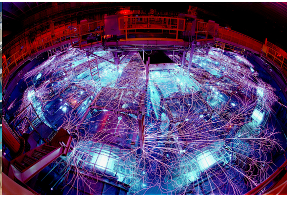
*Exceptional service in the national interest*

Sandia National Laboratories



# The Q Compiler

## for Verifying High-Consequence Controls

Jon Aytac

6/27/18

U.S. DEPARTMENT OF ENERGY NNSA National Nuclear Security Administration

## Motivation

- Suppose you believe that
  - Designs of High Consequence Systems should come with formal proofs of safety and reliability

## Motivation

- Suppose you believe that
  - Designs of High Consequence Systems should come with formal proofs of safety and reliability
  - Or, at least, Designers of High Consequence Systems should be able to check whether their designs satisfy some safety and reliability properties

## Motivation

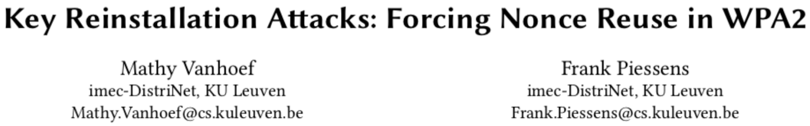
- Suppose you believe that
  - Designs of High Consequence Systems should come with formal proofs of safety and reliability
  - Or, at least, Designers of High Consequence Systems should be able to check whether their designs satisfy some safety and reliability properties
- So you give a presentation to decision makers advocating the adoption of formal methodologies

## Motivation

- Suppose you believe that
  - Designs of High Consequence Systems should come with formal proofs of safety and reliability
  - Or, at least, Designers of High Consequence Systems should be able to check whether their designs satisfy some safety and reliability properties
- So you give a presentation to decision makers advocating the adoption of formal methodologies
- They don't have time to read everything, but they like to stay abreast of what's going on in the literature...

## Disaster!

and they show you this paper of Vanhoef and Piessens in 2017

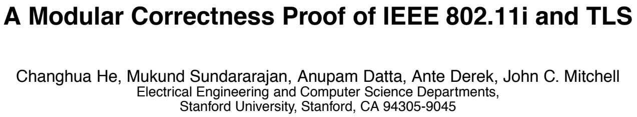


**Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2**

Mathy Vanhoef
imec-DistriNet, KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
imec-DistriNet, KU Leuven
Frank.Piessens@cs.kuleuven.be

which describes security vulnerabilities in a protocol *proven secure* in this paper of He et al in 2005

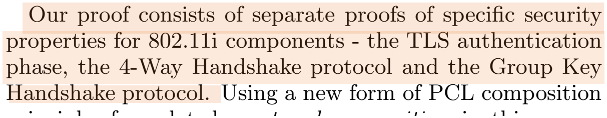


**A Modular Correctness Proof of IEEE 802.11i and TLS**

Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, John C. Mitchell
Electrical Engineering and Computer Science Departments,
Stanford University, Stanford, CA 94305-9045

## Disaster!

He et al Proved Security Properties for a Composition of Protocols ...

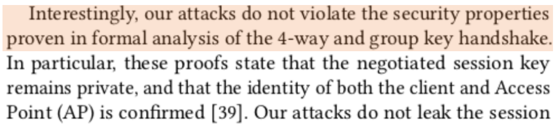


Our proof consists of separate proofs of specific security properties for 802.11i components - the TLS authentication phase, the 4-Way Handshake protocol and the Group Key Handshake protocol. Using a new form of PCL composition

*He et al*

## Disaster!

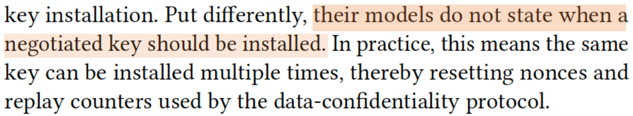
... The Attack didn't Violate Those Properties ...


Interestingly, our attacks do not violate the security properties proven in formal analysis of the 4-way and group key handshake. In particular, these proofs state that the negotiated session key remains private, and that the identity of both the client and Access Point (AP) is confirmed [39]. Our attacks do not leak the session

*Vanhoef and Piessens*

## Disaster!

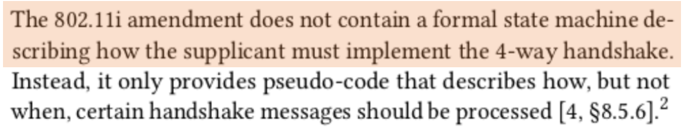
... The Crucial Properties Were Temporal...


key installation. Put differently, their models do not state when a negotiated key should be installed. In practice, this means the same key can be installed multiple times, thereby resetting nonces and replay counters used by the data-confidentiality protocol.

*Vanhoef and Piessens*

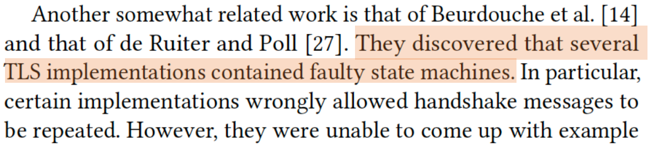
## Disaster

... and the 802.11i amendment didn't specify the protocols in such a way that questions about temporal properties could even be posed ...


The 802.11i amendment does not contain a formal state machine describing how the supplicant must implement the 4-way handshake. Instead, it only provides pseudo-code that describes how, but not when, certain handshake messages should be processed [4, §8.5.6].[2]

*Vanhoef and Piessens*

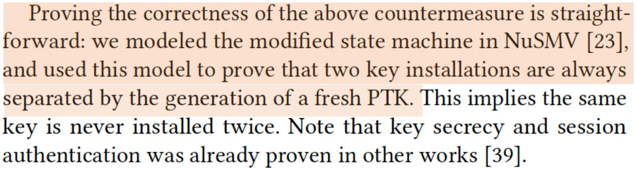
# … Worse yet, the Map isn't the Territory



Another somewhat related work is that of Beurdouche et al. [14] and that of de Ruiter and Poll [27]. They discovered that several TLS implementations contained faulty state machines. In particular, certain implementations wrongly allowed handshake messages to be repeated. However, they were unable to come up with example

*Vanhoef and Piessens*

## ... Vanhoef and Piessens Proved CounterMeasure Correctness in NuSMV...



Proving the correctness of the above countermeasure is straightforward: we modeled the modified state machine in NuSMV [23], and used this model to prove that two key installations are always separated by the generation of a fresh PTK. This implies the same key is never installed twice. Note that key secrecy and session authentication was already proven in other works [39].

*Vanhoef and Piessens*

## Jedi Mind Trick

- So then the decision makers say

## Jedi Mind Trick

- So then the decision makers say
- We need a language for the specification of temporal behavior, preferably one designers would actually use

## Jedi Mind Trick

- So then the decision makers say
- We need a language for the specification of temporal behavior, preferably one designers would actually use
- We need to compile these descriptions into languages suitable for proving (e.g. Why3) and checking (e.g. NuSMV) temporal properties about compositions of specifications

## Jedi Mind Trick

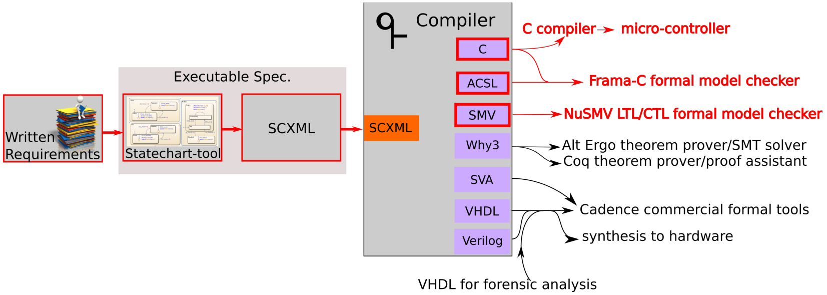
- So then the decision makers say
- We need a language for the specification of temporal behavior, preferably one designers would actually use
- We need to compile these descriptions into languages suitable for proving (e.g. Why3) and checking (e.g. NuSMV) temporal properties about compositions of specifications
- For scalability's sake, the compiler should allow compositional reasoning about systems

## Jedi Mind Trick

- So then the decision makers say
- We need a language for the specification of temporal behavior, preferably one designers would actually use
- We need to compile these descriptions into languages suitable for proving (e.g. Why3) and checking (e.g. NuSMV) temporal properties about compositions of specifications
- For scalability's sake, the compiler should allow compositional reasoning about systems
- We need to prove those properties descend to implementations

## What a Coincidence

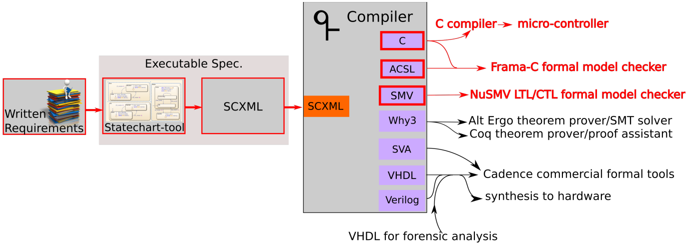
At this point, you say, we just so happen to have already written a tool, the $\mathcal{Q}$ compiler, to do just that!



Written Requirements
Executable Spec.
Statechart-tool
SCXML
Compiler
SCXML
C
ACSL
SMV
Why3
SVA
VHDL
Verilog
C compiler
micro-controller
Frama-C formal model checker
NuSMV LTL/CTL formal model checker
Alt Ergo theorem prover/SMT solver
Coq theorem prover/proof assistant
Cadence commercial formal tools
synthesis to hardware
VHDL for forensic analysis

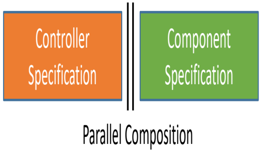
## Q Compiler

Designers draw statechart-like diagrams in their specifications, so $Q$ captures these in a statechart-like formal language


Written Requirements
Executable Spec.
Statechart-tool
SCXML
SCXML
Q Compiler
C
ACSL
SMV
Why3
SVA
VHDL
Verilog
C compiler → micro-controller
Frama-C formal model checker
NuSMV LTL/CTL formal model checker
Alt Ergo theorem prover/SMT solver
Coq theorem prover/proof assistant
Cadence commercial formal tools
synthesis to hardware
VHDL for forensic analysis

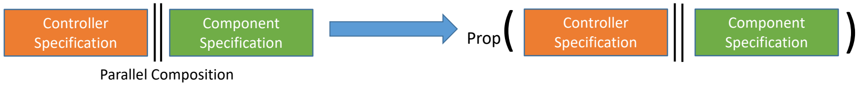
and compiles them into a multitude of languages. We focus here on how $Q$ compiles into ACSL and SMV to enable a workflow with FramaC and NuSMV.

# WorkFlow: Boxes and Arrows



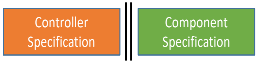
The designer writes abstract specifications for the controller and the component in a Statechart-like language, along with system level properties about their composition

## WorkFlow: Boxes and Arrows

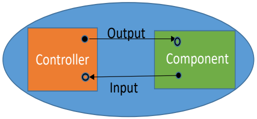


$Q$ then generates an LTL model of the composition of abstract specifications along with the system level properties. These properties may now be checked with model checking tools like NuSMV

# WorkFlow: Boxes and Arrows



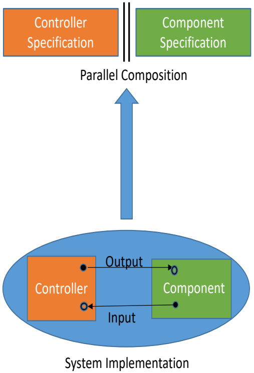
Parallel Composition



System Implementation

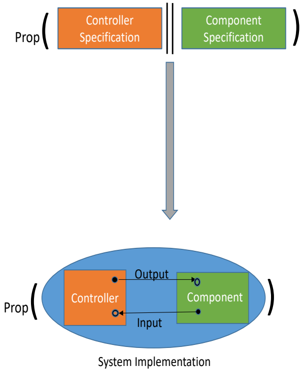
The Engineers then implement the controller in C

## WorkFlow: Boxes and Arrows



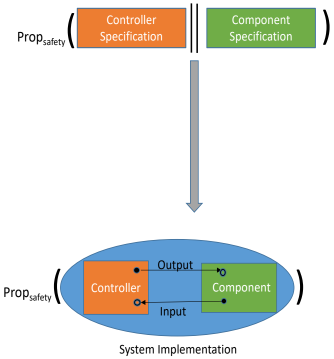
The Engineers need to present some kind of evidence that the specified system abstracts the system implementation

# WorkFlow: Boxes and Arrows



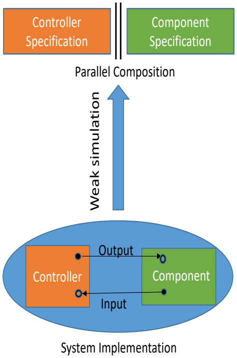
In particular, the evidence of abstraction must be sufficient for proofs of properties about the abstract system to imply proofs of those properties about the system implementation

# WorkFlow: Boxes and Arrows



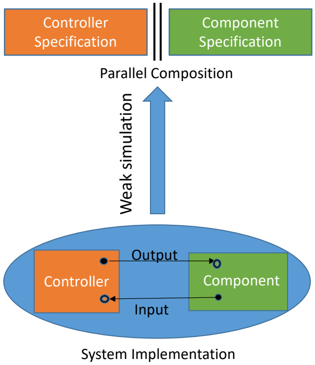
In our context, we are allowed to focus on stuttering invariant safety properties

## WorkFlow: Boxes and Arrows



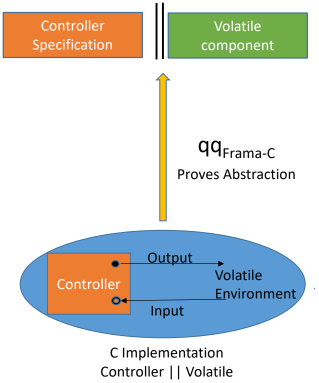
The engineer must therefore present evidence that the com- position of specifications *weakly simulates* the composition of implementations

# WorkFlow: Boxes and Arrows



Only Frama-C meets our requirements for reasoning about C programs, but Frama-C analyzes *sequential* programs. *Fortunately, our C program interacts with components through memory mapped I/O*

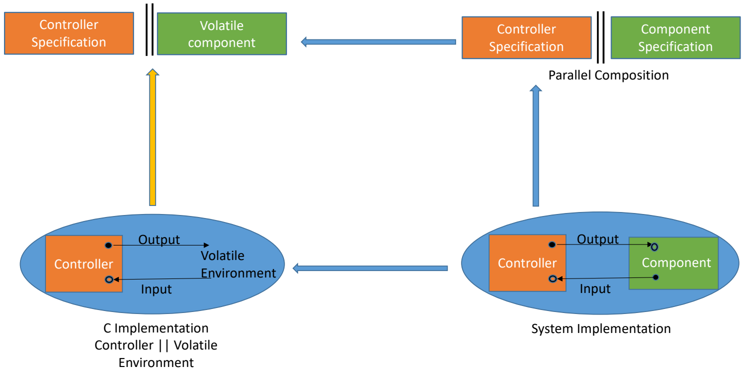
# WorkFlow: Boxes and Arrows



So we ask the engineer to present evidence from which $Q$ can construct , in ACSL, a proof obligation that *the composition of the abstract controller specification with a volatile environment weakly simulates the composition of the implementation with a volatile environment*

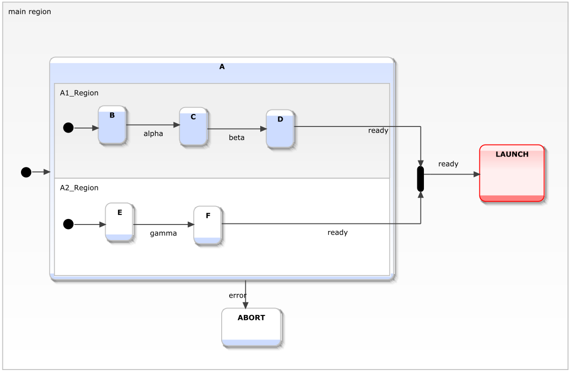
# Workflow: Boxes and Arrows

The proof of this yellow, weak simulation arrow is discharged by Frama-C's Weakest Precondition plugin. The trick is to obtain the blue arrows by construction. Then proof of the yellow arrow gives us our goal, the right column, for free



## The Boxes are StateChart-*like*

The designer writes in a Statechart-*like* language, by which we mean a specification language for reactive programs,inductively defined through hierarchic and parallel composition of transition systems.



## The Boxes: LTS

A labelled transition system $P$ : LTS is a tuple $(S_P, \mathcal{A}_P, \dot{\rightarrow}_P, P_0)$, with $S_P$ the set of states, $\mathcal{A}_P$ an alphabet, the transition relation $\dot{\rightarrow}_P$: $\mathcal{A}_P \rightarrow \mathcal{P}(S_P \times S_P)$ a map from alphabet to relations on states, and $P_0$ a set of initial states

## The Boxes: C Programs as LTS

### Example

- a *program point* and an *execution environment* (which maps variables to values) together constitute the *program state* of a C program.

## The Boxes: C Programs as LTS

### Example

- a *program point* and an *execution environment* (which maps variables to values) together constitute the *program state* of a C program.
- The program is given by the data of a map from program points to commands

## The Boxes: C Programs as LTS

### Example

- a *program point* and an *execution environment* (which maps variables to values) together constitute the *program state* of a C program.
- The program is given by the data of a map from program points to commands
- Given a program state, the evaluation of the command found at that program state's program point defines a *transition* to new program state.

## The Boxes: C Programs as LTS

### Example

- a *program point* and an *execution environment* (which maps variables to values) together constitute the *program state* of a C program.
- The program is given by the data of a map from program points to commands
- Given a program state, the evaluation of the command found at that program state's program point defines a *transition* to new program state.
- The *label* of that transition is some predicate on the execution environment.

## The Boxes: C Programs as LTS

### Example

- a *program point* and an *execution environment* (which maps variables to values) together constitute the *program state* of a C program.
- The program is given by the data of a map from program points to commands
- Given a program state, the evaluation of the command found at that program state's program point defines a *transition* to new program state.
- The *label* of that transition is some predicate on the execution environment.
- In this way, we can think about C programs as labelled transition systems.

# The Arrows: Simulation Relations

## Definition

- For any $P, Q$ : LTS with $\mathcal{A}_P = \mathcal{A}_Q$, a relation $R \subseteq S_P \times S_Q$ is a **simulation relation** if and only if $\forall (p, q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$

$$p \overset{\alpha}{\to}_P p' \Rightarrow \exists q' \in S_Q : \left(q \overset{\alpha}{\to}_Q q' \wedge (p', q') \in R\right)$$

$Q$ simulates $P$, written $P \preceq Q$ or $P \to Q$, if $P_0 \subset R^{-1}(Q_0)$.

# The Arrows: Simulation Relations

## Definition

- For any $P, Q$ : LTS with $\mathcal{A}_P = \mathcal{A}_Q$, a relation $R \subseteq S_P \times S_Q$ is a **simulation relation** if and only if $\forall (p, q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$

$$p \overset{\alpha}{\rightarrow}_P p' \Rightarrow \exists q' \in S_Q : \left( q \overset{\alpha}{\rightarrow}_Q q' \wedge (p', q') \in R \right)$$

  $Q$ simulates $P$, written $P \preceq Q$ or $P \rightarrow Q$, if $P_0 \subset R^{-1}(Q_0)$.

- The relation $R$ is said to be a **witness** for $P \preceq Q$, and $Q$ is said to be an **abstraction** of $P$. When the witness is a function, the simulation relation is said to be a **refinement**.

## The Arrows: Simulation Relations

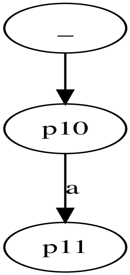
### Proposition

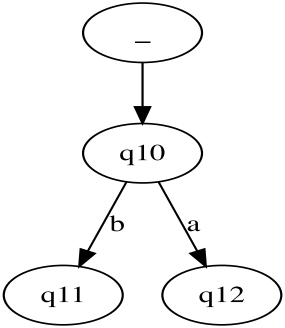
*This preorder is sound in that* $P \preceq Q \Rightarrow$ *trace*$(P) \subseteq$ *trace*$(Q)$. *So, if* $P \preceq Q$, *then for any* $\mathbb{P}_Q : \mathrm{Prop}(Q)$ *temporal property on* $Q$ *and* $\mathbb{P}_P : \mathrm{Prop}(P)$ *the corresponding temporal property on* $P$, $\mathbb{P}_Q \Rightarrow \mathbb{P}_P$.

## An Example

### Example

Here, $P_1 \preceq Q_1$





$P_1$

$Q_1$

## Memory Mapped I/O, Volatile Variables and the Terminal Abstraction

### Definition

The terminal abstraction over the alphabet $\mathcal{A}$ is the labeled transition system $1_{\mathcal{A}_0} = \left(\star, \mathcal{A}, \rightarrow_{1_{\mathcal{A}}}, \star\right)$ with $\forall \alpha \in \mathcal{A}. \rightarrow_{\mathcal{A}}$: $\alpha \rightarrow \star \times \star$.

# Memory Mapped I/O, Volatile Variables and the Terminal Abstraction

## Definition

The terminal abstraction over the alphabet $\mathcal{A}$ is the labeled transition system $1_{\mathcal{A}_0} = \left(\star, \mathcal{A}, \rightarrow_{1_{\mathcal{A}}}, \star\right)$ with $\forall \alpha \in \mathcal{A}. \rightarrow_{\mathcal{A}}: \alpha \rightarrow \star \times \star$.

## Proposition

*The terminal abstraction over a given alphabet abstracts any machine of the same alphabet* $\forall P = \left(S_P, \mathcal{A}_P, \dot{\rightarrow}_P, P_0\right)$*,* $P \preceq 1_{\mathcal{A}_P}$

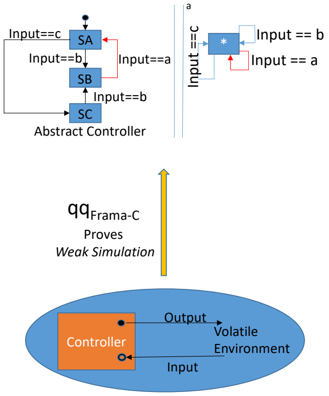
## Memory Mapped I/O and the Terminal Abstraction

Example

Let "volatile uint8_t $\star$I;" a declaration of a reference to a volatile variable of type uint8_t. Then, $\mathcal{A}_I = \{(\star I == 0), \cdots, (\star I == 255)\}$, and the presence of a volatile variable in a $C$ program is an **asynchronous** parallel composition with $1_{\mathcal{A}_I}$ , which we'll write as, $C\|^a 1_{\mathcal{A}_I}$

## Proving Spec Simulates Implementation

So we will use the Q compiler and FramaC to prove



C Implementation
Controller || Volatile
Environment

# Proving Spec Simulates Implementation

So the designer gives a formal specification of their design in our statechart-like language, e.g. the ABC example

$$\begin{tikzcd} & A \arrow[dl, bend right, "c"'] \arrow[dr, bend left, "b"] & \\ C \arrow[ur, bend right, "b"'] & & B \arrow[ul, bend left, "a"'] \end{tikzcd}$$

The implementer of the C program *should give a witness that their program is simulated by this abstract transition system.*

# Proving Spec Simulates Implementation

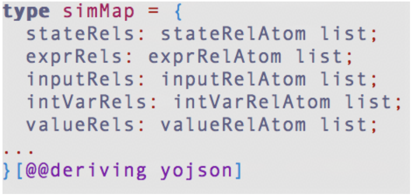
A witness that $Q$ simulates $P$ ($P \preceq Q$ or $P \to Q$) decomposes into a relation on labels $R_{\mathcal{A}}$ and a relation on the states $R_{\mathcal{S}}$

$$\begin{tikzcd}
\mathcal{A}_P \arrow[r, "\dot{\to}_P"] \arrow[d, "f_{[R_{\mathcal{A}}]}"] & \mathcal{P}(S_P \times S_P) \arrow[d, "f_{[R_{\mathcal{S}}]}"] \\
\mathcal{A}_Q \arrow[r, "\dot{\to}_Q"] & \mathcal{P}(S_Q \times S_Q)
\end{tikzcd}$$

such that $f_{[R_{\mathcal{A}}]} \circ \dot{\to}_P \subseteq \dot{\to}_Q \circ f_{[R_{\mathcal{A}}]}$.

## Proving Spec Simulates Implementation

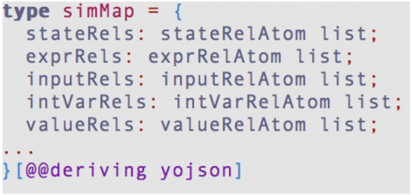
- For C programs thought of as LTS, the labels are expressions over the execution environment.



```
type simMap = {
  stateRels: stateRelAtom list;
  exprRels: exprRelAtom list;
  inputRels: inputRelAtom list;
  intVarRels: intVarRelAtom list;
  valueRels: valueRelAtom list;
...
}[@@deriving yojson]
```

# Proving Spec Simulates Implementation

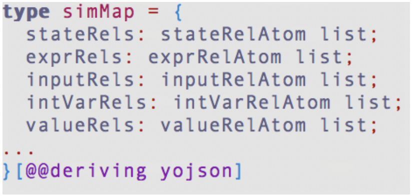
- For C programs thought of as LTS, the labels are expressions over the execution environment.
- so $R_{\mathcal{A}}$ may be given by a relation on internal variables, a relation on values, and a relation on inputs,



```
type simMap = {
  stateRels: stateRelAtom list;
  exprRels: exprRelAtom list;
  inputRels: inputRelAtom list;
  intVarRels: intVarRelAtom list;
  valueRels: valueRelAtom list;
...
}[@@deriving yojson]
```

# Proving Spec Simulates Implementation

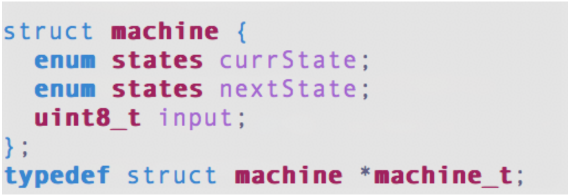
- For C programs thought of as LTS, the labels are expressions over the execution environment.
- so $R_{\mathcal{A}}$ may be given by a relation on internal variables, a relation on values, and a relation on inputs,
- So the implementer proposes $R_{\mathcal{A}}$ and $R_{\mathcal{S}}$ via a JSON file described by the OCaml simMap type below



```
type simMap = {
  stateRels: stateRelAtom list;
  exprRels: exprRelAtom list;
  inputRels: inputRelAtom list;
  intVarRels: intVarRelAtom list;
  valueRels: valueRelAtom list;
...
}[@@deriving yojson]
```

## Proving Spec Simulates Implementation

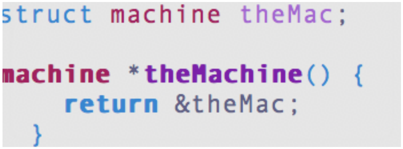
$R_S$ should be pairs of program states in the abstract and concrete program. For our simple ABC example, the relevant execution context is held by a type



```
struct machine {
  enum states currState;
  enum states nextState;
  uint8_t input;
};
typedef struct machine *machine_t;
```

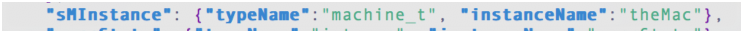
## Proving Spec Simulates Implementation

In the ABC example, the particular instance of the type carrying the execution environment part of the program state is declared in the C as



```
struct machine theMac;

machine *theMachine() {
    return &theMac;
  }
```

so the implementer specifies this in the JSON as



```
    "sMInstance": {"typeName":"machine_t", "instanceName":"theMac"},
```

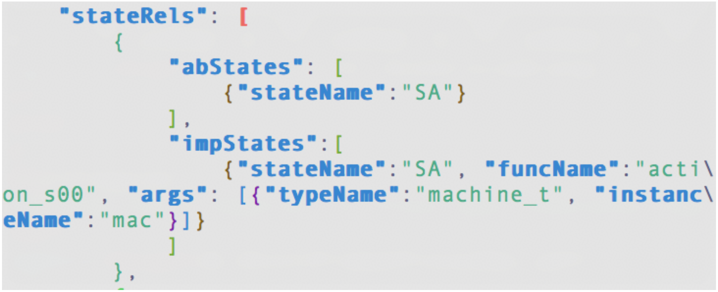
## Proving Spec Simulates Implementation

In the ABC example, the initial and final program states in the abstraction correspond in the C program to the entry and exit program points given the execution environment

```
void action_s00(machine_t mac) {
  mac->currState = 0x00;
  printf("State A (0x%02x)\n", mac->currState);
  if(!read_packet(mac)) {
    error();
  }
  if(mac->input == 'b') {
    mac->nextState = SB;
  }
  else if(mac->input == 'c') {
    mac->nextState = SC;
  }
  else {
    printf("Error on input: %c\n", mac->input);
  }
}
```

## Proving Spec Simulates Implementation

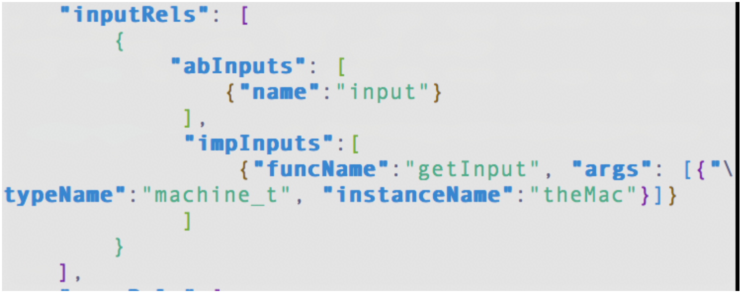
So the implementer gives the relation on states as



```
    "stateRels": [
        {
            "abStates": [
                {"stateName":"SA"}
            ],
            "impStates":[
                {"stateName":"SA", "funcName":"acti\
on_s00", "args": [{"typeName":"machine_t", "instanc\
eName":"mac"}]}
            ]
        },
```

# Proving Spec Simulates Implementation

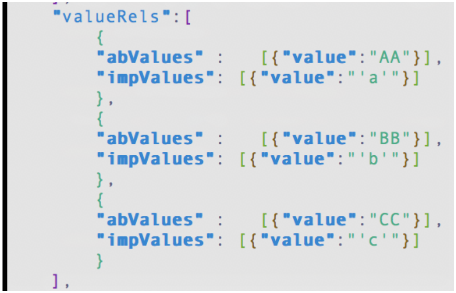
there being, in this example, no internal variables, the map on alphabets factors through a map on input variables



```
    "inputRels": [
        {
            "abInputs": [
                {"name":"input"}
            ],
             "impInputs":[
                 {"funcName":"getInput", "args": [{"\
typeName":"machine_t", "instanceName":"theMac"}]}
             ]
        }
    ],
```

# Proving Spec Simulates Implementation

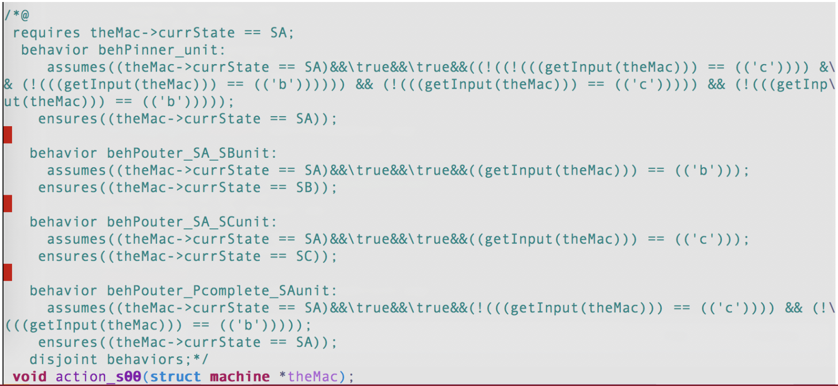
and values



```
    "valueRels":[
        {
        "abValues" :    [{"value":"AA"}],
        "impValues": [{"value":"'a'"}]
        },
        {
        "abValues" :    [{"value":"BB"}],
        "impValues": [{"value":"'b'"}]
        },
        {
        "abValues" :    [{"value":"CC"}],
        "impValues": [{"value":"'c'"}]
        }
    ],
```

# Proving Spec Simulates Implementation

Q then generates the ACSL annotations positing the pre and post conditions from the specification and the simulation relation. A short clang program searches through the C implementation's code base for the program points specified by the stateRel and annotates:



```
/*@
 requires theMac->currState == SA;
  behavior behPinner_unit:
     assumes((theMac->currState == SA)&&\true&&\true&&((!((!(((getInput(theMac)) == (('c')))) &\
& (!(((getInput(theMac)) == (('b')))))) && (!(((getInput(theMac)) == (('c'))))) && (!(((getInp\
ut(theMac)) == (('b')))));
    ensures((theMac->currState == SA));

   behavior behPouter_SA_SBunit:
     assumes((theMac->currState == SA)&&\true&&\true&&((getInput(theMac)) == (('b')));
    ensures((theMac->currState == SB));

   behavior behPouter_SA_SCunit:
     assumes((theMac->currState == SA)&&\true&&\true&&((getInput(theMac)) == (('c')));
    ensures((theMac->currState == SC));

   behavior behPouter_Pcomplete_SAunit:
     assumes((theMac->currState == SA)&&\true&&\true&&(!(((getInput(theMac)) == (('c')))) && (!\
(((getInput(theMac)) == (('b')))));
    ensures((theMac->currState == SA));
   disjoint behaviors;*/
 void action_s00(struct machine *theMac);
```

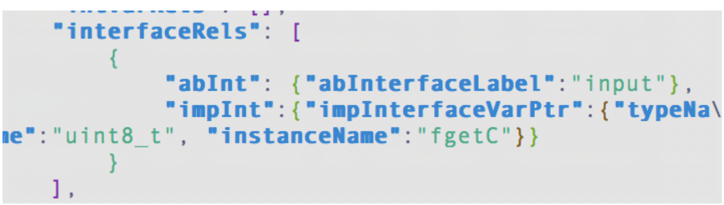
## Proving Spec Simulates Implementation

The C program interacts with its environment through memory mapped IO

```
volatile uint8_t *fgetC = (uint8_t *)INPUT_ADDRESS;
int read_packet(machine_t mac) {
  uint8_t c = *fgetC;
  mac->input = (char)c;
  return ('a' <= c && c <= 'z');
}
```

## Proving Spec Simulates Implementation

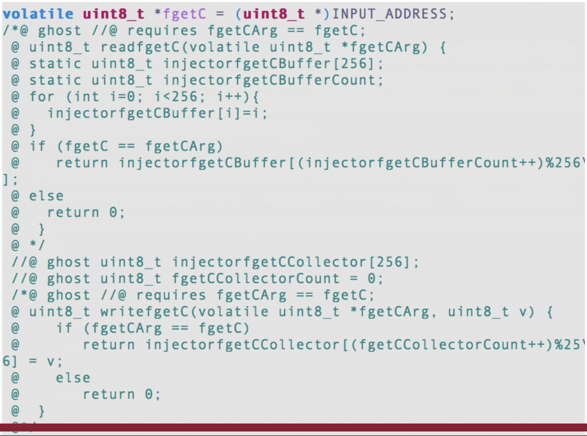
so the implementer must relate the interface to the environment in the abstraction to a memory mapped I/O interface in the implementation of a given type



```
    "interfaceRels": [
        {
            "abInt": {"abInterfaceLabel":"input"},
            "impInt":{"impInterfaceVarPtr":{"typeNa\
me":"uint8_t", "instanceName":"fgetC"}}
        }
    ],
```

## Proving Spec Simulates Implementation

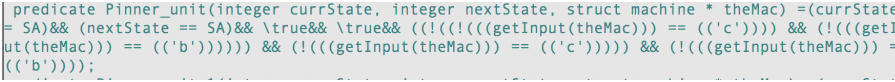
Q generates the ACSL describing the corresponding terminal abstraction $1_{\mathcal{A}_{fgetc}}$, and a short clang program searches through the codebase for the declaration of a volatile variable of that name and that type



```
volatile uint8_t *fgetC = (uint8_t *)INPUT_ADDRESS;
/*@ ghost //@ requires fgetCArg == fgetC;
 @ uint8_t readfgetC(volatile uint8_t *fgetCArg) {
 @ static uint8_t injectorfgetCBuffer[256];
 @ static uint8_t injectorfgetCBufferCount;
 @ for (int i=0; i<256; i++){
 @   injectorfgetCBuffer[i]=i;
 @ }
 @ if (fgetC == fgetCArg)
 @    return injectorfgetCBuffer[(injectorfgetCBufferCount++)%256\
];
 @ else
 @   return 0;
 @  }
 @ */
 //@ ghost uint8_t injectorfgetCCollector[256];
 //@ ghost uint8_t fgetCCollectorCount = 0;
 /*@ ghost //@ requires fgetCArg == fgetC;
 @ uint8_t writefgetC(volatile uint8_t *fgetCArg, uint8_t v) {
 @    if (fgetCArg == fgetC)
 @       return injectorfgetCCollector[(fgetCCollectorCount++)%25\
6] = v;
 @    else
 @       return 0;
 @  }
```

# Proving Spec Simulates Implementation

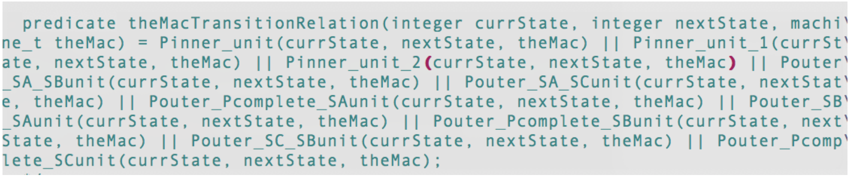
and defines a collection of ACSL predicates



```
 predicate Pinner_unit(integer currState, integer nextState, struct machine * theMac) =(currState
= SA)&& (nextState == SA)&& \true&& \true&& ((!((!(((getInput(theMac))) == (('c')))) && (!(((getI
ut(theMac))) == (('b')))))) && (!(((getInput(theMac))) == (('c'))))) && (!(((getInput(theMac))) =
(('b'))));
```

# Using the Q Compiler to Prove Spec Simulates Implementation

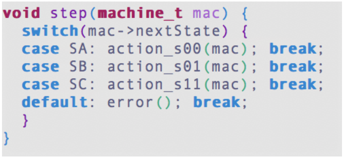
which form clauses in the transition relation



```
  predicate theMacTransitionRelation(integer currState, integer nextState, machi\
ne_t theMac) = Pinner_unit(currState, nextState, theMac) || Pinner_unit_1(currSt\
ate, nextState, theMac) || Pinner_unit_2(currState, nextState, theMac) || Pouter\
_SA_SBunit(currState, nextState, theMac) || Pouter_SA_SCunit(currState, nextStat\
e, theMac) || Pouter_Pcomplete_SAunit(currState, nextState, theMac) || Pouter_SB\
_SAunit(currState, nextState, theMac) || Pouter_Pcomplete_SBunit(currState, next\
State, theMac) || Pouter_SC_SBunit(currState, nextState, theMac) || Pouter_Pcomp\
lete_SCunit(currState, nextState, theMac);
```

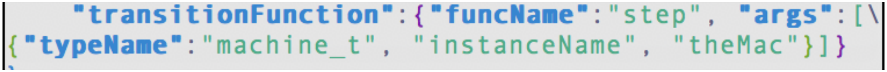
## Using the Q Compiler to Prove Spec Simulates Implementation

in the ABC example, the transition relation in this design is implemented as an actual function on the executin environment



```
void step(machine_t mac) {
  switch(mac->nextState) {
  case SA: action_s00(mac); break;
  case SB: action_s01(mac); break;
  case SC: action_s11(mac); break;
  default: error(); break;
  }
}
```

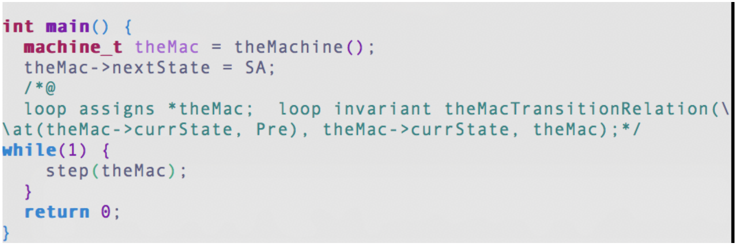
and the implementer has related the abstract transition function to this program point



```
    "transitionFunction":{"funcName":"step", "args":[\
{"typeName":"machine_t", "instanceName", "theMac"}]}
```

# Using the Q Compiler to Prove Spec Simulates Implementation

For any envocation of that function found within a while loop, the clang program annotates with the corresponding loop invariant

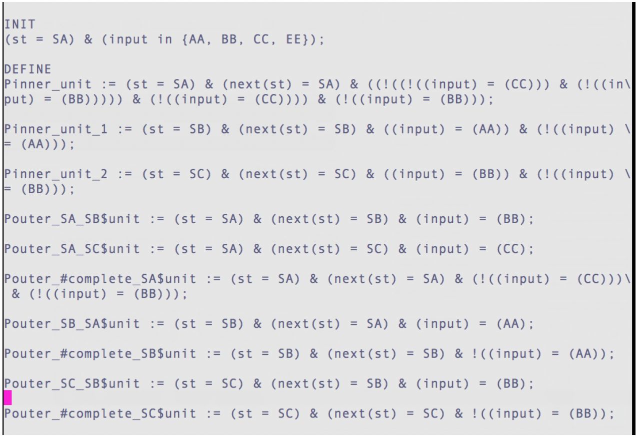


```
int main() {
  machine_t theMac = theMachine();
  theMac->nextState = SA;
  /*@
  loop assigns *theMac;  loop invariant theMacTransitionRelation(\
\at(theMac->currState, Pre), theMac->currState, theMac);*/
while(1) {
    step(theMac);
  }
  return 0;
}
```

We then pose this invariant as a proof obligation to FramaC's WP plugin.

## Proving LTL properties about the Spec

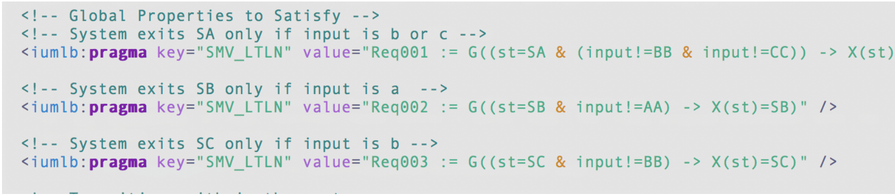
The Q tool produces from the same abstraction a description of the same transition system as a collection of LTL constraints, e.g. in NuSMV



```
INIT
(st = SA) & (input in {AA, BB, CC, EE});

DEFINE
Pinner_unit := (st = SA) & (next(st) = SA) & ((!((!((input) = (CC))) & (!((in\
put) = (BB))))) & (!((input) = (CC)))) & (!((input) = (BB)));

Pinner_unit_1 := (st = SB) & (next(st) = SB) & ((input) = (AA)) & (!((input) \
= (AA)));

Pinner_unit_2 := (st = SC) & (next(st) = SC) & ((input) = (BB)) & (!((input) \
= (BB)));

Pouter_SA_SB$unit := (st = SA) & (next(st) = SB) & (input) = (BB);

Pouter_SA_SC$unit := (st = SA) & (next(st) = SC) & (input) = (CC);

Pouter_#complete_SA$unit := (st = SA) & (next(st) = SA) & (!((input) = (CC)))\
 & (!((input) = (BB)));

Pouter_SB_SA$unit := (st = SB) & (next(st) = SA) & (input) = (AA);

Pouter_#complete_SB$unit := (st = SB) & (next(st) = SB) & !((input) = (AA));

Pouter_SC_SB$unit := (st = SC) & (next(st) = SB) & (input) = (BB);

Pouter_#complete_SC$unit := (st = SC) & (next(st) = SC) & !((input) = (BB));
```

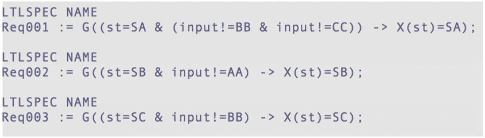
## Proving LTL properties about the Spec

Any proof obligations the designer may have posited



```
<!-- Global Properties to Satisfy -->
<!-- System exits SA only if input is b or c -->
<iumlb:pragma key="SMV_LTLN" value="Req001 := G((st=SA & (input!=BB & input!=CC)) -> X(st)

<!-- System exits SB only if input is a  -->
<iumlb:pragma key="SMV_LTLN" value="Req002 := G((st=SB & input!=AA) -> X(st)=SB)" />

<!-- System exits SC only if input is b -->
<iumlb:pragma key="SMV_LTLN" value="Req003 := G((st=SC & input!=BB) -> X(st)=SC)" />
```

can be checked in NuSMV



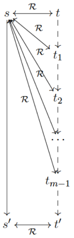
```
LTLSPEC NAME
Req001 := G((st=SA & (input!=BB & input!=CC)) -> X(st)=SA);

LTLSPEC NAME
Req002 := G((st=SB & input!=AA) -> X(st)=SB);

LTLSPEC NAME
Req003 := G((st=SC & input!=BB) -> X(st)=SC);
```

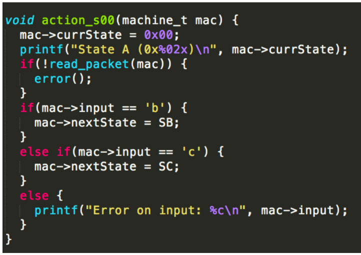
What does that say about the implementation?

## What have we proved?

When FramaC’s WP plugin discharges these proof obligations, the implementer’s simMap witnesses *weak simulation*

```
    behavior behPouter_SA_SBunit:
      assumes((theMac->currState == SA)&&\true&&\true&&\
((theMac->input)) == (('b')));
```



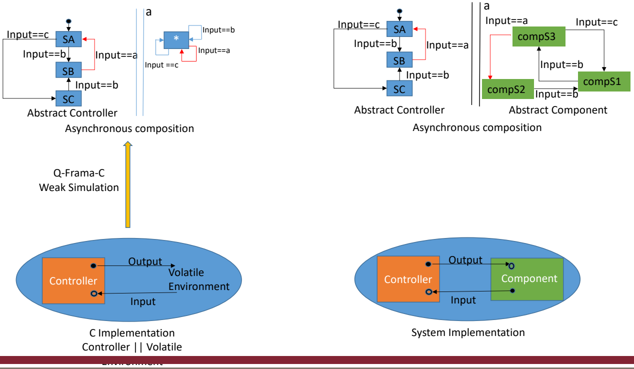


```
void action_s00(machine_t mac) {
  mac->currState = 0x00;
  printf("State A (0x%02x)\n", mac->currState);
  if(!read_packet(mac)) {
    error();
  }
  if(mac->input == 'b') {
    mac->nextState = SB;
  }
  else if(mac->input == 'c') {
    mac->nextState = SC;
  }
  else {
    printf("Error on input: %c\n", mac->input);
  }
}
```

```
    ensures((theMac->currState == SB));
```

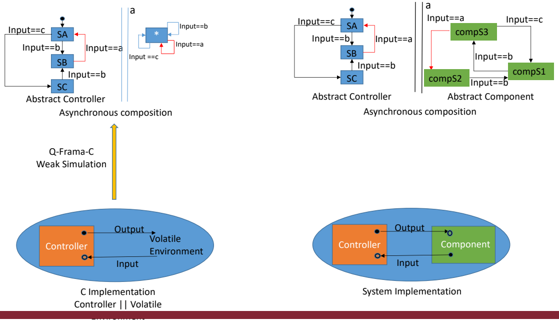
# Proving LTL Properties of Compositions with Implementations

- So Q with FramaC proves weak simulation (in yellow) of the implementation asynchronously composed with a volatile environment by the asynchronous composition of the spec with a volatile environment



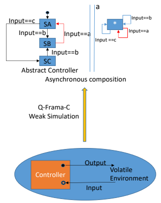
# Proving LTL Properties of Compositions with Implementations

- Q with FramaC proves weak simulation (in yellow) of the implementation composed with a volatile environment by the asynchronous composition of the spec with the environment
- Q can generate models and LTL proof obigations for a composition of abstractions (upper right hand corner)

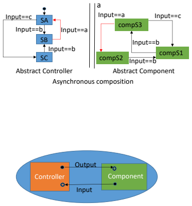


# Proving LTL Properties of Compositions with Implementations

- Q with FramaC proves weak simulation (in yellow) of the implementation composed with a volatile environment by the asynchronous composition of the spec with the environment
- Q can generate models and LTL proof obligations for a composition of abstractions (upper right hand corner)
- *What does that say about the implementation (lower right corner)?*



C Implementation
Controller || Volatile



System Implementation

## The $\|$ between Boxes: Parallel Composition of LTS

To reduce state space, and to model the composition of hardware within clock domains, we actually have in Q a synchronous composition

### Definition

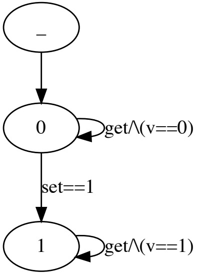
The Synchronous composition of $P, Q$ : LTS, written $P \| Q$ is given by the rule

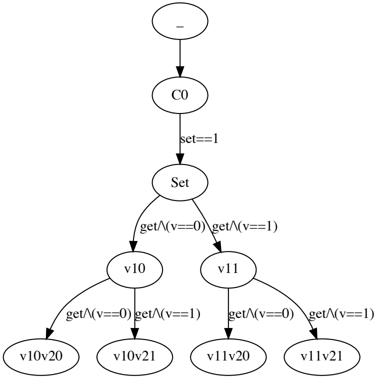
$$\frac{p \overset{\alpha}{\rightarrow}_P p' \quad q \overset{\alpha}{\rightarrow}_Q q'}{(p, q) \overset{\alpha}{\rightarrow}_{P\|Q} (p', q')}$$

# An example

## Example

$\forall q \in S_Q$, the synchronous composition of these two LTS's will never visit $(v10v21, q)$ or $(v11v20, q)$





## Moving Arrows past $\|$: Precongruence

Our synchronous $\|$ was a convenient choice for reducing statespace and for modeling hardware. More importantly, now, is that it is *compositional*, in that

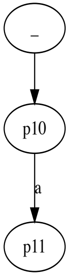
### Lemma

*Strong simulation $\preceq$ is a pre-congruence over $\|$, that is,*
$\forall O, P, Q$ : LTS.$P \preceq Q \Rightarrow P \| O \preceq Q \| O$

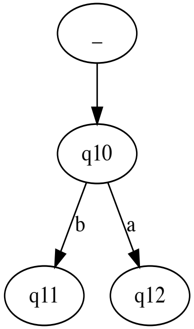
## An Example

### Example

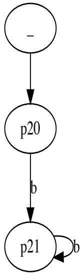
Here, $P_1 \preceq Q_1$, and $P_1 \| P_2 \preceq Q_1 \| P_2$



$P_1$



$Q_1$



$P_2$

## Synchronous, Asynchronous, Strong, Weak

We can recover asynchronous composition *and* weak simulation through a completion with the right alphabet

### Definition

Let $\mathcal{A}$ an alphabet, and let $M_{\mathcal{A}}$ : LTS $\to$ LTS :: $P \mapsto P$ *the completion over* $\mathcal{A}$ be the transition system $\left(S_P, \mathcal{A}_P \bigsqcup \mathcal{A}, \dot{\to}, S_0\right)$ with the new transition relation given by rules

$$\frac{p \overset{\alpha}{\to}_P p'}{p \overset{\alpha}{\to}_{\mathsf{M}_{\mathcal{A}(P)}} p'} \qquad\qquad \frac{\alpha \in \mathcal{A}}{p \overset{\alpha}{\to}_{\mathsf{M}_{\mathcal{A}(P)}} p} \tag{2.1}$$

## Synchronous, Asynchronous, Strong, Weak

Let $P, Q$ : LTS with alphabets $\mathcal{A}_P$ and $\mathcal{A}_Q$. We can recover the asynchronous composition from the synchronous composition via the completion once given a decomposition of their meet $\mathcal{A}_P \sqcap \mathcal{A}_Q$ into disjoint $\mathcal{A}_{O_P}$ (labels output by $P$) and $\mathcal{A}_{O_Q}$ (labels output by $Q$). Let $\mathcal{A}_{Q/P} := \mathcal{A}_{O_Q} \sqcup (\mathcal{A}_Q \setminus \mathcal{A}_P)$ *the asynchronous alphabet of $Q$ over $P$*

## Synchronous, Asynchronous, Strong, Weak

Let $P, Q$ : LTS with alphabets $\mathcal{A}_P$ and $\mathcal{A}_Q$. We can recover the asynchronous composition from the synchronous composition via the completion once given a decomposition of their meet $\mathcal{A}_P \sqcap \mathcal{A}_Q$ into disjoint $\mathcal{A}_{O_P}$ (labels output by $P$) and $\mathcal{A}_{O_Q}$ (labels output by $Q$). Let $\mathcal{A}_{Q/P} := \mathcal{A}_{O_Q} \sqcup (\mathcal{A}_Q \setminus \mathcal{A}_P)$ *the asynchronous alphabet of* $Q$ *over* $P$

### Proposition

*Let* $Q, P$ : LTS *share alphabet* $\mathcal{A}_P = \mathcal{A}_Q$ *and let* $\mathcal{A}_E$ *the alphabet of their environment.* $Q$ *weakly simulates* $P$ *(i.e.* $P \preceq_W Q$*) if and only if* $P \preceq \mathsf{M}_{\mathcal{A}_{E/Q}}(Q)$, *or* $P \to \mathsf{M}_{\mathcal{A}_{E/Q}}(Q)$

## Synchronous, Asynchronous, Strong, Weak

### Lemma

*The asynchronous composition is in strong simulation's kernel with the synchronous composition of completions*

$\forall P, Q : LTS.P \|^a Q \simeq \mathsf{M}_{\mathcal{A}_{Q/P}}(P) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$

## Synchronous, Asynchronous, Strong, Weak

Lemma

*The asynchronous composition is in strong simulation's kernel with the synchronous composition of completions*

$\forall P, Q : LTS. P \|^a Q \simeq \mathsf{M}_{\mathcal{A}_{Q/P}}(P) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$

Lemma

$P \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \simeq P$

## Synchronous, Asynchronous, Strong, Weak

Lemma

*The asynchronous composition is in strong simulation's kernel with the synchronous composition of completions*

$\forall P, Q : LTS. P \|^a Q \simeq \mathsf{M}_{\mathcal{A}_{Q/P}}(P) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$

Lemma

$P \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \simeq P$

Lemma

*Q asks* FramaC-WP *to prove*

$\mathsf{M}_{\mathcal{A}_{Q/P}}(P \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q})) \overset{\mathrm{q}_{\mathrm{FramaC}}}{\leftarrow} \mathsf{M}_{\mathcal{A}_{Q/P}}(C) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q})$

## Soundness and Compositionality of Q

- By terminality of the terminal abstraction, $1_{\mathcal{A}_Q} \overset{!}{\leftarrow} Q$.

## Soundness and Compositionality of Q

- By terminality of the terminal abstraction, $1_{\mathcal{A}_Q} \overset{!}{\leftarrow} Q$.
- By functoriality of the completion,

$$\mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \overset{\mathsf{M}_{\mathcal{A}_{P/Q}}}{\leftarrow} \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$$

## Soundness and Compositionality of Q

- By terminality of the terminal abstraction, $1_{\mathcal{A}_Q} \overset{!}{\leftarrow} Q$.
- By functoriality of the completion,

  $\mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \overset{\mathsf{M}_{\mathcal{A}_{P/Q}}}{\leftarrow} \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$
- Therefore, by precongruence of simulation ($\succeq$ or $\leftarrow$) over $\|$,

## Soundness and Compositionality of Q

- By terminality of the terminal abstraction, $1_{\mathcal{A}_Q} \overset{!}{\leftarrow} Q$.
- By functoriality of the completion,

$$\mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \overset{\mathsf{M}_{\mathcal{A}_{P/Q}}}{\leftarrow} \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$$

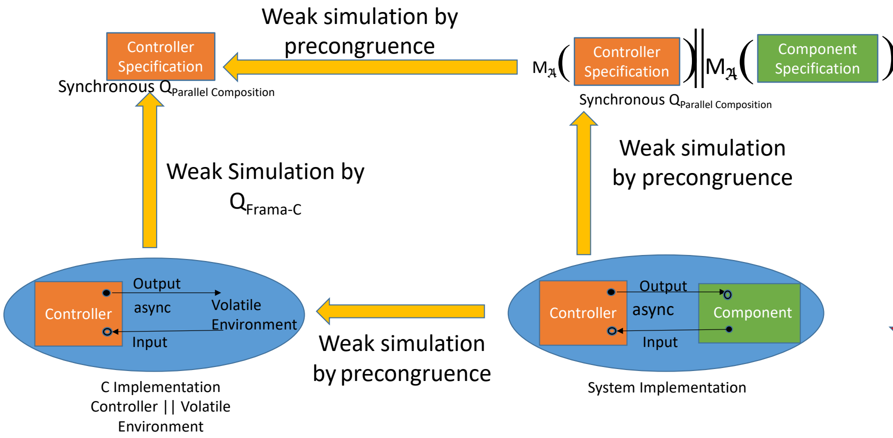
- Therefore, by precongruence of simulation ($\succeq$ or $\leftarrow$) over $\|$,

### Theorem (Main Theorem)

*The following is a commuting diagram of* LTS *with, as maps of* LTS, *simulations*

$$\begin{tikzcd}
\mathsf{M}_{\mathcal{A}_{Q/P}}(P \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q})) & \mathsf{M}_{\mathcal{A}_{Q/P}}(P) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q) \arrow[l, "\mathsf{M}_{\mathcal{A}_{Q/P}}(!)"'] \\
\mathsf{M}_{\mathcal{A}_{Q/P}}(C) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(1_{\mathcal{A}_Q}) \arrow[u, "\mathsf{q}_{\mathsf{FramaC}}"'] & \mathsf{M}_{\mathcal{A}_{Q/P}}(C) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q) \arrow[l, "!"] \arrow[u]
\end{tikzcd}$$

# What Was Accomplished

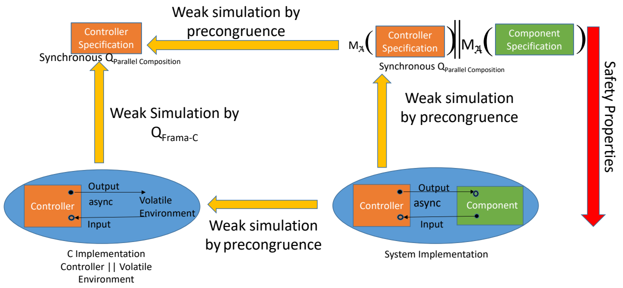


Weak simulation by precongruence
Controller Specification
Synchronous Q_Parallel Composition
M_𝔄 ( Controller Specification ) || M_𝔄 ( Component Specification )
Synchronous Q_Parallel Composition
Weak Simulation by Q_Frama-C
Weak simulation by precongruence
Controller
Output
async
Input
Volatile Environment
Weak simulation by precongruence
Controller
Output
async
Input
Component
C Implementation Controller || Volatile Environment
System Implementation

## What Was Accomplished

Thus we have accomplished our goal -

- Given a proposed witness of simulation of a controller implementation by its spec, the Q compiler generates and emplaces the ACSL annotations from which FramaC generates the proof of simulation.

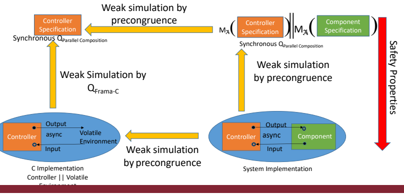


Controller Specification
Synchronous Q Parallel Composition
Weak simulation by precongruence
M(
Controller Specification
)||M(
Component Specification
)
Synchronous Q Parallel Composition
Safety Properties
Weak Simulation by Q Frama-C
Weak simulation by precongruence
Controller
Output
async
Volatile Environment
Input
C Implementation Controller || Volatile Environment
Weak simulation by precongruence
Controller
Output
async
Component
Input
System Implementation

## What Was Accomplished

Thus we have accomplished our goal -

- Given a proposed witness of simulation of a controller implementation by its spec, the Q compiler generates and emplaces the ACSL annotations from which FramaC generates the proof of simulation.
- The Q compiler computes the *asynchronous* composition of that specification with specifications for other components and constructs LTL models of the compositions along with high level properties in e.g. Why3 if they are to be proved or e.g. NuSMV if they are to be checked.

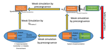


Controller Specification
Synchronous $Q_{\text{Parallel Composition}}$
Weak simulation by precongruence
$M_{\mathfrak{A}}($ Controller Specification $) \| M_{\mathfrak{A}}($ Component Specification $)$
Synchronous $Q_{\text{Parallel Composition}}$
Safety Properties
Weak Simulation by $Q_{\text{Frama-C}}$
Weak simulation by precongruence
Controller
Output
async
Input
Volatile Environment
Weak simulation by precongruence
Controller
Output
async
Input
Component
C Implementation Controller || Volatile
System Implementation

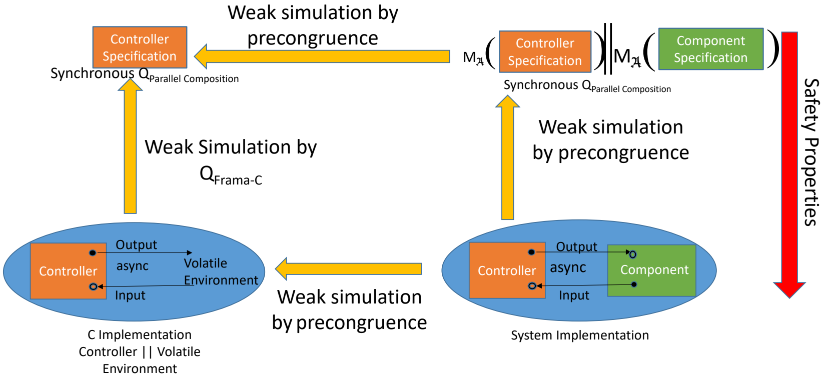
## What Was Accomplished

Thus we have accomplished our goal -

- Given a proposed witness of simulation of a controller implementation by its spec, the Q compiler generates and emplaces the ACSL annotations from which FramaC generates the proof of simulation.
- The Q compiler computes the composition of that specification with (the completion of the) specifications for other components, constructs LTL models of the compositions along with high level properties in e.g. Why3 if they are to be proved or e.g. NuSMV if they are to be checked.
- By the Main Theorem, any Safety Properties proved about that composition of specifications implies the same property about the actual System implementation



# What Was Accomplished


Weak simulation by precongruence
Controller Specification
Synchronous $Q_{\text{Parallel Composition}}$
$M_{\mathfrak{A}}$( Controller Specification )||$M_{\mathfrak{A}}$( Component Specification )
Synchronous $Q_{\text{Parallel Composition}}$
Safety Properties
Weak Simulation by $Q_{\text{Frama-C}}$
Weak simulation by precongruence
Controller
Output
async
Volatile Environment
Input
Weak simulation by precongruence
Controller
Output
async
Component
Input
C Implementation Controller || Volatile Environment
System Implementation

## What is Next for Q?

- Q is currently proprietary. However, there is internal discussion about potentially open sourcing it
- We would like to mechanize the metatheory: ideally Q would provide certificates in, e.g. Coq, that it is doing the right thing. For that it would be nice to have a formal semantics of ACSL.

## Thanks!

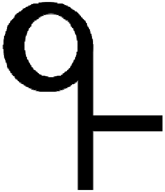
Robert Armstrong

Jon M. Aytac

Geoffrey Compton Hulette

Benjamin Curt Nilsen

Philip Alden Johnson-Freyd



Jackson Mayo

Jason Michnovicz

Karla Morris

Ratish J. Punnoose

Andrew Michael Smith

## Bonus Slide: What about Aeorai

- There is already a plugin for producing LTL properties about C programs: Aeorai

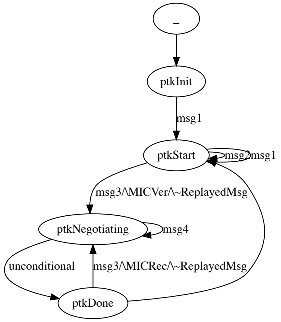
## Bonus Slide: What about Aeorai

- There is already a plugin for producing LTL properties about C programs: Aeorai
- We could use this for our purposes only after solving the following problem

## Bonus Slide: What about Aeorai

- There is already a plugin for producing LTL properties about C programs: Aeorai
- We could use this for our purposes only after solving the following problem
- Let $\mathbb{P}_{P\|^a Q}$ : Prop$(P\|^a Q)$ a temporal property on the asynchronous composition of $P$ and $Q$. Given the specification of $Q$, what temporal property $\mathbb{P}_P$ : Prop$(P)$ suffices to obtain $\mathbb{P}_{P\|^a Q}$. This is sometimes called the problem of producing quotient specifications

# Bonus Slide: What about WPA?



## Bonus Slide: Lattice Structure on Labels

The labels in a C program will have some lattice structure, i.e. a Boolean algebra of expressions over its variables, and this puts some constraints on $\dot{\rightarrow} \colon \mathcal{A} \rightarrow \mathcal{P}(S \times S)$,

$$\frac{\beta \Rightarrow \alpha \; p \overset{\alpha}{\rightarrow}_C p'}{p \overset{\beta}{\rightarrow}_C p'} \qquad \frac{p \overset{\alpha}{\rightarrow} p' \; p \overset{\beta}{\rightarrow}_C p'}{p \overset{\alpha \vee \beta}{\rightarrow}_C p'} \qquad \frac{}{p \overset{\perp}{\rightarrow}_C p'} \tag{2.2}$$

## Bonus Slide: Lattice Structure on Labels

The labels in a C program will have some lattice structure, i.e. a Boolean algebra of expressions over its variables, and this puts some constraints on $\dot{\rightarrow}: \mathcal{A} \rightarrow \mathcal{P}(S \times S)$,

$$\frac{\beta \Rightarrow \alpha \quad p \overset{\alpha}{\rightarrow}_C p'}{p \overset{\beta}{\rightarrow}_C p'} \qquad \frac{p \overset{\alpha}{\rightarrow} p' \quad p \overset{\beta}{\rightarrow}_C p'}{p \overset{\alpha \vee \beta}{\rightarrow}_C p'} \qquad \frac{}{p \overset{\perp}{\rightarrow}_C p'} \tag{2.2}$$

So $\dot{\rightarrow}$ is a Galois Connection. $f_{[R_{\mathcal{A}}]}$ must therefore be monotonic for our witness to be viable.

## Bonus Slide: Lattice Structure on Labels

- The labels in the abstraction are the expressions over abstract variables $\text{Var}_Q$ taking values in abstract value domain $\mathbb{V}_Q$.

## Bonus Slide: Lattice Structure on Labels

- The labels in the abstraction are the expressions over abstract variables $\mathsf{Var}_Q$ taking values in abstract value domain $\mathbb{V}_Q$.
- if $f_{[R_\mathcal{A}]}$ factors through a map of variables $f_{\mathsf{Var}} : \mathsf{Var}_Q \to \mathcal{P}(\mathsf{Var}_P)$ and a map of their value domains $f_{\mathbb{V}} : \mathbb{V}_Q \to \mathcal{P}(\mathbb{V}_P)$, then we can know $f_{[R_\mathcal{A}]}$ is monotonic, no further proof obligations required.

## Bonus Slide: Lattice Structure on Labels

- The labels in the abstraction are the expressions over abstract variables $\mathsf{Var}_Q$ taking values in abstract value domain $\mathbb{V}_Q$.
- if $f_{[R_{\mathcal{A}}]}$ factors through a map of variables $f_{\mathsf{Var}} : \mathsf{Var}_Q \to \mathcal{P}(\mathsf{Var}_P)$ and a map of their value domains $f_{\mathbb{V}} : \mathbb{V}_Q \to \mathcal{P}(\mathbb{V}_P)$, then we can know $f_{[R_{\mathcal{A}}]}$ is monotonic, no further proof obligations required.
- Otherwise, checking monotonicity of $f_{[R_{\mathcal{A}}]}$ requires reasoning about the lattice structure of the algebra of boolean expressions over the execution environment of the C program

So we present the data type provide required to construct the simulation witness $R$ as well as the information to generate $1_{\mathcal{A}_I}$ in a format encouraging the designer to give the witness without incurring additional proof obligations

```
type simMap = {
  stateRels: stateRelAtom list;
  exprRels: exprRelAtom list;
  inputRels: inputRelAtom list;
  intVarRels: intVarRelAtom list;
  valueRels: valueRelAtom list;
...
}[@@deriving yojson]
```

## Completion is not Strong

The completion is not, however, a strong endofunctor on LTS

$$\mathsf{M}_{\mathcal{A}_{Q/P}}(P \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)) \to \mathsf{M}_{\mathcal{A}_{Q/P}}(P) \| \mathsf{M}_{\mathcal{A}_{P/Q}}(Q)$$