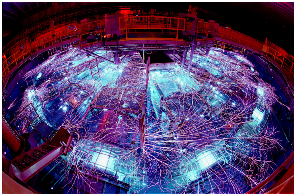
Exceptional service in the national interest

Sandia National Laboratories



Q: A Sound Verification Framework for Statecharts and Their Implementations

Samuel D. Pollard Robert C. Armstrong, John Bender, Geoffrey C. Hulette, Raheel S. Mahmood, Karla Morris, Blake C. Rawlings, Jon M. Aytac

Formal Techniques for Safety-Critical Systems, 7 December 2022, Auckland, NZ

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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. SAND No. SAND2022-16638 C

Introduction

Architecture

Refinement

Coq Formalization

Conclusion

Motivation

- Sandia National Labs is a US government research & development center
- Sandia develops software for high-consequence embedded control systems
- The cost for errors is very high
- Good use-case for formal methods
- Design features:
  - Asynchronous interacting components (e.g., across a bus)
  - Requirements documents in English and informal diagrams
  - Modeled in MATLAB Stateflow as an abstract model
  - Implemented in C
- From these, we require proofs of *system-level* properties.

Overview

- Sandia has the fortune of strong control over structure of C programs, hardware interface, and interaction with software developers and system engineers
- Long history of verification of models (e.g., TLA, SMV) and of implementations directly (e.g., SLAM [3]).
- However, existing research does not support compositional reasoning of state machines while also providing refinement proofs into C
- We developed Q Framework to address this gap and provide (mostly) automated refinement proofs

Stucture of This Presentation

1. Provide overview of Q Framework, piece by piece
   - Use a running example of a "secure coffee maker"
2. Describe our refinement argument between temporal properties of state charts and Frama-C [4] proof obligations
3. Give overview of our formalization of Q Framework in Coq
4. Related work, future work, conclusion

Introduction

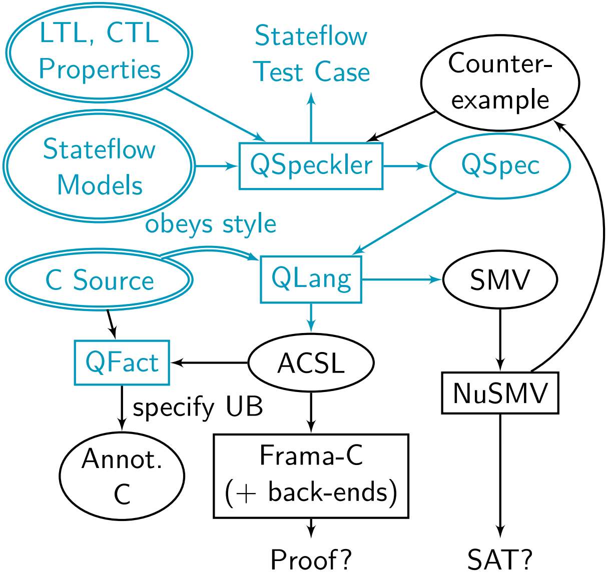
Architecture

Refinement

Coq Formalization

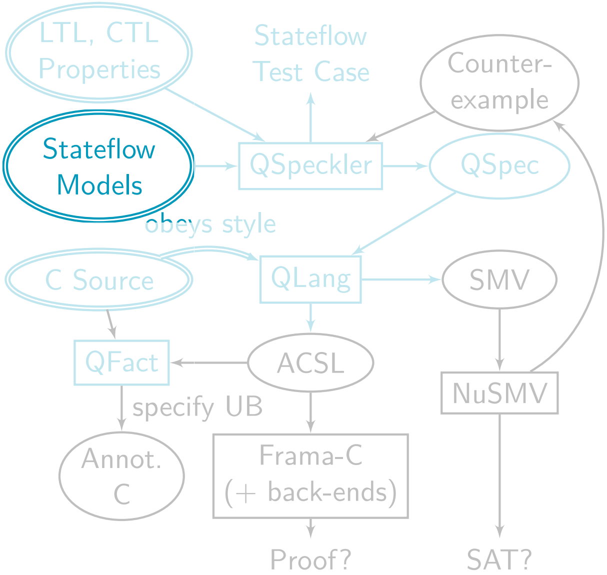
Conclusion

Overview

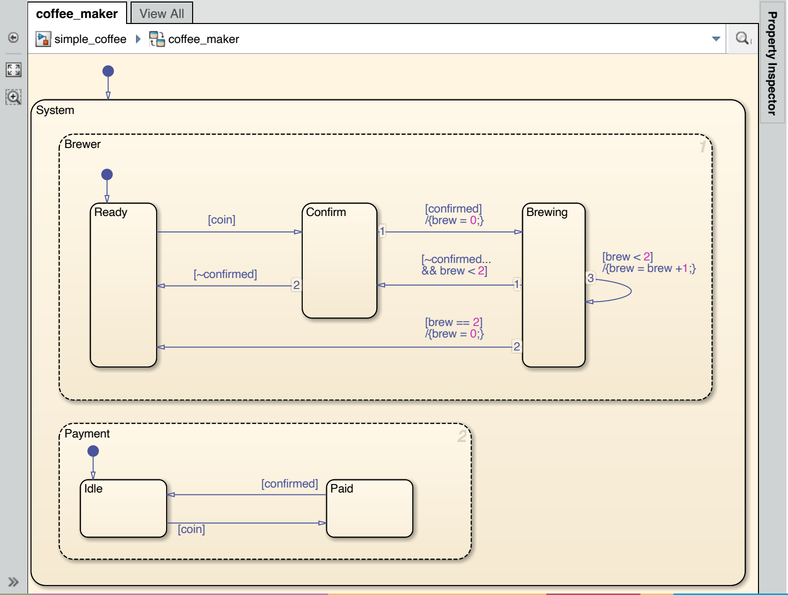


- Blue text Sandia-developed
- Double-struck require manual writing or enforcement

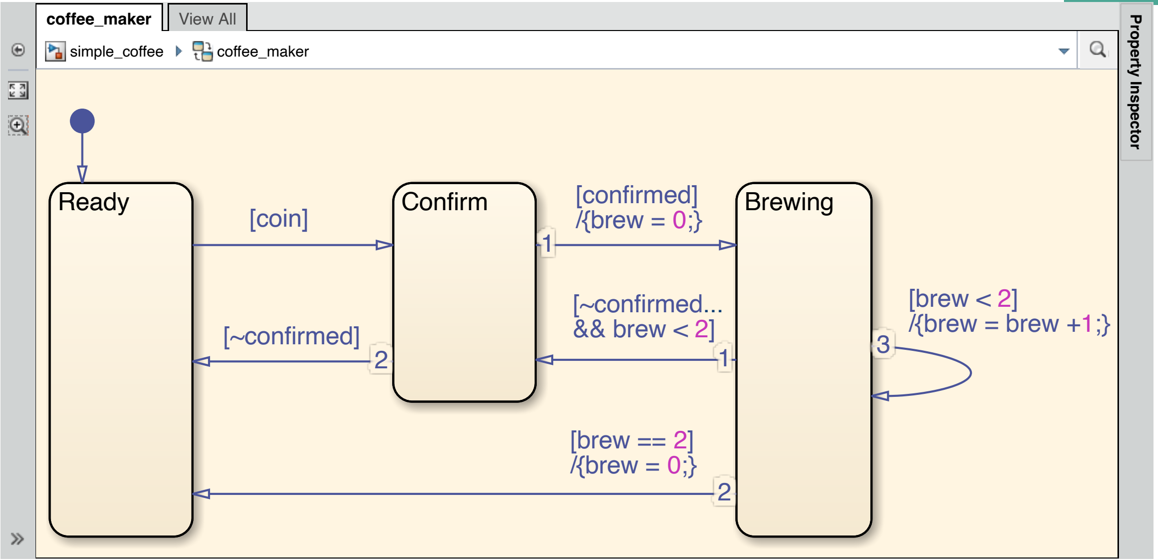
Stateflow


LTL, CTL
Properties
Stateflow
Test Case
Counter-
example
Stateflow
Models
QSpeckler
QSpec
obeys style
C Source
QLang
SMV
QFact
ACSL
NuSMV
specify UB
Annot.
C
Frama-C
(+ back-ends)
Proof?
SAT?

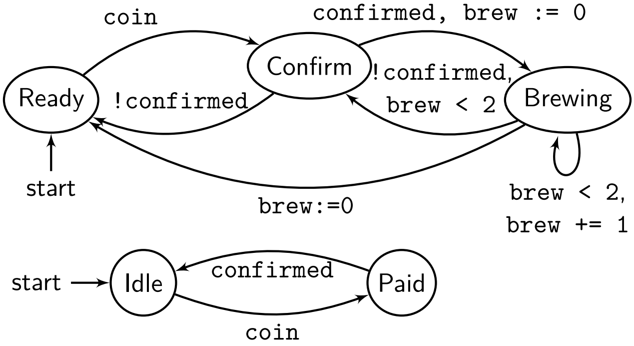
Coffee Maker in Stateflow



Coffee Maker in Stateflow (Zoomed)

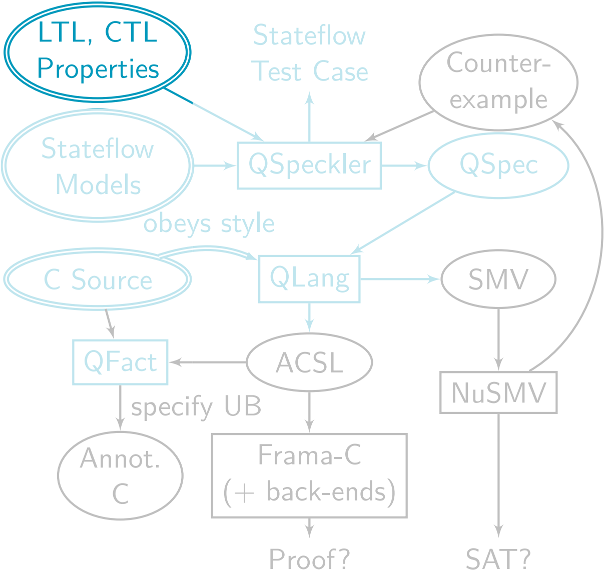


Coffee Maker State Machine

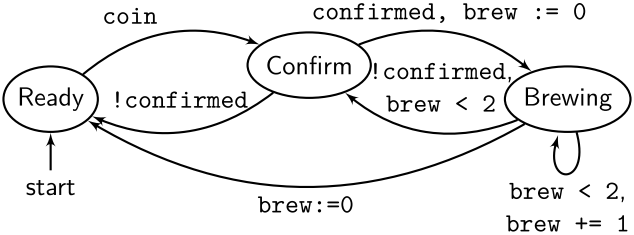


- Coffee maker with confirm and cancel buttons
- “payment” system which continuously pays and presses “confirm.”

LTL/CTL

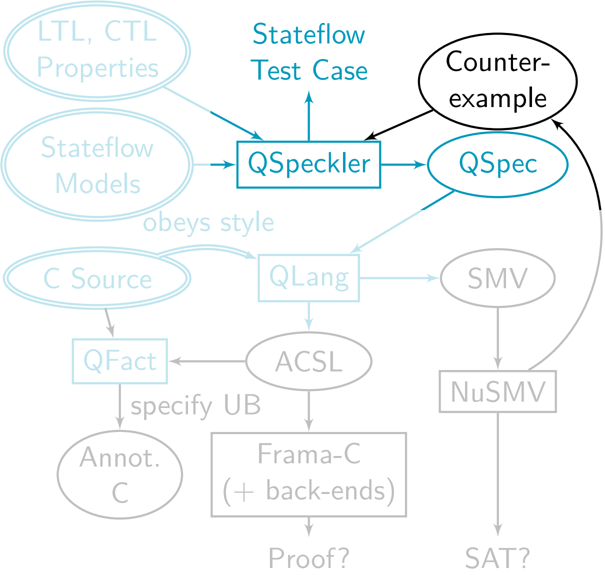

LTL, CTL Properties
Stateflow Test Case
Counter-example
Stateflow Models
QSpeckler
QSpec
obeys style
C Source
QLang
SMV
QFact
ACSL
NuSMV
specify UB
Annot. C
Frama-C (+ back-ends)
Proof?
SAT?

LTL/CTL



- Write properties based on requirements docs
- Example safety condition in CTL:
  - `AG !(state = confirm & brew = 2)`
  - The coffee maker should not be “confirmed” after coffee is done brewing
- We support LTL and CTL because NuSMV does

QSpec and QSpeckler

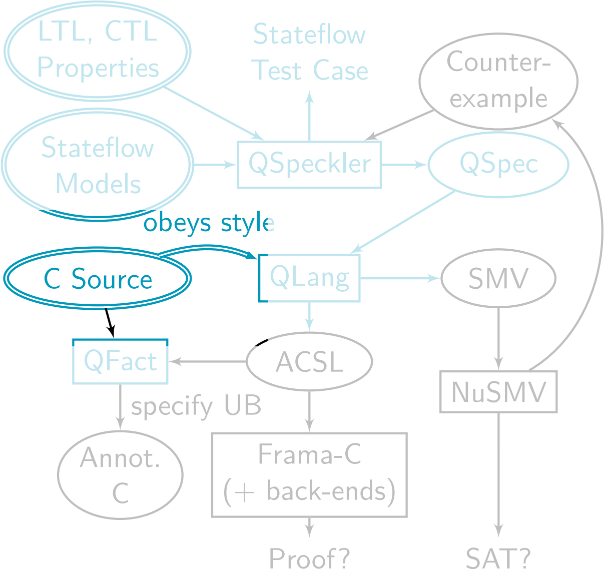


QSpec and QSpeckler

- QSpec inspired by SCXML
- QSpec files (right) aren't written by hand
- QSpeckler translates from Stateflow into QSpec
- QSpeckler understands MATLAB
  - Can generate a Stateflow test case from an SMV counterexample
  - QLang handles the translation into an SMV model

```
<?xml version="1.0" encoding="UTF-8"?>
<qspec> <!-- initialization -->
  <state id="System">
    <parallel>
      <sequential>
        <initial> <!-- ... --> </initial>
        <state id="Brewing">
          <transition label="Brewing_Brewing"
                      target="Brewing">
              <guard name="check_brewing"
               predicate="(< brew 2)"/>
              <assign location="brew"
               expr="(+ brew 1)"/>
          </transition>
        <!-- ... more states -->
      </sequential>
    <parallel>
  </state>
</qspec>
```

C Implementation

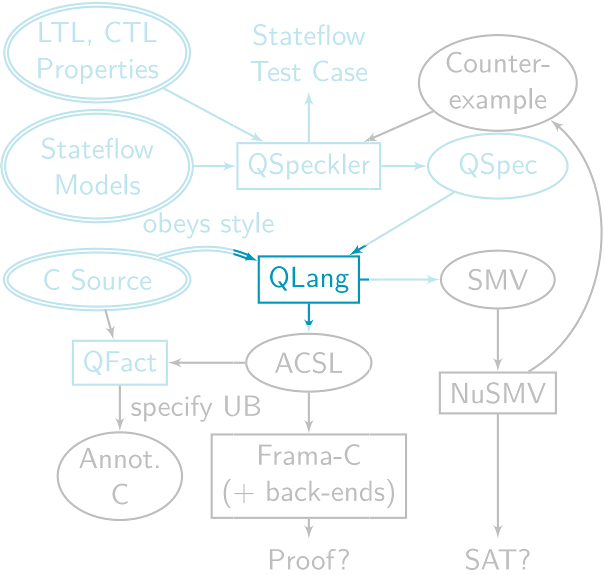


C Coding Standards

- Q Framework expects a restricted subset of C
- Must be able to map from Stateflow to C
- Separate all hardware access (memory-mapped I/O or volatile variables) into function calls
  - Axiomatize the hardware behavior
  - These specifications are written in Frama-C
- These are used for our soundness argument

```
/*@
requires \valid(unsigned char volatile *v);
requires fgetC == v;
ensures obs_t == \old(obs_t) + 1;
ensures \result \in (0 .. 255);
ensures \result <==>
  fgetCObs(obs_at(\old(obs_t)));
*/
uint8_t *volatile_load_uint8_t_(uint8_t *v);
```

QLang



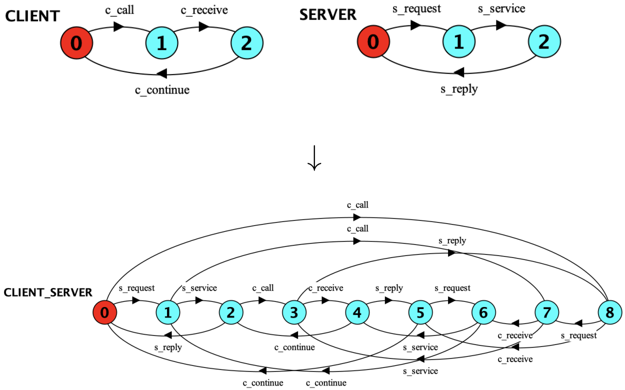
LTL, CTL Properties
Stateflow Test Case
Counter-example
Stateflow Models
QSpeckler
QSpec
obeys style
C Source
QLang
SMV
QFact
ACSL
NuSMV
specify UB
Annot. C
Frama-C (+ back-ends)
Proof?
SAT?

QLang

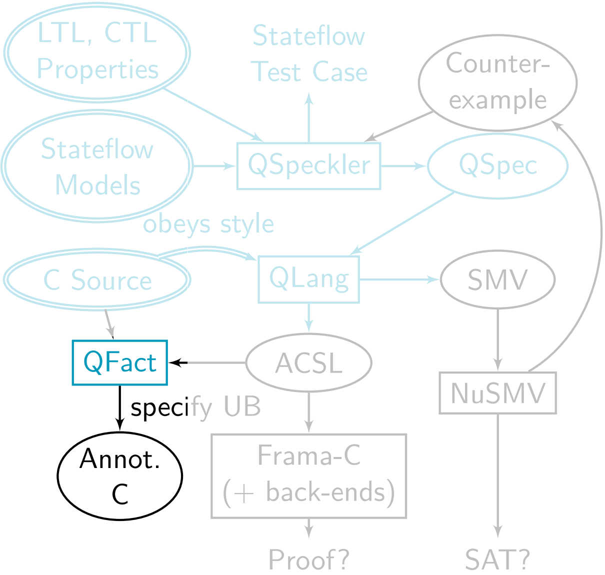
- Input
  1. QSpec (including the desired temporal properties)
  2. C program written in a constrained style
  3. Simulation map between Stateflow and C variables
- Output
  1. “flattened” SMV model
  2. C header file with ANSI C Specification Language (ACSL) annotations
     - These are the proof obligations to be proven by Frama-C
- QLang has several back-ends
  - The most interesting being SMV, but also, e.g., one for visualization

Flattening

- A flattened state chart has no nesting or parallel composition
- Benefit: simple implementation
- Concern: Exponential increase in size of model
  - Can pass onto NuSMV; in practice this sometimes helps
  - Future work to address this (e.g., assume-guarantee reasoning)



QFact


LTL, CTL Properties
Stateflow Test Case
Counter-example
Stateflow Models
QSpeckler
QSpec
obeys style
C Source
QLang
SMV
QFact
ACSL
NuSMV
specify UB
Annot. C
Frama-C (+ back-ends)
Proof?
SAT?

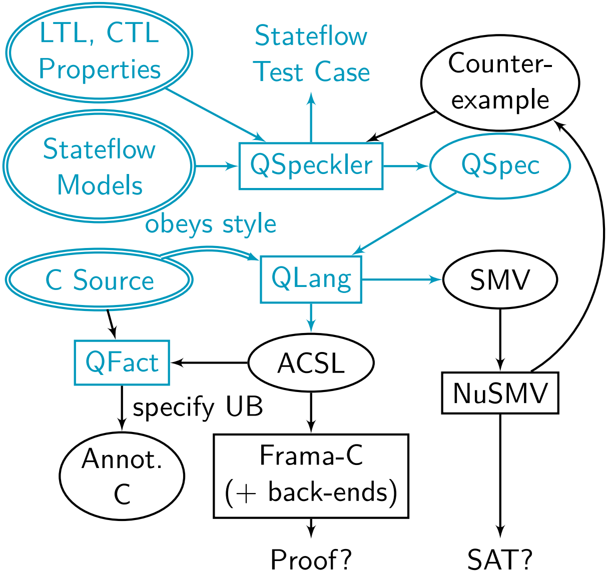
QFact

- Clang tool which annotates a C program with its ACSL specification
- Why is this necessary?

QFact

- Clang tool which annotates a C program with its ACSL specification
- Why is this necessary?
- C semantics are complex
  - Lots of implementation-defined, unspecified, and undefined behavior
  - e.g., evaluation order of function arguments
- Our trick: Convert from C $\rightarrow$ Clight, then back to C
  - Fortunately, CompCert has such a forward translation; we modify it do the reverse

QWorkflow



LTL, CTL Properties
Stateflow Test Case
Counter-example
Stateflow Models
QSpeckler
QSpec
obeys style
C Source
QLang
SMV
QFact
ACSL
NuSMV
specify UB
Annot. C
Frama-C (+ back-ends)
Proof?
SAT?

QWorkflow

- Orchestrate all the moving parts
- Provide:
  - Requirements documents (Microsoft Word, Visio)
  - Each requirement in the Word document has identifier
  - Stateflow model
  - C code
- Runs analysis, generates counterexample (if available), and links the status of each requirement to whether its proof completed in Frama-C and NuSMV

Introduction

Architecture

Refinement

Coq Formalization

Conclusion

The Goal of Q Framework, Restated

- Prove system-level temporal properties
  1. Prove the temporal properties hold for QSpecs
  2. Prove a given C program implements (refines) a component of the QSpec
- 1. is done by encoding QSpec model as SMV, then using NuSMV
- We next describe 2.
  - Generate ACSL function contracts
  - Use Frama-C to prove the C implements these contracts
  - Carefully chose our notions of refinement (model $\rightarrow$ C) and composition
  - With these, any properties we prove of the QSpec also hold for C implementation

Ghost State

- Observations *within* a function call may not be observable to Frama-C, but are observable behavior to C semantics
- Solve this with *ghost state*
- Frama-C annotation to describe whenever the ghost state changes

Frama-C specification:

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

In Clight, use pointer `fgetC`:

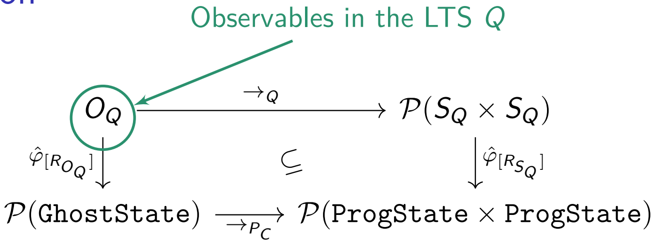
```
$1 = volatile_load_uint8_t_(fgetC);
```

Weak Simulation

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

- 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



- 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

Transition relation of Q

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

- 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

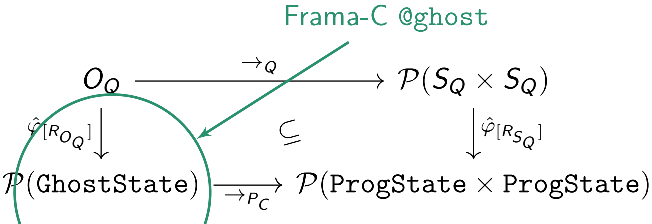
Weak Simulation

Relations over states in Q

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

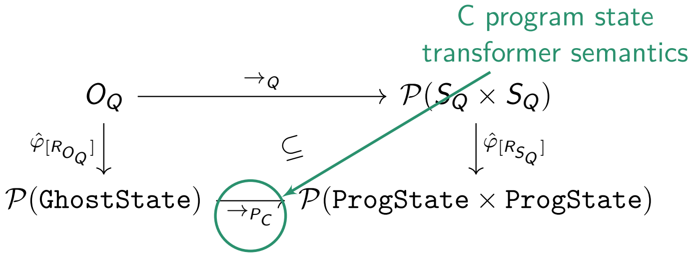
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\to_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation,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

Weak Simulation



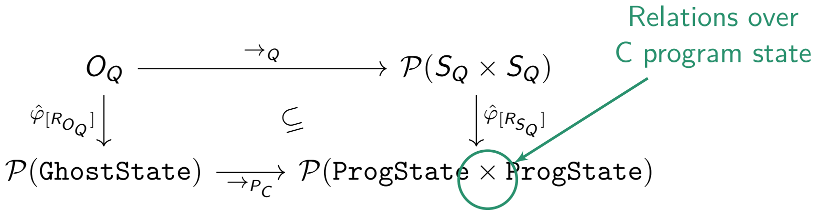
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\to_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation,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

Weak Simulation



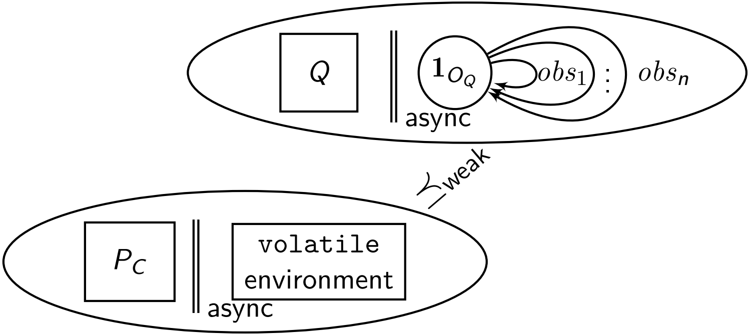
- Q is the abstract model (QSpec)
- P_C is the concrete implementation (C program)
- $\hat{\varphi}$ is a JSON file relating Stateflow variables to predicates over C variables.
- $\rightarrow_Q$ is a Galois connection between O_Q and $\mathcal{P}(S_Q \times S_Q)$
- This demonstrates a proof of weak simulation,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

Weak Simulation



- 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

Refinement



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

Introduction

Architecture

Refinement

Coq Formalization

Conclusion

Coq Formalization

- Have semantics of state charts in Coq
- Model of what we've implemented in Q Framework
- Also provide notion of refinement between two state charts

```
(* S is State
   E is environment (model vars) *)
Record Machine :=
{ m_initial : (S * E) -> Prop;
  m_terminal : (S * E) -> Prop;
  m_inner : S -> E -> E -> Prop;
  m_step : (S * E) -> (S * E) -> Prop
}.
Inductive Chart :=
| Unit : Chart
| Par : Chart -> Chart -> Chart
| Nest : Machine ->
  (S -> Chart) -> Chart.
```

Example: Must Go

```
Theorem qspec_must_go_ind :
  forall qchart qspec data
        cfg1 cfg2 env1 env2,
    qchart = semantics qspec data
    -> chart_step
        qchart
        (cfg1, env1)
        (cfg2, env2)
    -> chart_step_pred
        must_go_pred qchart
        (cfg1, env1)
        (cfg2, env2).
```

- Informally, if a top level state machine can step from $A \rightarrow B$, then it should guarantee that we cannot go from $A \rightarrow A$ as an inner step.
- Q Framework compositional over parallel composition, this states in which cases nested composition is compositional

Introduction

Architecture

Refinement

Coq Formalization

Conclusion

Related Work

- DeepSpec project and the Verified Software Toolchain (VST)
  - strongest assurance arguments
  - a full program logic for C
  - time-intensive
- Modeling with eventB [1], SMT, TLA+
- Trillium [5]: Coq proof of refinement between TLA+ specs and a DSL for specifying concurrent systems, AnerisLang

Future Work

- Add multiple observables per function call
- Size of flattened QSpec model causes scalability concerns
- Modularity of (Stateflow) design *should* allow some modular reasoning
  - Plan to add support for assume-guarantee, circular assume-guarantee reasoning for Q Framework
- We have Coq model of semantics and semantics of C, but not a formal proof of compilation between them
- Less restrictions on C code implementations
- Automatically generate some ACSL specs, especially for pure functions
  - To this effect, use Verified Software Toolchain's (VST) [2] symbolic executor

Conclusion

- Q Framework allows us to build compositional reasoning, and provides evidence that a C implementation refines a given state machine model
- Q has rather strict limitations on the structure of the C
- Future work of "One Q.E.D."
- Not open source, but examples can be found here: `https://github.com/sampollard/q-supplement`

References I

[1] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L.
Rodin: an open toolset for modelling and reasoning in Event-B.
International Journal on Software Tools for Technology Transfer 12, 6 (2010), 447–466.

[2] Appel, A. W.
Verified software toolchain.
In *Proceedings of the 20th European Conference on Programming Languages and Systems* (Saarbrücken, Germany, Mar. 2011), ESOP/ETAPS (LNCS 6602), Springer-Verlag, pp. 1–17.

[3] Ball, T., Cook, B., Levin, V., and Rajamani, S. K.
Slam and static driver verifier: Technology transfer of formal methods inside microsoft.
In *Integrated Formal Methods* (Berlin, Heidelberg, 2004), Springer Berlin Heidelberg, pp. 1–20.

[4] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B.
Frama-c.
In *Software Engineering and Formal Methods* (Thessaloniki, Greece, Oct. 2012), SEFM (LNCS 7504), Springer, pp. 233–247.

[5] Timany, A., Gregersen, S. O., Stefanesco, L., Gondelman, L., Nieto, A., and Birkedal, L.
Trillium: Unifying refinement and higher-order distributed separation logic.
arXiv, Sept. 2021.
Available at `https://arxiv.org/abs/2109.07863`.