SPY: Local Verification of Global Protocols

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu

Imperial College London, UK

Abstract. This paper presents a toolchain for designing deadlock-free multiparty global protocols, and their run-time verification through automatically generated, distributed endpoint monitors. Building on the theory of multiparty session types, our toolchain implementation validates communication safety properties on the global protocol, but enforces them via independent monitoring of each endpoint process. Each monitor can be internally embedded in or externally deployed alongside the endpoint runtime, and detects the occurrence of illegal communication actions and message types that do not conform to the protocol. The global protocol specifications can be additionally elaborated to express finer-grained and higher-level requirements, such as logical assertions on message payloads and security policies, supported by third-party plugins. Our demonstration use case is the verification of choreographic communications in a large cyberinfrastructure for oceanography [10].

1 Introduction

The application-level interactions in distributed systems and Web services often involve complex, high-level communication patterns between multiple parties. It is common for implementations of each participant to be written separately, or for a system to be constructed by composing separate services managed by different administrative domains. Implementations are also commonly based on informal protocol specifications, and thus informal verification mechanisms, and can be prone to concurrency errors such as communication mismatch (e.g. the arrival of an unexpected message or request of an unsupported service operation) and deadlock (e.g. party A waits to receive a message from B while B is waiting for a message from A). This is why the need for rigorous description and verification of protocols has been observed in many different contexts.

The Scribble language [6,12] (foundation of the JBoss Savara project [13]) is a formal protocol description language developed towards tackling this challenge. The goal of Scribble is to provide an intuitive engineering language and tools, based on the theory of multiparty session types (MPST) [7], for specifying and reasoning about message passing protocols and their implementations. As a verification technique, the previously published implementations of MPST focus on static type checking of protocol specifications against endpoint processes. Well-typed processes are guaranteed to enjoy properties such as communication-safety (all processes conform to a globally agreed communication protocol) and deadlock-freedom. Static session type checking in these mainstream languages, however, requires support in the form of the language extensions and pre-compiler processing to be tractable.

In this paper, we demonstrate a toolchain (SPY: Session Python) for runtime verification of distributed Python programs against Scribble protocol specifications. Our aim is to adapt the MPST protocol verification techniques to runtime verification in order to be directly applicable to standard mainstream languages. Due to the distributed setting, our toolchain works to enforce a *global* protocol by decomposing it into *local* specifications to be independently monitored at each endpoint. Runtime verification can also be more practical for enforcing advanced protocol features, e.g. we have extended our version of Scribble to support annotations for logical assertions, which would be more complicated to verify statically, even conservatively and with language extensions.

Given a Scribble specification of a global protocol, our toolchain validates consistency properties, such as race-free branch paths, and generates Scribble (i.e. syntactic) local protocol specifications for each participant (role) defined in the protocol. At runtime, an independent monitor (internal or external) is assigned to each Python endpoint. When a session between the endpoints is initiated, each monitor retrieves the local protocol for its endpoint, and generates the corresponding finite state machine (by an extension of the algorithm in [4]) to verify the local trace of communication actions executed during the session. The evaluation of assertions is handled through a third-party engine.

To summarise the main features and characteristics of our toolchain: (1) it is based on a specification language [6,12] with a formal semantics [3,2] (with proof of the soundness of local monitoring of global protocols), and is the first implementation of runtime verification for this theory; (2) protocol specifications can be decorated to perform third-party validation of constraints beyond the core message passing protocol; (3) monitoring is decentralised with each participant verified locally and therefore no synchronisation between monitors is needed; (4) two kinds of monitor, internal (synchronous) and external (asynchronous), are implemented; and (5) the toolchain has been integrated into an industry project [10] for the verification of RPC services and multiagent protocols [11].

The rest of the paper illustrates the key steps of the toolchain, outline its usage requirements and discusses current applications. A discussion of related work and additional examples can be found within the same volume [8]. The source code of the tools and performance benchmarks are available from the project website [9].

2 Multiparty Session Types and Runtime Verification

We illustrate our toolchain through an introductory example, an online payment application, which we call OnlineWallet (Fig. 1). The scenario involves three parties: a Client (C), a Payment Server (S) and a separate Authenticator (A). At the start of a session, C sends its login details to A, and A replies to inform C and S whether the authentication is successful or not. If so, C and

S enter a loop: 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. In the first case, C sends the payee and amount to S, and the loop is repeated. In the other case, or if the authentication failed, the session ends.

```
global protocol OnlineWallet
      (role S, role C, role A) {
  login(id:string, pw:string)
      from C to A;
  choice at A {
    login_ok() from A to C, S;
    rec LOOP {
      account(balance:int,
        overdraft:int) from S to C;
      choice at C {
        @<amount <= balance+overdraft>
        pay(payee:string, amount:int)
            from C to S;
        continue LOOP;
      } or {
        quit() from C to S; }}
  } or {
    login_fail(error:string)
        from A to C, S; }}
```

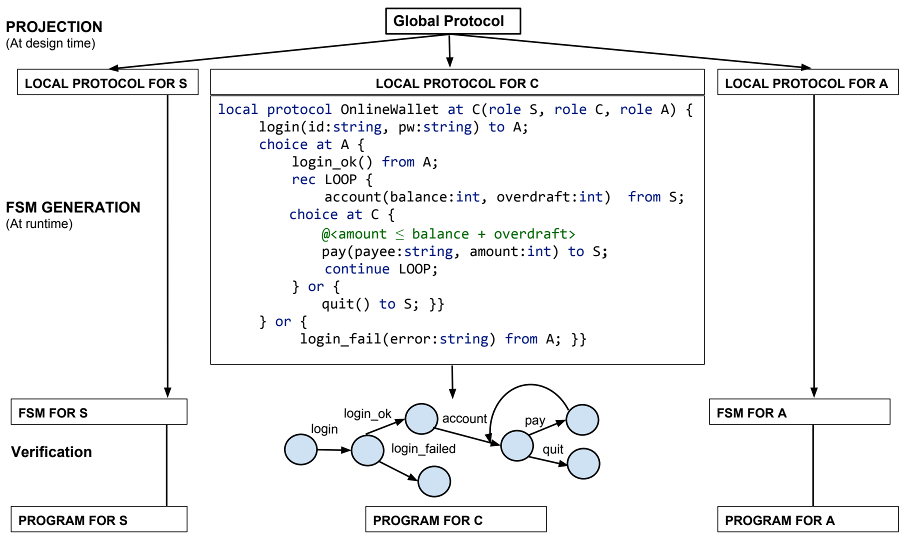
Fig. 1. OnlineWallet protocol in Scribble

Our toolchain performs the verification across several levels, as explained below.

Global protocol correctness The first level of verification is in the design of the global protocol. The Scribble in Fig. 1 describes interactions between session participants from the global perspective using message passing sequences, branching (choice) and recursion. Each message has an operator (a label) and a payload. The toolchain validates that the protocol is coherent and deadlock-free, and thus projectable [7] for each role. For example, in each case of a `choice` construct, the deciding party (e.g. `at A`) must correctly communicate the decision outcome unambiguously to all other roles involved; a `choice` is badly-formed if the actions of the deciding party would cause a race condition on the selected case between the other roles, or if it is ambiguous to another role whether the decision has already been made or is still pending. The interested reader may refer to [6,12] for a comprehensive overview of the Scribble syntax, a tutorial, and further references to the formal conditions for protocol correctness.

Local protocol conformance The second level is runtime verification to ensure that each endpoint program conforms to the core protocol structure *according to its role*. There are two main factors. First, we verify that the type (operation and payload) of each message matches its specification (operations can be mapped directly to message headers, or to method calls, class names or other relevant artefacts in the program). Second, we verify that the flow of interactions is correct, i.e. interaction sequences, branches and recursions proceed as expected, respecting the explicit dependencies (e.g. `m1() from A to B; m2() from B to C;` imposes an input-output causality at B). These measures rule out errors, e.g. communication mismatches, that violate the permitted protocol flow.

Fig. 2 outlines the concrete verification steps. First, local protocols are mechanically generated from the validated global protocol. A local protocol is essentially a view of the global protocol from the perspective of one role. The projection algorithm works by identifying the message exchanges where the participant is involved, and disregarding the rest while preserving the overall structure of the global protocol. Each local protocol has a corresponding FSM, generated by the monitor at runtime. When a party requests to start or join a session, the initial message specifies which role it intends to play. Its monitor retrieves



```
local protocol OnlineWallet at C(role S, role C, role A) {
     login(id:string, pw:string) to A;
     choice at A {
         login_ok() from A;
         rec LOOP {
             account(balance:int, overdraft:int)  from S;
         choice at C {
             @<amount ≤ balance + overdraft>
             pay(payee:string, amount:int) to S;
             continue LOOP;
         } or {
             quit() to S; }}
     } or {
          login_fail(error:string) from A; }}
```

Fig. 2. Global specification to local runtime verification methodology

the local specification based on the protocol name and the role. Fig. 2 gives the local protocol and associated FSM for the client role C (we omit the protocols for S and A). The FSM encodes the flow of local communication actions, with transitions fired by the input and output of the permissible messages.

Policy validation The final level of verification enables the elaboration of Scribble protocols using annotations (@<...> in Fig. 1 and 2). The annotations function as API hooks to the verification framework: they are not verified by the MPST monitor itself, but delegated to a third-party engine. Various policy domains (e.g. security policies) can be enforced by integrating engines for predicates on endpoint state, automata-based properties, etc., as extensions to the core protocol monitor. Our current implementation uses a Python library for evaluating basic predicates (e.g. the overdraft check in Fig. 1), which is sufficient for the application protocols we have developed with [11]. At runtime, the monitor passes the annotation information, along with the FSM state information, to the appropriate policy engine to perform the additional checks or calculations. To plug in an external validation engine, our toolchain API requires modules for parsing and evaluating the annotation expressions specified in the protocol.

3 Toolchain Requirements and Evaluation

3.1 Monitor Requirements

Positioning The network monitoring in our theory imposes *complete mediation* of communications: no communication action should have an effect unless the message is mediated by the monitor. The tool implements this principal for both inline and outline monitor configurations. Inline monitoring relies on internal message interception: the local conversation runtime, in place at each endpoint, synchronously passes every message (on arrival or prior to dispatch) through the monitor component. Outline monitoring is realised by dynamically modifying the application-level network configuration to (asynchronously) route every message through a monitor. Our prototype is built over an Advance Messaging Queue Protocol (AMQP) [1] transport, where we use the AMQP exchange-to-exchange binding functionality to perform the message rerouting. A monitor dispatcher is assigned to each network endpoint as a conversation gateway. The dispatcher can create new routes and spawn new monitor processes if needed, to ensure the scalability of this approach.

Message format To monitor Scribble conversations, our toolchain relies on a small amount of message meta data that we refer to as the Scribble header, but is actually embedded into the message payload (for more flexible interoperability). Messages are processed depending on the message type, as recorded in the header. There are two kinds of conversation messages: *initialisation* (exchanged when a session is started, carrying information such as the protocol name and the role of the monitored process) and *in-session* (carrying the message operation and the sender/receiver roles). Initialisation messages are used for routing reconfiguration, while in-session messages are checked for protocol conformance.

Conversation API Our toolchain is accompanied by a message-passing library for implementing Python endpoint applications, that augments message payloads with the conversation information required for monitoring. The library API concisely exposes the core MPST primitives [3,2] for (1) initiating and joining a conversation and (2) asynchronous message dispatch and consumption by the participants. The API can be used directly by the programmer as a standalone conversation library, or as a complementary support module by another library to handle the formatting of conversation messages for monitoring.

3.2 Evaluation

Our work is applied to and running within the Ocean Observatories Initiative (OOI) [10,11], an ongoing project to establish a cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. The OOI architecture relies on the combination of high-level protocol specifications (to express how the infrastructure services should be used) and distributed run-time monitoring to regulate the behaviour of every application within the system, for which the present toolchain is used. Performance measurements for our current implementation (the project is at release two of a planned four) show a reasonable overhead (13% percent per message call, see [9] for the full benchmarks). The overhead is mostly due to just-in-time FSM generation, which we believe can be reduced by caching or pre-generation of the FSM for each protocol. We also note that the relative overhead due to FSM generation decreases as the length of the conversation increases.

Our collaboration in the OOI project has had interesting impacts on our work and research. First, the practical requirements, emerging from their use cases, led to the several advances of the MPST theory and the Scribble language (interruptible conversations [8], generic protocols [5] and protocol annotations). Second, we found that many OOI use cases can be categorised into a small set of parameterised protocols. As an example, the majority of service-oriented protocols, with diverse message signatures, are now derived from a single parametrised RPC service protocol; rather than requiring a Scribble protocol per application instance, one parameterised protocol can be provided per application library. This is a convenient approach because we have observed that developers are (so far) often not accustomed to writing protocols explicitly and formally. Finally, the integration of our toolchain proceeded from the specification and verification of the smaller, lower-level protocols in the OOI system, such as RPC. In general, the kinds of bugs detected by our toolchain (e.g. messages to/from the wrong participant) did not frequently arise for these smaller protocols; however, this starting point enabled a straightforward, non-intrusive integration (not a single line of existing application code was changed) that eased the adoption of the tool by the developers. The next phase of the ongoing integration is to port the more complex application protocols to Scribble, given the monitoring infrastructure (independent of the protocol size) is already in place: our toolchain is able to verify any Scribble protocol using the single generic monitor implementation.

References

1. Advanced Message Queuing Protocol homepage. `http://www.amqp.org/`.
2. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65, 2013.
3. T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC'11*, volume 7173 of *LNCS*, pages 25–45, 2012.
4. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
5. K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Deniélou, and N. Yoshida. Structuring Communication with Session Types. In *COB'12*, LNCS, 2012. To appear.
6. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
7. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
8. R. Hu, R. Neykova, N. Yoshida, and R. Demangeon. Towards practical interruptible conversations. This volume.
9. Session Python (SPY) resource page. `http://www.doc.ic.ac.uk/~rn710/spy/`.
10. Ocean Observatories Initiative. `http://www.oceanobservatories.org/`.
11. Scribble-OOI collaboration. `https://confluence.oceanobservatories.org/display/CIDev/OOI+Use+Cases+in+Scribble`.
12. Scribble project home page. `http://www.scribble.org`.
13. JBoss Scribble site. `http://www.jboss.org/scribble`.