Session Types Go Dynamic or How to Verify Your Python Conversations

Rumyana Neykova
Imperial College London, United Kingdom
`rumi.neykova10@imperial.ac.uk`

This paper presents the first implementation of session types in a dynamically-typed language - Python. Communication safety of the whole system is guaranteed at runtime by monitors that check the execution traces comply with an associated protocol. Protocols are written in Scribble, a choreography description language based on multiparty session types, with addition of logic formulas for more precise behaviour properties. The presented framework overcomes the limitations of previous works on the session types where all endpoints should be statically typed so that they do not permit interoperability with untyped participants. The advantages, expressiveness and performance of dynamic protocol checking are demonstrated through use case and benchmarks.

1 Introduction

The study of multiparty session types (MPST) has explored a type theory for distributed programs which can ensure, for any typable programs, a full guarantee of deadlock-freedom and communication safety (all processes conform to a globally agreed communication protocol) through static type checking. However, a static verification is not always feasible and dynamic approaches have several advantages. First, when access to the source code is restricted dynamic verification enables to detect and ensure the correctness of external untyped components. Second, constraints on the message payload are easier to check dynamically. Third, as shown in this paper, dynamic checking is less obstructive to the source code, because it does not require extensions of the host language as in the existing works on session types.

In this paper we present a toolchain for session-based programming (hereafter conversation programming) in Python that uses MPST-protocols to dynamically verify the communication safety of the running system. Conversation programming in Python resembles the standard development methodology for MPST-based frameworks (Fig. 1). It starts by specifying the intended interactions (choreography) as a global protocol in the protocol description language Scribble [13]. Then Scribble local protocols are generated mechanically for each participant (role) defined in the protocol. After that processes for each role are implemented using MPST operations exposed by Python conversation library. An external monitor is assigned to each endpoint.
During communication initiation the monitor retrieves the local protocol for its process and converts it to a finite state machine (FSM). The FSM continuously checks at runtime that each interaction (execution trace) is correct. If all participants comply to their protocols, the whole communication is guaranteed to be safe [6]. If participants do not comply, violations (such as deadlocks and communication mismatch) are detected and optionally ignored.

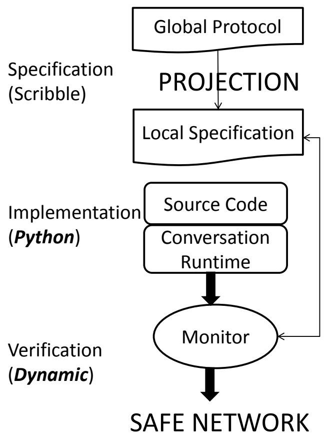

Figure 1: Development methodology

The presented framework brings several non-trivial contributions to MPST works. First, Scribble is extended with logic assertions (constraints on the message payload). Second, implementing MPST in a dynamic language requires different code augmentation techniques. For that purpose, we have defined a minimal, but sufficient and extendable format for conversation message headers. Third, we show that using FSMs for MPST checking has reasonable overhead. The algorithm used to convert local session types to FSM is based on [7], however we have optimised it to avoid the state explosion for parallel sub-protocols and have extended it for the new Scribble constructs. Finally, the Python API is more flexible compared to other session types language extensions, because it supports different programming styles (event-driven and thread-based, see Fig. 4). From the existing implementations only SJ [9] features event-driven programming, but it has more strict typing rule. To the best of our knowledge, this is the first implementation of session types for decentralised monitoring. Our practical framework is inspired by the formal model of MPST runtime safety enforcement presented in [6, 5]. In the aforementioned works conformance to stipulated global protocols is guaranteed at runtime through local monitoring.

The rest of the paper illustrates the key features of our conversation framework, the Python runtime and its API (§ 2), it also gives overview of the monitoring tool, along with its benchmarks (§ 3). § 5 discusses future work and concludes. The code for the runtime and the monitor tool and example applications are available from [14].

2 Conversation Programming in Python

This section illustrates the stages of our framework and its implementation through a use case. Step 1 and 2 illustrate the use case specification in Scribble, while Step 3 presents one of the main contributions of the paper – a python API for conversation programming. We present a use case obtained from our industrial partners Ocean Observatory Institute (OOI) [11] (use case UC.R2.13 ”Acquire Data From Instrument”). OOI aims to establish cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor. Their architecture relies on distributed run-time monitoring to regulate the behaviour of third-party applications within the system. Part of the monitor tool presented in this paper is already integrated in their system as an internal monitor.

Step 1: Global Protocol. The Scribble global protocol for the use case is listed in Fig. 2. Scribble describes interactions between session participants through message passing sequences, branches and recursion. Each message has a label (an operator) and a payload. The first line declares the Data Acquisition protocol and three participant roles – a User (U), an Agent service (A) and an Instrument (I). The overall scenario is as follows: U requests via A to start streaming a list of resources from I (line 2–3). At Line 4 I makes a choice wether to continue the interaction or not. If I supports the requested resource the communication continues and A starts to poll resources from I and streams them to U (line 6–15). Line 10 shows the new assertion construct and restricts I to send data packages that are less than 512MB. The presented assertion extension is inspired by [4]. However, we do not stick to a predefined logic, but allow various policy languages to be incorporated inside an assertion construct.

Step 2: Global-to-local Protocol Projection. Local protocols specify the communication behaviour for each conversation participant. An example of a local protocol (the local protocol for role A is given in Fig. 2. A local protocol is essentially a view of the global protocol from the perspective of one participant role and as such it is mechanically projected from the global protocol. Projection basically works by identifying the message exchanges where the participant is involved, and disregarding the rest, while preserving the overall interaction structure of the global protocol. The assertions are similarly

```
global protocol DataAquisition(role U,
  role A, role I) {
  Request(string:info) from U to A;
  Request(string:info) from A to I;
  choice at I {
  Support from I to A;
  rec Poll{
    Poll from A to I;
    choice at I {
      @{size(data) ≤ 512}
      Raw(data) from I to A ;
      Formatted(data) from I to U;
      Poll;
      } or {
        Stop from I to A;
        Stop from A to U;}}
  } or {
    NotSupported from I to A;
    Stop from A to I;
    Stop from A to U;}}
```

```
local protocol DataAquisition at A(role U,
  role A, role I) {
  Request(string:info) from U;
  Request(string:info) to I;
  choice at I {
  Support from I;
  rec Poll{
    Poll to I;
    choice at I {
      @{size(data) ≤ 512}
      Raw(data) from I;

      Poll;
      } or {
        Stop from I;
        Stop to U;}}
  } or {
    NotSupported from I;
    Stop to I;
    Stop to U;}}
```

Figure 2: Global Protocol (left) and Local Protocol for role A (right)

preserved by projection where relevant.

Step 3: Process Implementation. Fig. 4 illustrates the conversation API by presenting two alternative implementations in Python for the User process. Our Python conversation API offers a high level interface for safe conversation programming and maps basic session calculus primitives to lower-level communication actions on a concrete transport (AMQP [1] in this case). The implementation is built on top of Pika [12], a widely used AMQP client library for Python. Fig. 3 lists the basic API methods. In short, the API provides functionality for (1) session initiation and joining and (2) basic send/receive. Each message embeds in its payload a conversation header. The header contains session information either for monitor initialisation (in case of *invitation* messages), or session checking (in case of *in-session* messages).

```
# session initiation bla
create(protocol, inv_config.yml)
# accept an invitation
join(self, role, principal_name)
# send a msg
send(self, to_role, op, payload)
# receive a msg
recv(self, from_role)
# receive asynchronously
recv_async(self, from_role, callback)
# close the connection
stop()
```

Figure 3: Conversation API

Conversation initiation The `Conversation.create` method initiates a new conversation. It creates a fresh conversation id and the required AMQP objects (principal exchange and queue), and sends an invitation message for each role specified in the protocol. Invitation mechanism is needed to map the role names to concrete addressable entities on the network (principals) and to propagate this mapping to all participants. Invitation header carries a conversation id, a role, a principal name (resolvable to a network address) and a name for a Scribble local specification file. In our example, the User starts a session and sends invitation to all other participants. Once the invitations are sent and accepted, a session is established and the intended message exchange can start. An invitation for a role is accepted using the `Conversation.join` method. It establishes an AMQP connection and, if one does not exist, creates an invitation queue on which the invitee waits to receive an invitation.

Conversation message passing The API provides standard send/receive primitives. Send is asynchronous, meaning that a basic send does not block on the corresponding receive; however, the basic re-

```python
class ClientApp(BaseApp):
  def start(self):
    c = Conversation.create('DataAquisition',
         'config.yml')
    c.join('U', 'alice')

    resource_request = c.receive('U')
    c.send('I', resource_request)
    req_result = c.receive('I')

    if (req_result == SUPPORTED):
      c.send('I', 'Poll')
      op, data = c.receive('I')
      while (op != 'Stop'):
        formatted_data = format(data)
        c.send('U', fomratted_data)
      c.send('U', stop)
    else:
        c.send('U, I', stop)
        c.stop()
```

```python
class ClientApp(BaseApp):
  def start(self):
    c = Conversation.create('DataAquisition',
          'config.yml')
    c.join('U', 'alice')
    c.receive_async('U', on_request_received)

  def on_request_received(self, conv, op, msg):
    if (op == SUPPORTED):
      conv.send('I', 'Poll')
      conv.receive_async('I', 'on_data_received')
    else: conv.send('I, U', 'Stop')

  def on_data_received(self, conv, op, payload):
    if (operation != 'Stop'):
      formatted_data = format(payload)
      c.send('U', formatted_data)
    else:
        conv.send('U', 'Stop')
        conv.stop()
```

Figure 4: Python standard (left) and event-driven (right) implementation of the User process

ceive does block until the complete message has been received. An asynchronous receive (`receive_async`) is also provided to support event-driven usage of the conversation API. We have demonstrated two different implementations for the the User process (threaded and event-driven). Both versions require the same monitor for checking. The primitives for sending and receiving specify the name of the sender and receiver role respectively. The runtime resolves the role name to the actual network destination by coordinating with the in-memory conversation routing table created as a result of the conversation invitation. All messages are sent/received as a tuple of an operation and a payload. The API does not mandate how the operation field should be treated, allowing the runtime freedom to interpret the operation name various ways, e.g. as a plain message label, an RMI method name, etc. Syntactic sugar such as automatic dispatch on method calls based on the message operation is possible. More examples of programs using the API can be found in [14].

3 Dynamic Verification

3.1 Monitoring Implementation

To guarantee global safety our monitoring framework imposes *complete mediation* of communications: no communication action should have an effect unless the message is mediated by the monitor. We use the AMQP's functions to reroute each outgoing/incoming message to its associated monitor. Routing is configured during session initialisation.

Figure 5 depicts the main components and internal workflow of our prototype monitor. The lower part relates to session initiation. The invitation message carries (a reference to) the local type for the invitee and the session id (global types can be exchanged if the monitor has the facility for projection.) The monitor generates the FSM from the local type following [7]. Our implementation differs from [7] in the treatment of parallel sub-protocols (i.e. unordered message sequences). For efficiency, the monitor generates nested FSMs for each session thread, avoiding the potential state explosion that comes from constructing their product. FSM generation has therefore polynomial time and space cost in the length

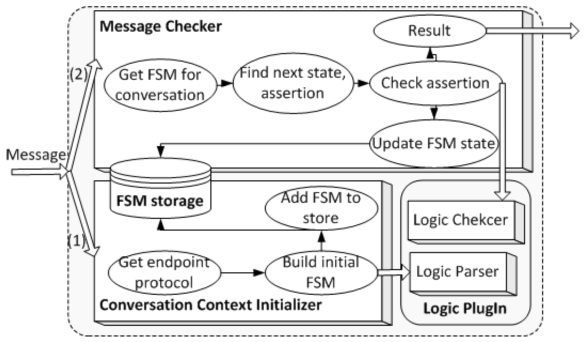

Figure 5: Monitor components and workflow. The messages are processed depending on their type: (1) Invitation Messages and (2) Conversation Messages.

of the local type. The (nested) FSM is stored in a hash table with session id as the key. Due to MPST well-formedness conditions (message label distinction), any nested FSM is uniquely identifiable from any unordered message (i.e. session FSMs are deterministic). Transition functions are similarly hashed, each entry having the shape: $(current\_state, transition) \mapsto (next\_state, assertion, var)$ where *transition* is a triple $(label, sender, receiver)$, and *var* is the variable binder for the message payload.

The upper part of the Figure relates to in-session messages, which carry the session id (matching an entry in the FSM hash table), sender and receiver fields, and the message label and payload. This information allows the monitor to retrieve the corresponding FSM (the message signature is matched to the FSM's transition function). Any associated assertions are evaluated by invoking an external logic engine; a monitor can be configured to use various logic engines, for example, logic engines that support the validation of assertions, automata-based specifications (such as security automata), or state updates. The current implementation uses a Python predicate evaluator, which is sufficient for the example protocol specifications that we have tested so far.

3.2 Benchmarks

These benchmarks measure the communication overhead introduced by our prototype monitor implementation. The results show that the core FSM-related functionality of the monitor adds little overhead in comparison to a dummy monitor that performs plain message forwarding.

Benchmark framework. We measure the time to complete a session between client and server endpoints connected to a single-broker AMQP network. Three benchmark cases are compared. The main case (Monitor) is fully monitored, i.e. FSM generation and message checking are enabled for both the client and server. The base case for comparison (Forwarder) has the client and server in the same configuration, but with dummy monitors that perform only message forwarding. For reference, the final case (No Monitor) tests direct AMQP communication between the server and client, i.e. messages are routed directly from an exchange to their destination queues (no intermediate forwarding). Naturally, forwarding-based mediation incurs additional latencies; the actual internal overhead of the monitor is given by the first two benchmark cases. This benchmark framework is applied to three scenarios:

1. Increasing *session length* (number of messages), for protocol:
$$\mu X.\mathtt{S} \to \mathtt{C}\{\mathtt{OK}().\mathtt{C} \to \mathtt{S}\{\mathtt{ACK}().X\},\mathtt{KO}().\mathtt{end}\}$$
Session length is the number of times the recursion is repeated.
2. Increasing *protocol size* (increasing number of parallel states). We repeatedly compose the base pattern to construct bigger protocols for nested FSM generation.
$$\mathtt{S} \to \mathtt{C}\{\mathtt{OK}().\mathtt{end}\} \mid \mathtt{C} \to \mathtt{S}\{\mathtt{ACK}().\mathtt{end}\}$$

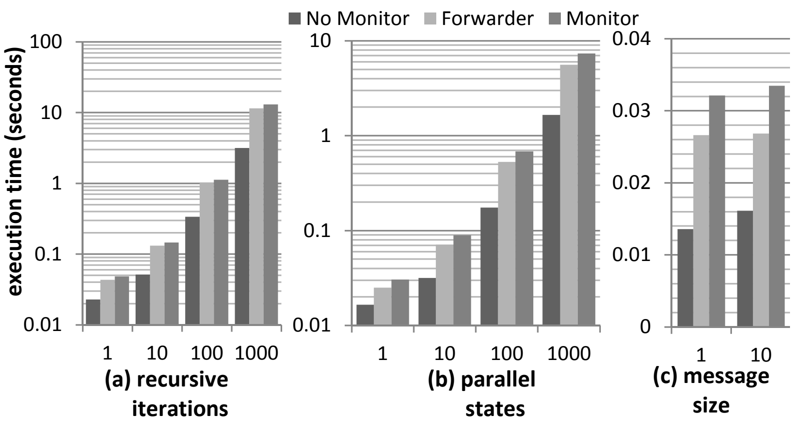

Figure 6: Microbenchmarks comparing end-to-end monitor performance

3. Increasing *payload size* (message size), using protocol from (1).

Benchmark environment and results. The server and client endpoint processes, both monitors and the RabbitMQ broker (2.7.0/R13B03) are all run on separate machines with the same specification: Intel Core2 Duo 2.80 GHz and 4 GB main memory, running 64-bit Ubuntu 11.04 (kernel 2.6.38) and connected via gigabit Ethernet. Latency between each node is measured to be 0.24 ms on average (ping 64 bytes). The benchmark applications are executed using Python 2.7.1.

Figure 6 presents the results for the three benchmark scenarios. Each chart gives the mean time (y-axis) for the client and server to complete one session after repeating the benchmark 100 times for each parameter configuration (session length/parallel states/message size). Scenario (3) message size is measured for session length 1. For all three scenarios, the results show that the overhead of the monitor due to FSM generation and FSM-based message checking, the baseline cost in the current framework, are acceptable (around 20%). Non-communication related computation in more realistic applications and higher latency environments will both contribute to decreasing the relative overhead. For scenario (1) in chart (a), note that the relative overhead decreases (from 12% to 9%) as the session length increases, because the one-time FSM generation cost becomes less prominent. Although our implementation work is ongoing, we believe these results confirm the feasibility of our approach. As expected, the forwarding configuration incurs extra latencies (due to the reciprocal shape of the benchmark protocol) in comparison to the (No Monitor) case. The full source code and raw results of these benchmarks, and additional tests using protocols with assertions, can be obtained from the project homepage [14].

4 Related Work

The work closest to ours is that by Ancona et al. [2]. It explores session types protocols as a test framework for multiagent systems (MAS). A global session type is specified as cyclic Prolog terms in Jason (a MAS development platform) and verified through test monitors. Their global types are less expressive in comparison with the language presented in this paper (due to restricted arity on forks and the lack of assertions). Their monitor is centralised and global safety properties are not discussed. Kruger et al. [10] propose a run-time monitoring framework, projecting MSCs to FSM-based distributed monitors. They use aspect-oriented programming techniques to inject monitors into the implementation of the components. Our outline monitoring verifies conversation protocols and does not require such monitoring-specific augmentation of programs. Gan [8] follows a similar but centralised approach to Kruger et al.

Works on monitoring BPEL languages can also be compared. Baresi et al. [3] develop a run-time monitoring tool with assertions. However, a major difference is that BPEL approaches do not treat or prove global safety. BPEL is expressive, but does not support distribution and is designed to work in a centralised manner.

5 Conclusion and Future Work

We have shown that session types are amendable for dynamic verification. Our implementation automates distributed monitoring by generating FSMs from local protocol projections. Further benchmarks are needed to compare the conversation API with existing network libraries and to investigate its performance. Future work includes also the incorporation of more elaborate handling of error cases into monitor functionality, extending Scribble and automatic generation of services stubs. Although our implementation work is ongoing, the results confirm the feasibility of our approach. We believe this work is an important step towards a better, safer world of easier to speak and easier to understand distributed conversations.

Acknowledgments. I would like to dedicate this paper to the memory of Kohei Honda, who is a constant source of inspiration to me and whose guidance was invaluable. I thank my supervisor Nobuko Yoshida for her constant support and ideas, my colleagues Raymond Hu and Pierre-Malo Denilou for the discussions about the framework; the anonymous reviewers for useful comments and corrections; and Tzu-Chun Chen for her valuable feedback and her inspirational work on the formal system behind the presented work. This work is partially supported by VMWare PhD studentship and EPSRC EP/G015635/1.

References

[1] *Advanced Message Queuing Protocols (AMQP) homepage*. `http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`.

[2] Davide Ancona, Sophia Drossopoulou & Viviana Mascardi (2012): *Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason*. In: *DALT'12*, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-37890-4_5`.

[3] Luciano Baresi, Carlo Ghezzi & Sam Guinea (2004): *Smart monitors for composed services*. In: *ICSOC '04*, pp. 193–202. Available at `http://doi.acm.org/10.1145/1035167.1035195`.

[4] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A theory of design-by-contract for distributed multiparty interactions*. In: *CONCUR*, *LNCS* 6269, pp. 162–176. Available at `http://dx.doi.org/10.1007/978-3-642-15375-4_12`.

[5] Tzu chun Chen (2013): *Theories for Session-based Governance for Large-scale Distributed Systems*. Ph.D. thesis, Queen Mary, University of London.

[6] Tzu-Chun Chen et al. (2012): *Asynchronous Distributed Monitoring for Multiparty Session Enforcement*. In: *TGC'11*, LNCS, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-30065-3_2`.

[7] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In: *ESOP*, LNCS, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-28869-2_10`.

[8] Yuan Gan et al. (2007): *Runtime monitoring of web service conversations*. In: *CASCON '07*, ACM, pp. 42–57. Available at `http://doi.ieeecomputersociety.org/10.1109/TSC.2009.16`.

[9] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida & Kohei Honda (2010): *Type-Safe Eventful Sessions in Java*. In: *ECOOP'10*, *LNCS* 6183, Springer-Verlag, pp. 329–353. Available at `http://dx.doi.org/10.1007/978-3-642-14107-2_16`.

[10] Ingolf H. Krüger, Michael Meisinger & Massimiliano Menarini (2010): *Interaction-based Runtime Verification for Systems of Systems Integration*. *J. Log. Comput.* 20(3), pp. 725–742. Available at `http://dx.doi.org/10.1093/logcom/exn079`.

[11] *OOI*. `http://www.oceanobservatories.org/`.

[12] *AMQP for Python (PIKA)*. `https://github.com/pika/pika`.

[13] *Scribble Project homepage*. `http://www.scribble.org`.

[14] *Full version of this paper*. `http://www.doc.ic.ac.uk/~rn710/spy`.