Refinements for Multiparty Message-Passing Protocols: Specification-Agnostic Theory and Implementation (Artifact)

Martin Vassor
University of Oxford, UK

Nobuko Yoshida
University of Oxford, UK

Abstract

Multiparty message-passing protocols are notoriously difficult to design, due to interaction mismatches that lead to errors such as deadlocks. Existing protocol specification formats have been developed to prevent such errors (e.g. multiparty session types (MPST)). In order to further constrain protocols, specifications can be extended with refinements, i.e. logical predicates to control the behaviour of the protocol based on previous values exchanged. Unfortunately, existing refinement theories and implementations are tightly coupled with specification formats.

This artifact accompanies [1]. It presents an implementation of the framework presented in this paper.

2012 ACM Subject Classification Software and its engineering → Specification languages; Theory of computation → Assertions; Theory of computation → Concurrency

Keywords and phrases Message-Passing Concurrency, Session Types, Specification

Digital Object Identifier 10.4230/DARTS.10.2.23

Funding Work supported by: EPSRC EP/T00006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462, EP/X015955/1n NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank Burak Ekici, Marco Giunti, Ping Hou, Amrita Suresh, and Fangyi Zhou for their insightful suggestions

Related Article Martin Vassor and Nobuko Yoshida, "Refinements for Multiparty Message-Passing Protocols: Specification-Agnostic Theory and Implementation", in 38th European Conference on Object-Oriented Programming (ECOOP 2024), LIPIcs, Vol. 313, pp. 41:1–41:29, 2024.
`https://doi.org/10.4230/LIPIcs.ECOOP.2024.41`

Related Conference 38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2024 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

The artifact allows to run the set of examples presented in Section 7.2 if the related article. It also contains scripts to run the evaluation benchmarks presented in the said section.

2 Content

The artifact package includes a virtual machine containing:

- the source code of Rumpsteak extended with refinements
- the source code of the three additional programs written for the verification of the conditions for decentralised refinement assertion (`scr2dot`, `mpst_unroll` and `dynamic_verify`).

- benchmark scripts
- a copy of the related article
- a README file explaining how-to use the artifact
- the source of 3rd-party programs needed to compile our tools.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://zenodo.org/records/11173622`. The latest update of the artifact (fixing bugs), is available at `https://zenodo.org/doi/10.5281/zenodo.10535050`.

The md5sum and the size (see below) refer to version 1.0.5, the latest at time of publishing.

4 Tested platforms

The provided virtual machine was run with VirtualBox 6.1.50 on a Ubuntu 22.04 machine (Intel i7-6700 @ 3.40GHz × 8 with 16 GiB of memory).

5 License

The artifact is available on CC-BY-SA-NC license. The CC-BY-SA-NC license only regards the content of the folder "refinement-automata-benchmarks" in the virtual machine.

This virtual machine contains software licensed under the MIT license (rumpsteak), GPL3 (nuscr), and others (c.f. ubuntu operating system). Please refer to each software component to learn about their respective license.

6 MD5 sum of the artifact

8ef53937b5b645043e52feebff85092a

7 Size of the artifact

12.6 GiB

A Running the benchmarks

A.1 Table 1b

- Open a terminal an go to "~/Desktop/refinement-automata-benchmarks"
- run the script `dynamic_verify.sh`, which will sample a set of micro benchmarks
- for each micro benchmark, the script outputs something like the following, which corresponds to the values reported in Table 1b, for the said benchmark:

A.2 Table 1c

- Open a terminal an go to "~/Desktop/refinement-automata-benchmarks"
- run the script `compare.sh`, which will same a set of micro benchmarks
- for each micro benchmark, the script outputs something like the following, which corresponds to the values reported in Table 1c, for the said benchmark:

```
********* Analysis NAME_OF_BENCHMARK **********
***** 1st quartile, median, 3rd quartile (vanilla) *******
[3 lines with the values]
***** 1st quartile, median, 3rd quartile (refinements) *******
[3 lines with the values]
***** 1st quartile, median, 3rd quartile (difference) *******
[3 lines with the values, showing the empirical overhead]
****** Mann-Whitney test (null-hypothesis: refinement is greater than vanilla) ******
MannwhitneyuResult(statistic=[...], pvalue=[The p-value])
```

B Adding a new example

To implement a new example, the easiest way is to add the example in the Rumpsteak folder (it is also possible to create a standalone project and include Rumpsteak as a dependency).

The example is to be added in the `examples/Running\ examples` subfolder:

```
$ cd ~/Desktop/rumpsteak-refined_mpst
$ cd examples/Running\ examples/
$ mkdir my_project
$ cd my_project
```

The first part to create a new example is to write a Scribble file, representing the protocol. For instance, let's Copy/Paste the ping-pong example, and add refinements:

```
(*# RefinementTypes #*)

global protocol PingPong(role A, role B)
{
    rec t {
        Ping(x: int {x > 0}) from A to B;
        Pong(x: int {x > 0}) from B to A;
        continue t;
    }
}
```

Following Figure 7, we first want to verify the localisation of the variables. First, we unpack and build the tools (`scr2dot`, `mpst_unroll` and `dynamic_verify`), and their dependencies:

```
$ cd /tmp
$ unzip ~/Desktop/scr2dot-main.zip
$ unzip ~/Desktop/mpst_unroll-main.zip
$ unzip ~/Desktop/dynamic_verify-main.zip
$ cd scr2dot-main
$ dune build
$ cd ..
$ unzip ~/Desktop/dot-parser-v0.1.zip # dependency for mpst_unroll
$ mv dot-parser-v0.1 dot-parser
$ cd mpst_unroll-main
$ cargo build
$ cd ../dynamic_verify-main
$ cargo build
```

We can finally check the validity of our example:

```
$ cd ~/Desktop/rumpsteak-refined_mpst/examples/Running\ examples/my_project
$ /tmp/scr2dot-main/_build/default/scr2dot.exe my_project.nuscr \
| /tmp/mpst_unroll-main/target/debug/mpst_unroll \
| /tmp/dynamic_verify-main/target/debug/parser
Refinements can be dynamically checked.
```

(Of course, if you want to see the intermediate steps, you can run the 3 commands one after each other.)

Now that we now the protocol can be dynamically checked, we can implement it. First, we need to obtain the local types of the participants, obtained from `nuscr`. We also do a bit of renaming (naming the generated automata with the name of the participant, and using rust types for integers):

```
$ nuscr --fsm A@PingPong my_project.nuscr | sed "s/digraph G/digraph A/" \
| sed s/int/i32/ > A.dot
$ nuscr --fsm B@PingPong my_project.nuscr | sed "s/digraph G/digraph B/" \
| sed s/int/i32/ > B.dot
```

We now generate the Rust API. We first need to build the generator:

```
$ cd ~/Desktop/rumpsteak-refined_mpst/generate
$ cargo build
$ cd ../examples/Running\ examples/my_project
$ ../../../target/debug/rumpsteak-generate --name PingPong \
A.dot B.dot > my_project.rs
```

We finally need to implement the protocol, by filling the file `my_project.rs`.

For the sake of this example, let's countdown from 10 to 0. This will eventually violate the refinement and halt the process.

```
async fn a(role: &mut A) -> Result<(), Box<dyn Error>> {
    try_session(role, HashMap::new(), |s: PingPongA<'_, _>| async {
        let mut x = 10;
        let mut s = s;
        loop {
            let cont_rec = s.0.send(Ping(x)).await?;
            let (Pong(y), cont) = cont_rec.receive().await?;
            s = cont;
            x = y-1;
        println!("Role A received {}", y);
        }
    })
    .await
}

async fn b(role: &mut B) -> Result<(), Box<dyn Error>> {
    try_session(role, HashMap::new(), |s: PingPongB<'_, _>| async {
        let mut s = s;
```

```
        loop {
            let (Ping(x), cont_snd) = s.0.receive().await?;
            println!("Role B received {}", x);
            s = cont_snd.send(Pong(x-1)).await?;
        }
    })
    .await
}

fn main() {
    let mut roles = Roles::default();
    executor::block_on(async {
        try_join!(a(&mut roles.a), b(&mut roles.b)).unwrap();
    });
}
```

We can now run the example, which, as expected, halts when the refinement is violated.

```
$ cp my_project.rs ~/Desktop/rumpsteak-refined_mpst/examples/
$ cd ~/Desktop/rumpsteak-refined_mpst
$ cargo run --example my_project
Role B received 10
Role A received 9
Role B received 8
Role A received 7
Role B received 6
Role A received 5
Role B received 4
Role A received 3
Role B received 2
Role A received 1
thread 'main' panicked at /home/rmpst/Desktop/rumpsteak-refined_mpst/src/lib.rs:
192:14:
called `Result::unwrap()` on an `Err` value: ()
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

References

1 Martin Vassor and Nobuko Yoshida. Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation. In *38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria*, pages 41:1–41:29, 2024. doi:10.4230/LIPIcs.ECOOP.2024.41.