OOPSLA ’20 Artifacts #11 Overview

Statically Verified Refinements for Multiparty Protocols

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova and Nobuko Yoshida

Our paper presents **Session***, a toolchain for specifying message passing protocols using **Refined Multiparty Session Types** and safely implementing the distributed endpoint programs in **F***.

This artifact submission contains the following:

- An overview **TODO** of the artifact (this document).
- The main artifact, a Docker image **TODO**.
- The md5 hash of the artifact file is **TODO**.

For better usability, please use the online **TODO** version of this document.

This overview describes the steps to assess the practical claims of the paper using the artifact.

1 Getting Started

1.1 Run the Artifact (Docker Image)

For the OOPSLA’20 artifact evaluation, please use the docker image provided:

TODO: Replace with how to unzip the docker image and run.

0. Install docker.
1. Download the artifact file (assume the filename is `artifact.tar.gz`)
2. Unzip the artifact file. `bash  gunzip artifact.tar.gz`
3. You should see the tar file `artifact.tar` after last operation.
4. Load the docker image `bash  docker load < artifact.tar`
5. You should see in the end of the output after last operation: **TODO: Change the tag**
   `Loaded image: docker.pkg.github.com/sessionstar/oopsla20-artifact/artifact:latest`

6. Run the docker container:
   `bash docker run -it -p 3000:3000 docker.pkg.github.com/sessionstar/oopsla20-artifact/artifact:latest`
7. The Docker image comes with an installation of vim and nano for editing. If you wish to install additional software for editing or other purposes, you may obtain sudo access with the password `sessionstar`.
8. The instructions in this overview assume you are in the `/home/sessionstar/examples` directory.

1.2 Artifact Layout

The artifact is built from this commit **TODO** in the sessionstar GitHub repository.

The artifact contains the following: * The directories `scribble-java` and `ScribbleCodeGenOCaml` comprise the full source code of the toolchain. * The `sessionstar` command, available on the command line `$PATH`, performs the Scribble protocol to F* API generation (e.g. the F *callback signatures)* The directory `FStar` contains a checkout of the F* compiler, patched to enable `TCP_NODELAY` flag for benchmarking purposes. * The directory `examples` contains the source code for the various examples, including the HigherLower running example from the paper (Section 2) and those listed in Table 2 in the paper. * The sub-directory `scripts` contains scripts for executing the benchmarks from Table 1 and Table 2 in the paper. * The directory `template` contains template files to help you through writing and testing your own examples.

1.3 Quick Test

We provide several scripts that allow you to quickly run the main examples of the paper.

A step by step explanation on how to verify the claims of the paper, how to use the toolchain, and how to test each example separately is deferred to later sections (§2 and §3) of this document.

1.3.1 Test that at all examples can be executed

To verify and execute all implemented examples:

```
cd examples
make
make run
```

§ 2.5 explains how to run each example separately.

1.3.2 Run the benchmarks for Table 1 (Sections 5.2 and 5.3 in the paper).

To execute the benchmark experiment once:

```
python3 scripts/pingpong.py
```

The produced table corresponds (up to column renaming) to Table 1 from the paper.

(**TODO:** explain the script arguments: 1. explain the option to adjust n, 2. option to adjust how many times the example are run; 3.explain the option to run remotely)

§ 2.1 explains in details how to compare the produced results with the paper.

1.3.3 Run the benchmarks for Table 2 (Section 5.4 in the paper).

To compile all applications implemented with **Session*** (Table 2):

```
python3 scripts/examples.py
```

The produced table corresponds (up to column renaming) to Table 2 from the paper.

§ 2.2 explains in details how to compare the produced results with the paper.

2 Step-by-Step Instructions

The purpose of this section is to describe in details the steps required to assess the artifact associated with our paper. We would like you to be able to:

- reproduce our benchmarks from Table 1, Section 5.2 and 5.3. For that purpose, complete §2.1 of this document.
- compile the examples, reported in Table 2, Section 5.4. For that purpose, complete §2.2 of this document.
- test the running example (HigherLower) from the paper, described in Section 2. For that purpose, complete §2.3 of this document.

Additionally, you can test and modify any of the examples we have implemented (§2.4), as well as implement and verify your own protocols using our toolchain.

Note on performance: Measurements in the paper are taken using a machine with Intel i7-7700K CPU (4.20 GHz, 922 4 cores, 8 threads), 16 GiB RAM, operating system Ubuntu 18.04. Depending on your test machine, the absolute values of the measurements produced in §2.1 and §2.2 may differ slightly from the paper. Nevertheless, the claims stated in the paper should be preserved.

2.1 Run and verify the benchmarks for Table 1 (Sections 5.2 and 5.3).

The purpose of this set of benchmarks is to demonstrate the scalabilty of our tool on protocols of increasing length (as explained in Section 5.2). We also measure the execution overhead of our implementation by comparing it against an implementation without session types or refinement types, which we call bare implementation (as explained in Section 5.3).

To reproduce the benchmarks reported in the paper run the script with an argument of 30 (**TODO: (verify the argument and what it means**). Note that the script will take a considerable time to complete **TODO: (how much approx: XXX)**:

```
python3 scripts/pingpong.py 30
```

Compare the results with the results reported in Table 1, taking into account that the absolute values may differ. Verify the associated claim (Section 5.3, line 971-972):

Despite the different protocol lengths, there are no significant changes in execution time

The produced table contains the following columns. In brackets we give the name of the corresponding columns from Table 1. * `Gen Time (CFSM)` - the time taken for Scribble to generate the CFSM (`CFSM`) * `Gen Time (F*)` - the time taken for the code generation tool to convert the CFSM to F* (`F* APIs`) * `TC Time (Gen.)` - the time taken for the generated APIs to type-check in F* (`Gen. Code`) * `TC Time (Impl)` - the time taken to time check the implementation (`Callbacks`)

The script runs the example 30 times and displays the average.

(TODO: 1. explain the option to adjust n, 2. option to adjust how many times the example are run; 3.explain the option to run remotely)

Note: The result in the paper run the experiments under a network of latency of 0.340ms (64 bytes ping), while the script runs the examples in the same docker container.

2.3 Run the main example (HigherLower) of the paper (Section 2).

The purpose of this section is to give you a quick walk through of using the toolchains to implement and verify a protocol. We focus on the running example - HigherLower.scr. For high-level overview of the toolchain refer to §A.1

:one: **Generate.** The first step of our toolchain is the generation of callback signatures from Scribble protocols. The `sessionstar` command takes a file name, a protocol name and a role. To generate the callback file for role A for the HigherLower protocol, i.e `HigherLower/HigherLower.scr`:

`bash  sessionstar HigherLower/HigherLower.scr HigherLower A` The `sessionstar` command (1) generates a CFSM and (2) produces the callback signatures in F*. It produces the corresponding files: - `HigherLower_A.fsm` - contains the CFSM for role A - `GeneratedHigherLowerA.fst` - contains the generated API, as callback signatures, for role A.

:two: **Implement and compile.** A user has to implement the program logic for each callback from the generated API file (`GeneratedHigherLowerA.fst`).

After we implement the program logic for role A using the callback signatures produced in the previous step, we can verify that the implementation is correct by running the F* type checker.

A sample implementation of role A is given in `HigherLower/A/HigherLowerA_CallbackImpl.fst`. To compile this implementation for endpoint A, we first move the generated file to the correct folder, and then we build the endpoint using the F* compiler:

```
mv GeneratedHigherLowerA.fst HigherLower/A
make -C HigherLower/A main.ocaml.exe
```

The above command generates the binary for role A, `main.ocaml.exe`.

:three: **Execute.** Repeat the above steps (generation and compilation for role B and C). After all endpoints have been implemented and their binaries have been generated, we can run them. To run all endpoins issue all the commands:

```
HigherLower/B/main.ocaml.exe &
HigherLower/C/main.ocaml.exe &
HigherLower/A/main.ocaml.exe &
```

The above command runs the three endpoins, i.e A, B and C.

2.4 Observe Refinement Violations

Next we highlight how protocol violations are ruled out by static refinement typing, which is ultimately the practical purpose of **Session***.

a. **Refinement violations:** Change the implementation for role B. Below we suggest two modifications to the HigherLower/B/HigherLowerB_CallbackImpl.fst file.

First, ensure that the current implementation for B is correct :

```
sessionstar HigherLower/HigherLower.scr HigherLower B
mv GeneratedHigherLowerB.fst HigherLower/B
make -C HigherLower/B main.ocaml.exe
```

After each modification, compile and observe that an error is reported. Note that since we are not changing the protocol, you do not need to run sessionstar again, it is enough to run the F* type checker using `make -C HigherLower/B main.ocaml.exe`

Suggested modifications: - Option 1: Modify the condition for the lose case (Line 32) from `t=1` to `t=0` - Option 2: Comment the lose case (Line 32-33). Note: the syntax for comments in F* is (* commented code *).

b. **Use of proof-irrelevant variables:** To demonstrate how our toolchain uses reasoning with latent information, we will modify the protocol HigherLower, and we will compile the implementation for role C.

First, we verify that the implementation of C is correct:

```
sessionstar HigherLower/HigherLower.scr HigherLower C
mv GeneratedHigherLowerC.fst HigherLower/C
make -C HigherLower/C main.ocaml.exe
```

Suggested modifications: - Option 1: Modify the implementation (HigherLower/C /HigherLowerC_CallbackImpl.fst file) such that the higher case sends a variable that is lower than the current

one. For example change Line 34 from `next := (Mkstate72?.x st) + 1)` to `next := (Mkstate72?.x st) - 1)`. Compile the endpoint (`make -C HigherLower/C main.ocaml.exe`) to observer an error.

- Option 2: Modify the protocol (HigherLower.scr) by removing all constraints for x that depend on n.
  - Change Line 19 from `@'n>x && t>1'` to `@'t>1`, and
  - Change Line 23 by commenting `n=x'` (comment in Scribble is `//`), and
  - Change Line 31 from @‘((n<x || n>x) && t=1)’ to `@'t=1'`

Since we changed the protocol, new callback signatures have to be generated. Generate new callback signatures and compile:

```
bash  sessionstar HigherLower/HigherLower.scr HigherLower C  mv GeneratedHigherLowerC.fst HigherLower/C  make -C HigherLower/C main.ocaml.exe
```

❗__Note__ on syntax discrepancies:

There are small syntax discrepancies between Scribble syntax and the paper. For details, see §[A.1.1]((#discrepancy) and §[A.1.2]((#syntax).

2.5 Run Through Other Examples (Optional)

To build a selected example from Table 2:

```
make build-[name of the example]
```

To run a selected example from Table 2: You can run them using:

```
make run-[name of the example]
```

See the Makefile for more details.

Each examples is in a separate folder. The folder contains: - The protocol, specified in Scribble - a file with extension `.scr` - a folder for each role in a protocol. Each role folder contains: - Generated API file (automaticaly generated by the toolchain) - the name convention of such files is `Generated[ProtocolName][RoleName].fst`. Note that this file will be generated only after running the `sessionstar` command on the target protocol. - Callback Implementation file (specific to the implementation, should be implemented by the user) - the name convention of such files is `[ProtocolName][RoleName]_CallbackImpl.fst` - Boilerplate files (non-specific to the implementation): - `Payload.fst` - specifies serialisation of the payload types (e.g. int, strings) - `Network.fst` - standard communication functions for send/receive

Below we briefly explain each example:

- Two Buyer
  - source folder: examples/TwoBuyer
  - explanation:Two Buyer is a canonical example for demonstrating business logic interactions. It specifies a negotiation between two buyers and a seller to purchase a book. The Seller S sends the price of the book to Buyer A and Buyer B. The refinement ensures that the seller quotes the same price to both buyers. A and B negotiate and buyer B accepts to buy the book only if A contributes more to the purchase.
- Negotiation
  - source folder: examples/Negotiation
  - explanation: This is a recursive protocol that describes a service agreement proposal between a producer P and a consumer C. The protocol starts by the producer P sending an initial proposal to C, the proposal contains the price of the service. Then C can either accept the proposal, or can send a counter proposal. The refinements ensure that when an offer is accepted the confirmed price and the offer price are the same.
- Fibonacci
  - source folder: examples/Fibonacci
  - explanation: The protocol specify a computation of a fibonacci sequence. The specified refinements

ensure that each number (produced by role B) is the sum of two preceding numbers (provided by role A). Hence, the implementation is guaranteed to compute a fibonacci sequence.

- Travel Agency
  - source folder: examples/TravelAgency
  - explanation: This is a W3C Choreographies use case, and the running example from Hu et al. 2008. The protocol depicts the interactions between a client (C), the travel agency (A) and a travel service (S). Customer requests and receives by the Agency the price for a desired journey. This exchange may be repeated an arbitrary number of times for different journeys under the initiative of Customer. Customer either accepts an offer from Agency or decides that none of the received quotes are satisfactory. If the offer is accepted, the Service handles the payment.
- Calculator
  - source folder: examples/Calculator
  - explanation: a distributed service for addition of two numbers. The recursive protocol allows a client to repeatedly send an operation request (e.g addition) with two numbers, and receive back the result (the sum of the two numbers).
- SH
  - source folder: examples/SH
  - explanation: SH is short for Sutherland-Hodgman algorithm. It is a 3-role protocol for polygon clipping. It takes a plane, and the vertices of a polygon as a series of points; and produces vertices for the polygon restricted to one side of the plane. This is the running example from Neykova et al. 2018
- OnlineWallet
  - source folder: examples/OnlineWallet
  - explanation: This is the running example from Neykova et al. 2013. It represents an online payment application between client C, bank S, and an Authentication service A. In each iteration, S sends C the current account status, and C has the choice to make a payment (but only for an amount that would not overdraw the account) or end the session.
- Ticket
  - source folder:examples/Ticket
  - explanation: This is the running example from Bocchi et al. 2013, where a buyer negotiates with the seller and bank for buying a purchase. The refinements ensure that the buyer has to increase the price during negotiations until an agreement is reached. In addition, the value of the (last) offer and the payment must be equal.
- HTTP
  - source folder: examples/HTTP
  - explanation: It is a minimal specification of the Hypertext Transfer protocol protocol. The refinements ensure the validity of the status code that are used. We have implemented an HTTP server in F*. The example can interoperate with HTTP clients, e.g Chrome, Firefox, etc.

Appendix (Additional Information)

A.1 Toolchain Overview

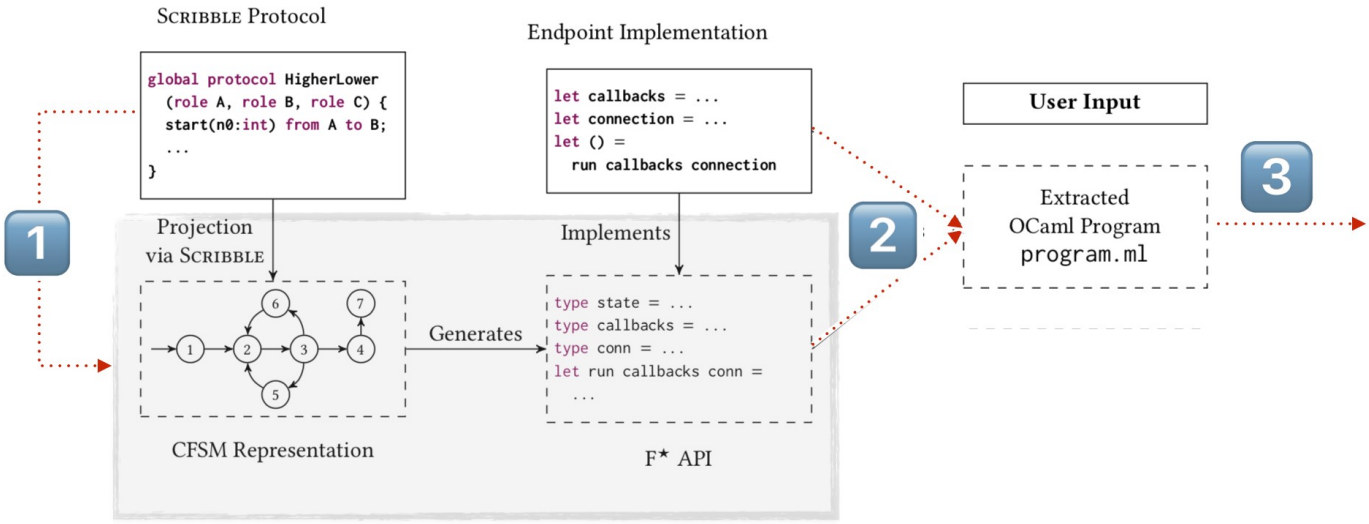
The following is a quick recap of the **Session*** toolchain, as presented in the paper.

In a nutshell. The toolchain allows users to specify, implement and verify *refined* multiparty protocols and programs. Protocol specifications are based on our Refined Multiparty Session Types (RMPST), which express data dependent protocols via refinement types on interactions and message payloads, i.e.,

protocols with logical constraints on message value and control-flow.

How. Users write protocol specifications in our extension of the Scribble protocol description language, and implement the endpoint programs in F* using refinement-typed APIs generated from the protocol by the toolchain. The F* compiler statically verifies each endpoint program and its refinements to ensure that the program follows the protocol.

The steps of the toolchain, as exercised in 2.3, are outlined below (the figure corresponds to Fig. 2 in the paper).



The **Session*** starts by writing the protocol in our extended Scribble: this artifact supplies the protocols for all the examples in the paper. This overview then runs through the following steps.

1. **Generate** – for each role in a given Scribble protocol, we generate an F* API for implementing that role (by way of a CSFM representation). This is done using the `sessionstar` command, which is available on the command line path in the artifact conatiner and produces the following files. Outputs: (a dot file representation of the CFSM) and an F* API file.
2. **Implement and compile** – the user supplies the application logic for each endpoint by implementing the I/O callback function types of the generated API. The implementation is verified by the F* compiler. Outputs: executable binaries.
3. **Execute** – with a well-typed endpoint program for each role, we can execute the protocol. For this artifact, we run all endpoints within the same container as separate processes communicating asynchronously via TCP localhost (i.e., with the same communication semantics as geographically distributed TCP connections). Outputs: safe execution of the refined multiparty protocol.

A.1.1 Discrepancies between the Paper and the Artifact

There are a few discrepancies between the implementation of our extended Scribble in the artifact and that presented in the paper.

- There are small syntactic differences.
  - Refined state variable declarations were written in the paper: e.g., Fig.
    `aux global protocol Aux(role A, role B, role C) @'B[n: int{0<=n<100}, t: int{0<t}]`
    whereas the implementation in the artifact requires syntax like:
    `aux global protocol Aux(role A, role B, role C) @'B[n: int = 0, t: int = 1] (0<=n && n<100) && 0<t`
    The syntax in our implementation declares state variables with a default initial expression (which happen to be irrelevant to the HigherLower example), and the refinements of each variable are written as a combined assertion following the declarations.
  - The code in the paper uses some compacted notation (e.g., ≤, ∧) which the implementation requires in longer form (e.g., `<=`, `&&`).
- Our implementation of Scribble includes an additional protocol validation step prior to the F* API

generation. This validation is an optional bonus, and the toolchain as presented in the paper does not depend on it.

A.1.2 Syntax of Refined Scribble

Our extended Scribble is based on the global types of our Refined MPST as defined in the paper (Section 4). The syntax and key features are already mostly demonstrated by the HigherLower example (2.3). The following summarises the syntax using another compact example.

```
module Foo;  // Corresponds to the file name, i.e., Foo.scr

type <fstar> "..." from "..." as int;  // The "..." are currently irrelevant

// A "main" protocol
global protocol MyProto(role A, role B, role C) {
    // Interaction sequences -- with payload variables and refinements
    One(x1: int) from A to B;        @'x1>0'
    Two(x2: int) from A to C;        @'x2=x1'
    // A subprotocol -- essentially inlined
    do MyProtoAux(A, B, C);        @'B[x1] C[x2]'  // State variable arguments
}

// "aux" protocols are used as subprotocols from main protocols
aux global protocol MyProtoAux(role A, role B, role C)
                                  // State variable declarations and assertion
                                  @'B[xB: int = 0] C[xC: int = 0] xB>=0' {
    // "Directed" choice -- refinements specify control flow as well as message values
    choice at B {
        Two(curr: int) from B to C;    @'xB>0 && curr=xB'
        Three(orig: int) from C to A;  @'orig=xC'
        // A (tail) recursive subprotocol -- translated: \mu X ... X
        do MyProtoAux(A, B, C);    @'B[xB-1] C[xC]'
    } or {
        Bye() from B to C;         @'xB=0'
        Bye() from C to A;
        // End of protocol
    }
}
```

- Usability warning: Some user errors are reported with a full stack trace – you may find a basic error message at the top of the trace.
- To match our RMPST in the paper, choices must be **directed**: this means the initial messages inside each choice case must be sent from the choice subject (the `at` role) to the *same* role in all cases.
  - Our implementation expands slightly on the core theory presented in the paper by allowing a so-called merge of "third-party" branch cases with *distinct* labels in projection. E.g., for the third-party `C`
    `choice at A {  One() from A to B; Two() from A to C;  } or {  Three() from A to B; Four() from A to C;  }`
    This is a common extension in the literature, e.g., Yoshida et al.
- Regarding refinements, our current implementation supports `int` variables only, with operations for comparisons (`<`, `<=`, `=`, `>=`, `>`), addition (`+`) and connectives (`&&`, `||`). We can readily extend with additional sorts and functions based on what Z3 supports.

A.2 Implementing your own protocols

The template folder contains an initial setup for implementing your own programs. It contains some boilerplate files: * Makefile –to compile the program * payload.fst and network.fst – for basic communication operations.

Below, we outline the main steps required to implement the client for a simple calculator protocol that

supports addition: 1. Copy the template directory and rename it to mycalc 2. Inside the new directory create a Scribble protocol (you can copy the Calculator.scr protocol from the examples/Calculator) 3. Generate an F* API using `sessionstar`. We assume you are in the examples directory (the parent directory of mycalc)

```
sessionstar mycalc Calculator C
```

4. Implement the logic for all callbacks, generated in the previous step

- Create a new file CalcC_CallbackImpl.fst
- Import the generated F* file (GeneratedCalculatorC.fst), and other relevant libraries you may need
- Implement the business logic for all callbacks. The easiest way is to copy all F* callback signatures from GeneratedCalculatorC.fst, and implement them. We have given you a head start with the skeleton below.

```
module CalcC_CallbackImpl

open GeneratedCalcC
open FStar.All

// these are some local variables that you can use
let x : ref int = alloc 0
let y : ref int = alloc 1

(* implement the callback record *)
let callbacks : callbacksC = {

 (*state20Onsend : (st: state20) -> ML (state20Choice st);*)
 state20Onsend = (fun _ ->
    (* this is a choice state so you need to return either Choice20Sum1 or Choice20Quit
      ...maybe you need an if statement ... *)
  );

  (*state22OnsendSum2 : (st: state22) -> ML (int);*)
  state22OnsendSum2 = (fun _ r1 ->
    ...you need to return a number, it can be any number
  );

  (*state23OnreceiveResult : (st: state23) -> (r1: int{((r1) = ((Mkstate23?.x1 st) + (Mkstate23?.y1 st)))})
        -> ML (unit);*)
 state23OnreceiveResult = (fun _ r1 ->
   the value that is receivbed is r1, here you can decide what to do with it,
   ...you can either return unit (), you can print the receive result,
   or you can save it.
  );

  (*state24OnreceiveTerminate : (st: state24) -> (_dummy: unit) -> ML (unit);*)
  state24OnreceiveTerminate = (fun _ _ -> ()) // this is the terminating state
}
```

Hint: the Calculator folder contains the full implementation, and you can use it for reference. 5. Check that your implementation is correct. Below command assumes that you are in the examples folder (the parent directory of the mycalc folder)

```
make -C mycalc main.ocaml.exe
```

A.3 Debugging tips

- If you have problems compiling the examples, try:

```
rm .depend;
```

```
make clean-[example-name]
```

- A socket error `ECONNREFUSED`:
  - the error indicates that you have not started the roles in the expected order, you should always start the “server” role (the role that listens for connections) first.