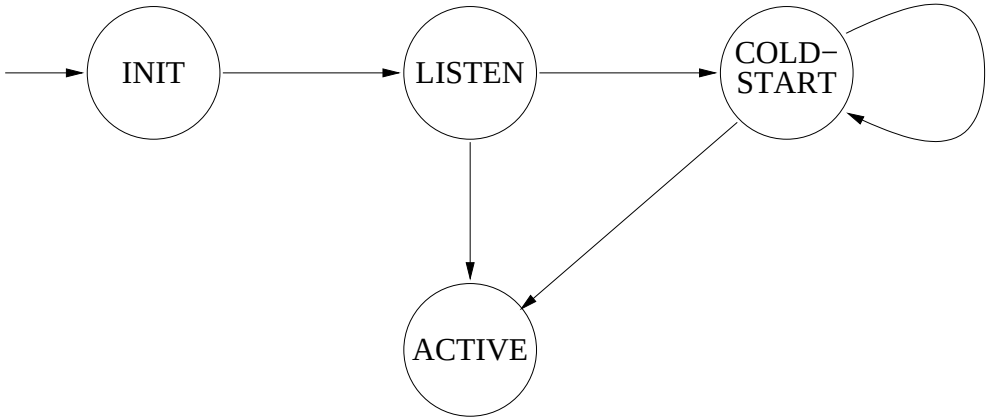
TTA

- Related to TTE, but with a hub/bus topology
- In normal mode, the nodes share the bus using TDMA
- This requires all nodes to be synchronized and follow the same cyclic schedule
- Full TTA: uses two hubs to tolerate failure of one of them

Startup Protocol

- Brings system from unsynchronized to the normal, synchronized mode
- This requires both nodes and hubs to synchronize
- During startup, the hub resolves message collisions

Node Startup



Timing

- Length of a round: $\tau^{round} = N.\tau$
- Start of node i's TDMA slot in a round: $\tau_i^{startup} = (i-1).\tau$
- Timeouts used during startup:

$$\begin{aligned} \tau_i^{listen} &= 2\tau^{round} + \tau_i^{startup} \\ \tau_i^{coldstart} &= \tau^{round} + \tau_i^{startup} \end{aligned}$$

- Transmission delay: must be smaller than $\tau/2$

Simplified Startup Model in SAL

Main simplification: no failures

- Hub only has to deal with collision, no node failures to mask
- The hub is modeled in SAL as a calendar:

```
message: TYPE = { cs_frame, i_frame };

calendar: TYPE =
  [# flag: ARRAY IDENTITY OF bool,  % nodes that will get the frame
     content: message,              % frame being transmitted
     origin: IDENTITY,              % sender
     send: TIME,                    % transmission time
     delivery: TIME                 % reception time
   #];
...
empty?(cal: calendar): bool = FORALL (i: IDENTITY): NOT cal.flag[i];
...
i_frame_pending?(cal: calendar, i: IDENTITY): bool =
    cal.flag[i] AND cal.content = i_frame;
...
consume_event(cal: calendar, i: IDENTITY): calendar =
    cal WITH .flag[i] := false;
```

Simplified Startup: Node Model

```
node[i: IDENTITY]: MODULE =
  BEGIN
    INPUT  time: TIME
    OUTPUT timeout: TIME
    OUTPUT slot: IDENTITY     % slot and pc need to be output
    OUTPUT pc: PC             % to be read by the abstraction module
    GLOBAL cal: calendar
  ...
  TRANSITION
   ...
   % end of listen phase: broadcast cs frame, move to coldstart state
   [] listen_to_coldstart:
      pc = listen AND time = timeout -->
         pc' = coldstart;
         timeout' = time + tau_coldstart(i);
         cal' = bcast(cal, cs_frame, i, time)
   ...
   % reception of a cs_frame in the coldstart state:
   % synchronize on the sender and move to active state
   [] cs_frame_in_coldstart:
      pc = coldstart AND cs_frame_pending?(cal, i) AND time = event_time(cal, i) -->
         pc' = active;
         timeout' = time + slot_time - propagation;
         slot' = frame_origin(cal, i);
         cal' = consume_event(cal, i)
```

Simplified Startup: Clock Module

```
clock: MODULE =
  BEGIN
    INPUT time_out: TIMEOUT_ARRAY
    INPUT cal: calendar
    OUTPUT time: TIME
  INITIALIZATION
    time = 0
  TRANSITION
    [ time_elapses:
      time_can_advance(cal, time_out, time) -->
           time' IN { t: TIME | is_next_event(cal, time_out, t) } ]
  END;
  ...

  nodes: MODULE = % asynchronous composition of modules node[1] to node[N]

  tta: MODULE = clock [] nodes;
```

TTA Model: Summary

Calendar-Based Model in SAL

- A bounded calendar to model communication channel with delays
- Events in the calendar are frames being transmitted
- Fixed bound on transmission delays
- Timeouts used to specify node behaviors
- Two variants
  - Simplified TTA: no node failures
  - TTA: one node may be Byzantine faulty (no hub failure)

Verification

Expected Synchronization Property

```
synchro: THEOREM
     system |- G(FORALL (i, j: IDENTITY):
           pc[i] = active AND pc[j] = active AND
                    time < time_out[i] AND time < time_out[j] =>
                        time_out[i] = time_out[j] AND slot[i] = slot[j]);
```

Proof Method

- k-induction for obvious lemmas
- abstraction (disjunctive invariant method, Rushby) defined in SAL and verified by induction
- synchronization property implied by the abstraction

Synchronization Property Proved by Induction

Proof Steps

- Auxiliary invariants: proved by k-induction with $k = 1$

```
time_aux1: LEMMA
  tta |- G(FORALL (i: IDENTITY): time <= time_out[i]);

time_aux2: LEMMA
  tta |- G(empty?(cal) OR (cal.send <= time AND time <= cal.delivery));

delivery_delay1: LEMMA
  tta |- G(FORALL (i: IDENTITY):
             event_pending?(cal, i) => event_time(cal, i) = cal.send + propagation);
```

- Final step: k-induction at depth 8:

```
sal-inf-bmc -v 3 -d 8 -i -l time_aux1 -l time_aux2
     -l delivery_delay1 simple_startup4 synchro
...
proved.
total execution time: 258.71 secs
```

Limitation

- This works only for a TTA instance with $N = 2$ nodes!

Constructing More Scalable Proofs

General Approach

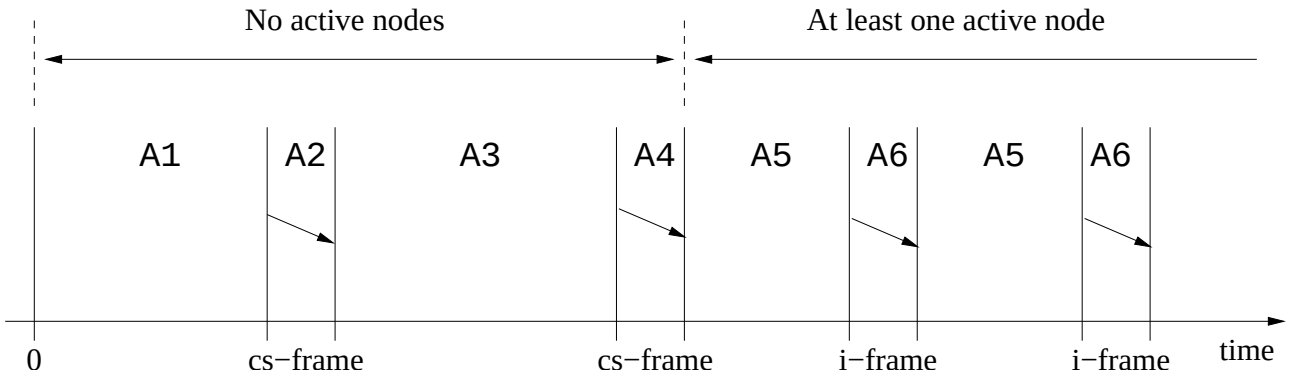
- All proof steps by usual induction (i.e., k-induction with $k = 1$)
- Main issue: find an inductive invariant (as usual)

Methodology

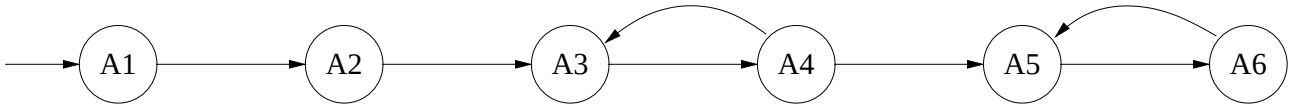
- Bounded model checker used as a tool to find the inductive invariant
- Mechanize an abstraction method based on disjunctive invariants and verification diagrams (Manna & Pnueli, 1994; Rushby, 2000)
- Construction of a correct abstraction done incrementally, guided by counterexamples

Verification Diagram for the Simplified TTA Startup

Protocol Execution



Verification Diagram



Verification Diagram (cont'd)

Example abstraction predicates

```
A1 = empty?(cal) AND (FORALL (i: IDENTITY): pc[i] = init OR pc[i] = listen);

A2 = cs_frame?(cal) AND pc[cal.origin] = coldstart
      AND (FORALL (i: IDENTITY):
              pc[i] = init OR pc[i] = listen OR pc[i] = coldstart)
      AND (FORALL (i: IDENTITY): pc[i] = coldstart =>
              NOT event_pending?(cal, i)
              AND time_out[i] - cal.send >= tau_coldstart(i)
              AND time_out[i] - time <= tau_coldstart(i))
      AND (FORALL (i: IDENTITY): pc[i] = listen =>
             event_pending?(cal, i) OR time_out[i] >= cal.send + tau_listen(i));
```

Showing that the abstraction is correct

- One proof obligation per abstract state: e.g., $A_1(x) \wedge T(x, x') \Rightarrow A_1(x') \vee A_2(x')$
- One more for the initial states: $I(x) \Rightarrow A_1(x)$

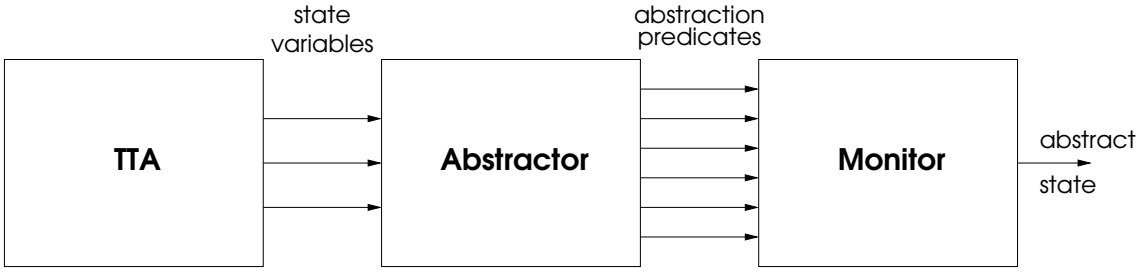
Result

- If the abstraction is correct then $A_1 \vee A_2 \vee \ldots \vee A_6$ is an inductive invariant

Showing the Abstraction Correct in SAL

Auxiliary Modules

- Abstractor: defines Boolean state variables `A1` to `A6`
- Monitor: the verification diagram extended with a `bad` state



Correctness

- Abstraction is correct iff `state /= bad` is invariant
- If the abstraction is correct then `state /= bad` is an inductive invariant (modulo some auxiliary lemmas)

Monitor and Auxiliary Invariants

Monitor Fragment

```
monitor: MODULE =
  BEGIN
    INPUT A1, A2, A3, A4, A5, A6: BOOLEAN
    LOCAL state: abstract_state
  INITIALIZATION
    state = a1
  TRANSITION
     [ state = a1 -->
         state' = IF A1' THEN a1 ELSIF A2' THEN a2 ELSE bad ENDIF
    [] state = a2 -->
         state' = IF A2' THEN a2 ELSIF A3' THEN a3 ELSE bad ENDIF
       ...
    [] ELSE --> state' = bad
```

Auxiliary Invariants

```
abstract_a1: LEMMA system |- G(state = a1 => A1);

abstract_a2: LEMMA system |- G(state = a2 => A2);
```

Abstraction Correctness

```
abstract_invar: LEMMA system |- G(state /= bad);
```

Verification Method: Summary

Direct Proof by k-induction

- For general lemmas (e.g., `time_aux1`, `time_aux2`, `delivery_delay1`)
- For more complex results if you're lucky

Proof by Abstraction

- Define abstraction predicates: `abstractor` module
- Specifies abstraction diagram: `monitor` module
- Show the abstraction correct: `bad` abstract state unreachable
- Last step: show that the invariant `state /= bad` implies the synchronization property (`state /= bad` is equivalent to $A_1 \vee \ldots \vee A_6$)
- All the proofs are done by induction with $k = 1$ (except the last one, $k = 0$)

Modeling Fault

Fault Model for Simplified TTA

- The hub does not fail, at most one faulty node
- The faulty node is Byzantine

SAL Specification

- id of the faulty node: its value is unspecified + a new node state

```
faulty_node: NODE_ID;

PC: TYPE = { init, listen, coldstart, active, faulty };
```

- the faulty node may transition to the `faulty` state at any time

```
...
pc = init AND i = faulty_node -->
   pc' = faulty;
   timeout' IN { x: TIME | time < x };
```

Modeling Faulty Node

In the `faulty` state, the faulty node can send anything at arbitrary times

```
[] faulty_i_frame:
   pc = faulty AND time = timeout -->
      cal' = send_frame(cal, i_frame, i, time + delta1);
      timeout' IN { t: TIME | t > time + propagation }

[] faulty_cs_frame:
   pc = faulty AND time = timeout -->
      cal' = send_frame(cal, cs_frame, i, time + delta1);
      timeout' IN { t: TIME | t > time + propagation }
 ...
```

Verification Results

Example: TTA with 10 nodes

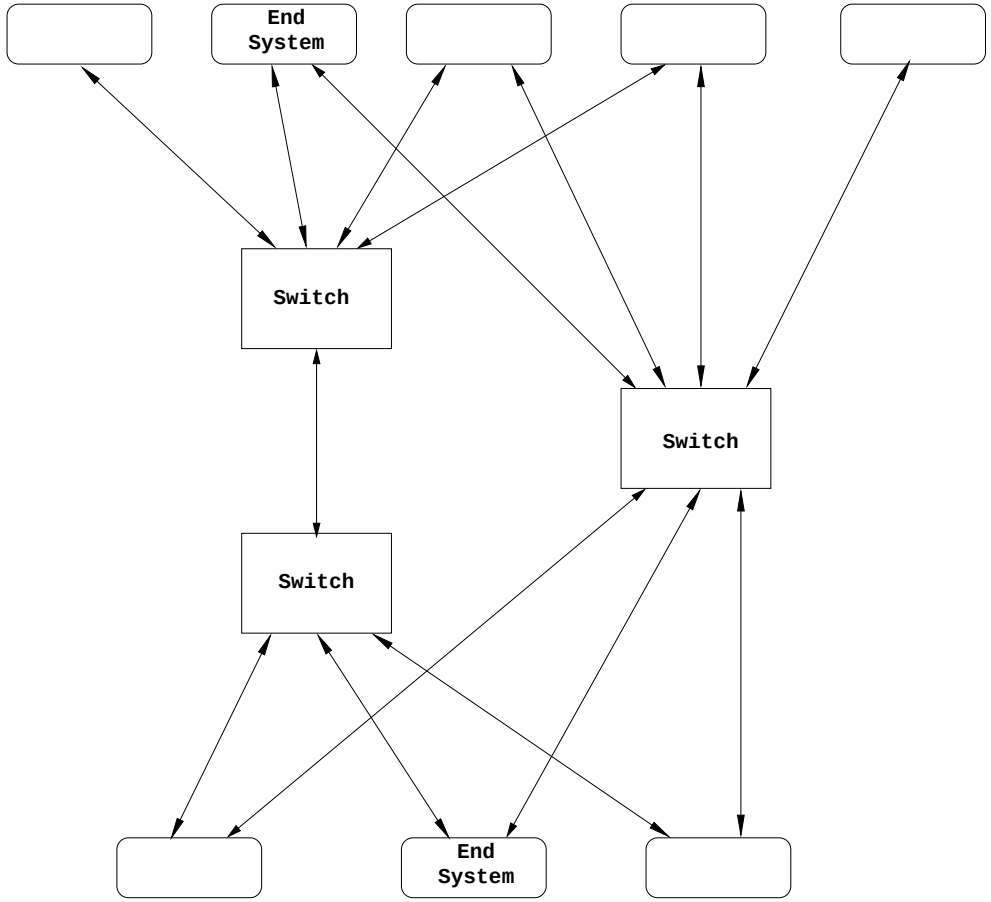
- Transition System Size
  - Simplified Startup (no faults): 13 real variables, 39 discrete variables
  - Fault-tolerant Startup: 25 real variables, 54 discrete variables
- Proof times in seconds

Simplified Startup				Fault-Tolerant Startup			
lemmas	abstract.	synchro	total	lemmas	abstract.	synchro	total
18.59	113.73	2.41	134.73	62.86	44.76	8.3	115.92

On MacBook Pro: 2.66 GHz, Intel Core 2 Duo, using Yices 1.0.34 as the SMT solver

Clock Synchronization in TTE

TTE Architecture



Clock Synchronization Problem

Clocks

- A physical clock C is imperfect: it can drift from real time at some small rate ρ
  Given two times $t_0 < t_1$ we have

$$(t_1 - t_0)(1 - \rho) \quad \leqslant \quad C(t_1) - C(t_0) \quad \leqslant \quad (t_1 - t_0)(1 + \rho)$$

- Given enough time, a slow clock and a fast clock will eventually differ by a large amount
  - in TTE, this will lead to loss of the common time base
  - then real-time communication service will fail (frames can collide)

To Keep TTE Nodes Synchronized

- periodically run a fault-tolerant clock synchronization protocol

Clock Synchronization Protocol in TTE

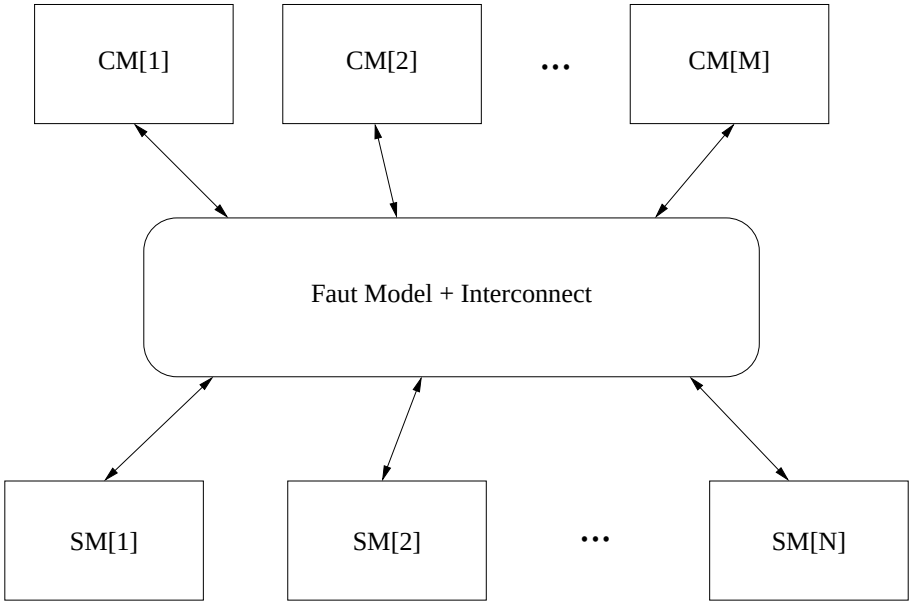
Node Roles

- Synchronization Masters (SMs):
  - broadcast their local clock values
- Compression Masters (CMs):
  - apply a compression function to the clock values they receive from the SMs (this computes a fault-tolerant average)
  - broadcast the compression value to all nodes
- Synchronization Clients (SCs:)
  - receive a compression value from one or more SMs
  - apply a correction to their local clock based on these

Fault Models

- Single-fault scenario: one Byzantine Faulty SM
- Two faults scenario: two faulty SMs, with inconsistent omission faults

Model Structure



Synchronization Master Model

```
SM_STATE: TYPE = { sm_send, sm_correct, sm_drift };

SM: MODULE =
  BEGIN
    INPUT
      compression: ARRAY CM_ID OF CLOCK
    OUTPUT
      state: SM_STATE,
      clock: CLOCK
    INITIALIZATION
      state = sm_send;
      clock = 0;
    TRANSITION
      [ state = sm_send -->
          state' = sm_correct;

     [] state = sm_correct -->
          state' = sm_drift;
          clock' = (compression[1] + compression[2])/2; % correction

     [] state = sm_drift -->
          clock' IN { x : CLOCK | clock - max_drift <= x AND x <= clock + max_drift };
          state' = sm_send;
      ]
   END;
```

Compression Function

Clock Values Received by a CM

- a set of n values $C_0, \ldots, C_n$
- sort them in increasing order: $v_0 \leqslant v_1 \leqslant \ldots \leqslant v_n$

Compression Function in TTE Draft Standard

$$compression = \begin{cases} v_0 & \text{if } n = 1 \\ (v_0 + v_1)/2 & \text{if } n = 2 \\ v_1 & \text{if } n = 3 \\ (v_1 + v_2)/2 & \text{if } n = 4 \\ v_2 & \text{if } n = 5 \\ (v_{K-1} + v_{n-K})/2 & \text{otherwise} \end{cases}$$

Compression Function

Clock Values Received by a CM

- a set of n values $C_0, \ldots, C_n$
- sort them in increasing order: $v_0 \leqslant v_1 \leqslant \ldots \leqslant v_n$

A Better Compression Function

$$compression = \begin{cases} v_0 & \text{if } n = 1 \\ (v_0 + v_1)/2 & \text{if } n = 2 \\ v_1 & \text{if } n = 3 \\ (v_1 + v_2)/2 & \text{if } n = 4 \\ (v_1 + v_3)/2 & \text{if } n = 5 \\ (v_{K-1} + v_{n-K})/2 & \text{otherwise} \end{cases}$$

Compression Master Model

```
CM: MODULE =
  BEGIN
    ...

    TRANSITION
      [ state = cm_receive AND sort(sm_reading, sm_valid, 3, perm') -->
          %% received 3 clock readings (i.e., two faulty SMs)
          state' = cm_correct;
          compression' = sm_reading[perm'[2]];

     [] state = cm_receive AND sort(sm_reading, sm_valid, 4, perm')  -->
          %% received 4 clock readings
          state' = cm_correct;
          compression' = (sm_reading[perm'[2]] + sm_reading[perm'[3]])/2;

     [] state = cm_receive AND sort(sm_reading, sm_valid, 5, perm') -->
          %% received 5 clock readings
          state' = cm_correct;
          compression' = sm_reading[perm'[3]];
     ...
```

Fault Model

Nodes are modeled as non-faulty

Fault assumptions are specified as constraints on what's received from a faulty node

- sm_valid[j][i]: true if CM_j receives something from SM_i, false if CM_j receives nothing from SM_i
- sm_reading[j][i]: clock value that CM_j received from SM_i
- Constraints:
  - If SM_i is nonfaulty then
    `sm_reading[j][i] = sm_clock[i]` and `sm_valid[j][i] = true`
  - If SM_i is omissive faulty then
    `sm_reading[j][i] = sm_clock[i]` and `sm_valid[j][i]` can be either true or false.
  - If SM_i is Byzantine then
    both `sm_reading[j][i]` and `sm_valid[j][i]` are arbitrary.

Verification Problem

Goal

- show that the revised compression function is actually better

Approach

- find/guess the best achievable clock precision Q
- show that the following invariant holds

```
cm_clock_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= Q * max_drift);
```

- show that the following property does not hold

```
cm_clock_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] < Q * max_drift);
```

Results

Original Compression Function: Q is 4

```
cm_clock_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 4 * max_drift);

cm_clock_distance2_strict: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] < 4 * max_drift);
```

Revised Compression Function: Q is 3

```
cm_clock_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 3 * max_drift);

cm_clock_distance2_strict: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] < 3 * max_drift);
```

Conclusion

SMT Solvers in Fault-Tolerant System Verification

- Enable automated or almost automated analysis of infinite state systems (including time and faults)
- Can do much more than finding bugs (induction is the key)
- Currently, still require some human guidance to find lemmas
  But it's easier since SMT solvers give counterexamples when proof fails

Before SMT solvers, model checking was barely applicable to this type of problems, verifications used to require interactive theorem proving (slow, required effort and expertise).

The challenge is to automate the discovery of inductive invariants

- possible methods: interpolants (McMillan), template-based techniques (Tiwari & Gulwani, Kahsai et al.), extensions of IC3 to non-Boolean domains
- not widely applied to the domain of fault-tolerant verification yet

References & Related Work

Alur & Dill, *A Theory of Timed Automata*, Theoretical Computer Science, Vol. 126, Issue 2, April 1994.

Abadi & Lamport, An Old-Fashioned Recipe for Real Time, in *Real Time: Theory and Practice*, LNCS 600, 1992.

Lamport, Real-Time Model Checking is Really Simple, in *Correct Hardware Design and Verification Methods*, LNCS 3725, 2005.

Audemard, Cimatti, Kornilowicz & Sebastiani, Bounded Model Checking for Timed Systems, in *Formal Techniques for Networked and Distributed Systems (FORTE'2002)*, LNCS 2529, 2002.

Dutertre & Sorea, Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata, in *Formal Techniques, Modelling, and Analysis of Timed and Fault-Tolerant Systems*, LNCS 3253, 2004.

References & Related Work

Steiner & Dutertre, SMT-Based Formal Verification of a TTEthernet Synchronization Function, in *Formal Methods for Industrial Critical Systems*, LNCS 6371, 2010.

Steiner & Dutertre, Automated Formal Verification of the TTEthernet Synchronization Quality, in *NASA Formal Methods*, LNCS 6617, 2011.

Bruttomesso, Carioni, Ghilardi & Ranise, Automated Analysis of Parametric Timing-Based Mutual Exclusion Algorithms, in *NASA Formal Methods*, LNCS 7226, 2012.

Manna & Pnueli, Temporal Verification Diagrams, in *Theoretical Aspects of Computer Software*, LNCS 789, 1994.

Rushby, Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification, in *Computer-Aided Verification*, LNCS 1855, 2000.

References & Related Work

McMillan, Interpolation and SAT-Based Model Checking, in *Computer-Aided Verification*, LNCS 2742, 2003.

Gulwani & Tiwari, Constraint-Based Approach for Analysis of Hybrid Systems, in *Computer-Aided Verification*, LNCS 5123, 2008.

Kahsai, Ge & Tinelli, Instantiation-Based Invariant Discovery, in *NASA Formal Methods*, LNCS 6617, 2011.

Bradley, SAT-Based Model Checking Without Unrolling, in *Verification, Model Checking, and Abstract Interpretation*, LNCS 6538, 2011.