Towards Type-Level Model Checking for Distributed Protocols

Xin Zhao and Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden
{xizhao,phaller}@kth.se

Abstract

Developing correct distributed systems is notoriously difficult. Message-passing is a popular abstraction to simplify their implementation, supported by a number of projects including Akka and Orleans. However, the verification of these message-passing programs is still a challenge. There are recent works on type systems for verifying the behavior of message-passing applications but focused on the process/actor level, keeping system components such as transaction protocols unverified.

In this paper, we present preliminary work on ProtoMC, a type system that supports model checking for the correctness of distributed protocols. Our goal for ProtoMC is to compose processes verified using type-level model checking into protocols such that a well-typed program can be directly used as a verified implementation for a specific protocol.

1 Motivation

Model-checking is widely used in many distributed protocols such as 2PC, Paxos, and Raft. However, there might always exist a mismatch between the actual implementation and the protocol specification that is used for verification. There are works on how to automatically transform a specification to Implementation; however, due to the different choice of languages, a comprehensive approach remains elusive. Effpi [3] points out a type-based method that can specify and verify message-passing applications and ensure the type soundness of the program.

According to our experience in developing distributed systems, a more exciting target is to verify the implementation of specific protocols and even the composition of several protocol components. We want to extend Effpi to achieve the goal. However, the verification of complete protocols and their composition is challenging for three primary reasons: (1) Effpi focuses on verifying the safety/liveness properties of message-passing programs. [5] points out that 12 out of 15 projects they studied did not entirely stick to the Actor model; thus, the reasoning about message-passing programs should integrate with other programming and concurrency paradigms. (2) Effpi uses continuations to manipulate the "future" of the message passing, it increases the difficulty for developing a Hoare-stype logic reasoning. (3) Composition of verified protocols is first studied in Disel [4] in 2018 and the technique has improvement space.

2 The ProtoMC Language

ProtoMC is a programming language based on type-level model checking for distributed systems. In addition to Effpi, we add Hoare logic to check invariants guaranteed by protocols.

Example: a client-server system We write the following Effpi-style code to set up a client-server system where the server responds to client requests.

```
case class Req(key: Int, replyTo: ActorRef[Resp])
case class Resp(key: Int, res: Int, replyTo: ActorRef[Req])

def client(req: ActorRef[Req], key: Int): Actor[Resp] = {
  send(req, Req(key, self)) >>
  read { resp: Resp =>
    println("Result is: " + resp.res)
  }
}

def server(): Actor[Req] = {
  forever {
    read { req: Req =>
      send(req.replyTo, Resp(key,f(key),self))
    }
  }
}
```

In the above code, a client sends a `Req` message containing a request `key` and waits for the response from the server, and the server returns a response with the same `key`. Effpi can check the safety and liveness properties of the client-server system, e.g., whether the server responds to the client request or not; however, it cannot check whether the client receives the result for which it previously made a request, i.e., with a matching `key`. For each command, we create a Hoare triple $\{P\}S\{Q\}$ where P is the precondition, and Q is the postcondition. The variable pool is to record a set of requests that are going to be replied to, and m represents the received message. For example in client:

```
{pool = rs ∧ key ∈ dom(f)}
send(req, Req(key, self)) >>
{pool = (key, req) ⊎ rs}

{pool = (key, req) ⊎ rs ∧ m = Resp(key,res,req)}
  read { resp: Resp =>
    println("Result is: " + resp.res)
  }
{pool = rs }
```

We check the precondition for read statements such that the program is only correct if the received message contains the same key that appears in the pool. In this way, we are able to check the weak causality of the client protocol.

In order to integrate Hoare triples into a type system, we introduce *Hoare types* which augment types with pre- and postconditions.

Core ProtoMC. We propose the syntax for ProtoMC which extends Effpi with separation logic.

t	$::=$	$t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t' \mid \mathsf{chan}() \mid p \mid \ \ldots$	terms
v	$::=$	$\lambda x.t \mid \mathbb{C} \mid \ \ldots$	values
$\mathbb{C}$	$::=$	$\mathbb{a}, \mathbb{b}, \mathbb{c} \ldots$	channel objects
p	$::=$	$\mathsf{end} \mid \mathsf{send}(t, t', t'') \mid \mathsf{recv}(t, t') \mid t \vert\vert t'$	processes
T	$::=$	$\{P\}\{Q\}\tau$	Hoare types
τ	$::=$	basic types | channel types | process types	basic Effpi types

A Hoare type $\{P\}\{Q\}\tau$ is used to type a computation with a precondition P and a postcondition Q, computing a result of type τ.

We integrate Hoare types with Effpi's type system by extending the typing judgement with an additional predicate. Sent and Received are two auxiliary functions for calculating the state

$\{P\}$ and $\{Q\}$ after sending or receiving a message.

$$\frac{\Gamma \vdash \mathsf{send}(t_1, t_2, t_3) : \tau \quad \mathsf{Sent}(t_1, t_2, t_3, pr) \sqsubseteq (P, Q)}{\Gamma; pr \vdash \mathsf{send}(t_1, t_2, t_3) : \{P\}\{Q\}\tau} \text{ (T-S{\scriptsize END})}$$

$$\frac{\Gamma \vdash \mathsf{recv}(t_1, t_2) : \tau \quad \mathsf{Received}(t_1, t_2, pr) \sqsubseteq (P, Q)}{\Gamma; pr \vdash \mathsf{recv}(t_1, t_2) : \{P\}\{Q\}\tau} \text{ (T-R{\scriptsize ECEIVE})}$$

3 Challenges

We are still working on completing the semantics of ProtoMC. Here, we list some challenges for the design and implementation:

- A well-defined type system and complete soundness proof.
- Implementation of a verification system. We are currently devising the implementation of extended Effpi.
- Case studies for popular distributed protocols such as 2PC, Paxos, and Raft. The focus would be on creating state invariants for complex protocols.
- Experiments on the performance of such verification tools. We are considering the same approach as used in previous work, in order to evaluate the overhead and efficiency using well-known protocol implementations.

4 Related Work

Although session types have been studied for many years, Effpi [3] is one of the few systems that implement session types and build on top of a solid foundation. However, as we mentioned in the motivation section, the verification for only message passing is not enough for practical programs.

Diesel [4] is a type system that first studies the composition of the verified protocols. It gives us the motivation to extend Effpi with separation logic.

Actris [2] provides a functional correctness proof for concurrent programs with a mix of message passing, it is an extension of Iris project [1] which is a higher-order concurrent separation logic framework implemented and verified in the proof assistant Coq. [6] is another work based on Iris provides the first completely formalized tool for verification of concurrent programs with continuations.

References

[1] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program*, 28, article e20, 2018.

[2] Jesper Bengtson Jonas Kastberg Hinrichsen and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), to appear.

[3] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Proc. of 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2019*, pages 502–516. ACM, 2019.

[4] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018.

[5] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In Giuseppe Castagna, editor, *Proc. of 27th European Conf. on Object-Oriented Programming, ECOOP 2013*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2013.

[6] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP):105:1–105:28, 2019.