Specification-Based Verification and Testing of Internet Protocols

David Nowak and Yutaka Oiwa

RCIS, AIST

Summary

A research project funded by NICT
From October 2010 to March 2013
(Jointly with Lepidum, Inc.)

Objective:

Design a methodology to certify implementation of Internet protocols

Leader:

Etsuya Shibayama (AIST & The University of Tokyo)

AIST Participants:

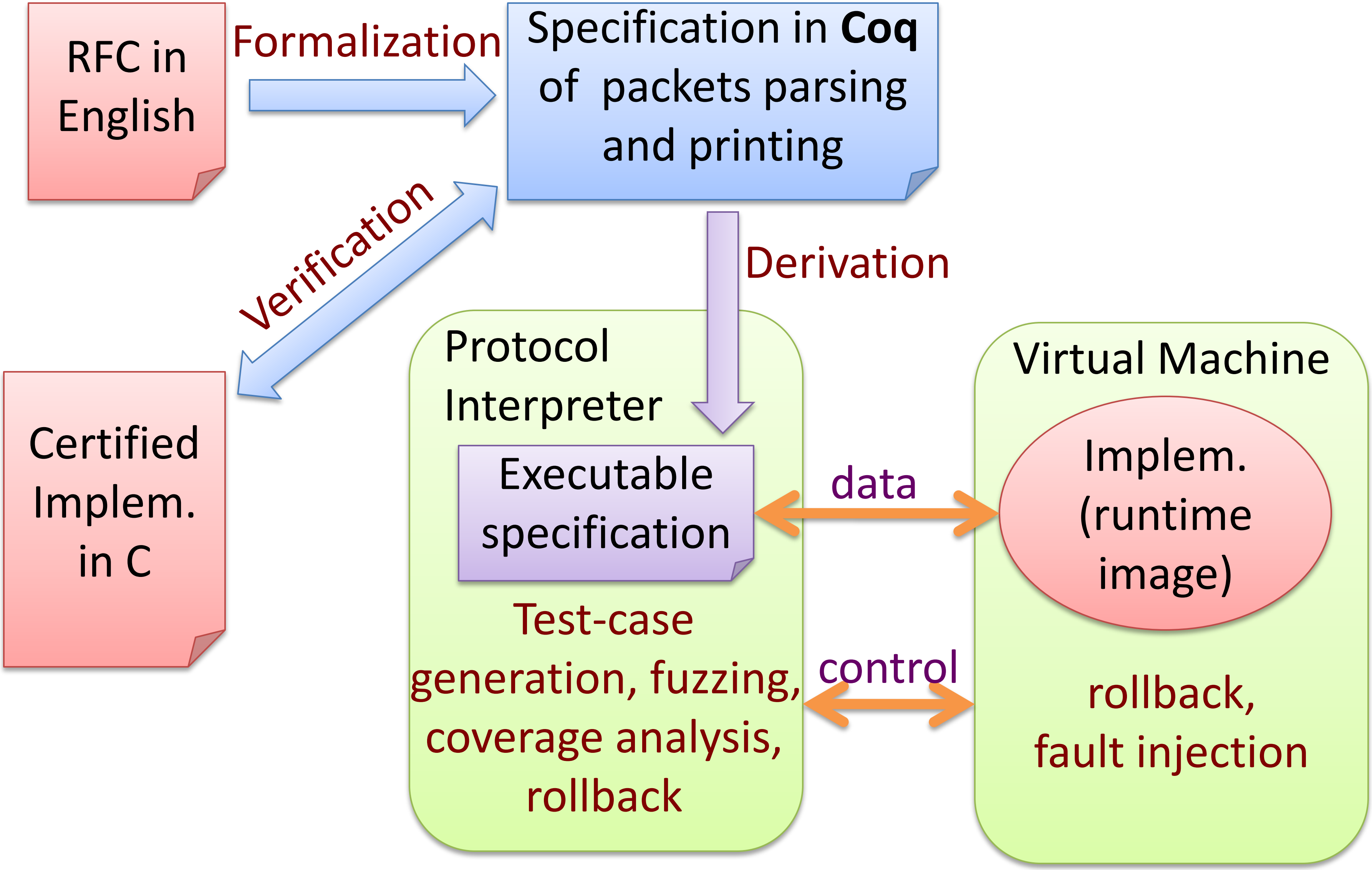
- Reynald Affeldt (AIST)
- David Nowak (AIST)
- Yutaka Oiwa (AIST)
- Kuniyasu Suzaki (AIST)

Complementarity of formal verification and testing

We combine theorem proving and testing.

	Proof Assistant	Model checking	Testing
Technique	Mathematical proof	Abstraction, State-space exploration	Sampling
Result	Correctness proof	Correctness proof, Bug discovery	Bug discovery
Object	Source code	Abstracted source code	Runtime behavior
Scalability	Low	Medium	High
Expressivity	High (Higher-order logic)	Low (Temporal logic)	Medium

Overview of the project



RFC in English

Formalization

Specification in **Coq** of packets parsing and printing

Verification

Certified Implem. in C

Derivation

Protocol Interpreter

Executable specification

Test-case generation, fuzzing, coverage analysis, rollback

data

control

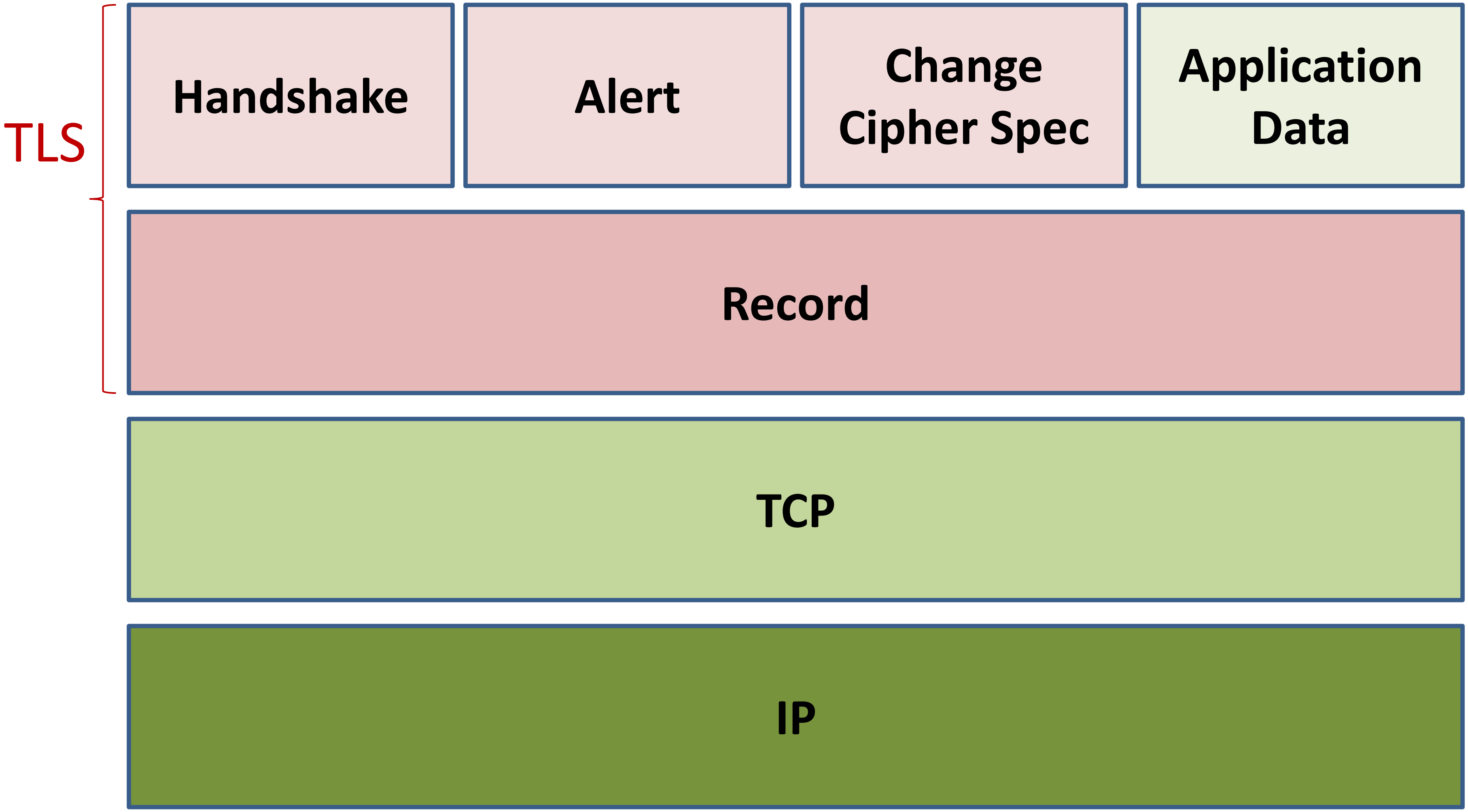
Virtual Machine

Implem. (runtime image)

rollback, fault injection

Case study: TLS

A cryptographic layer on top of existing communication protocols


TLS

Handshake

Alert

Change Cipher Spec

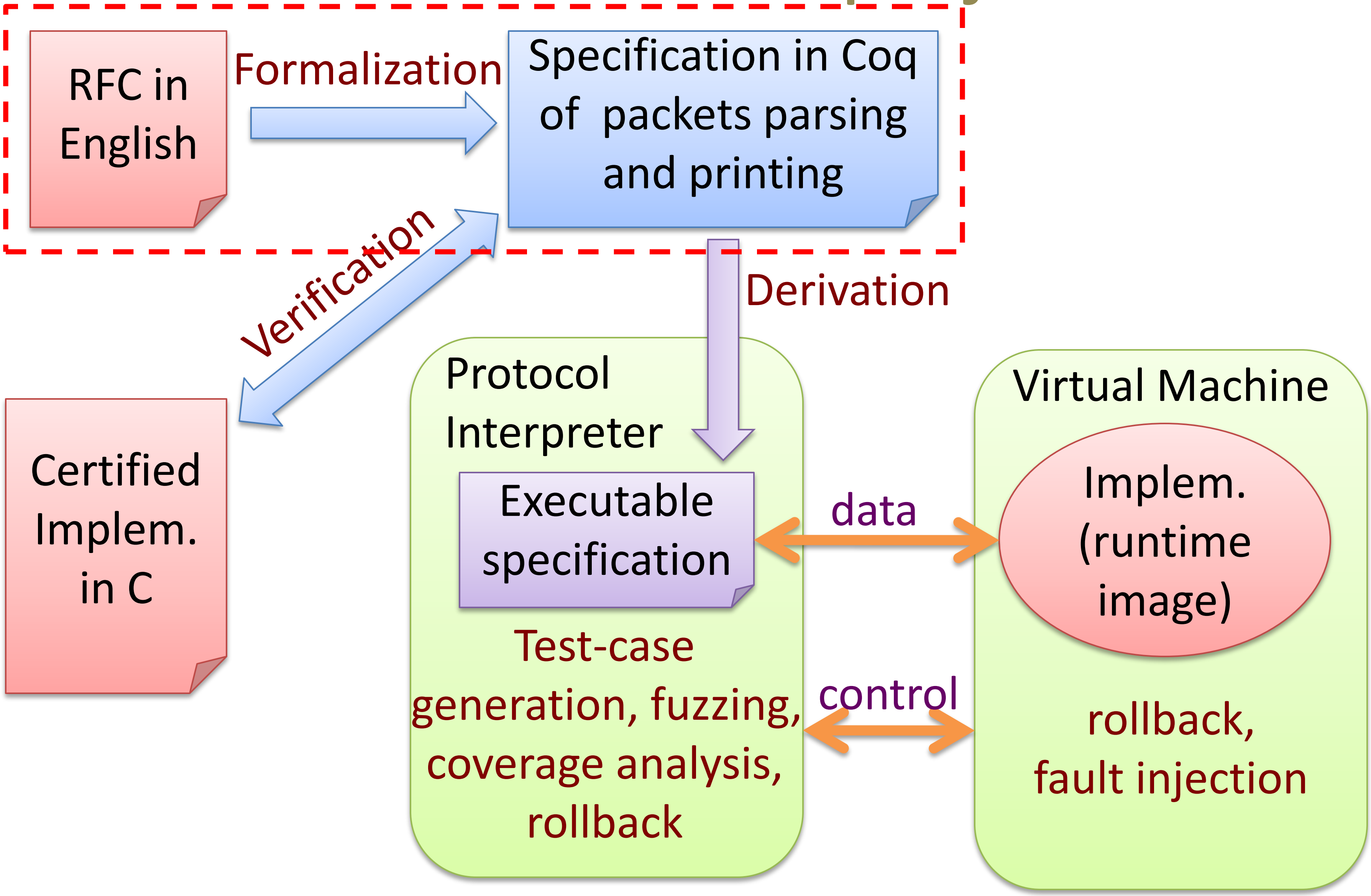
Application Data

Record

TCP

IP

Overview of the project



RFC in English

Formalization

Specification in Coq of packets parsing and printing

Verification

Derivation

Certified Implem. in C

Protocol Interpreter

Executable specification

Test-case generation, fuzzing, coverage analysis, rollback

data

control

Virtual Machine

Implem. (runtime image)

rollback, fault injection

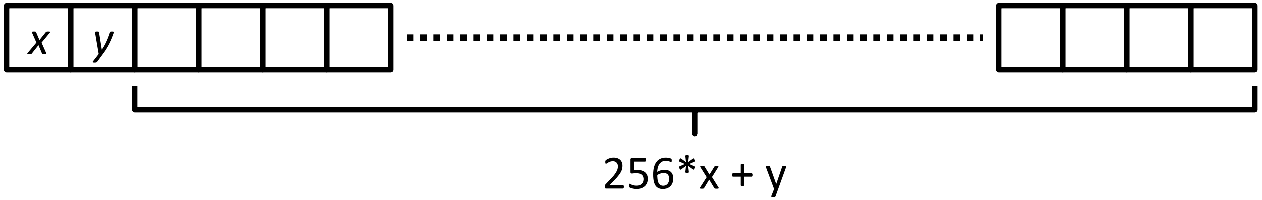
Data-dependent parsing for TLS

- To specify TLS packets, there is a need for dependent types.

Example:

```
Record Handshake : Set := {
  msg_type : HandshakeType ;
  body : HandshakeType_type msg_type
}.
```

- We also need dependent types to parse variable-length packets.



- We will use dependent types available in the proof assistant **Coq**.

Parsing monad

A monad is an abstract data type that allows to embed imperative features in a purely functional language (like Coq).

```
Definition state : Type :=
  in_channel * out_channel * nat.

Definition parser (A:Type) : Type :=
  state -> exception (A * state).

Definition ret {A:Type}(a:A) : parser A :=
  fun s => value (a,s).

Definition bind {A B:Type}(p:parser A)(f:A -> parser B) :
    parser B :=
  fun s =>
    match p s with
    | value (a,s') => f a s'
    | error msg => error msg
    end.
```

Dependent parsing monad

For data-dependent parsing, we need a dependently typed *bind*:

```
Definition bind_dep
    {A:Type}{B:A->Type}(p:parser A)(f:forall a:A, parser (B a)) :
    parser {a:A & B a} :=
  fun s =>
    match p s with
    | value (a,s') =>
        match f a s' with
        | value (b, s'') => value (existT (fun a => B a) a b, s'')
        | error msg => error msg
        end
    | error msg => error msg
    end.
```

Example: parsing handshake packets

We parse in order the packet’s type, its length and its body:

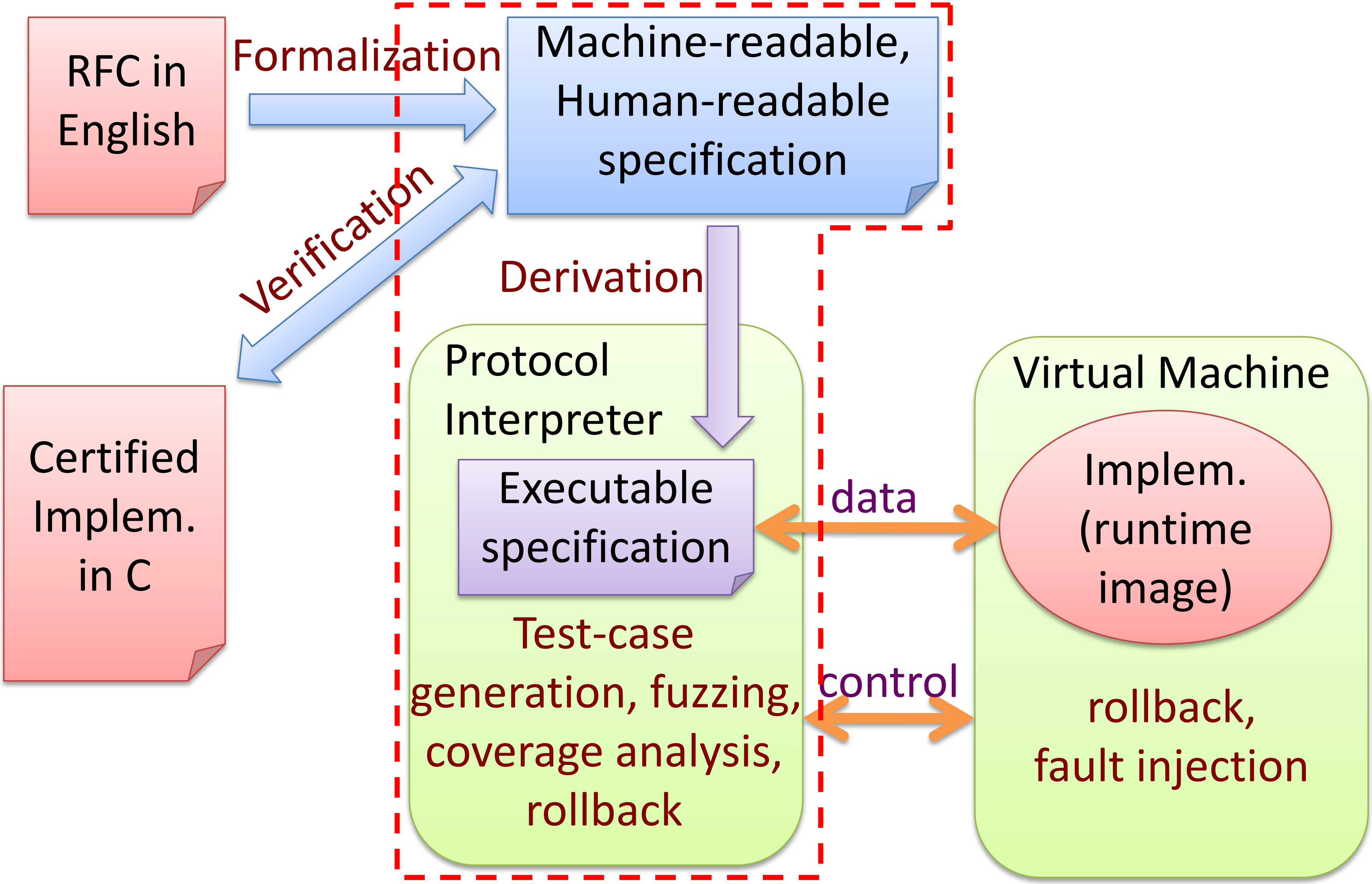
```
Definition parse_Handshake' :
    parser {ht : HandshakeType & HandshakeType_type ht} :=
  bind_dep parse_HandshakeType (
    fun ht =>
      len <<= parse_Z 3 ;
      match ht return parser (HandshakeType_type ht) with
      | hello_request => parse_exact (Zabs_nat len) parse_HelloRequest
      | client_hello => parse_exact (Zabs_nat len) parse_ClientHello
      | server_hello => parse_exact (Zabs_nat len) parse_ServerHello
      | certificate => parse_exact (Zabs_nat len) parse_Certificate
      | server_hello_done => parse_exact (Zabs_nat len) parse_ServerHelloDone
      end
  ).

Definition parse_Handshake : parser Handshake :=
  h <<= parse_Handshake' ;
  ret {| msg_type := projT1 h ; body := projT2 h |}.
```

Parsing/Printing monad

- In some sense, they are inverse to each other.
- Parsers and printers are often implemented separately.
  - → Redundancy
  - → Potential inconsistency
- We will use **invertible syntax descriptions**:
  a monadic approach that unifies parsing and printing

Overview of the project



RFC in English

Formalization

Machine-readable, Human-readable specification

Verification

Certified Implem. in C

Derivation

Protocol Interpreter

Executable specification

Test-case generation, fuzzing, coverage analysis, rollback

data

control

Virtual Machine

Implem. (runtime image)

rollback, fault injection

Roadmap of the following parts:

- To design a specification language that is:
  - Precise enough to extract parsing/encoding programs automatically
    - Parsing part done in work in the the previous slides
    - *This Coq work shows us how to do theoretic things here, very precisely!*
      - *Type design, dependent types, etc…*
      - *If it can be done in Coq, we can do it anywhere* :-)
  - Human-friendly enough to be read as document
    - Add some “human-friendly” nuts where needed
      - E.g. BNF, state machine description, etc…
    - Add hints for “fuzzing” in the next stage

Roadmap of the following parts:

- Extract “correct” testing program (reference impl.) from the specification
  - Just interprets the input specification
- Derive “incorrect” testing program, too
  - Mix the “fuzzings” with the extracted program
    - E.g. trailing garbage, data overrun,
      incorrect input for case branches…
    - Needs more hints in the input specification than above

Roadmap of the following parts:

- Build a black-box test program
    - Talk with black-box test targets
    - Non-deterministically choose several “correct” or “incorrect” behaviors on the fly
- Use program rollback to trace all possible cases to see what happens
    - Test all “correct” and “incorrect” cases thoroughly
    - Use VM to rollback execution of target program

Roadmap of the following parts:

- (Rough) Plans
  - *FY2010: explore design choices in Coq*
    - *See what will be needed to model parsing problems*
  - FY2011: Design spec language for “correct” cases
    - Auto-extract testing program from the spec
    - Build a test-bed system for black-box testing
  - FY2012: testing “incorrect” cases
    - Add fuzzing hints to the language
    - Integrates with VM roll-back control