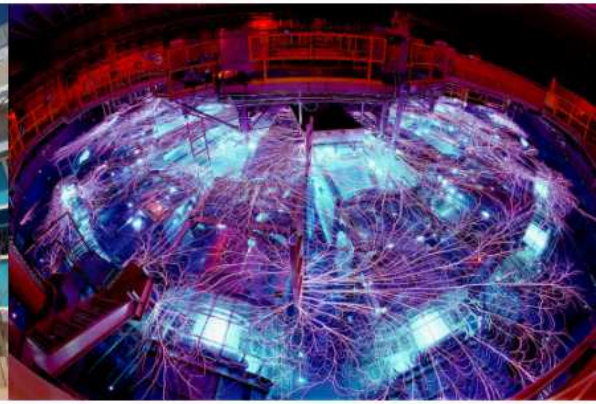
Exceptional service in the national interest

National Laboratories

SAND2019-0788C





The Q Compiler

for Verifying High-Consequence Controls

Jon Aytac

6/18/18

U.S. DEPARTMENT OF ENERGY

oratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia,
neywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract

Motivation

Sandia National Laboratories

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

Motivation

Sandia National Laboratories

- 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

Sandia National Laboratories

- 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

Sandia National Laboratories

- 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

The Modular Correctness Proof Sounds Nice and All

Sandia National Laboratories

> 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

But Some Crucial Properties...

Sandia National Laboratories

> 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

… Were not in the Model…

Sandia National Laboratories

> 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

… Which Captured How but not When!

Sandia National Laboratories

> 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

Temporal Logic Proof of Fix's Correctness

Sandia National Laboratories

> 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

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

Jedi Mind Trick

- So then the decision makers say

Jedi Mind Trick

Sandia National Laboratories

- 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

Sandia National Laboratories

- 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

Sandia National Laboratories

- 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

Sandia National Laboratories

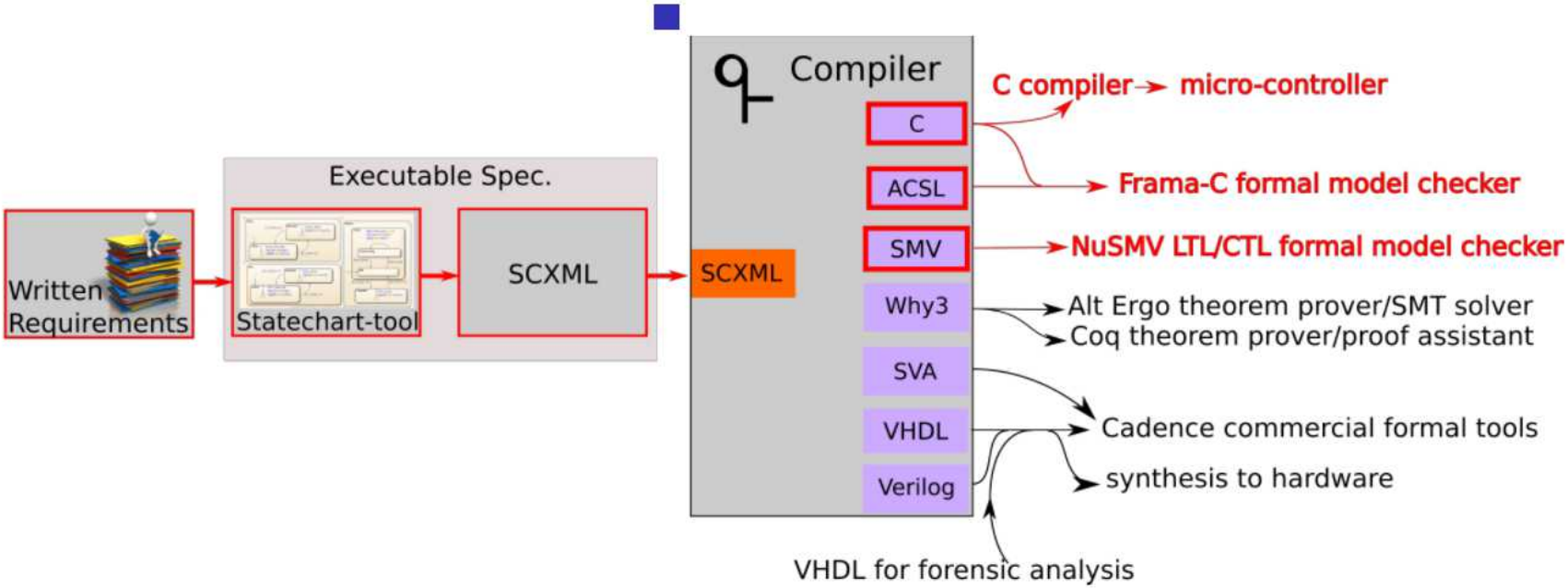
- So then the decision makers say
- We need a language for the specification of temporal behavior, preferably one designers would actually use
- We need to compile these descriptions into languages suitable for proving (e.g. Why3) and checking (e.g. NuSMV) temporal properties about compositions of specifications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

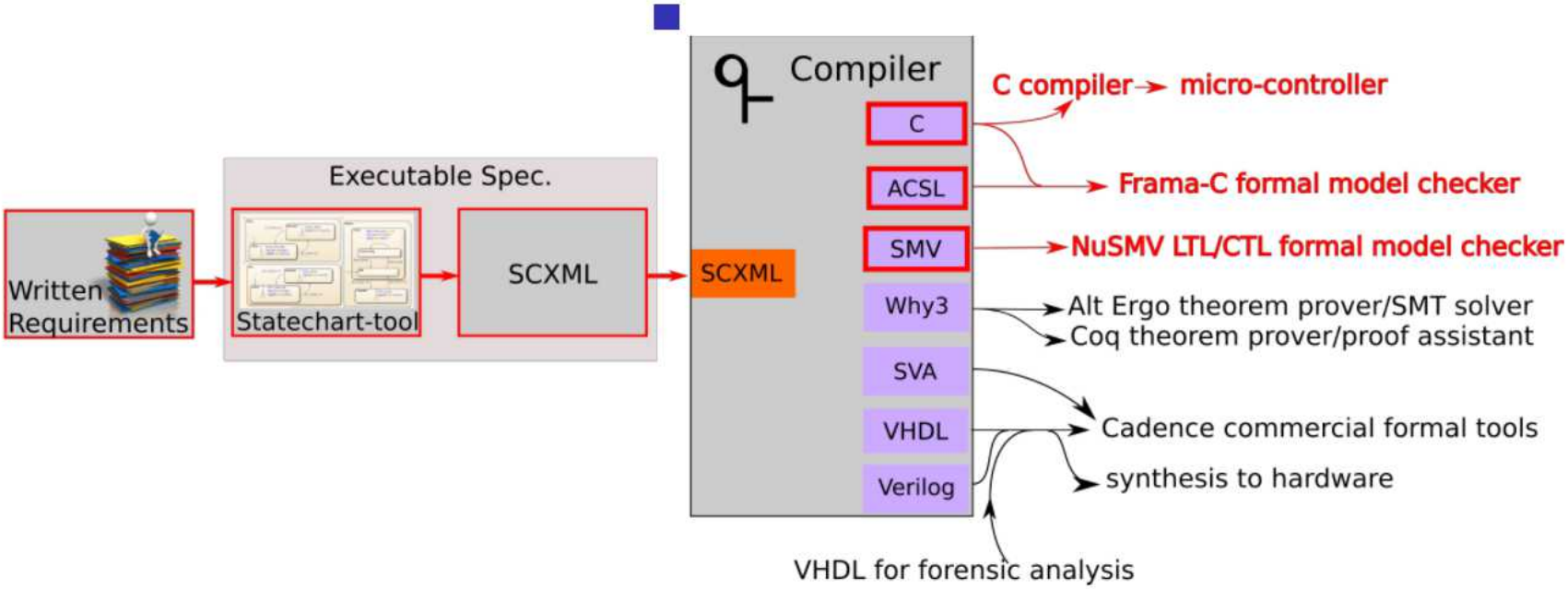
What a Coincidence

- At this point, you say, we just so happen to have already written a tool, the Q compiler, to do just that!



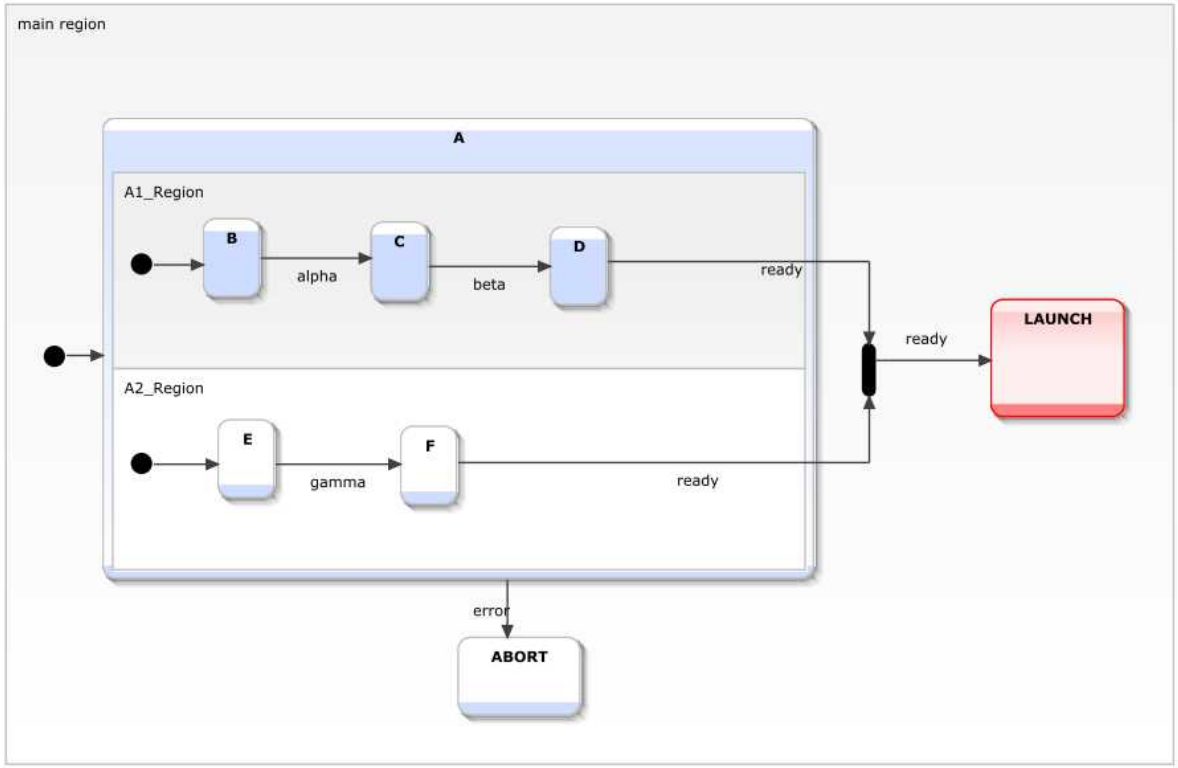
What a Coincidence

- At this point, you say, we just so happen to have already written a tool, the Q compiler, to do just that!

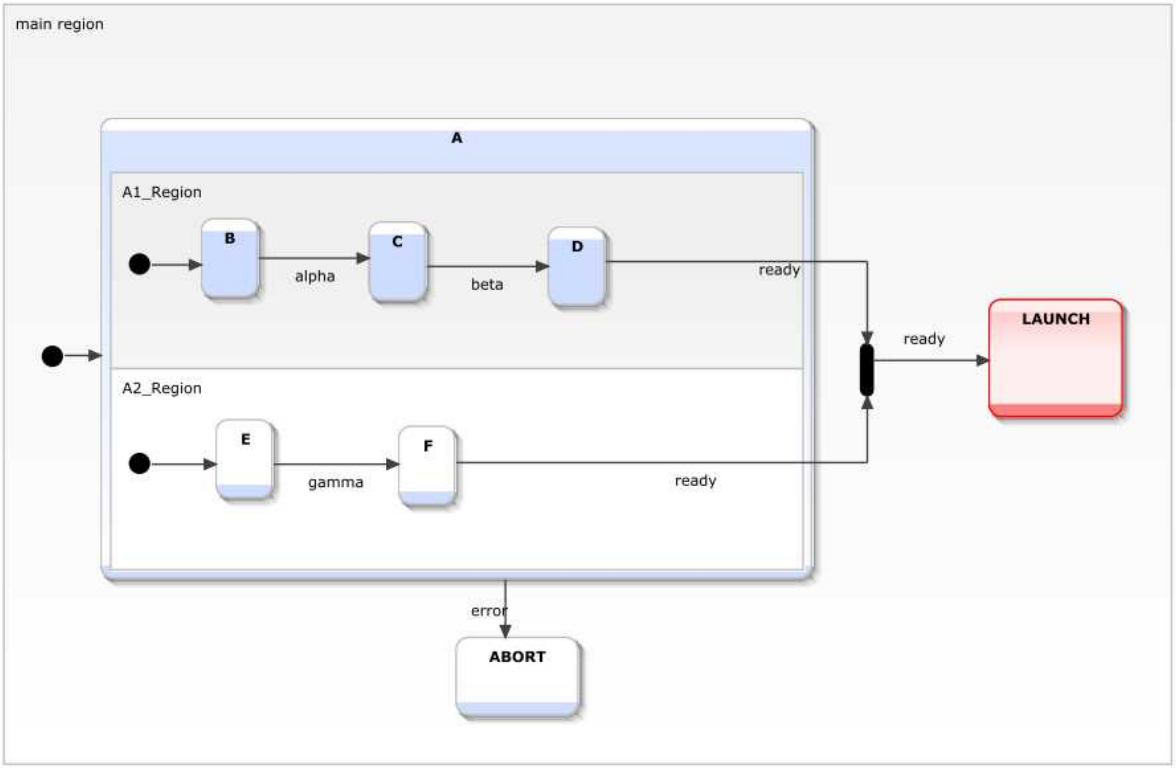


- Designers draw statechart-like diagrams in their specifications, so capture these in a statechart-like formal language

By Statechart-like, we mean a specification language for reactive programs,inductively defined through hierarchic and parallel composition of finite state machines.



By Statechart-like, we mean a specification language for reactive programs,inductively defined through hierarchic and parallel composition of finite state machines.



We say Statechart-like because we want to be free to choose whichever definition allows compositional reasoning.

FramaC and Microcontrollers

- Suppose a controller is written in C, and this C module works in concert with other components

FramaC and Microcontrollers

Sandia National Laboratories

- Suppose a controller is written in C, and this C module works in concert with other components
- Our controller interacts with other components through memory-mapped IO

FramaC and Microcontrollers

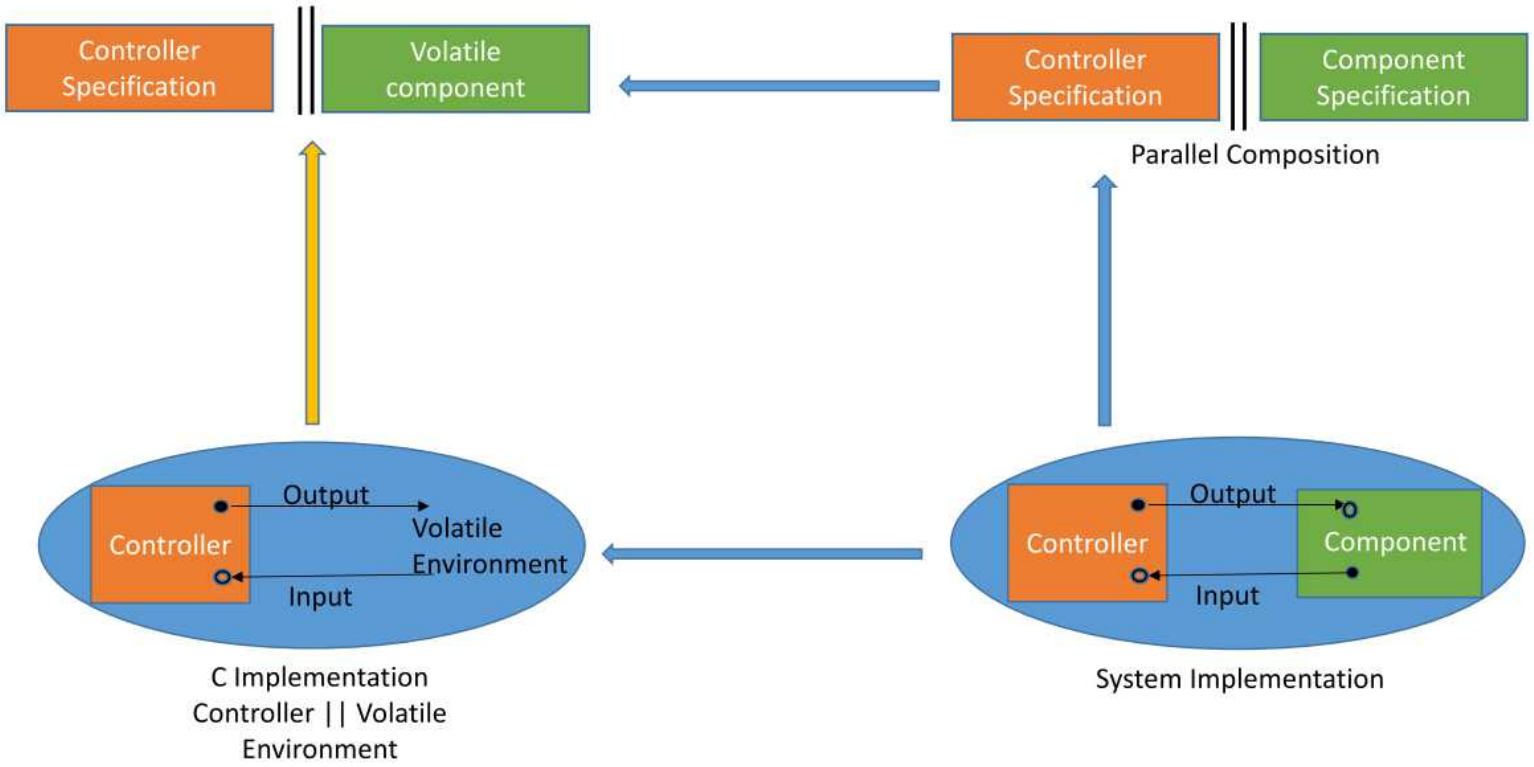
Sandia National Laboratories

- Suppose a controller is written in C, and this C module works in concert with other components
- Our controller interacts with other components through memory-mapped IO
- Only Frama-C meets our requirements for reasoning about C programs, but Frama-C analyzes *sequential* programs.

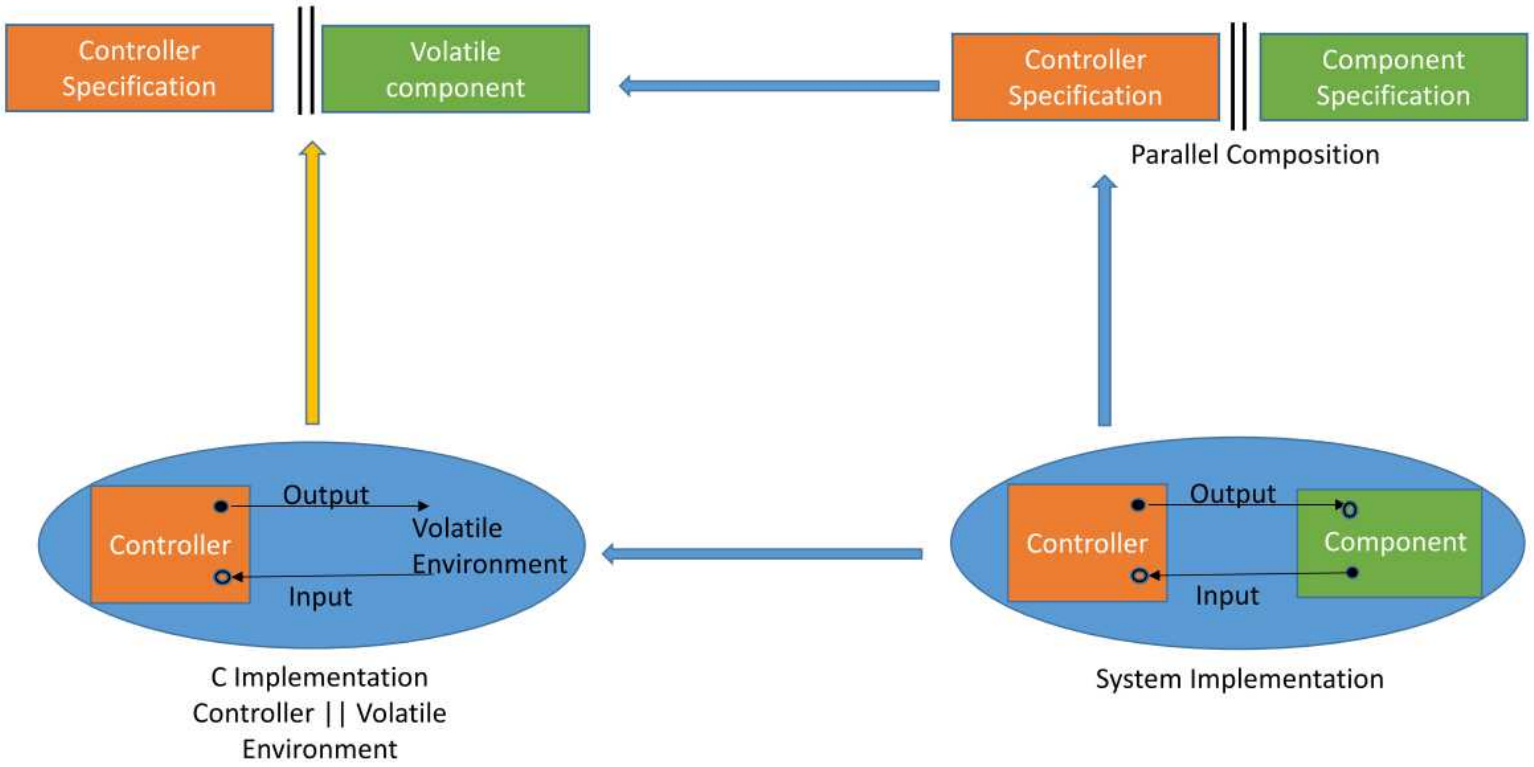
Composition in C through Memory-mapped IO

A volatile declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function.

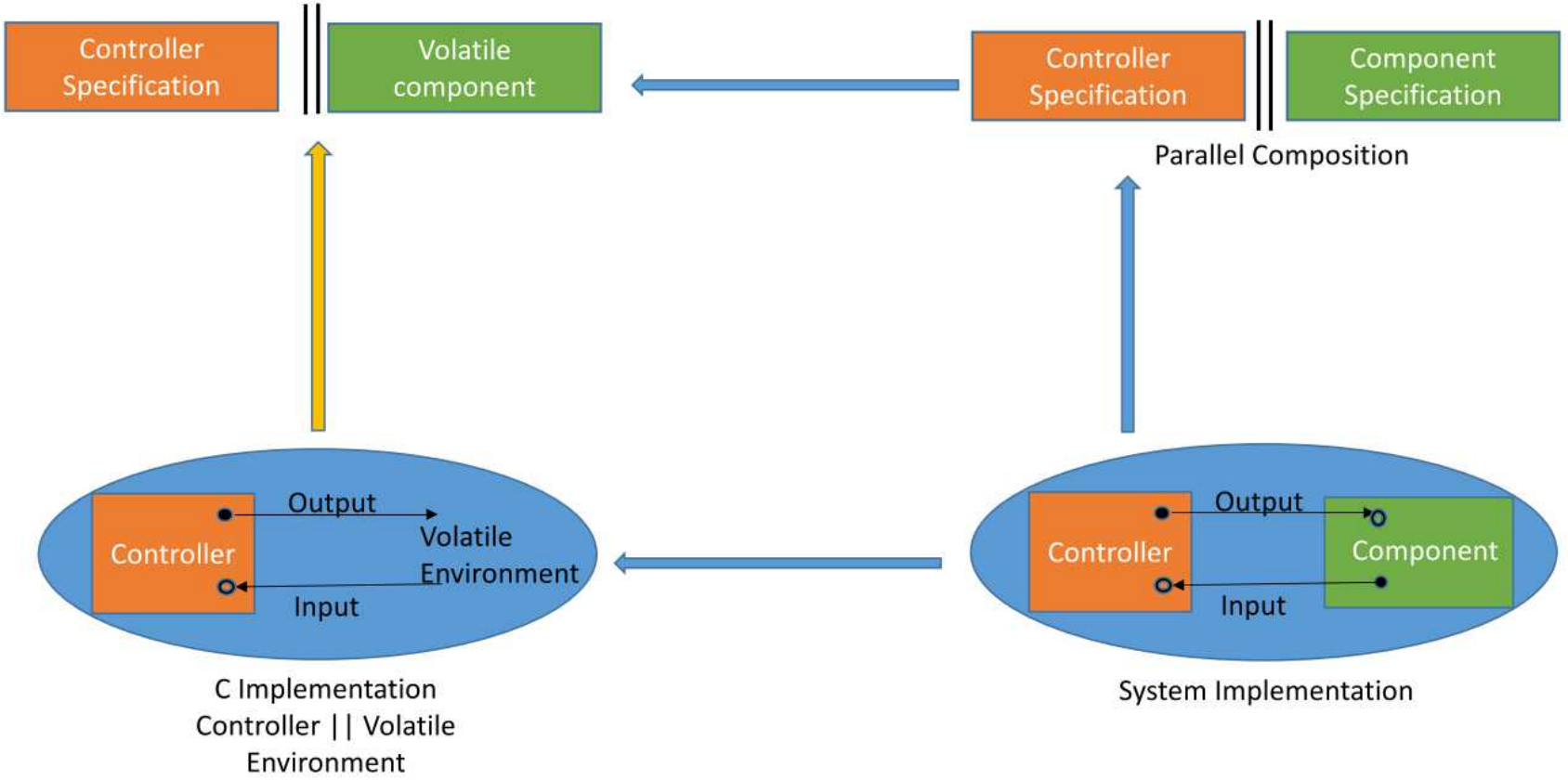
- So we want a way to prove all the arrows to point from a program to its, in a sense soon made more precise, abstraction. Then every safety property of the abstract Parallel Composition is a safety property of the System Implementation



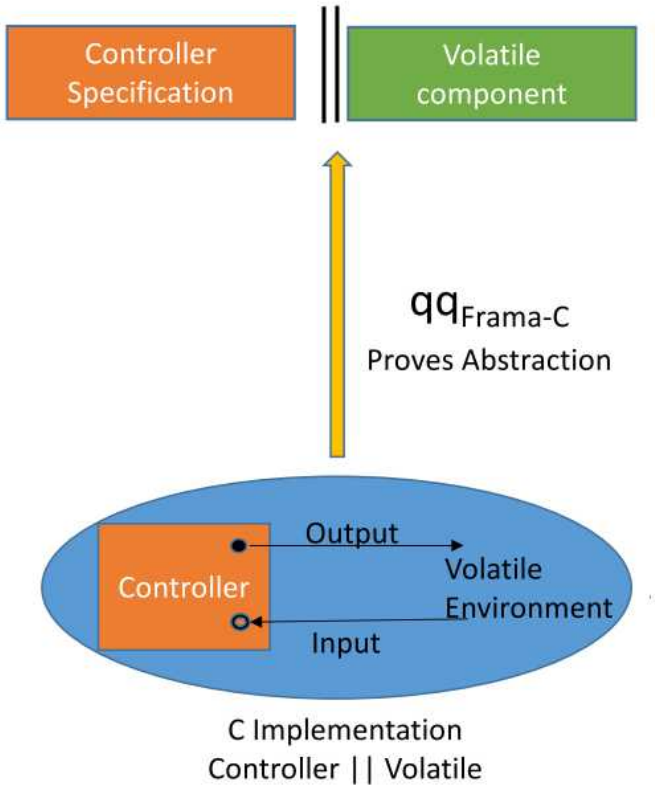
- So we want a way to prove all the arrows to point from a program to its, in a sense soon made more precise, abstraction. Then every safety property of the abstract Parallel Composition is a safety property of the System Implementation
- We want a way to prove the yellow arrow, and, preferrably, a way to get the blue arrows for free.



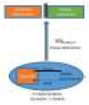
The punchline will be, yes, the arrows will come for free, but, as will become clear, we will have to be careful!



Proving Abstraction of a C Program

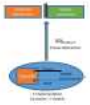


The Ovals: C Programs as Transition Systems



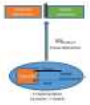
- Recall that a program point and an execution environment together constitute state of an imperative program.

The Ovals: C Programs as Transition Systems



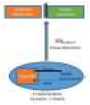
- Recall that a program point and an execution environment together constitute state of an imperative program.
- The program is a map from program points to Commands, and evaluating the command found at a program point in an execution environment produces a transition of execution environments.

The Ovals: C Programs as Transition Systems



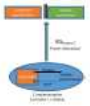
- Recall that a program point and an execution environment together constitute state of an imperative program.
- The program is a map from program points to Commands, and evaluating the command found at a program point in an execution environment produces a transition of execution environments.
- So we can think about our programs as *labelled transition systems*.

The Ovals: C Programs as Transition Systems



- Recall that a program point and an execution environment together constitute state of an imperative program.
- The program is a map from program points to Commands, and evaluating the command found at a program point in an execution environment produces a transition of execution environments.
- So we can think about our programs as *labelled transition systems*.
- A labelled transition system P 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 (with properties to be discussed later), and P_0 a set of initial states.

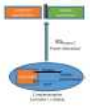
The Arrows: Simulation Relations



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 $\forall (p,q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$ $p \xrightarrow{\alpha}_P p' \Rightarrow \exists q' \in S_Q : \left(q \xrightarrow{\alpha}_Q q' \wedge (p', q') \in R\right)$

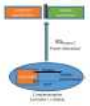
The Arrows: Simulation Relations



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 $\forall (p,q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$ $p \xrightarrow{\alpha}_P p' \Rightarrow \exists q' \in S_Q : \left(q \xrightarrow{\alpha}_Q q' \wedge (p', q') \in R\right)$
- A state q **simulates** a state p, written $p \preceq q$ if $\exists$ a simulation relation R with $(p,q) \in R$. Q **simulates** P, written $P \preceq Q$ or $P \rightarrow Q$, if $\forall p_0 \in P_0 \exists q_0 \in Q_0 . p_0 \preceq q_0$, so that $P_0 \subset R^{-1}(Q_0)$.

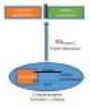
The Arrows: Simulation Relations



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 $\forall (p,q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$ $p \xrightarrow{\alpha}_P p' \Rightarrow \exists q' \in S_Q : \left(q \xrightarrow{\alpha}_Q q' \wedge (p', q') \in R\right)$
- A state q **simulates** a state p, written $p \preceq q$ if $\exists$ a simulation relation R with $(p,q) \in R$. Q **simulates** P, written $P \preceq Q$ or $P \rightarrow Q$, if $\forall p_0 \in P_0 \exists q_0 \in Q_0 . p_0 \preceq q_0$, so that $P_0 \subset R^{-1}(Q_0)$.

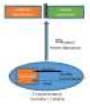
The Arrows: Simulation Relations



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 $\forall (p,q) \in R, \alpha \in \mathcal{A}_P, p' \in S_P$ $p \xrightarrow{\alpha}_P p' \Rightarrow \exists q' \in S_Q : \left(q \xrightarrow{\alpha}_Q q' \wedge (p', q') \in R\right)$
- A state q **simulates** a state p, written $p \preceq q$ if $\exists$ a simulation relation R with $(p,q) \in R$. Q **simulates** P, written $P \preceq Q$ or $P \to Q$, if $\forall p_0 \in P_0 \exists q_0 \in Q_0 . p_0 \preceq q_0$, so that $P_0 \subset R^{-1}(Q_0)$.
- 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**.

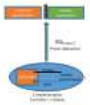
The Arrows: Simulation Relations



Proposition

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

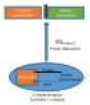
Volatile Variables and the Terminal Abstraction



Definition

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

Volatile Variables and the Terminal Abstraction



Definition

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

Proposition

This abstracts everything $\forall P = \left(S_P, \mathcal{A}_P, \dot{\rightarrow}_P, P_0\right), P \preceq 1_{\mathcal{A}_P}$

Volatile Variables and the Terminal Abstraction

Definition

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

Proposition

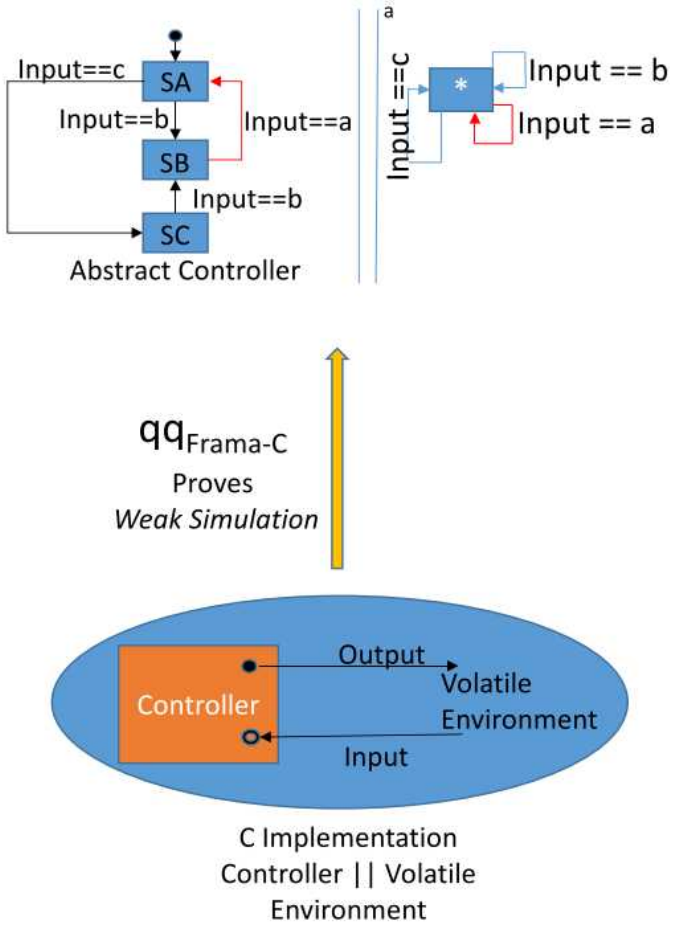
This abstracts everything $\forall P = \left(S_P, \mathcal{A}_P, \dot{\rightarrow}_P, P_0\right), P \preceq 1_{\mathcal{A}_P}$

Example

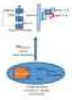
Let "volatile uint8_t I;" a declaration of a reference to a volatile variable of type uint8_t. Then, $\mathcal{A}_I = \{(I == 0), \cdots, (I == 256)\}$, the presence of a volatile variable in a C program is an **asynchronous** parallel composition with $1_{\mathcal{A}}$, which we'll write as, $C \|^a 1_{\mathcal{A}_I}$

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

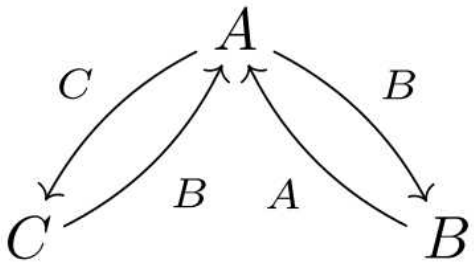
So we use the Q compiler and FramaC to prove



Using the Q Compiler and FramaC to Prove Spec Simulates Implementation



The designer gives a formal specification of their design in our statechart-like language, e.g.



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

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

A witness that Q simulates P ($P \preceq Q$ or $P \rightarrow Q$) decomposes into a relation on labels and a relation on the states

$$\begin{array}{ccc} \mathcal{A}_P & \xrightarrow{\dot{\rightarrow}_P} & \mathcal{P}(S_P \times S_P) \\ \downarrow f_{[R_{\mathcal{A}}]} & & \downarrow f_{[R_{\mathcal{S}}]} \\ \mathcal{A}_Q & \xrightarrow{\dot{\rightarrow}_Q} & \mathcal{P}(S_Q \times S_Q) \end{array}$$

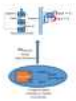
such that $f_{[R_{\mathcal{A}}]} \circ \dot{\rightarrow}_P \subseteq \dot{\rightarrow}_Q \circ f_{[R_{\mathcal{A}}]}$. *Is the implementer free to pick any relation?*

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

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 $\dot{\rightarrow}: \mathcal{A} \rightarrow \mathcal{P}(S \times S)$,

$$\frac{\beta \Rightarrow \alpha \; p \overset{\alpha}{\rightarrow}_C p'}{p \overset{\beta}{\rightarrow}_C p'} \qquad \frac{p \overset{\alpha}{\rightarrow} p' \; p \overset{\beta}{\rightarrow}_C p'}{p \overset{\alpha \vee \beta}{\rightarrow}_C p'} \tag{1.1}$$

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation



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 $\dot{\rightarrow}$: $\mathcal{A} \rightarrow \mathcal{P}(S \times S)$,

$$\frac{\beta \Rightarrow \alpha \; p \overset{\alpha}{\rightarrow}_C p'}{p \overset{\beta}{\rightarrow}_C p'} \qquad \frac{p \overset{\alpha}{\rightarrow} p' \; p \overset{\beta}{\rightarrow}_C p'}{p \overset{\alpha \vee \beta}{\rightarrow}_C p'} \tag{1.1}$$

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

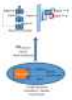
Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

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

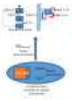
Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

- The labels in the abstraction are the expressions over abstract variables 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.

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation



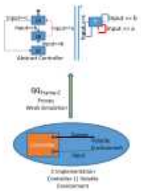
- The labels in the abstraction are the expressions over abstract variables Var_Q taking values in abstract value domain $\mathbb{V}_Q$.
- if $f_{[R_\mathcal{A}]}$ factors through a map of variables $f_{\mathsf{Var}} : \mathsf{Var}_Q \to \mathcal{P}(\mathsf{Var}_P)$ and a map of their value domains $f_{\mathbb{V}} : \mathbb{V}_Q \to \mathcal{P}(\mathbb{V}_P)$, 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



Sandia National Laboratories

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

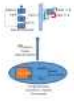
```
type simMap = {
  stateRels: stateRelAtom list;
  exprRels: exprRelAtom list;
  inputRels: inputRelAtom list;
  intVarRels: intVarRelAtom list;
  valueRels: valueRelAtom list;
...
}[@@deriving yojson]
```



Recall that the state of a C program C is given by a program point $l \in \text{Loc}_C$ and an execution environment $\varphi \in \mathcal{P}(\mathbb{V}^{\text{Var}})$, so, generically, R_S should be pairs of expressions and locations 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;
```

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation



The particular instance of the machine type carrying the execution environment of concern to us is

```
struct machine theMac;

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

so the implementer specifies

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

Using the Q Compiler and Frama C to Prove Spec Simulates Implementation

and the relevant program locations are at functions implementing the actions of the transition system

```
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);
  }
}
```

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

The implementer then gives the relation on states as

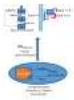
```
    "stateRels": [
        {
            "abStates": [
                {"stateName":"SA"}
            ],
            "impStates":[
                {"stateName":"SA", "funcName":"acti\
on_s00", "args": [{"typeName":"machine_t", "instanc\
eName":"mac"}]}
            ]
        },
```

Using the Q Compiler and FramaC to Prove Spec Simulates Implementation

the map on alphabets factors through a map on input variables

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

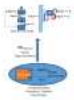
Using the Q Compiler to Prove Spec Simulates Implementation



and values

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

Using the Q Compiler to Prove 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');
}
```

so the implementer must specify that relation as well

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

Sandia National Laboratories

so Q generates the contract describing $1_{\mathcal{A}_{fgetc}}$, and a short program using clang searches through the codebase for the declaration of a volatile variable of that name at the appropriate scope

```
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;
 @  }
 @*/
```

Using the Q Compiler to Prove Spec Simulates Implementation

Sandia National Laboratories

Q then generates the ACSL annotations positing the Hoare-tuples described by the Spec. A short clang program searches through the code database for the program location/execution environment specified by the stateRel and annotates:

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

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

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

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

Using the Q Compiler to Prove Spec Simulates Implementation

Sandia National Laboratories

and defines a collection of clauses

```
 predicate Pinner_unit(integer currState, integer nextState, machine *theMac) =\
(currState == SA)&& (nextState == SA)&& \true&& \true&& ((!((!(((theMac->input))\
 == (('c')))) && (!(((theMac->input)) == (('b')))))) && (!(((theMac->input)) == \
(('c'))))) && (!(((theMac->input)) == (('b'))));
```

Using the Q Compiler to Prove Spec Simulates Implementation

Sandia National Laboratories

which form 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

the implementaiton transition relation in this design is implemented as an actual function

```
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

Sandia National Laboratories

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

Sandia National Laboratories

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

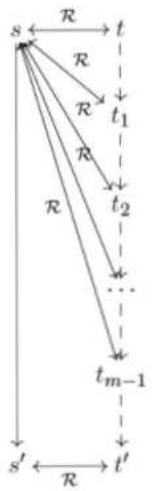
```
LTLSPEC NAME
Req001 := G((st=SA & (input!=BB & input!=CC)) -> X(st)=SA);

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, we don't, however, obtain a proof of the strong simulation defined above. Rather, we get a witness to a proof of *weak simulation*

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



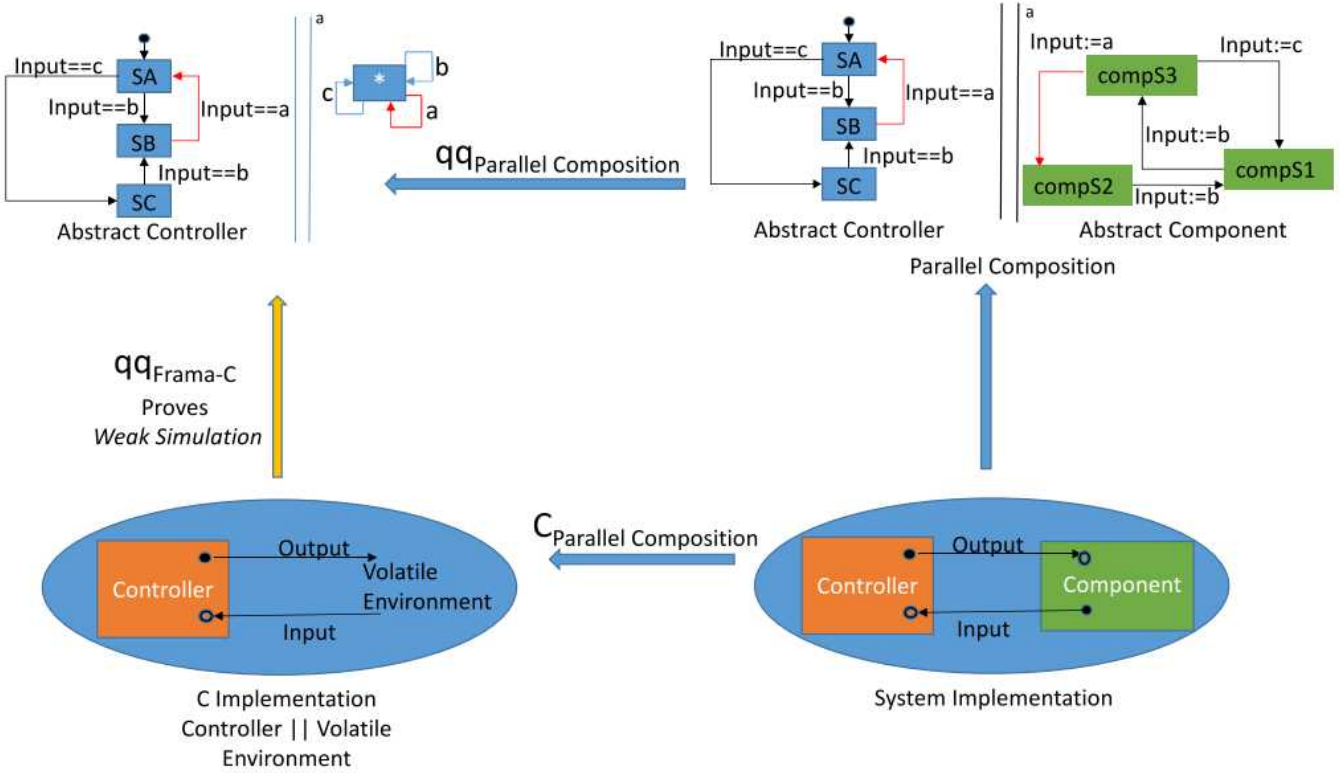

```
    ensures((theMac->currState == SB));
```

```
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);
  }
}
```

So we obtain safety properties.

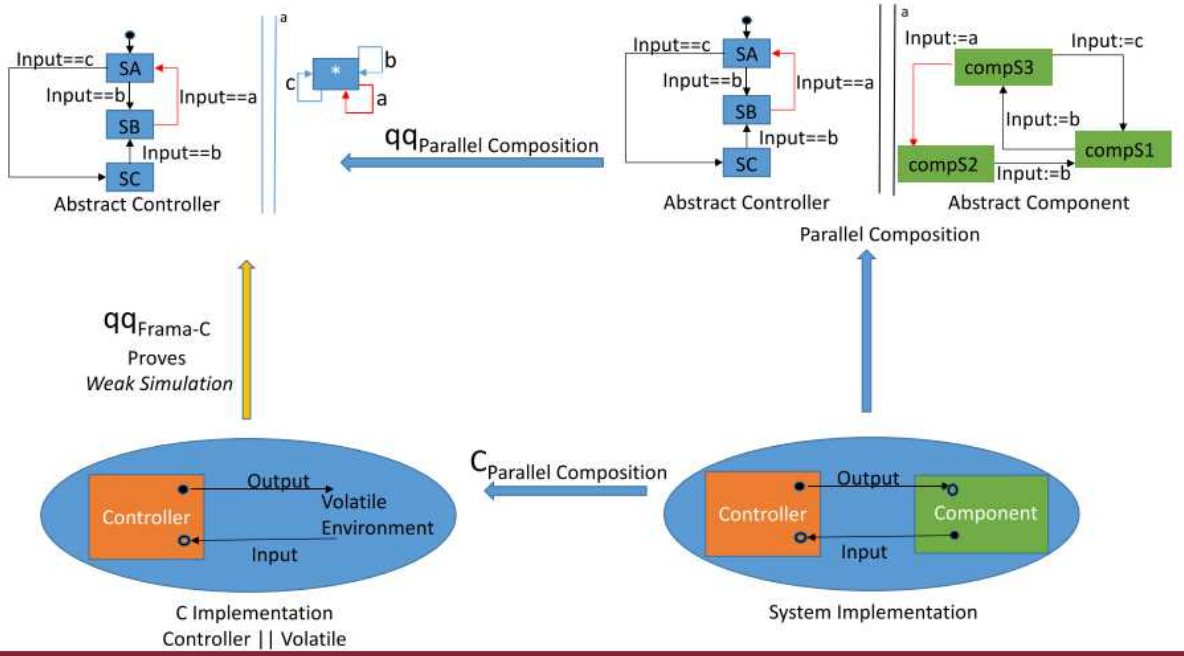
Proving LTL properties about Compositiins of Specs and Compositions of their Implementations

- 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



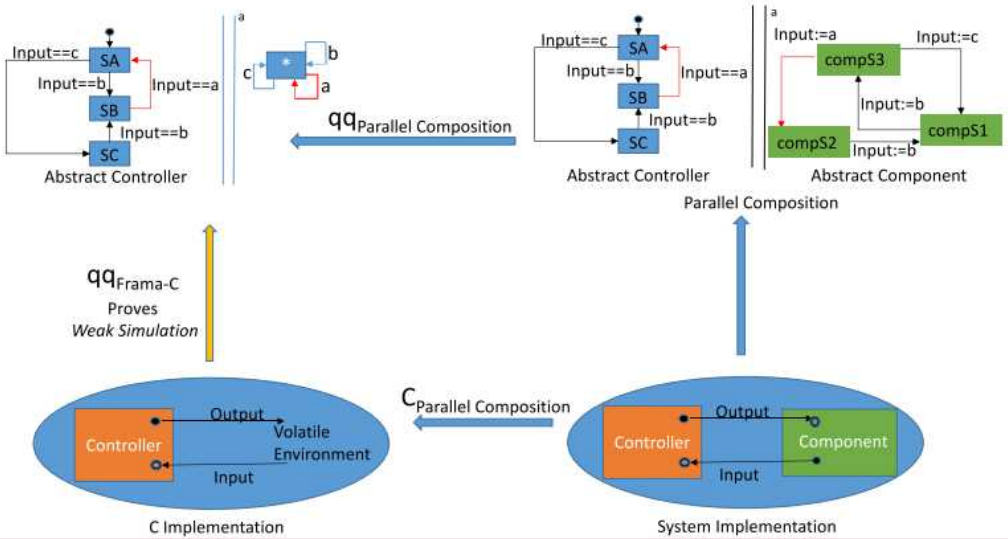
Proving LTL properties about Compositiins of Specs and Compositions of their Implementations

- 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 generates models and LTL proof obigations for a composition of abstractions (upper right hand corner)



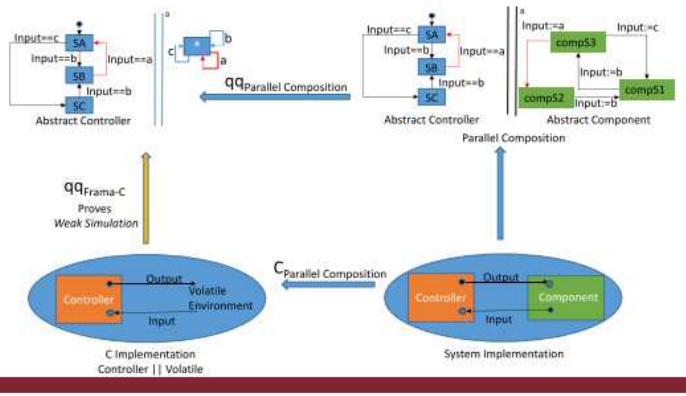
Proving LTL properties about Compositiins of Specs and Compositions of their Implementations

- 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 generates models and LTL proof obigations for a composition of abstractions (upper right hand corner)
- we want those proofs, along with the yellow arrow, to give proofs about the concrete system.

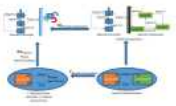


Proving LTL properties about Compositiins of Specs and Compositions of their Implementations

- 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 generates models and LTL proof obigations for a composition of abstractions (upper right hand corner)
- we want those proofs, along with the yellow arrow, to give proofs about the concrete system.
- For this, we need the blue arrows to be (weak?) simulation maps



Synchronous, Asynchronous, Strong, Weak



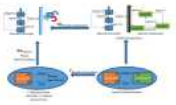
We'd like to reduce state space, and we'd like to model composition in hardware, so we actually have in Q a synchronous composition

Definition

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

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

Synchronous, Asynchronous, Strong, Weak



We'd like to reduce state space, and we'd like to model composition in hardware, so we actually have in Q a synchronous composition

Definition

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

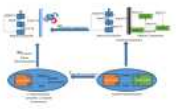
$$\frac{p \xrightarrow{\alpha}_P p' \; q \xrightarrow{\alpha}_Q q'}{(p,q) \xrightarrow{\alpha}_{P\|Q} (p',q')}$$

Lemma

Strong simulation $\preceq$ is a pre-congruence over $\|$, that is,

$\forall O, P, Q : \mathsf{LTS}. P \preceq Q \Rightarrow P \| O \preceq Q \| O$

Synchronous, Asynchronous, Strong, Weak



We can recover asynchronous composition and weak simulation through a completion

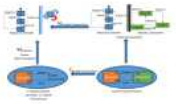
Definition

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

$$\frac{p \overset{\alpha}{\to}_P p'}{p \overset{\alpha}{\to}_{\mathsf{M}_{\mathcal{A}}} p'} \qquad \frac{\alpha \in \mathcal{A}}{p \overset{\alpha}{\to}_{\mathsf{M}_{\mathcal{A}}} p'} \tag{1.2}$$

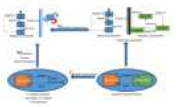
where $\Delta : \text{Set} \to \text{Set} :: S \mapsto \{(s, s) | s \in S\} \subseteq S \times S$.

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 $\mathcal{A}_{O_Q}$ and O_Q (labels output by Q). Let $\mathcal{A}_{L_Q} := \mathcal{A}_{O_Q} \sqcup \left(\mathcal{A}_Q \setminus \mathcal{A}_P\right)$

Synchronous, Asynchronous, Strong, Weak

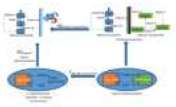


Let P, Q : LTS with alphabets $\mathcal{A}_P$ and $\mathcal{A}_Q$. We can recover the asynchronous composition from the synchronous composition via the completion once given a decomposition of their meet $\mathcal{A}_P \sqcap \mathcal{A}_Q$ into disjoint $\mathcal{A}_{O_P}$ (labels output by P) and $\mathcal{A}_{O_Q}$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

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 $P \preceq \mathsf{M}_{\mathcal{A}_{L_E}}(Q)$, or $P \to \mathsf{M}_{\mathcal{A}_{L_E}}(Q)$

Synchronous, Asynchronous, Strong, Weak

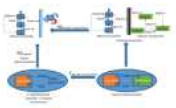


Lemma

The asynchronous composition is in the kernel of the strong simulation relation with the composition of completions

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

Synchronous, Asynchronous, Strong, Weak



Lemma

The asynchronous composition is in the kernel of the strong simulation relation with the composition of completions

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

Lemma

FramaC-WP *proves*

$$\mathsf{M}_{\mathcal{A}_{L_Q}}(P \| \mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q})) \overset{q_{\text{FramaC}}}{\leftarrow} \mathsf{M}_{\mathcal{A}_{L_Q}}(C) \| \mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q})$$

Soundness and Compositionality of Q

By terminality $1_{\mathcal{A}_Q} \leftarrow Q :!.$

Soundness and Compositionality of Q

By terminality $1_{\mathcal{A}_Q} \leftarrow Q$:!. By functoriality, $\mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q}) \leftarrow \mathsf{M}_{\mathcal{A}_{L_P}}(Q)$

Soundness and Compositionality of Q

By terminality $1_{\mathcal{A}_Q} \leftarrow Q$:!. By functoriality, $\mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q}) \leftarrow \mathsf{M}_{\mathcal{A}_{L_P}}(Q)$ Therefore, by precongruence of simulation ($\succeq$ or $\leftarrow$) over $\|$,

Soundness and Compositionality of Q

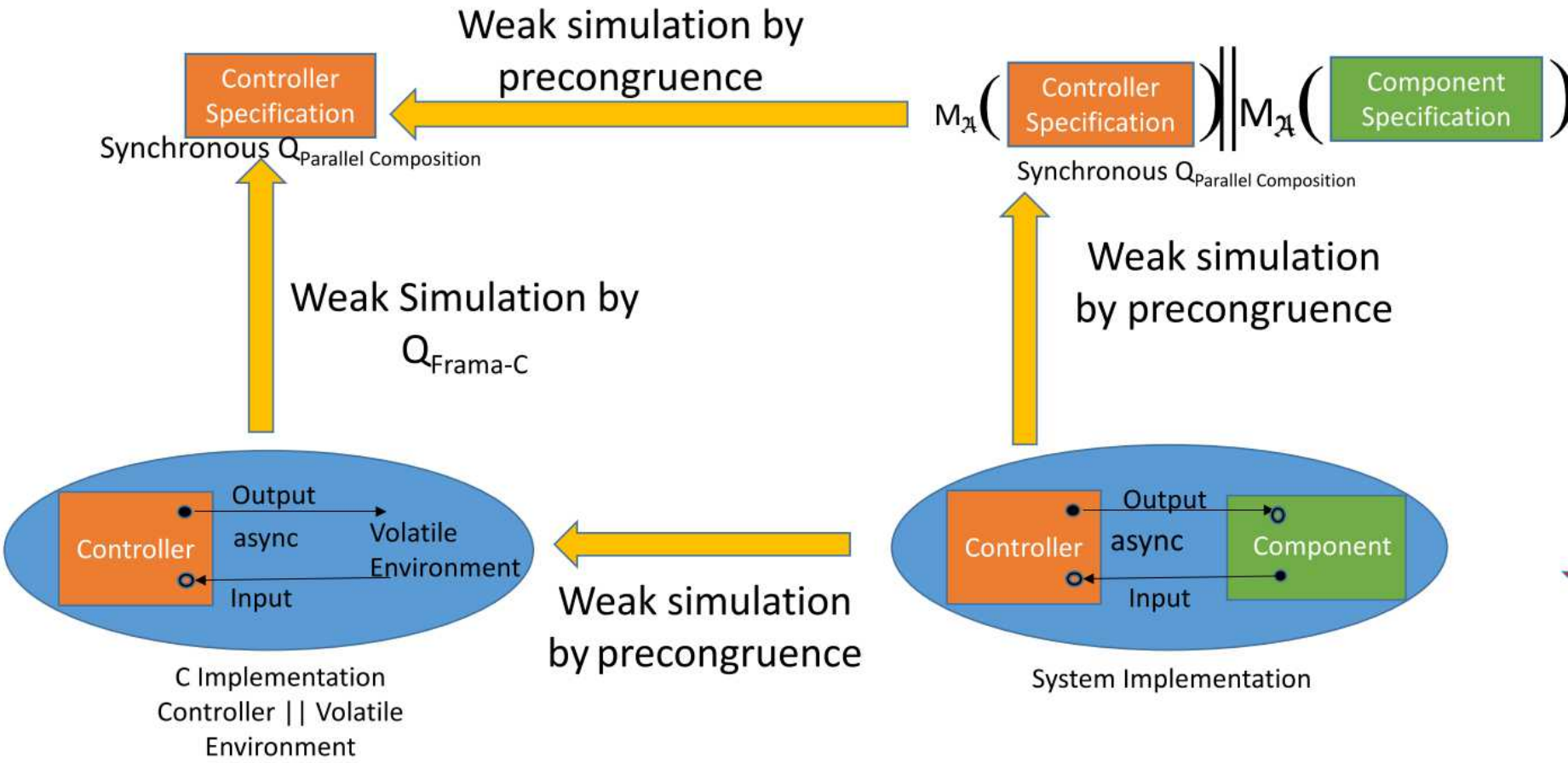
By terminality $1_{\mathcal{A}_Q} \leftarrow Q :!.$ By functoriality, $\mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q}) \leftarrow \mathsf{M}_{\mathcal{A}_{L_P}}(Q)$ Therefore, by precongruence of simulation ($\succeq$ or $\leftarrow$) over $\|$,

Theorem (Main Theorem)

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

$$\begin{array}{ccc}
\mathsf{M}_{\mathcal{A}_{L_Q}}(P \| \mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q})) & \xleftarrow{\mathsf{M}_{\mathcal{A}_{L_Q}}(!)} & \mathsf{M}_{\mathcal{A}_{L_Q}}(P) \| \mathsf{M}_{\mathcal{A}_{L_P}}(Q) \\
\uparrow q_{\mathsf{FramaC}} & & \uparrow \\
\mathsf{M}_{\mathcal{A}_{L_Q}}(C) \| \mathsf{M}_{\mathcal{A}_{L_P}}(1_{\mathcal{A}_Q}) & \xleftarrow[!]{} & \mathsf{M}_{\mathcal{A}_{L_Q}}(C) \| \mathsf{M}_{\mathcal{A}_{L_P}}(Q)
\end{array}$$

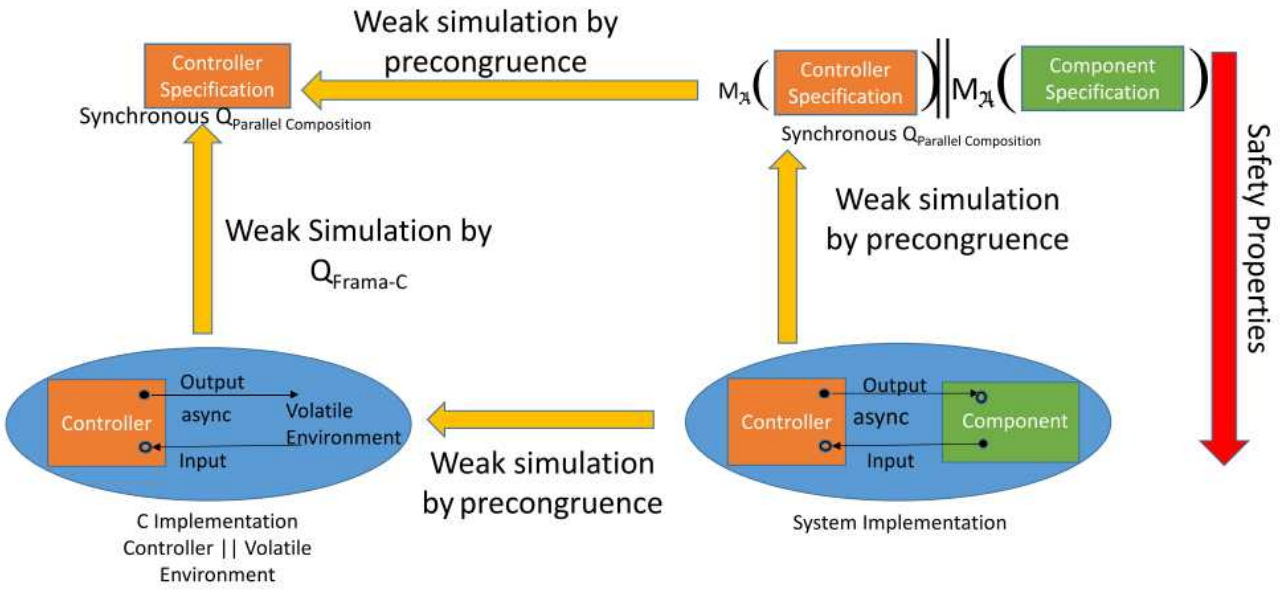
What Was Accomplished



What Was Accomplished

Thus we have accomplished our goal -

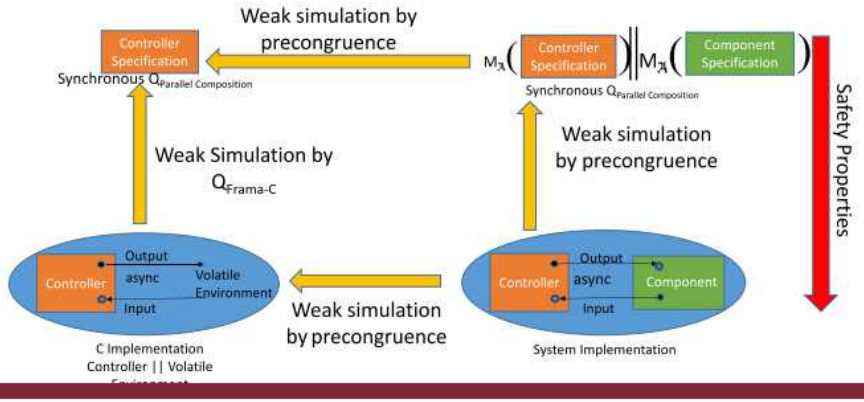
- Given a proposed witness of simulation of a controller implementation by its spec, the Q compiler generates and emplaces the ACSL annotations from which FramaC generates the proof of simulation.



What Was Accomplished

Thus we have accomplished our goal -

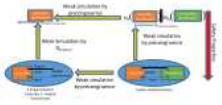
- Given a proposed witness of simulation of a controller implementation by its spec, the Q compiler generates and emplaces the ACSL annotations from which FramaC generates the proof of simulation.
- The Q compiler computes the composition of that specification with (the completion of the) specifications 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 checked.



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 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