# VLSM: A General Coq Framework for Reasoning About Faulty Distributed Systems

Vlad Zamfir[1,2], Denisa Diaconescu[2,3], Wojciech Kolowski[2],
Brandon Moore[2], Karl Palmskog[2,4], Traian Florin Serbanuta[2,3],
Ioan Teodorescu[3]

[1]Ethereum Foundation [2]Runtime Verification, Inc.
[3]University of Bucharest [4]KTH Royal Institute of Technology

`https://github.com/runtimeverification/vlsm`

# Context: Message-Passing Distributed Systems

- stateful components communicate (only) by sending messages
- messages (usually) delivered over a network
- asynchronous: no firm upper bound on message delivery time

Many application areas:

- state machine replication (key-value stores)
- scientific computing
- blockchains and cryptocurrencies

# Faults in Distributed Systems

Distributed systems can go wrong in many ways:

- messages arriving out-of-order
- component crashes (with recovery?)
- messages getting corrupted or lost
- components misbehaving (adversarially?)

Several techniques can be used to address faults:

- stable component storage and state checkpointing
- component redundancy
- cryptographic message signatures
- trusted hardware

# Message Passing Distributed Systems in Coq

- attractive application area due to high risk of errors
- executable code small in size compared to theory
- many tricky environmental assumptions

Examples from the literature:

- **Verdi**, PLDI'15/CPP'16, applied to Raft protocol, crash fault-tolerant state-machine replication
- **Chapar**, POPL'18, causally consistent key-value store
- **Velisarios**, ESOP'18, applied to PBFT protocol, Byzantine fault-tolerant state-machine replication
- **Disel**, POPL'18, applied to 2PC protocol
- **Aneris**, ESOP'20, applied to load balancing and 2PC protocol

# VLSM Overview

- a Coq framework for modeling and reasoning about distributed systems subject to faults, building on a (pen-and-paper) theory of *Validating Labeled State transition and Message production systems*
- a single VLSM can represent a local component view
- global system view comes from composing multiple VLSMs and lifting the local “validity constraint” of each component
- framework focuses on modeling distributed systems subject to *equivocation* rather than Byzantine faults

# VLSM Definition

A VLSM is a tuple $\mathcal{V} = (L, S, S_0, M, M_0, \tau, \beta)$, such that

- $L$ is a set of labels for transitions
- $S$ is a non-empty set of states
- $S_0 \subseteq S$ is a non-empty set of initial states
- $M$ is set of messages
- $M_0 \subseteq M$ is a set of initial messages
- $\tau : L \times S \times M? \to S \times M?$ is a transition function
- $\beta \subseteq L \times S \times M?$ is a predicate filtering the inputs for the transition function

where $M? = M \cup \{\otimes\}$ and $\otimes$ stands for "no message".

# VLSM Coq Encoding

```
Record VLSMType (message : Type) : Type :=
 { state : Type; label : Type; }.

Record VLSMMachine {message : Type} (T : VLSMType message) : Type := {
 initial_state_prop : state T → Prop;
 initial_state : Type := {s : state T | initial_state_prop s};
 s0 :  Inhabited initial_state;
 initial_message_prop : message → Prop;
 initial_message : Type := {m : message | initial_message_prop m};
 transition : label T → state T ∗ option message →
  state T ∗ option message;
 valid : label T → state T ∗ option message → Prop;
}.

Record VLSM (message : Type) : Type := mk_vlsm
 { vtype :> VLSMType message; vmachine :> VLSMMachine vtype; }.
```

# Rationale for Coq Definition

```
Record VLSM (message : Type) : Type := mk_vlsm
 { vtype :> VLSMType message; vmachine :> VLSMMachine vtype; }.
```

Q1: why parameterize the `VLSM` Coq definition over a message type?

A1: we often need to consider VLSMs over the same set of messages

Q2: why split the `VLSM` Coq definition into a "VLSM type" and a "VLSM machine"?

A2: we often need to consider VLSMs over the same labels/states

# Free Composition of VLSMs

Let $\{\mathcal{V}_i\}_{i=1}^n$ be an indexed set of VLSMs over a set of messages $M$. The free composition of $\{\mathcal{V}_i\}_{i=1}^n$ is $\sum_{i=1}^n \mathcal{V}_i = (L, S, S_0, M, M_0, \tau, \beta)$

- $L = \bigcup_{i=1}^n \{i\} \times L_i$ is the disjoint union of labels
- $S = \prod_{i=1}^n S_i$ is the product of states
- $S_0 = \prod_{i=1}^n S_{i,0}$ is the product of initial states
- $M$ is the same set of messages as for each $\mathcal{V}_i$
- $M_0 = \bigcup_{i=1}^n M_{i,0}$ is the union of all initial messages
- $\tau : L \times S \times M? \to S \times M?$ is defined component-wise using labels,

$$\tau(\langle j, l_j\rangle, \langle s_1, \ldots s_n\rangle, m) =$$
$$\big(\langle s_1, \ldots, s_{j-1}, \tau_j^s(l_j, s_j, m), s_{j+1}, \ldots s_n\rangle, \tau_j^m(l_j, s_j, m)\big)$$

- $\beta \subseteq L \times S \times M?$ is defined component-wise using labels,

$$\beta(\langle j, l_j\rangle, \langle s_1, \ldots, s_n\rangle, m) = \beta_j(l_j, s_j, m)$$

# VLSM Composition Constraints

A composition constraint $\varphi$ is a predicate additionally filtering the inputs for the composed transition function, i.e.,

$$\varphi \subseteq L \times S \times M?$$

The constrained VLSM composition under $\varphi$ of $\{\mathcal{V}_i\}_{i=1}^{n}$ is the VLSM which has the same components as the free composition, except for the validate predicate which is further constrained by $\varphi$

$$\left(\sum_{i=1}^{n} \mathcal{V}_i\right)\Big|_{\varphi} = (L, S, S_0, M, M_0, \tau, \beta \cap \varphi)$$

The constrained VLSM composition might have fewer valid states/messages than the free VLSM composition.

# VLSM Composition in Coq

- in Coq, we allow the composition of an indexed set of VLSMs `IM : index -> VLSM message`, where `index` has decidable equality
- the state of a composite VLSM becomes a dependent product, yielding a particular state from the corresponding component for each index
- the label type is a dependent sum pairing an index with a label from the corresponding component

```
Definition composite_state : Type := ∀ n : index, vstate (IM n).
Definition composite_label : Type := sigT (fun n ⇒ vlabel (IM n)).
```

# The Parity VLSM Example

- our framework tutorial begins with the parity VLSM $\mathcal{P}$
- parity is the property of integers of being even or odd
- $\mathcal{P}$ stores a tuple and continually decrements one of the tuple's elements while a constraint is checked at each transition step

`https://github.com/runtimeverification/vlsm/blob/master/theories/VLSM/Core/Examples/Parity.v`

# Parity VLSM Definition

Let $\mathcal{P}$ be the following VLSM:

- $L = \{d\}$, an arbitrary singleton set
- $S = \{\langle n, i \rangle \mid n, i \in \mathbb{Z}\}$
- $S_0 = \{\langle n, n \rangle \mid n \geq 0\}$
- $M = \mathbb{Z}$
- $M_0 = \{2\}$
- for any integers $n, i$, and $j$, there is a transition

$$\langle n, i \rangle \xrightarrow[j \to 2j]{d} \langle n, i - j \rangle$$

- the validity constraint predicate is defined as

$$\beta = \{(d, \langle n, i \rangle, j) \mid i \geq j \geq 1\}$$

# Parity VLSM Transitions and Traces

$\mathcal{P}$ transition example:

$$\langle 5, 4\rangle \xrightarrow[10 \to 20]{d} \langle 5, -6\rangle.$$

$\mathcal{P}$ valid trace example:

$$\langle 8, 8\rangle \xRightarrow[4 \to 8]{d} \langle 8, 4\rangle \xRightarrow[2 \to 4]{d} \langle 8, 2\rangle \xRightarrow[2 \to 4]{d} \langle 8, 0\rangle$$

$\mathcal{P}$ valid trace example:

$$\langle 5, 5\rangle \xRightarrow[2 \to 4]{d} \langle 5, 3\rangle \xRightarrow[2 \to 4]{d} \langle 5, 1\rangle \xrightarrow[1 \to 2]{d} \langle 5, 0\rangle$$

# Parity VLSM Properties

### Lemma

$m \in M_{\mathcal{P}} \setminus \{\otimes\}$ *iff* $m = 2^p$, *where* $p \geq 1$.

"valid non-empty messages are always a positive power of two"

### Theorem

$\langle n, i\rangle \in S_{\mathcal{P}}$ *iff* $n \geq i \geq 0$, *where* $n$ *and* $i$ *have the same parity.*

"integers in valid states are non-negative with the first greater than or equal to the second, and are either both even or both odd"

# Parity VLSM Coq Encoding (Using Std++)

```
Definition ParityLabel : Type := unit.
Definition ParityState : Type := Z ∗ Z.
Definition ParityMessage : Type := Z.

Definition ParityType : VLSMType ParityMessage :=
 {| state := ParityState; label := ParityLabel; |}.

Definition ParityComponent_initial_state_prop (st : ParityState) :=
  st.1 >= 0 ∧ st.1 = st.2.

Definition ParityComponent_transition
  (l : ParityLabel) (s : ParityState) (om : option ParityMessage)
  : ParityState ∗ option ParityMessage :=
  match om with
  | Some j  ⇒ ((s.1, s.2 − j), Some (2 ∗ j))
  | None    ⇒ (s, None)
  end.
```

# Parity VLSM Coq Encoding, Continued

```
Definition ParityComponentValid (l : ParityLabel) (st : ParityState)
 (om : option ParityMessage) : Prop :=
  match om with
  | Some msg ⇒ msg <= st.2 ∧ 1 <= msg
  | None     ⇒ False
  end.

Definition ParityComponent_initial_state_type : Type :=
  {st : ParityState | ParityComponent_initial_state_prop st}.

Program Definition ParityComponent_initial_state :
  ParityComponent_initial_state_type := exist _ (0, 0) _.
Next Obligation.
Proof. done. Defined.
```

# Parity VLSM Coq Encoding, Continued

```
Instance ParityComponent_Inhabited_initial_state_type :
 Inhabited (ParityComponent_initial_state_type) :=
  populate (ParityComponent_initial_state).

Definition ParityMachine : VLSMMachine ParityType :=
{|
 initial_state_prop := ParityComponent_initial_state_prop;
 initial_message_prop := fun (ms : ParityMessage) ⇒ ms = 2;
 s0 := ParityComponent_Inhabited_initial_state_type;
 transition := fun l '(st, om) ⇒ ParityComponent_transition l st om;
 valid := fun l '(st, om) ⇒ ParityComponentValid l st om;
|}.

Definition ParityVLSM : VLSM ParityMessage :=
 {| vtype := ParityType; vmachine := ParityMachine; |}.
```

# Parity VLSM Coq Encoding, Continued

### Lemma

*$m \in M_{\mathcal{P}} \setminus \{\otimes\}$ iff $m = 2^p$, where $p \geq 1$.*

```
Lemma parity_valid_messages_powers_of_2 :
 ∀ (om : option ParityMessage), om <> None →
  ((option_valid_message_prop ParityVLSM om) ↔
   (∃ p : Z, p >= 1 ∧ om = Some (2 ^ p))).
Proof. (* ... *) Qed.
```

### Theorem

*$\langle n, i \rangle \in S_{\mathcal{P}}$ iff $n \geq i \geq 0$, where $n$ and $i$ have the same parity.*

```
Theorem parity_valid_states_same_parity :
  ∀ (s : ParityState),
  valid_state_prop ParityVLSM s ↔
   ((Z.Even s.2 ↔ Z.Even s.1) ∧ s.1 >= s.2 ∧ s.2 >= 0).
Proof. (* ... *) Qed.
```

# VLSMs and Equivocation

- **equivocation** refers to claiming different beliefs about the state of the protocol to different parts of the system in order to steer the protocol-following components into making inconsistent decisions
- an equivocating component may claim a bit is 0 to one part of the system, and 1 to the other
- an equivocating component behaves as if running multiple copies of the protocol
- for VLSMs, equivocation fault tolerance analysis takes place of Byzantine fault tolerance analysis

# Equivocation-Limited Message Observer (ELMO) Protocols

- ELMO is a VLSM that checks message validity, ensures that the component does not self-equivocate, and allows receiving a message only if this will not bring the total weight of locally-visible equivocating components above a fixed equivocation threshold
- an ELMO protocol is a constrained composition of ELMO components which ensures that the global equivocation exhibited by the system remains below a fixed threshold
- we say that ELMO components are "validating" for the composition with a limited amount of global equivocation

```
Inductive Label : Type := Receive | Send.
Inductive State : Type :=
 MkState { obs : list Observation; adr : Address; }
with Observation : Type :=
 MkObservation { label : Label; message : Message; }
with Message : Type := MkMessage { state : State; }.
```

# Reduction of Byzantine Behavior to Equivocation

We can model Byzantine behavior for a VLSM `M` by considering all the valid traces it can produce when being freely composed with an arbitrary VLSM:

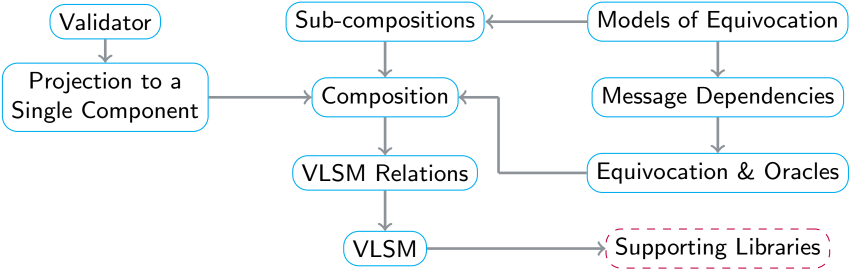
```
Definition byzantine_trace_prop (tr : vTrace M) :=
 ∃ (M' : VLSM message) (Proj := binary_free_composition_fst M M'),
  valid_trace_prop Proj tr.
```

# Reduction of Byzantine Behavior to Equivocation

When `M` is instantiated with a constrained composition of VLSMs whose every component is able to validate whether received messages can be validly produced by the constrained composition, we are able to resist exposure to arbitrary Byzantine behavior:

```
Lemma composite_validator_byzantine_traces_are_not_byzantine
{message : Type} ‘{EqDecision index} (IM : index → VLSM message)
(constraint : composite_label IM →
  composite_state IM * option message → Prop)
(X := composite_vlsm IM constraint)
(Hvalidator : ∀ i : index,
 component_message_validator_prop IM constraint i) :
∀ (tr : vTrace X), byzantine_trace_prop X tr →
 valid_trace_prop X tr.
```

# Library Module Overview



# Current Framework Coq Code Statistics

### Whole Project

Spec LOC: 19605, Proof LOC: 22242, Comment LOC: 3919

### Support Library

Spec LOC: 3331, Proof LOC: 4203, Comment LOC: 359

### VLSM Library and Applications

Spec LOC: 16274, Proof LOC: 18039, Comment LOC: 3560

# Usage of Coq's Stdlib

- earlier, VLSM framework only used Stdlib, partly due to dependency aversion
- Stdlib has many useful typeclasses, but not `Decision`
- this led to ad-hoc conversions between `bool` and `Prop`
- also problematic ad-hoc use of automation tactics (`auto`, `intuition`)

# Embracing Typeclasses via Std++

- MathComp/SSReflect not an option due to heavy typeclass use
- Std++ allowed to purge many `bool` uses through `Decision`
- Std++ provided SSReflect-like finishers
- typeclass hierarchy shallow enough to use unbundled typeclasses
- fewer problems with divergence when consistently using typeclass `Hint Mode`

# Itauto as Replacement for Intuition Tactic

- "`intuition.`" used to mean "`intuition auto with *.`"
- "`tauto.`" means "`intuition fail.`"
- unclear when we get propositional solving or not
- we adopted the Itauto plugin and `itauto` tactic as replacement for `intution`/`tauto` (and `auto`)
- `congruence` and `lia` are popular leaf tactics
- Itauto now part of the Coq Platform
- due to advice from Itauto developer, we stay away from `smt` tactic (Nelson-Oppen combinations)

# Axiom Management

- we use functional extensionality and sometimes classical logic
- recent problem: `itauto` pulling in `classic` via NNPP whenever `Classical_Prop` module is required
- solved by separating `itauto` and `ctauto` (also upstream)
- `make validate` is not precise enough for good axiom analysis, see Coq issue #17345

# Reasoning on VLSM Traces

- VLSM traces can be finite or infinite ...
- ... but is this typically known up front?
- ongoing investigation of encoding of *possibly-infinite* LTS traces
- assume finiteness and infiniteness precisely when needed
- positive vs. negative coinduction: do we care about subject reduction?
- need better automation for coinduction, currently experimenting with `coinduction` by Damien Pous
- more LTL-like machinery for temporal reasoning would be nice

# Positive vs. Negative Finiteness

```
Set Primitive Projections.
Variant traceF (trace : Type) : Type :=
| TnilF (a : A) | TconsF (a : A) (b : B) (tr : trace).
CoInductive trace : Type := go { _observe : traceF trace }.
Notation trace' := (traceF trace).
Definition observe (tr : trace) : trace' :=
 _observe tr.
Inductive finite' : trace' → Prop :=
| Fin_Tnil : ∀ a, finite' (TnilF a)
| Fin_Tcons : ∀ (a : A) (b : B) tr,
   finite' (observe tr) → finite' (TconsF a b tr).
Definition finite : trace → Prop := fun tr ⇒ finite' (observe tr).
```

# Development Practices

- continuous integration based on Docker-Coq with two latest released Coq versions
- at least one approving review required to merge a pull request
- coding conventions: a mix of Std++ and Stdlib, enforced during review
- pull requests with focused changes more quickly approved

# Conclusion

Project status:

- continuously developed on GitHub: `https://github.com/runtimeverification/vlsm`
- open source under the BSD-3-Clause license
- releases available on Coq's opam archive (`coq-vlsm`)
- pen-and-paper theory described in arXiv paper linked on GitHub

Ongoing work:

- additional examples and extensive VLSM tutorial
- additional consensus related applications