Sandia National Laboratories

Exceptional service in the national interest

Building Automated Proofs of Refinement Between State Machines and C

Samuel D. Pollard, Sandia National Labs

Frama-C Days

Maison de la Radio et de la Musique, Paris

14 June 2024, 11:30

SAND2024-07142C

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY

NNSA

Overview

- Sandia National Labs is a US government research & development center
- Sandia develops software for high-consequence embedded control systems

Livermore, California site

Overview

- The systems are relatively simple
- The cost for error is very high
- Requirements relatively complex
- A good use case for formal methods

Emergency Services Sector

Energy Sector

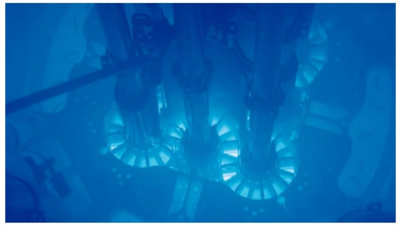
Financial Services Sector

Critical Manufacturing Sector

Dams Sector

Defense Industrial Base Sector

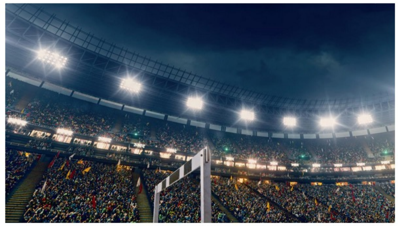
Information Technology



Nuclear Reactors, Materials,

Transportation Systems Sector

Chemical Sector



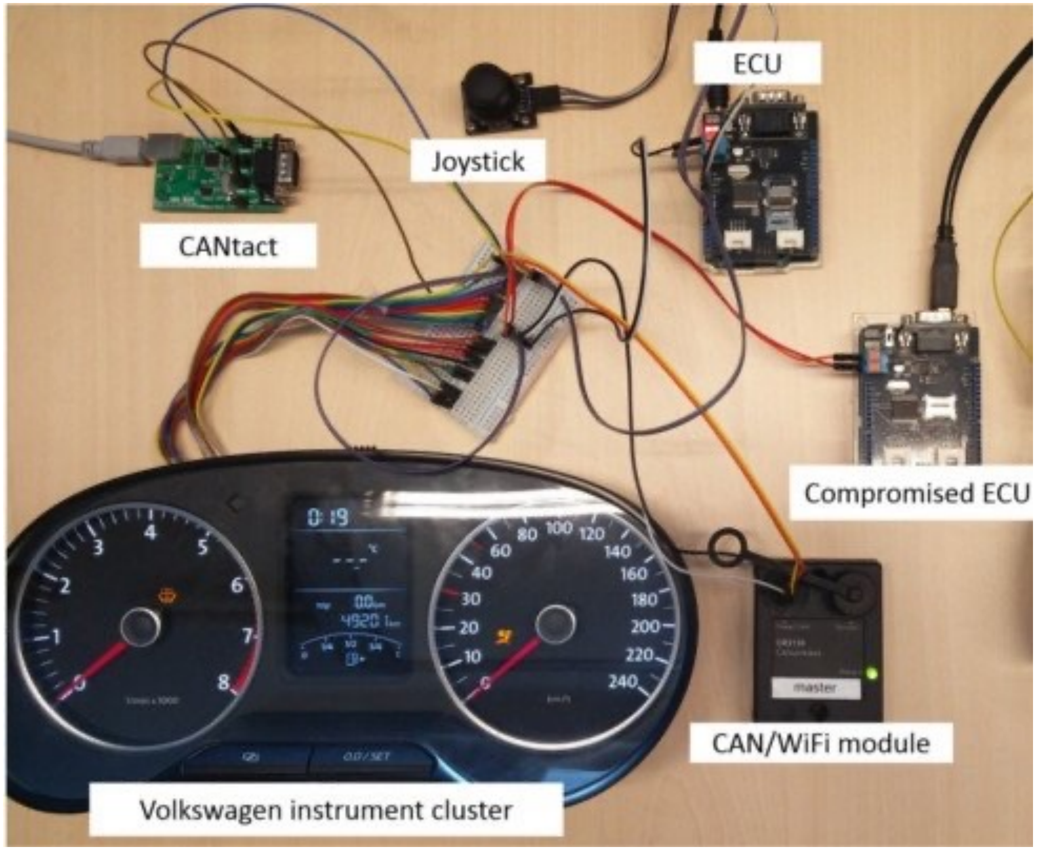
Commercial Facilities Sector

Communications Sector

https://www.cisa.gov/topics/critical-infrastructure-security-and-resilience/critical-infrastructure-sectors

Design Features of High Consequence Systems (HCS)

- Asynchronous interacting components
  - e.g., across a bus
- Requirements documents in English and informal diagrams
- Software implemented in C

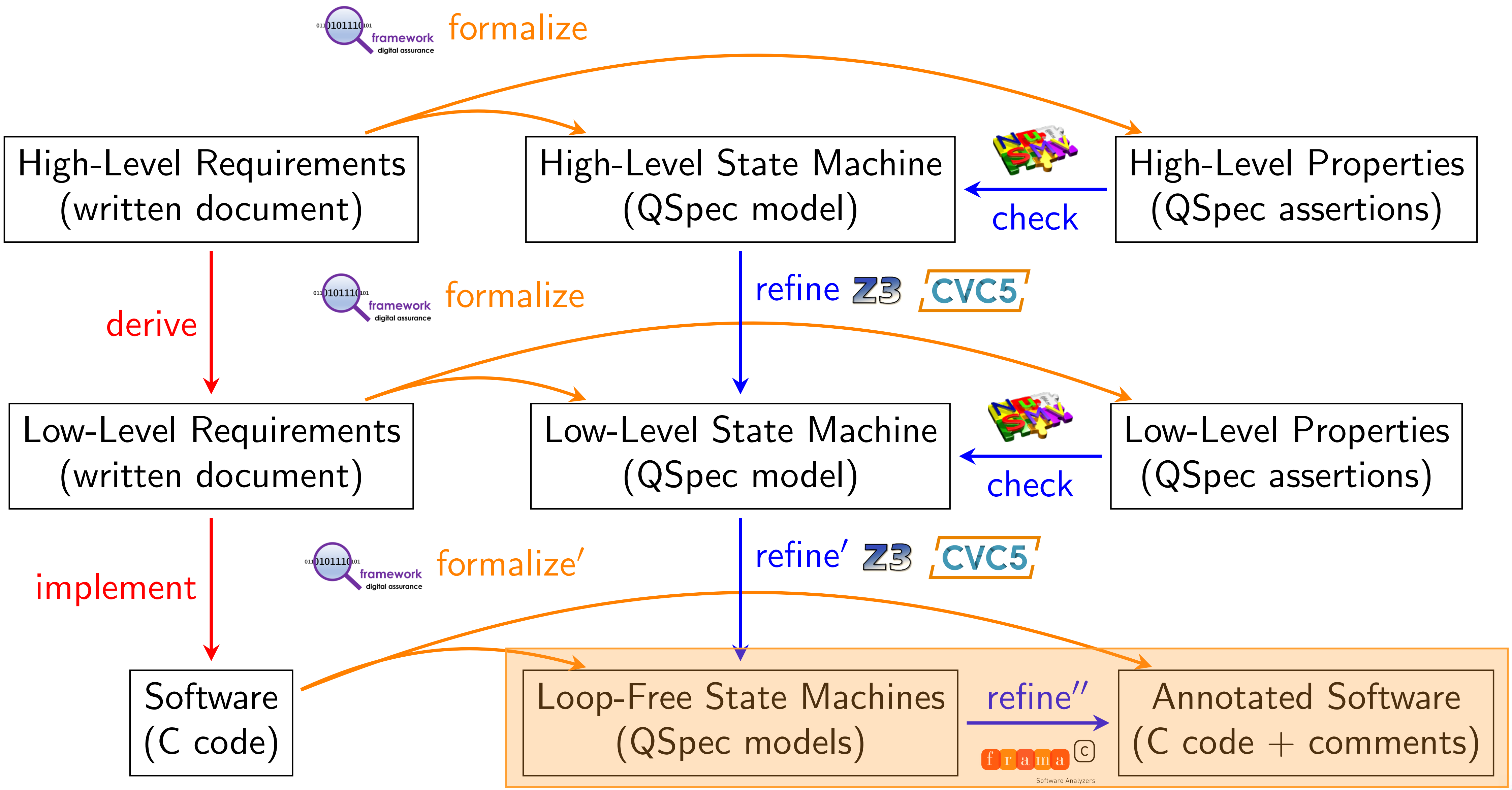


From these, we require proofs of *system-level* properties

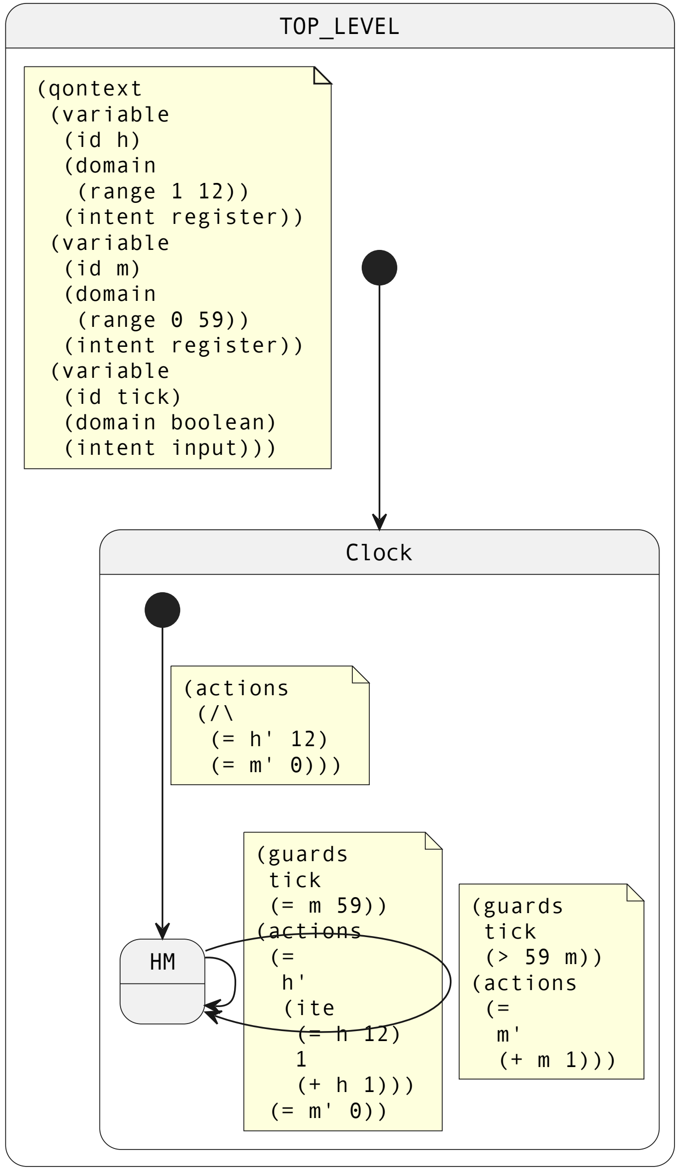
Introducing Q Framework

- Began in 2017
- Verify systems developed using model-based system design (MBSD)
- Leverage solvers for automation
  - NuSMV for LTL/CTL
  - Frama-C
- Currently has ~6 developers
- Part of a broader research group
  - hardware and software understanding
  - modeling, simulation, formal methods
- v1 in OCaml, v2 Haskell

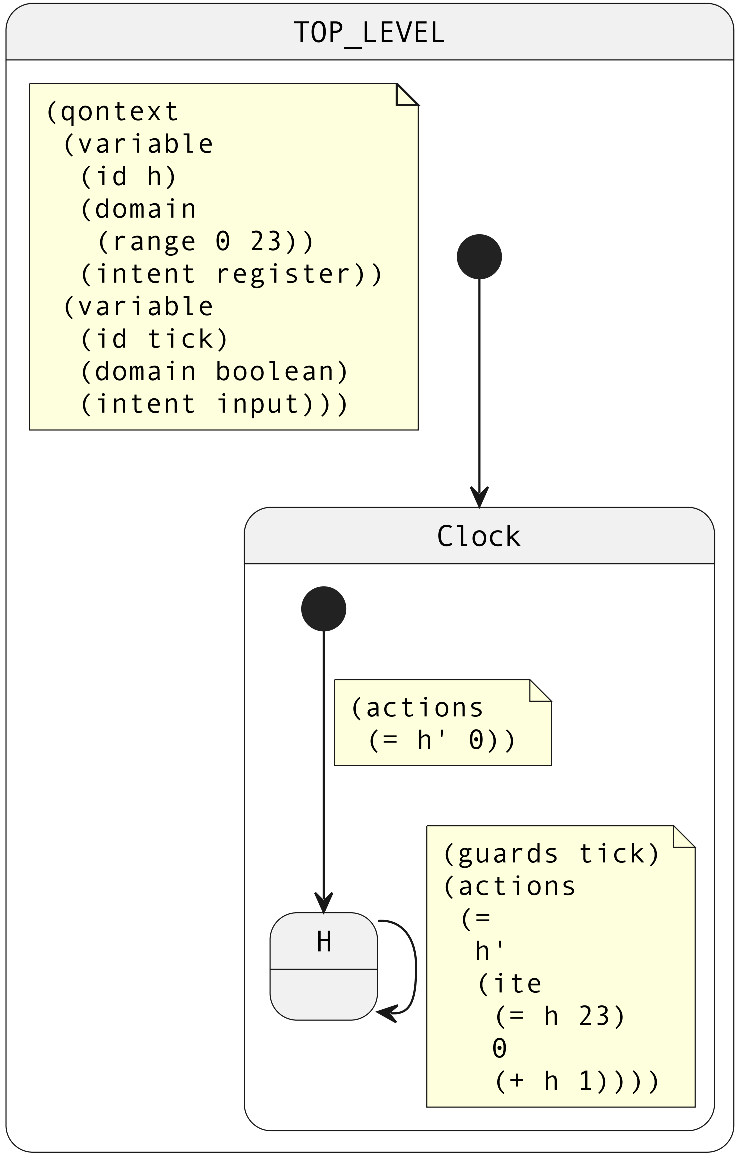
System-Level Refinement



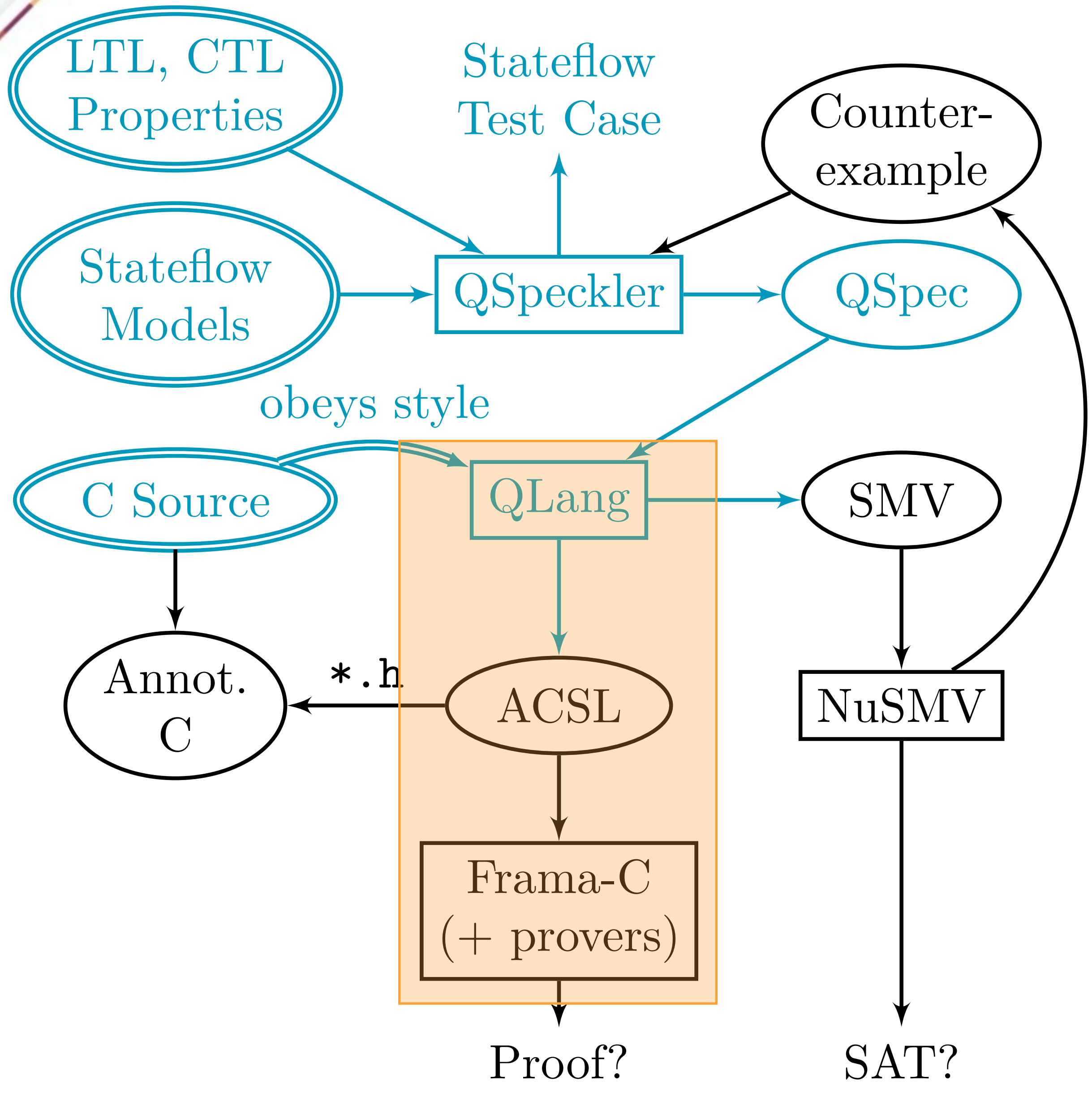
Modeling a Simple Clock



$$\preceq_{weak}$$

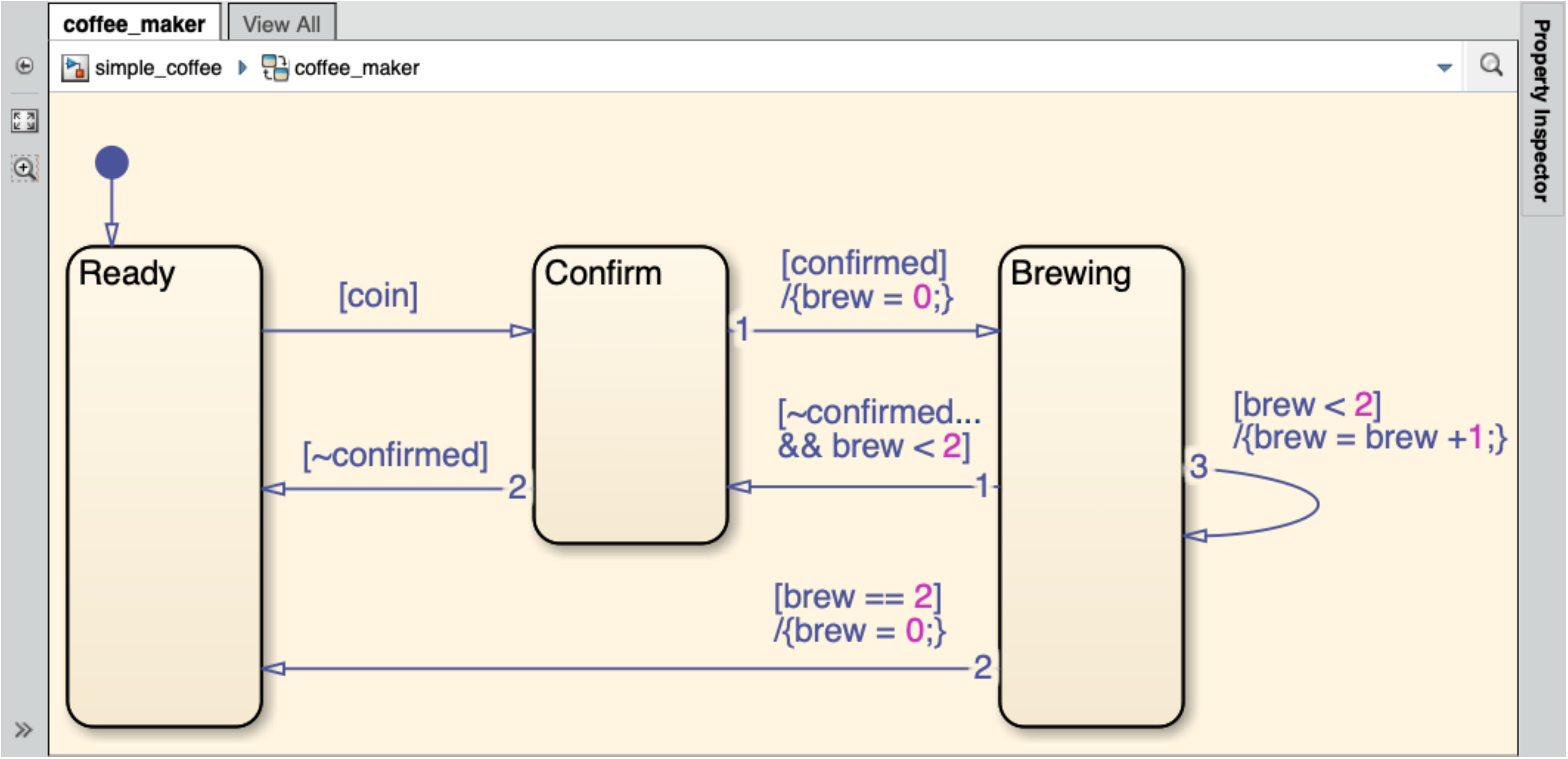


Architecture of Q Framework



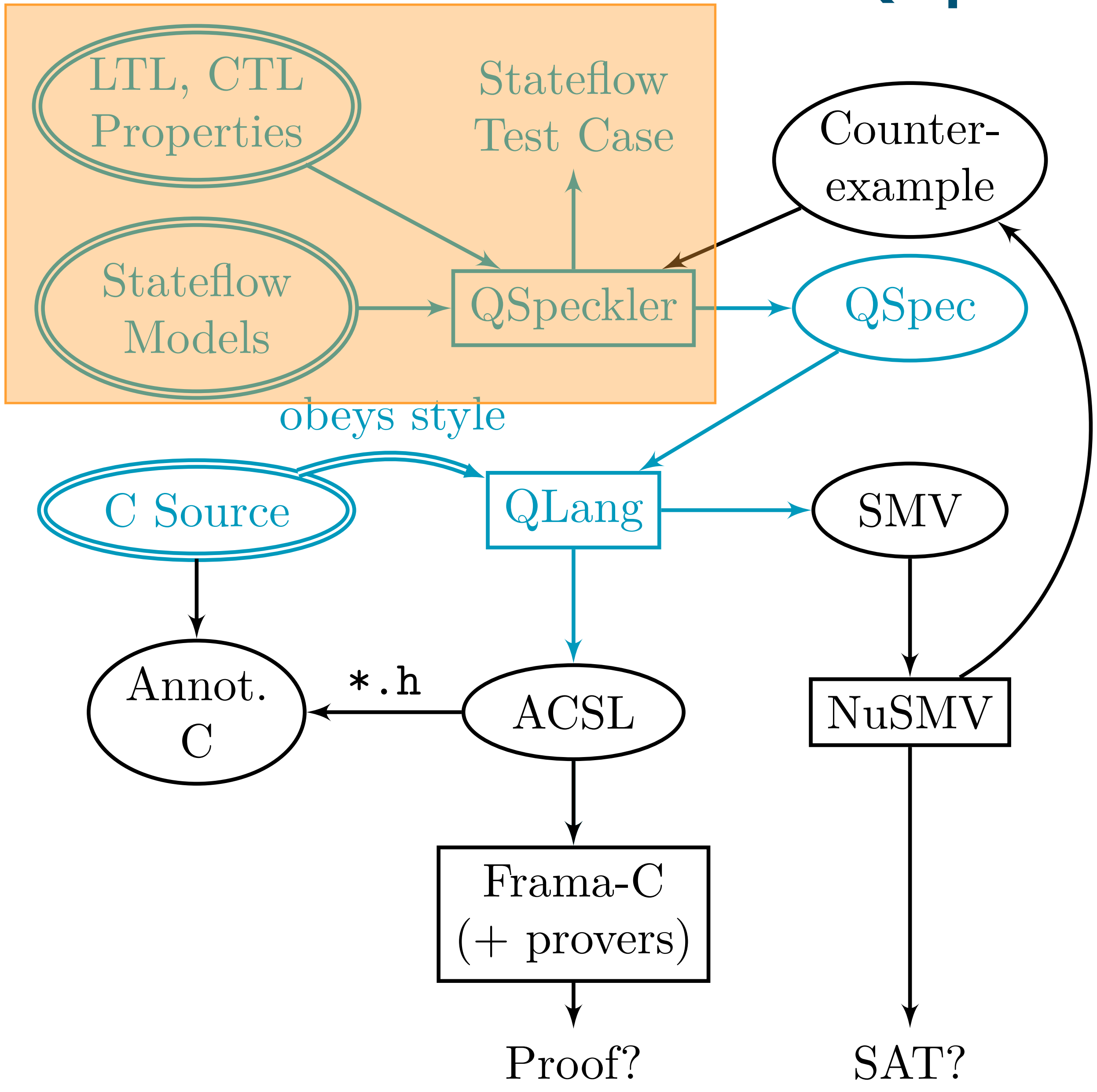
- Blue text: Sandia developed
- Double-struck: Written or checked by hand

Stateflow



coffee_maker
View All
simple_coffee
coffee_maker
Property Inspector
Ready
Confirm
Brewing
[coin]
[confirmed]
/{brew = 0;}
1
[~confirmed...
&& brew < 2]
1
[~confirmed]
2
[brew < 2]
/{brew = brew +1;}
3
[brew == 2]
/{brew = 0;}
2

Convert Stateflow to QSpec



- MATLAB App to generate SC-XML
- MATLAB expression parser
- Convenient UI for testing

QSpec

```xml
<?xml version="1.0" encoding="UTF-8"?>
<qspec>
  <!-- ... other initialization ... -->
  <sequential id="Clock">
    <variable id="tick" domain="boolean" intent="input"/>
    <variable id="h" domain="(range 0 23)" intent="register"/>
    <plain id="H"/>
    <transition type="initial" target="H">
      <assign id="h" ex="0"/>
    </transition>
    <transition source="H" target="H">
      <guard ex="tick"/>
      <assign id="h" ex="(ite (= h 23) 0 (+ h 1))"/>
    </transition>
  </sequential>
</qspec>
```

- Based on SCXML

Preliminaries

- A labeled transition system (LTS) is a triple

$$(S, O, \rightarrow)$$

  states, observations (labels), transition relation

- We are building a refinement between two LTSes

$$P_C \preccurlyeq_{weak} Q$$

  - P_C is a C program
  - Q is a QSpec

- Provided we can think of a C program as an LTS

Preliminaries

To define refinement, we first define partial correctness:

$$\{p\}f\{q\} := \forall s \in \mathtt{ProgState}. \qquad (1)$$
$$s \models p \implies (\forall s' \in \mathtt{ProgState}.\, s[\![f]\!]s' \implies s' \models q)\,,$$

WP's Hoare logic and predicate transformer semantics $[\![\cdot]\!]$

- But for Labeled Transition Systems, correctness is *stuttering-invariant trace equivalence.*

Comparing an LTS with C

- Strict refinement too strong
- Consider

$$\{p\}f\{q\}$$

  - Frama-C cannot describe intermediate states
  - Gives us modularity, but not observational refinement

Observable Events in C

- We require observational refinement
- We borrow CompCert’s notion
  - externally-visible reads and writes
- Nontermination not included here
  - Design requirement
    - infinite event loop with handler
    - handlers are loop free

```
struct machine;
while(true) {
  msg = read_msg();
  if (msg == A)
    handle_A(&m);
  else
    handle_other(&m);
}
```

Weak Simulation

- So, we map observables into transitions in the LTS:

$$P \preceq_{weak} Q := \forall (p,q) \in R, \alpha \in O_P, p' \in S_P. \tag{3}$$

$$p \xrightarrow{\alpha}_P p' \implies \exists q' \in S_Q. \left(q \xrightarrow{\tau^\star}\xrightarrow{\alpha}_Q \xrightarrow{\tau^\star} q' \wedge (p', q') \in R \right),$$

- τ is the silent transition
- S the set of states
- $R \subseteq S_P \times S_Q$
- O an observable (*Label* in typical LTL notation)

Handling Volatile Reads and Writes

- Require any access wrapped in a function call
- Axiomatize hardware access
- Use *ghost state*

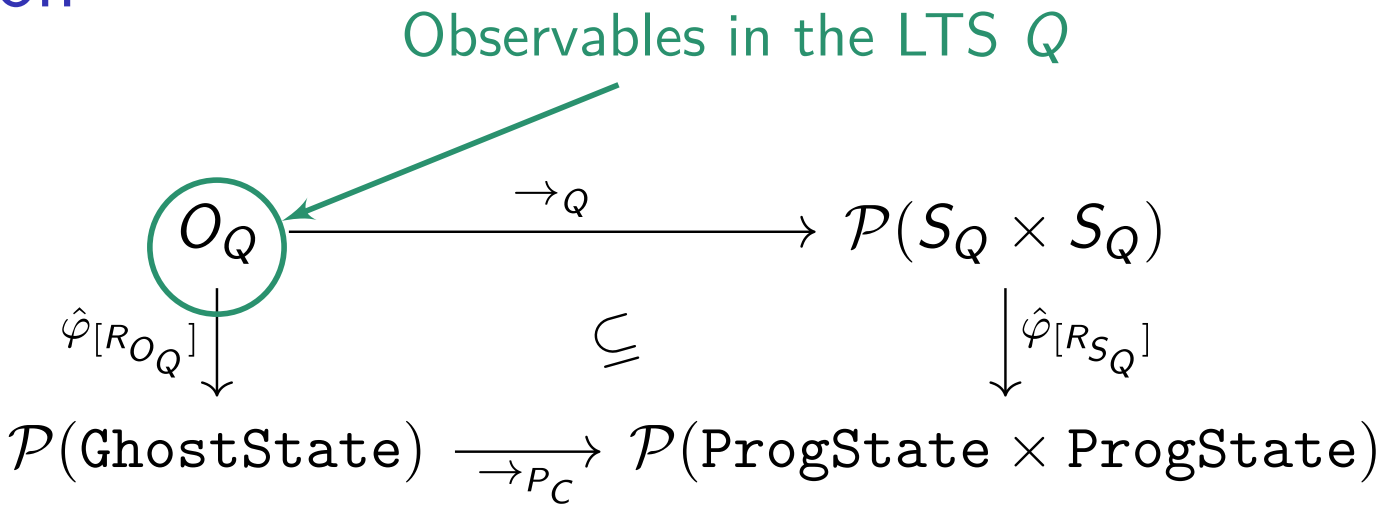
```
/*@
ghost int obs_t;
axiomatic model {
  type obs;
  logic obs obs_at(integer t);
  logic uint8_t fgetCObs(obs o);
} */
volatile uint8_t fgetCVal;
```

Weak Simulation

$$
\begin{array}{ccc}
O_Q & \xrightarrow{\;\to_Q\;} & \mathcal{P}(S_Q \times S_Q) \\
\Big\downarrow{\scriptstyle \hat{\varphi}[R_{O_Q}]} & \subsetneq & \Big\downarrow{\scriptstyle \hat{\varphi}[R_{S_Q}]} \\
\mathcal{P}(\texttt{GhostState}) & \xrightarrow[\to_{P_C}]{} & \mathcal{P}(\texttt{ProgState} \times \texttt{ProgState})
\end{array}
$$

- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\to_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Weak Simulation



- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Weak Simulation

Transition relation of Q

$$\begin{array}{ccc}
O_Q & \xrightarrow{\quad \rightarrow_Q \quad} & \mathcal{P}(S_Q \times S_Q) \\
\hat{\varphi}[R_{O_Q}] \downarrow & \subsetneq & \downarrow \hat{\varphi}[R_{S_Q}] \\
\mathcal{P}(\texttt{GhostState}) & \xrightarrow[\rightarrow_{P_C}]{} & \mathcal{P}(\texttt{ProgState} \times \texttt{ProgState})
\end{array}$$

- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

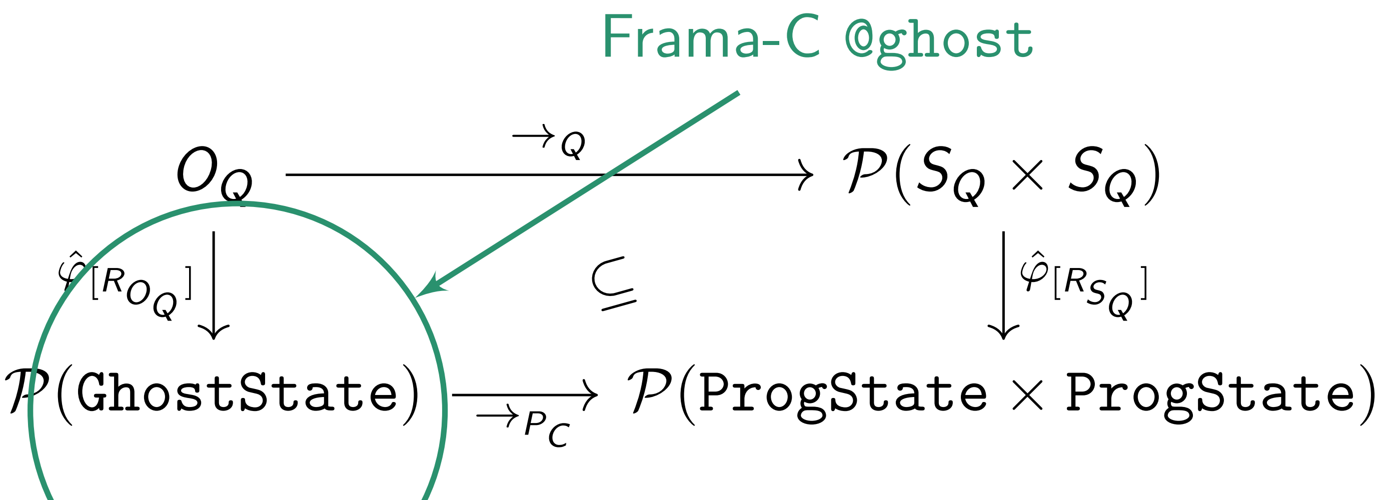
Weak Simulation

Relations over states in Q

$$
\begin{array}{ccc}
O_Q & \xrightarrow{\quad \rightarrow_Q \quad} & \mathcal{P}(S_Q \times S_Q) \\
\Big\downarrow \hat{\varphi}[R_{O_Q}] & \subsetneq & \Big\downarrow \hat{\varphi}[R_{S_Q}] \\
\mathcal{P}(\texttt{GhostState}) & \xrightarrow[\rightarrow_{P_C}]{} & \mathcal{P}(\texttt{ProgState} \times \texttt{ProgState})
\end{array}
$$

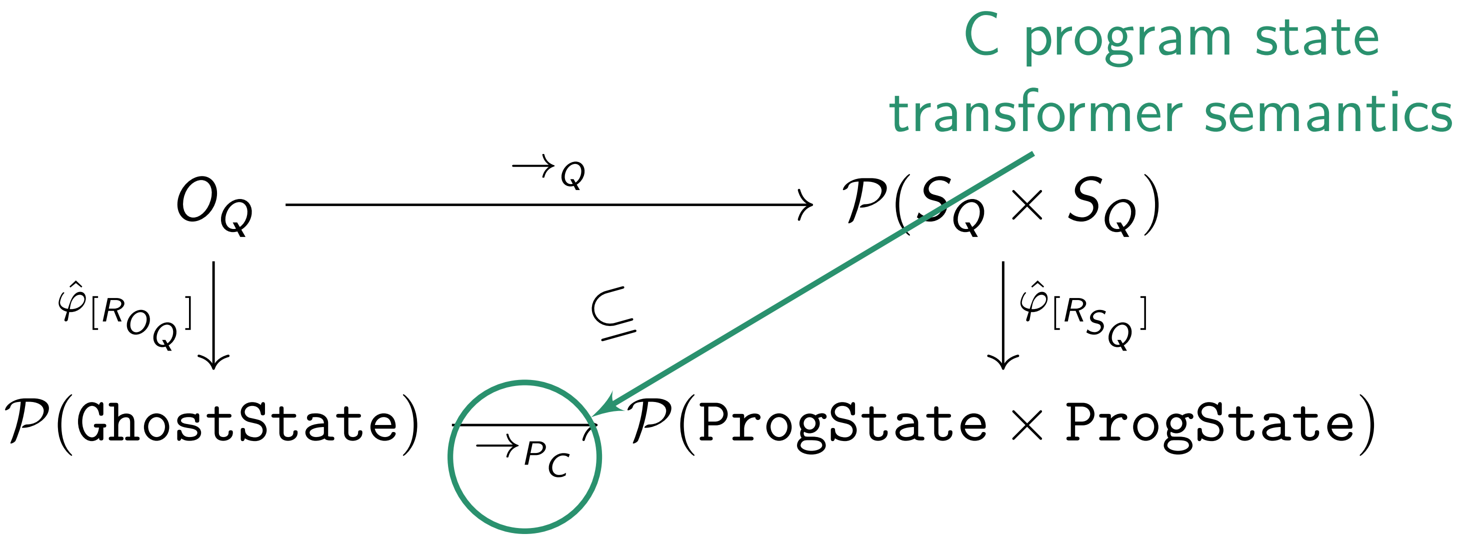
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Weak Simulation



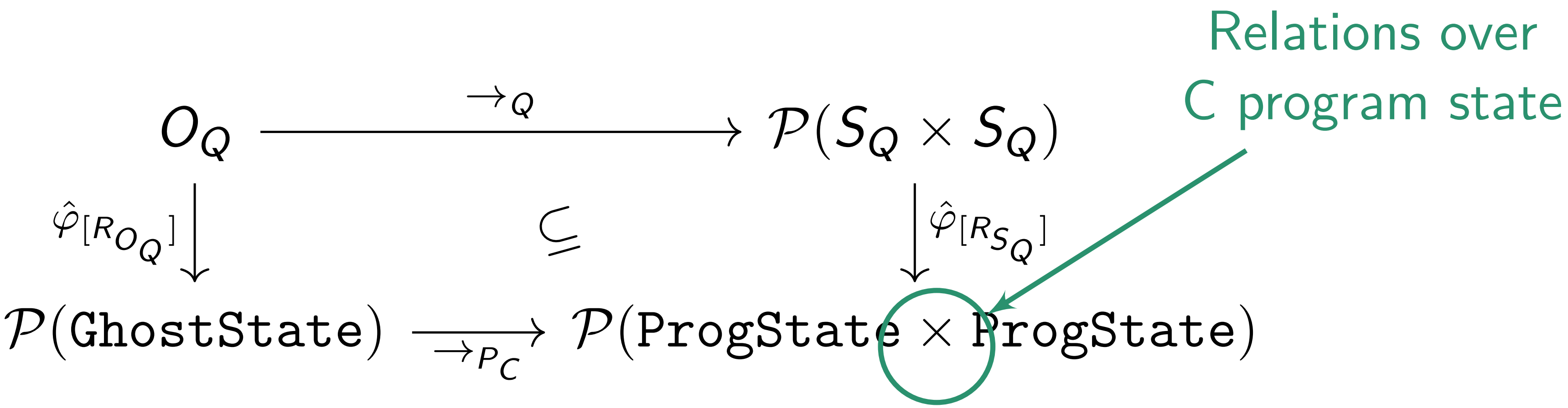
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Weak Simulation



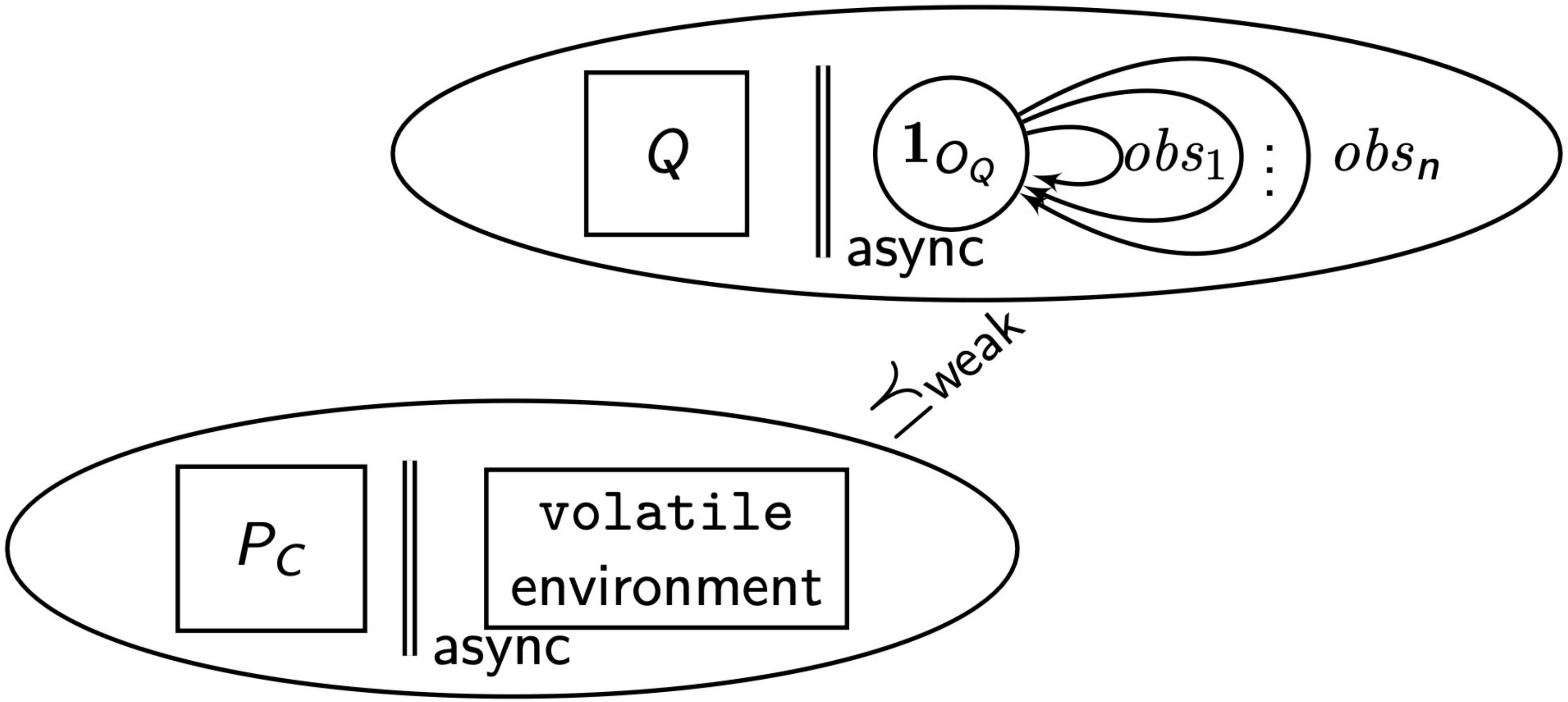
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Weak Simulation



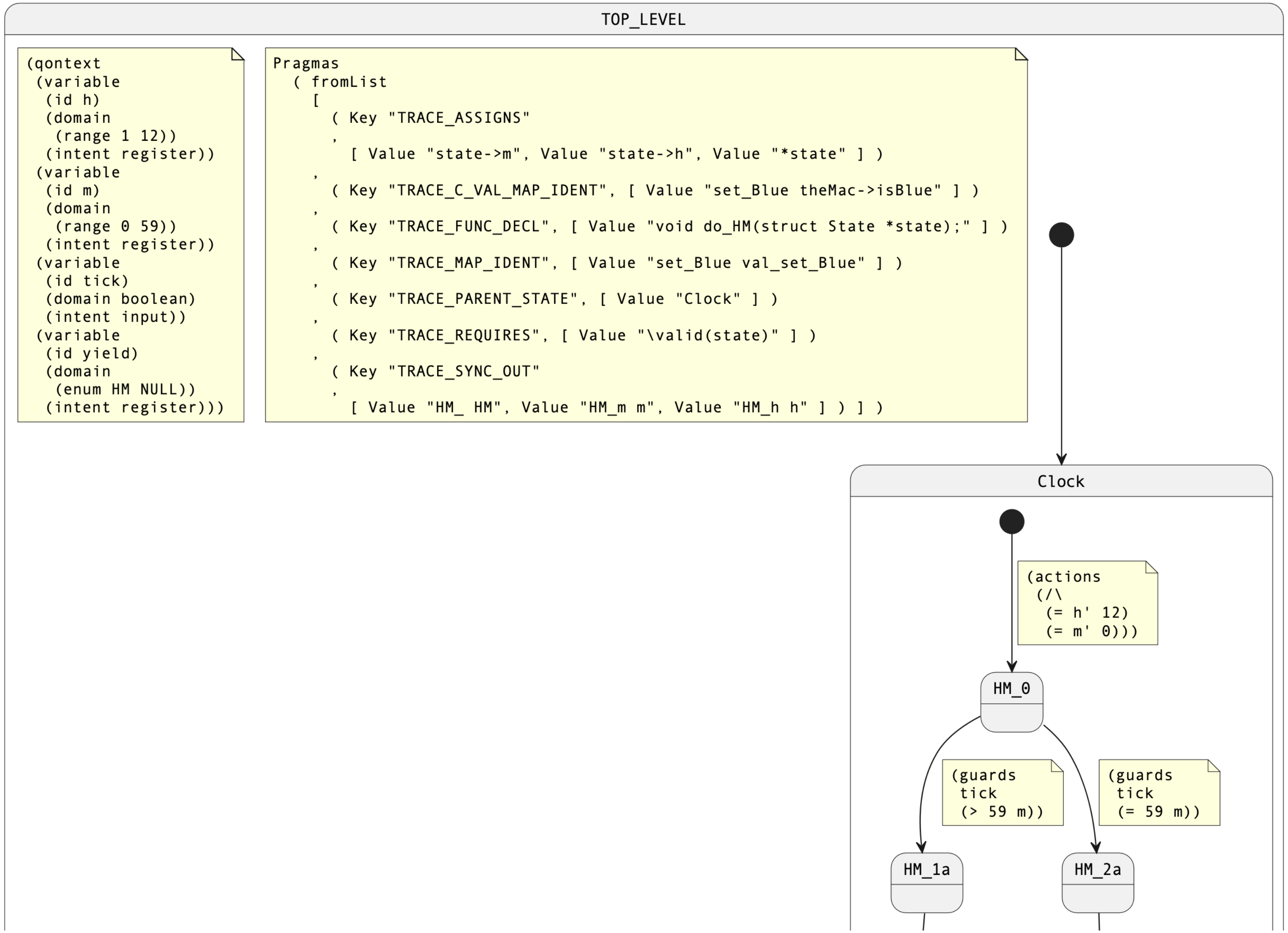
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\to_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation, provided we can think of P_C as a transition system: this is not trivial when considering C semantics

Refinement



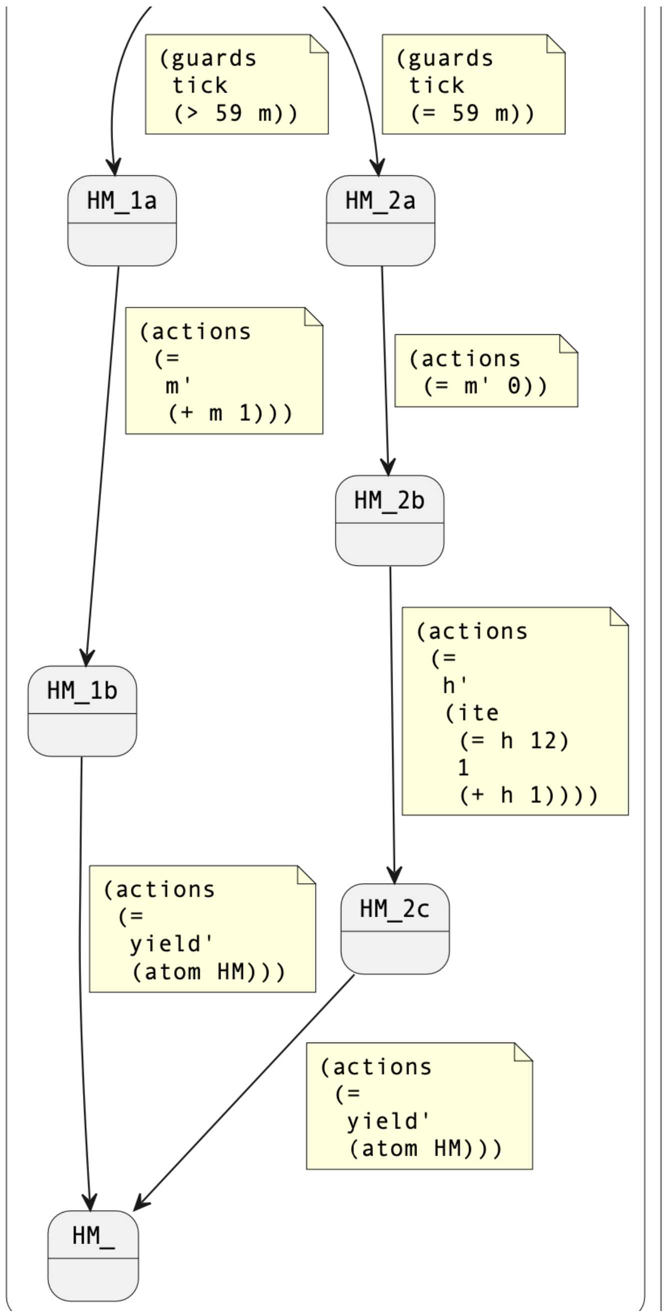
- Above: Composition of the model with an LTS with a single state **1**
- Below: Composition in the C program with an environment for volatiles

Example: Loop Free Machine



- Pragmas to link C with State Machine (Simulation Map)

Loop Free Machine



Generated ACSL

```
/* Generated with qlang */
/*@
axiomatic internal_states_Clock {
  logic gstate HM_1a;
  logic gstate HM_1b;
  logic gstate HM_2a;
  logic gstate HM_2b;
  logic gstate HM_2c;
}
predicate spec_step_Clock(integer t, integer ft) =
  \let gs0 = gstate_at(t+0);
  \let gs1 = gstate_at(t+1);
  (gs0 == HM_0
    && m_at(t+1, 0).tick
    && (59 > m_at(t+1, 0).m)
    && ft == 0 ==> gs1 == HM_1a) &&
        // ... other transitions */
```

```
/*@
// Behaviors -- all paths
behavior Path0000_Clock:
  assumes m_at(oracle_t+1, 0).tick
    && (59 > m_at(oracle_t+1, 0).m);
  ensures ...

behavior Path0001_Clock:
  ...
complete behaviors;
*/
void do_HM(struct State *state);
```

Voilà, the Trace Back-End

- Key idea
  - Enumerate all paths from initial state to terminals
  - Update ghost state, track all guards and actions along each path
  - Provided simulation map, this proves that C refines LTS
- Some Notes
  - Simulation Map can get complex; extra logic for:
    - handling nondeterminism (e.g., messages)
    - WP tactics
    - additional requires/ensures, error states,
  - Even so, most effort goes into generating & interpreting WP

Challenges

- Memory model
  - e.g., unions, bit-level operations
  - Granularity of `assigns` statements
- Counterexample generation
- Floating-point support is limited
- Scale: interpretating results from autogenerated proof obligations

```
Goal Check 'oracle_3' (file state.c, line 23):
Let a = L_m(oracle_t_0, ft_t_0).
Let a_1 = L_m(oracle_t_1, ft_t_3).
Let a_2 = shiftfield_F10_machine_nextState(theMac_0).
Let x = Mint_undef_0[a_2].
Let x_1 = Mint_0[a_2].
Let a_3 = shiftfield_F10_machine_currState(theMac_0).
Let m = Mint_0[a_3 <- 0].
Assume {
  Type: is_uint32_chunk(Mint_0) /\ is_bool(check_side_error_0) /\
      is_bool(old_val_bflushed_0) /\ is_bool(old_val_bit_delay_0) /\
      is_bool(old_val_dflushed_0) /\ is_bool(old_val_faultB_0) /\
      is_bool(old_val_faultD_0) /\ is_bool(old_val_side_err_0) /\
      is_uint32(old_val_set_Blue_0) /\ is_uint32(old_val_set_Green_0) /\
      is_uint32(old_val_set_Red_0) /\ is_sint32(ft_t_0) /\
      is_sint32(ft_t_1) /\ is_sint32(ft_t_2) /\ is_sint32(ft_t_3) /\
      is_sint32(ft_t_4) /\ is_sint32(ft_t_5) /\ is_sint32(old_t_plus_1_0) /\
      is_sint32(oracle_t_0) /\ is_sint32(oracle_t_1) /\
      is_sint32(oracle_t_2) /\ is_sint32(read_msg_if_ready_dev1_0) /\
      is_uint32_chunk(m) /\ is_uint32(x_1) /\ is_uint32(x) /\
      is_uint32_chunk(havoc(Mint_undef_0, m, theMac_0, 17)).
  (* Heap *)
  Type: (region(theMac_0.base) <= 0) /\ linked(Malloc_0).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rw(Malloc_0, a_3, 1).
  (* Call 'periodic_msg' *)
  Have: (x = x_1) /\ valid_rw(Malloc_0, theMac_0, 17).
  (* Call 'sync_code' *)
  Have: P_sync_t(old_val_bit_delay_0, old_val_side_err_0,
          old_val_set_Green_0, old_val_set_Red_0, old_val_set_Blue_0,
          old_val_dflushed_0, old_val_bflushed_0, old_val_faultD_0,
          old_val_faultB_0, ft_t_4, oracle_t_1, oracle_t_2, ft_t_5, 15).
  (* Call 'check_side_error' *)
  Have: ((a_1.F11_call_side_err) != 0) /\
      (((a_1.F11_val_side_err) != 0) <-> (check_side_error_0 != 0)) /\
      P_sync_ft(old_val_bit_delay_0, old_val_side_err_0, old_val_set_Green_0,
        old_val_set_Red_0, old_val_set_Blue_0, old_val_dflushed_0,
        old_val_bflushed_0, old_val_faultD_0, old_val_faultB_0, ft_t_3,
        oracle_t_1, ft_t_4, 7).
  (* Call 'read_msg_if_ready_dev1' *)
  Have: ((L_m(1 + oracle_t_1, 0).F11_is_ready_dev1) != 0) /\
      P_sync_t2(old_val_bit_delay_0, old_val_side_err_0, old_val_set_Green_0,
        old_val_set_Red_0, old_val_set_Blue_0, old_val_dflushed_0,
        old_val_bflushed_0, old_val_faultD_0, old_val_faultB_0,
        old_t_plus_1_0, ft_t_2, oracle_t_0, oracle_t_1, ft_t_3, 4).
  (* Then *)
  Have: read_msg_if_ready_dev1_0 != 0.
  (* Call 'msg_is_Red' *)
  Have: ((L_m(oracle_t_0, ft_t_1).F11_msg) = 8) /\
      P_sync_t0(old_val_bit_delay_0, old_val_side_err_0, old_val_set_Green_0,
        old_val_set_Red_0, old_val_set_Blue_0, old_val_dflushed_0,
        old_val_bflushed_0, old_val_faultD_0, old_val_faultB_0, ft_t_1,
        oracle_t_0, ft_t_2, 3).
  (* Call 'set_Red' *)
  Have: ((a.F11_call_set_Red) != 0) /\ ((a.F11_val_set_Red) = 3) /\
      P_sync_ft(old_val_bit_delay_0, old_val_side_err_0, old_val_set_Green_0,
        old_val_set_Red_0, old_val_set_Blue_0, old_val_dflushed_0,
        old_val_bflushed_0, old_val_faultD_0, old_val_faultB_0, ft_t_0,
        oracle_t_0, ft_t_1, 0).
}
Prove: oracle_t_0 = 3.
Prover Z3 4.11.0 returns Timeout (Qed:26ms) (20s)
```

Future Work

- Open Source: currently in the process
- Formalization in Coq
  - Some parts are proven in Coq
  - Want a formal proof of refinement
  - Composition: have parallel async, want nested composition
- Extend Hoare logic to better handle LTS to C refinement
- Check https://proof.sandia.gov/ for updates
- Thank you!