4

How to Verify Your Python Conversations

Rumyana Neykova and Nobuko Yoshida

Imperial College London, UK

Abstract

In large-scale distributed systems, each application is realised through interactions among distributed components. To guarantee safe communication (no deadlocks and communication mismatches) we need programming languages and tools that structure, manage, and policy-check these interactions. Multiparty session types (MSPT), a typing discipline for structured interactions between communicating processes, offers a promising approach. To date, however, session types applications have been limited to static verification, which is not always feasible and is often restrictive in terms of programming API and specifying policies. This chapter investigates the design and implementation of a runtime verification framework, ensuring conformance between programs and specifications. Specifications are written in Scribble, a protocol description language formally founded on MPST. The central idea of the approach is a runtime monitor, which takes a form of a communicating finite state machine, automatically generated from Scribble specifications, and a communication runtime stipulating a message format. We demonstrate Scribble-based runtime verification in manifold ways. First, we present a Python library, facilitated with session primitives and verification runtime. Second, we show examples from a large cyber-infrastructure project for oceanography, which uses the library as a communication medium. Third, we examine communication patterns, featuring advanced Scribble primitives for verification of exception handling behaviours.

4.1 Framework Overview

Figure 4.1 illustrates the methodology of our framework. The development of a communication-oriented application starts with the specification of the intended interactions (the choreography) as a *global protocol* using the Scribble protocol description language [9]. From a global protocol, the toolchain mechanically generates (*projects*) a Scribble *local protocol*, represented as a finite state machine (FSM), for each participant (abstracted as a *role*). As a session is conducted at run-time, the monitor at each endpoint validates the communication actions performed by the local endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the monitor's FSM. Each monitor thus works to protect (1) the endpoint from invalid actions by the network environment, and (2) the network from incorrectly implemented endpoints. Our runtime multiparty session types (MPST) [6] framework is designed in this way to ensure, using the decentralised monitoring of each local endpoint, that the session as a whole conforms to the original global protocol [1], and that unsafe actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints.

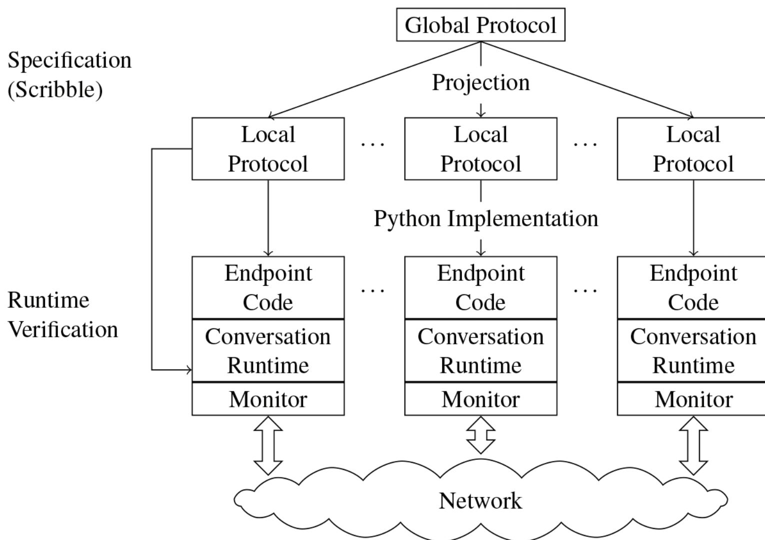


Figure 4.1 Scribble methodology from global specification to local runtime verification.

Outline. We outline the structure of this chapter. Section 4.2 demonstrates our runtime MPST framework through an example. We present a global-to-local projection of Scribble protocols, endpoint implementations, and local FSM generation. Section 4.3 demonstrates an API for conversation programming in Python that supports standard socket-like operations, as well as event-driven interface. The API decorates conversation messages with meta information required by the monitors to perform runtime verification. Section 4.4 discusses the monitor implementation, focusing on key architectural requirements of our framework. Section 4.5 presents an extension of Scribble with asynchronous session interrupts. This is a feature of MPST, giving a general mechanism for handling nested, multiparty exception handling. We present the extension of the Python API with a new construct for scopes, endpoint implementations, and local FSM generation for monitoring. Section 4.6 explains the correspondence between a theoretical model and our implementation.

4.2 Scribble-Based Runtime Verification

This section illustrates the stages of our framework and its implementation through a use case, emphasising the properties verified at each verification stage. The presented use case is obtained from our industrial partners Ocean Observatories Institute (OOI) [7] (use case UC.R2.13 ”Acquire Data From Instrument”). The OOI is a project to establish a cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. Its architecture relies on the combination of high-level protocol specifications of network services (expressed as Scribble protocols [8]) and distributed runtime monitoring to regulate the behaviour of third-party applications within the system. We have integrated our framework into the Python-based runtime platform developed by OOI [7].

4.2.1 Verification Steps

Global Protocol Correctness. The first level of verification is when designing a global protocol. A Scribble global protocol for the use case is listed in Figure 4.2 (left). Scribble describes interactions between session participants through message passing sequences, branches and recursion. Each message has a label (an operator) and a payload. The Scribble protocol in Figure 4.2 starts by protocol declaration, which specifies the name of the protocol, `Data`

```
1  global protocol DataAcquisition(
2  role U, role A, role I) {
3  Request(string:info) from U to A;
4  Request(string:info) from A to I;
5    choice at I {
6      Supported() from I to A;
7      rec Poll {
8        Poll() from A to I;
9        choice at I {
10       Raw(data) from I to A
11       @{size(data) <= 512};
12       Formatted(data) from A to U;
13       continue Poll;
14       } or {
15       Stop() from I to A;
16       Stop() from A to U;}}
17   } or {
18     NotSupported from I to A;
19     Stop() from A to I;
20     Stop from A to U;}}
```

```
local protocol DataAcquisition
  at A (role U, role A, role I)}
Request(string:info) from U;
Request(string:info) to I;
choice at I {
    Supported() from I;
    rec Poll {
      Poll() to I;
      choice at I {
      Raw(data) from I
      @{size(data) <= 512};
      Formatted(data) to U;
      continue Poll;
      } or {
      Stop() from I;
      Stop() to U;}}
  } or {
    NotSupported from I;
    Stop() to I;
    Stop to U;}}
```

Figure 4.2 Global Protocol (left) and Local Protocol (right).

`Acquisition`, and its participating roles – a User (`U`), an Agent service (`A`) and an Instrument (`I`). The overall scenario is as follows: `U` requests through `A` to start streaming a list of resources from `I` (line 3–4). At line 5 `I` makes a choice whether to continue the interaction or not. If `I` supports the requested resource, `I` sends a message `Supported` and the communication continues by `A` sending a `Poll` request to `I`. The raw resource data is sent from `I` to `A`, at `A` the data is formatted and forwarded to `U` (line 10–12). Line 11 demonstrates an *assertion* construct specifying that `I` is allowed to send data packages that are less than 512MB.

The Scribble toolchain validates that a protocol is well-formed and thus projectable for each role. For example, in each case of a choice construct, the deciding party (e.g. at `I`) 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.

Local protocol conformance. The second level of verification is performed at runtime and ensures that each endpoint program conforms to the local protocol structure. Local protocols specify the communication behaviour for each conversation participant. Local protocols are mechanically projected from a global protocol. A local protocol is essentially a view of the global protocol from the perspective of one participant role. Projection 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.

From the local protocols, an FSM is generated. At runtime, the endpoint program is validated against the FSM states. There are two main checks that are performed. First, we verify that the type (a label and payloads) of each message matches its specification (labels 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 a causality at `B`, which is obliged to receive the messages from `A` before sending a message to `C`).

Policy validation. The final level of verification enables the elaboration of Scribble protocols using annotations (@{} in Figure 4.2). Annotations function as API hooks to the verification framework: they are not verified by the MPST monitor itself, but are, instead, delegated to a third-party library. Our current implementation uses a Python library for evaluating basic predicates (e.g. the size check in Figure 4.2). At runtime, the monitor passes the annotated information, along with the FSM state information, to the appropriate library 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.

4.2.2 Monitoring Requirements

Positioning. In order to guarantee global safety, our monitoring framework imposes complete mediation of communications: communication actions should not have an effect unless the message is mediated by the monitor. The tool implements this principal for outline monitor configurations, i.e. the monitor is running as a separate application. 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 Advanced Message Queuing Protocol (AMQP) [1] transport. An AMQP is a publish-subscribe middleware. An AMQP network consists of a federation of distributed virtual routers (called brokers) and queues. 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, referred to as Scribble header, and embedded into the message payload. Messages are processed depending on their kind, as recorded in the first field of the Scribble 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 the ones checked for protocol conformance.

Principals and Conversation runtime. A principal (an application) implements a protocol behaviour using the Conversation API. The API is built on top of a Conversation Runtime. The runtime provides a library for instantiating, managing and programming Scribble protocols and serialising and deserializing conversation messages. The library is implemented as a thin wrapper over an existing transport library. The API provides primitives for creating and joining a conversation, as well as primitives for sending and receiving messages.

4.3 Conversation Programming in Python

The Python Conversation API is a message passing API, which offers a high-level interface for safe conversation programming, mapping the interaction primitives of session types to lower-level communication actions on concrete transports. The API primitives are displayed in Figure 4.3. In summary, the API provides functionality for (1) session initiation and joining, (2) basic send/receive.

Conversation Initiation. A session is initiated using the `create` method. It creates a fresh conversation id and the required AMQP objects (principal exchange and queue), and sends an invitation for each role specified in the protocol. Invitations are sent to principals.

Conversation API operation	Purpose
`create(protocol_name, config.yml)`	Initiate conversation, send invitations
`join(self, role, principal_name)`	Accept invitation
`send(role, op, payload)`	Send a message
`recv(role)`	Receive message from role
`recv_async(self, role, callback)`	Asynchronous receive

Figure 4.3 The core Python Conversation API operations.

We use a configuration file to provide the mapping between roles and principals. We give on the right an example of the configuration file (invitation section) for the `DataAcquisition` protocol. Principal names direct the routing of invitation message to the right endpoint. Each invitation carries a role, a principal name and a name for a Scribble local specification file. An invitation is accepted using the `Conversation.join` method. It establishes an AMQP connection and, if one does not exist, creates an invitation queue for receiving invitations.

```
invitations:
  -role: U
   principal name: bob
   local capability: DataAcquisition.spr
  -role: A
   principal name: allice
   local capability: DataAcquisition.spr
  -role: I
   principal name: carol
   local capability: DataAcquisition.spr
```

We demonstrate the usage of the API in a Python implementation of the local protocol projected for the Agent role. The local protocol is given in Figure 4.2 (right). Figure 4.4 (left) gives the Agent role implementation. First, the `create` method of the Conversation API initiates a new conversation instance of the `DataAcquisition` protocol, and returns a token that is used to join the conversation locally. The `config.yml` file specifies which network principals will play which roles in this session and the runtime sends invitation messages to each principal. The `join` method confirms that the endpoint is joining the conversation as the principal `alice` playing role `A`. Once the invitations are sent and accepted (via `Conversation.join`), the conversation is established and the intended message exchange can proceed. As a result of the initiation procedure, the runtime at every participant has a mapping (conversation table) between each role and their AMQP addresses.

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 receive does block until the message has been received. In addition, an asynchronous receive method, called `recv_async`, is provided to support event-driven usage of the conversation API. These asynchronous features map closely to those supported by

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

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

    if (req_result == 'Supported'):
      c.send('I', 'Poll')
      op, data = c.recv('I')

      while (op !=  'Stop'):
        formatted_data = format(data)
        c.send('U', formatted_data)
      c.send('U', stop)
    else:
        c.send([U, I], stop)
        c.stop()
```

```
class ClientApp(BaseApp):
  def start(self):
    c = Conversation.create(
    'DataAcquisition', 'config.yml')
    c.join('A', 'alice')
    c.recv_async('U', on_request)

  def on_request(self, conv, op, msg):
    if (op == SUPPORTED):
      conv.send('I', 'Poll')
      conv.recv_async('I', 'on_data')
    else: conv.send([I, U], 'Stop')

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

Figure 4.4 Python program for `A`: synchronous implementation (left) and event-driven implementation (right).

Pika[1], a Python transport library used as an underlying transport library in our implementations.

Each message signature in a Scribble specification contains an operation and payloads (message arguments). The API does not mandate how the operation field should be treated, allowing the flexibility to interpret the operation name in various ways, e.g. as a plain message label, an RMI method name, etc. We treat the operation name as a plain label.

Following its local protocol, the program for `A` receives a request from `U` and forwards the message to `I`. The `recv` returns a tuple, (label, payload) of the message. When the message does not have a payload, only the label is returned (`req_result = c.recv('I')`). The `recv` method can also take the source role as a single argument (`c.recv('I')`), or additionally the label of the desired message (`c.recv('I', 'Request')`). The `send` method called on the conversation channel `c` takes, in this order, the destination role, message operator and payload values as arguments. In our example, the received payload `resource_request` is forwarded without modifications to `I`. After `A` receives the reply from `I`, the program checks the label value `req_result` using conditional statements, `if (req_result== 'Supported')`. If `I` replies with `'Supported'`, `A` enters a loop, where it continuously sends a `'Poll'` requests to `I` and after receiving the result from `I`, formats the received data (`format(data)`) and resends the formatted result to `U`.

[1]http://pika.readthedocs.org/

Event-driven conversations. For asynchronous, non-blocking receives, the Conversation API provides `recv_async` to be used in an event-driven style. Figure 4.4 (right) shows an alternative implementation of the `user` role using callbacks. The method `recv_async` accepts as arguments a callback to be invoked when a message is received.

We first create a conversation variable similar to the synchronous implementation. After joining the conversation, `A` registers a callback to be invoked when a message from `U` is received (`on_request`). The callback executions are linked to the flow of the protocol by taking the conversation id as an argument (e.g. `conv`). It also accepts as arguments the label for the message (`op`) and the payload (`msg`). In the message handler for `Request`, the role `A` forwards the received payload to `I` and registers a new message handler for the next message. Although the event-driven API promotes a notably different programming style, our framework monitors both implementations in Figure 4.4 transparently without any modifications.

4.4 Monitor Implementation

Figure 4.5 depicts our outline monitor configuration. The interception mechanism is based on message forwarding. A principal has at least one queue for consuming messages, although the number of queues can be tuned to

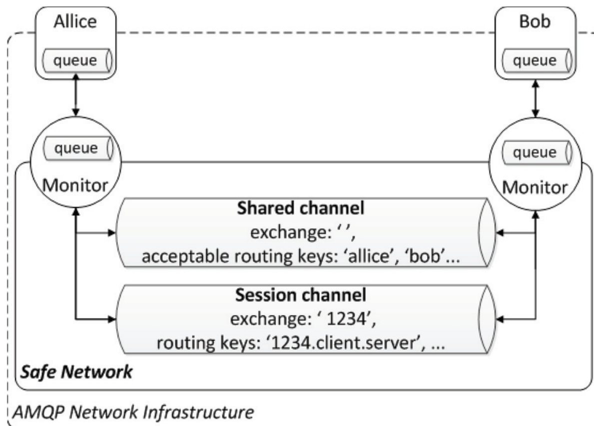


Figure 4.5 Configuration of distributed session monitors for an AMQP-based network.

use separate queues for invitations and roles. We outline a concrete scenario. Principal Alice is authenticated and connected to her local broker.

1. Authentication creates a network access point for Alice (the Monitor circle in Figure 4.5). The access point consists of a new conversation monitor instance, monitor queues (monitor as a consumer), and an exchange. Alice is only permitted to send messages to this exchange.
2. Alice initiates a new session (creates an exchange with id 1234 in Figure 4.5) and dispatches an invitation to principal Bob. The invitation is received and checked by Alice's monitor and then dispatched on the shared channel, from where it is rerouted to Bob's Monitor.
3. Bob's monitor checks the invitation, generates the local FSM and session context for Bob and Bob's role (for example client), and allocates a session channel (with exchange: 1234 and routing keys matching Bob's role ($1234.client.*$ and $1234.*.client$). The invitation is delivered to Bob's queue.
4. Any message sent by Alice (e.g. to Bob) in this session is similarly passed by the monitor and validated. If valid, the message is forwarded to the session channel to be routed. The receiver's monitor will similarly but independently validate the message.

Figure 4.6 depicts the main components and internal workflow of our prototype monitor. The lower part relates to conversation initiation. The *invitation* message carries (a reference to) the local protocol for the invitee and the conversation id. We use a parser generator (ANTLR[2]) to produce, from a Scribble local protocol, an abstract syntax tree with MPST constructs as nodes. The tree is traversed to generate a finite state machine, represented in Python as a hash table, where each entry has the shape:

$$(\mathit{current_state},\ \mathit{transition})\ \mapsto\ (\mathit{next_state},\ \mathit{assertion},\ \mathit{var})$$

where *transition* is a quadruple $(\mathit{interaction\ type}, \mathit{label}, \mathit{sender}, \mathit{receiver})$, *interaction type* is either *send* or *receive* and *var* is a variable binder for a message payload. We number the states using a generator of natural numbers. The FSM generation is based on the translation of local Scribble protocols to FSMs, presented in [5].

The upper part of Figure 4.6 relates to *in-conversation* messages, which carry the conversation id (matching an entry in the FSM hash table), sender and receiver fields, and the message label and payload. This information

[2] http://www.antlr.org/

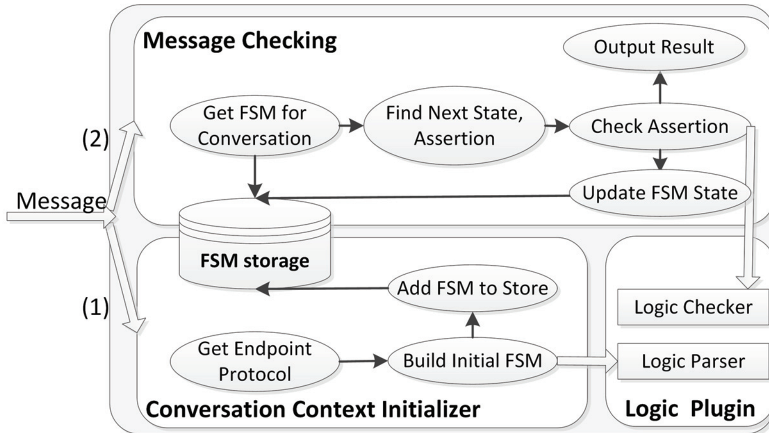


Figure 4.6 Monitor workflow for (1) invitation and (2) in-conversation messages.

allows the monitor to retrieve the corresponding FSM (by matching the message signature to the FSM's transition function). Assertions associated to communication actions are evaluated by invoking a library for Python predicate evaluation.

4.5 Monitoring Interruptible Systems

This section presents the implementation of a new construct for verifying *asynchronous multiparty session interrupts*. Asynchronous session interrupts express communication patterns in which the behaviour of the roles following the default flow through a protocol segment may be overruled by one or more other roles concurrently raising asynchronous interrupt messages. Extending MPST with asynchronous interrupts is challenging because the inherent communication race conditions that may arise conflict with the MPST safety properties. Taking a continuous stream of messages from a producer to a consumer as a simple example: if the consumer sends an interrupt message to the producer to pause or end the stream, stream messages (those already in transit or subsequently dispatched before the interrupt arrives at the producer) may well continue arriving at the consumer for some time after the interrupt is dispatched. This scenario is in contrast to the patterns permitted by standard session types, where the safety properties guarantee that no message is ever lost or redundant by virtue of disallowing all protocols with potential races.

This section introduces a novel approach based on reifying the concept of *scopes* within a protocol at the runtime level when an instance of the protocol is executed. A scope designates a sub-region of the protocol, derived from its syntactic structure, on which certain communication actions, such as interrupts, may act on the region as a whole. At run-time, every message identifies the scope to which it belongs as part of its meta data. From this information and by tracking the local progress in the protocol, the runtime at each endpoint in the session is able to resolve discrepancies in a protocol state by discarding incoming messages that have become irrelevant due to an asynchronous interrupt. This mechanism is transparent to the user process, and although performed independently by each distributed endpoint, preserves global safety for the session.

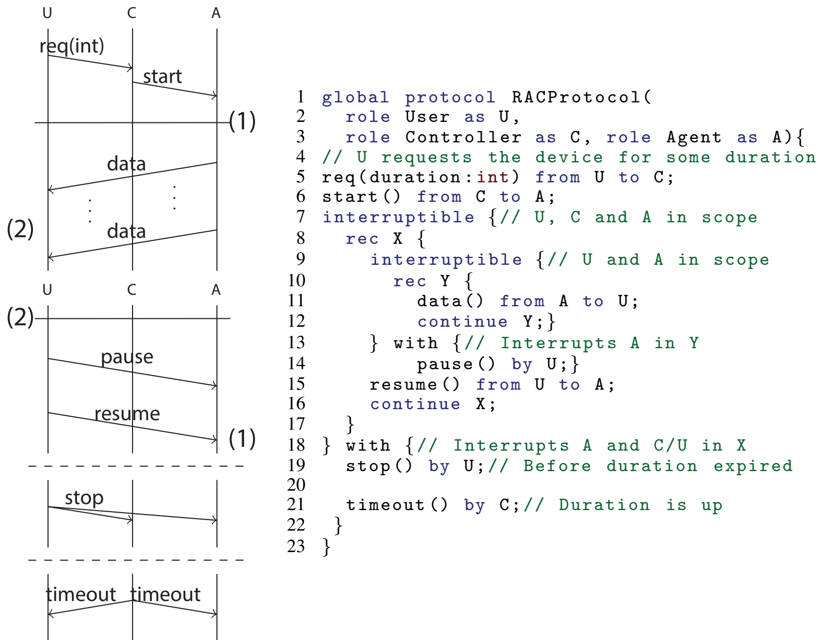
We integrate the new interrupt construct in our framework for runtime monitoring. The FSM generation is extended to support interruptible protocol scopes. We treat interruptible scopes by generating nested FSM structures. In the case of scopes that may be entered multiple times by recursive protocols, we use dynamic FSM nesting (conceptually, a new sub-FSM is created each time the scope is entered, and the sub-FSM is terminated once it reaches its end state or when an interrupt message is received) corresponding to the generation of fresh scope names in the syntactic model.

4.5.1 Use Case: Resource Access Control (RAC)

This section expands on how we extend Scribble to support the specification and verification of asynchronous session interrupts, henceforth referred to as just interrupts. Our running example is based on an OOI project use case, which we have distilled to focus on session interrupts.

Figure 4.7 (left) gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control (RAC) use case [8], regarding access control of users to sensor devices in the OOI cyberinfrastucture for data acquisition. In the OOI setting, a User interacts with a sensor device via its Agent proxy (which interacts with the device using a separate protocol outside of this example). OOI Controller agents manage concerns such as authentication of users and metering of service usage.

For brevity, we omit from the diagram some of the data types to be carried in the messages and focus on the *structure* of the protocol. The depicted interaction can be summarised as follows. The protocol starts at the top of the left-hand diagram. User sends Controller a `request` message to use a sensor




```
1  global protocol RACProtocol(
2    role User as U,
3    role Controller as C, role Agent as A){
4  // U requests the device for some duration
5  req(duration:int) from U to C;
6  start() from C to A;
7  interruptible {// U, C and A in scope
8    rec X {
9      interruptible {// U and A in scope
10       rec Y {
11         data() from A to U;
12         continue Y;}
13     } with {// Interrupts A in Y
14         pause() by U;}
15     resume() from U to A;
16     continue X;
17   }
18 } with {// Interrupts A and C/U in X
19   stop() by U;// Before duration expired
20
21   timeout() by C;// Duration is up
22  }
23 }
```

Figure 4.7 Sequence diagram (left) and Scribble protocol (right) for the RAC use case.

for a certain amount of time (the `int` in parentheses), and Controller sends a `start` to Agent. The protocol then enters a phase (denoted by the horizontal line) that we label (1), in which Agent streams `data` messages (acquired from the sensor) to User. The vertical dots signify that Agent produces the stream of data freely under its own control, i.e. without application-level control from User. User and Controller, however, have the option at any point in phase (1) to move the protocol to the phase labelled (2), below.

Phase (2) comprises three alternatives, separated by dashed lines. In the upper case, User *interrupts* the stream from Agent by sending Agent a `pause` message. At some subsequent point, User sends a `resume` and the protocol returns to phase (1). In the middle case, User interrupts the stream, sending both Agent and Controller a `stop` message. This is the case where User does not want any more sensor data, and ends the protocol for all three participants. Finally, in the lower case, Controller interrupts the stream by sending a `timeout` message to User and Agent. This is the case where, from Controller's view, the session has exceeded the requested duration, so Controller interrupts the other two participants to end the protocol. Note this diagram actually intends that `stop` (and `timeout`) can arise anytime after (1),

e.g. between `pause` and `resume` (a notational ambiguity that is compensated by additional prose comments in the specification).

4.5.2 Interruptible Multiparty Session Types

Figure 4.7 (right) shows a Scribble protocol that formally captures the structure of interaction in the Resource Access Control (RAC) use case and demonstrates the uses of our extension for asynchronous interrupts. Besides the formal foundations, we find the Scribble specification more explicit and precise, particularly regarding the combination of compound constructs such as choice and recursion, than the sequence diagram format, and provides firmer implementation guidelines for the programmer.

The protocol starts with a header declaring the protocol name (given as `RACProtocol` in Figure 4.7) and role names for the participants (three roles, aliased in the scope of this protocol definition as `U`, `C` and `A`). Lines 5 and 6 straightforwardly correspond to the first two communications in the sequence diagram, a User sends a request message, carrying an `int` *payload*, to the Controller and then the Controller replies with a `start()` message and an empty payload.

Then the intended communication in "phase" (1) and (2) in the diagram, is clarified in Scribble as two nested `interruptible` statements. The outer statement, on lines 7–22, corresponds to the options for User and Controller to end the protocol by sending the `stop` and `timeout` interrupts. An `interruptible`consists of a main body of protocol actions, here lines 8–17, and a set of interrupt message signatures, lines 18–22. The statement stipulates that each participant behaves by either (a) following the protocol specified in the body until finished for their role, or (b) raising or detecting a specified interrupt at any point during (a) and exiting the statement. Thus, the outer `interruptible` states that `U` can interrupt the body (and end the protocol) by a `stop()` message, and `C` by a `timeout()`.

The body of the outer `interruptible` is a labelled recursion statement with label `X`. The `continue X;` inside the recursion (line 16) causes the flow of the protocol to return to the top of the recursion (line 8). This recursion corresponds to the loop implied by the sequence diagram that allows User to pause and resume repeatedly. Since the recursion body always leads to the `continue`, Scribble protocols of this form state that the loop should be driven indefinitely by one role, until one of the interrupts is raised by *another* role. This communication pattern cannot be expressed in multiparty session types without `interruptible`.

The body of the X-recursion is the inner `interruptible`, which corresponds to the option for User to pause the stream. The stream itself is specified by the Y-recursion, in which A continuously sends `data()` messages to U. The inner `interruptible` specifies that U may interrupt the Y-recursion by a `pause()` message, which is followed by the `resume()` message from U before the protocol returns to the top of the X-recursion.

4.5.3 Programming and Verification of Interruptible Systems

We extend the Python API, presented in Section 4.3, to provide functionality for *scope* management for handling interrupt messages. We demonstrate the usage of the construct through an implementation of the local protocol projected for the `User` role. Figure 4.8 gives the local protocol and its implementation.

Similarly to the previous example from Section 4.3, the implementation starts by creating a conversation instance `c` of the Resource Access Control protocol (Figure 4.7) using method `create` (line 6, left) and `join`. The latter returns a conversation channel object for performing the subsequent communication operations.

Interrupt handling. The implementation of the User program demonstrates a way of handling conversation interrupts by combining conversation scopes

```
1 class UserApp(BaseApp):
2   user, controller, agent =
3       ['User', 'Controller', 'Agent']
4   def start(self):
5     self.buffer = buffer(MAX_SIZE)
6     conv = Conversation.create(
7              'RACProtocol', 'config.yml')
8     c = conv.join(user, 'alice')
9     c.send(controller, 'req', 3600)
10    with c.scope('timeout', 'stop') as c_x:
11      while not self.should_stop():
12        with c_x.scope('pause') as c_y:
13          while not self.buffer.is_full():
14            data = c_y.recv(agent)
15            self.buffer.append(data)
16          c_y.send_interrupt('pause')
17        use_data(self.buffer)
18        self.buffer.clear()
19        c_x.send(agent, 'resume')
20      c_x.send_interrupt('stop')
21    c.close()
```

```
local protocol RACProtocol
  at U (role C, role A){
  req(duration:int) to C;
  interruptible {
    rec X {
      interruptible {
        rec Y {
          data() from A;
          continue Y;
        }
      } with {
          pause() by U;
      }
      resume() to A;
      continue X;
    }
  } with {
      stop() by U;
      timeout() by C;
    }
}
```

Figure 4.8 Python implementation (left) and Scribble local protocol (right) for the User role for the global protocol from Figure 4.7.

with the Python `with` statement (an enhanced try-finally construct). We use `with` to conveniently capture interruptible conversation flows and the nesting of interruptible scopes, as well as automatic `close` of interrupted channels in the standard manner, as follows. The API provides the `c.scope()` method, as in line 10, to create and enter the scope of an **interruptible** Scribble block (here, the outer interruptible of the RAC protocol). The `timeout` and `stop` arguments associate these message signatures as interrupts with this scope. The conversation channel `c_x` returned by `scope` is a wrapper of the parent channel `c` that (1) records the current scope of every message sent in its meta data, (2) ensures every send and receive operation is guarded by a check on the local interrupt queue, and (3) tracks the nesting of scope contexts through nested `with` statements. The interruptible scope of `c_x` is given by the enclosing `with` (lines 10–20); if, e.g., a `timeout` is received within this scope, the control flow will exit the `with` block to line 21. The inner `with` (lines 12–16), corresponding to the inner interruptible block, is associated with the `pause` interrupt. When an interrupt, e.g. `pause` in line 16, is thrown (`send_interrupt`) to the other conversation participants, the local and receiver runtimes each raise an internal exception that is either handled or propagated up, depending on the interrupts declared at the current scope level, to direct the interrupted control flow accordingly. The delineation of interruptible scopes by the global protocol, and its projection to each local protocol, thus allows interrupted control flows to be coordinated between distributed participants in a structured manner.

The scope wrapper channels are closed (using the Python construct `with`) after throwing or handling an interrupt message. Since we assume asynchronous communication, there is a delay from the time when an interrupt mesasage is sent untill the time when the interrupt message is received by all participants. Hence, the monitor reacts differently when checking message sending (a check driven by the monitored participant) and message receive (an action driven by a message arriving in the queue of the monitor); the monitor discards the message in the latter case and marks the message as wrong in the former case. More precisely, when a monitor receives a message from a closed scope, it discards it as to accommodate for the delay in receiving of an interrupt message. However, if a participant that is monitoried attepts to send a message on a scope that is already closed (after an interrupt message has been recieved or after the participant has thrown interrupt himself) then the monitor flagges the interaction as an error. For example, using `c_x` after a `timeout` is received (i.e. outside its parent scope) will be flagged as an error. However, receiving messages on that scope will be

```
1  class UserApp(BaseApp):
2    def start(self):
3      self.buffer = buffer(MAX_SIZE)
4      conv = Conversation.create(
5            'RACProtocol', config.yml)
6      c = conv.join(user, 'alice')
7      # request 1 hour access
8      c.send(controller, 'req', 3600)
9      c_x = c.scope('timeout', 'stop')
10     c_y = c_x.scope('pause')
11     c_y.recv_async(agent, recv_handler)
12
13   def recv_handler(self, c, op, payload):
14     with c:
15       if self.should_stop():
16         c.send_interrupt('stop')
17       elif self.buffer.is_full():
18         self.process_buffer(c, payload)
19       else:
20         self.buffer.append(payload)
21       c.recv_async(agent, recv_handler)
22
23   def process_buffer(self, c, payload):
24     with c:
25       c_x = c.send_interrupt('pause')
26       use_data(self.buffer, payload)
27       self.buffer.clear()
28       c_x.send(agent, 'resume')
29       c_y = c_x.scope('pause')
30       c_y.recv_async(agent, recv_handler)
```

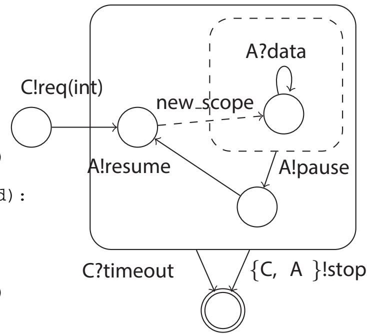


Figure 4.9 Event-driven conversation implementation for the User role (left) and Nested FSM generated from the User local protocol (right).

discarded and will not be dispatched to the application. In our example, the User runtime discards `data` messages that arrive after `pause` is thrown. The API can also make the discarded data available to the programmer through secondary (non-monitored) operations.

Message handlers with scopes. As demonstrated in Section 4.3, our Python API supports asynchronous receive through the primitive `recv_async`. The construct is used to register a method that should be invoked on message receive. To support event-driven programming with interrupts, we extend the implementation presented in Section 4.3. The difference is in the semantics of event handlers. More precisely, each event handler is associated with a scope. Therefore, if an interrupt is received, but the protocol state is not in the same scope as the scope written in the conversation header of the interrupt message, the interrupt will be discarded.

Figure 4.9 (left) shows an alternative implementation of the `User` role using callbacks. We first enter the nested conversation scopes according to the potential interrupt messages (lines 9 and 10). The callback method is then registered using the `recv_async` operation (line 11). The callback executions are linked to the flow of the protocol by taking the scoped channel

as an argument (e.g. `c` on line 13). Note that if the `stop` and `pause` interrupts were not declared for these scopes, line 16 and line 25 would be considered invalid by the monitor. When the buffer is full (line 17), the user sends the `pause` interrupt. After raising an interrupt, the current scope becomes obsolete and the channel object for the parent scope is returned. After the data is processed and the buffer is cleared, the `resume` message is sent (line 28) and a fresh scope is created and again registered for receiving data events (line 29). Our framework monitors both this implementation and that in Figure 4.8 transparently without any modifications.

4.5.4 Monitoring Interrupts

FSM generation for interruptible local protocols makes use of nested FSMs. Each `interruptible` induces a nested FSM given by the main interruptible block, as illustrated in Figure 4.9 (right) for the User local protocol. The monitor internally augments the nested FSM with a scope id, derived from the signature of the interruptible block, and an interrupt table, which records the interrupt message signatures that may be thrown or received in this scope. Interrupt messages are marked via the same meta data field used to designate invitation and in-conversation messages, and are validated in a similar way except that they are checked against the interrupt table. However, if an interrupt arrives that does not have a match in the interrupt table of the immediate FSM(s), the check searches upwards through the parent FSMs; the interrupt is invalid if it cannot be matched after reaching the outermost FSM.

4.6 Formal Foundations of MPST-Based Runtime Verification

In this section, we explain the correspondence between a theoretical model for MPST-based monitoring and the implementation, presented in this chapter. Our implementation is formalised in a theory for MPST-based verification of networks, first proposed in [3], and later extended in [1], and in [2]. The interrupt extension is formalised in [4]. [3] only gives an overview of the desired properties, and requires all local processes to be dynamically verified through the protections of system monitors, while [1] presents a framework for semantically precise decentralised run-time verification, supporting statically and dynamically verified components. In addition, the

routing mechanism of AMPQ networks is explicitly presented in [1], while in [3] it is implicit.

A delicate technical difference between the theory and the implementation lies in handling of out-of-order delivery of messages when messages are sent from different senders to the same receiver. Asynchrony poses a challenge in the treatment of out-of-order asynchronous message monitoring, and thus, to prevent false positive results, in the theoretical model, a type-level permutations of actions is required, e.g a monitor checks messages up to permutations. The use of global queues and local permutations is inefficient in practice, and thus we have implemented the theoretical model of a global queue as different physical queues. Specifically, we introduce a queue per pair of roles, which ensures messages from the same receivers are delivered in order and are not mixed with messages from other roles. This model is semantically equivalent to a model of a global indexed queue, permitting permutation of messages.

Next we explain the correspondence between the asynchronous π-calculus with fine-grained primitives for session initiation and our Python API.Also in [1] specifications are given as local types. Instead of using local types, for efficient checking, we use communicating finite state machines (CFSMs) generated from local Scribble protocols, which are equivalent to local types, as has been shown in [5].

Processes. Our Python API embodies the primitives of the asynchronous π-calculus with fine grained primitives for session initiation, presented in [1]. The correspondence is given in Figure 4.10. Note that the API does not stipulate the use of a recursion and a conditional, which appear in the syntax of session π-calculus, since these constructs are handled by native Python constructs. The `create` method, which, we remind, creates a fresh conversation id and the required AMQP objects (principal exchanges and queues), and sends an invitation for each role specified in the protocol, corresponds to the action $\overline{a}\langle s[r] : T\rangle$, which sends on the shared channel a, an

Conversation API operation	Purpose
`create(protocol_name, config.yml)`	$\overline{a}\langle s[r] : T\rangle$
`join(self, role, principal_name)`	$a(y[r] : T).P$
`send(role, op, payload)`	$k[r_1, r_2]!l\langle e\rangle$
`recv(role)`	$k[r_1, r_2]?\{l_i(x_i).P_i\}_{i\in I}$
`recv_async(self, role, callback)`	–

Figure 4.10 The core Python Conversation API operations and their session π-calulus counterparts.

invitation to join the fresh conversation s as the role of r with a specification T. In the implementation, this information is codified in the message header, which as we have explained contains the new session id (abstracted as s), the name of the local Scribble protocol (i.e. T) and the role (i.e. r). The invitation action $a(y[r] : T).P$ models session `join`. As a result of `join` new queues and a routing bindings are created. For example, when Bob joins a conversation with id of 1234 as the role of `client`, as shown in Figure 4.5, an AMQP binding `1234.client.*` is created, which ensures that all messages to the role of a `client` are delivered to Bob. The reduction rule for $a(y[r] : T).P$, in the semantics in [1], reflects this behaviour by adding a record in the routing table. The primitive for sending a message $k[r_1, r_2]!l\langle e\rangle$ corresponds to the API call `send`, and results in sending a message of type $s[r_1, t_2]!l\langle e\rangle$, which in the implementation is codified in the message header, consisting of session id s, sender r_1, receiver r_2, label l and a payload e.

Properties of monitored networks. Finally, we give an overview of the properties of monitored networks as presented in [1]. Due to the correspondence explained above, these properties are preserved in the context of the monitor implementation, presented in this chapter.

Local safety states that a monitored process respects its local protocol, i.e. that dynamic verification by monitoring is sound.

Local transparency states that a monitored process has equivalent behaviour to an unmonitored but well-behaved process, e.g. statically verified against the same local protocol.

Global safety states that a system satisfies the global protocol, provided that each participant behaves as if monitored.

Global transparency states that a fully monitored network has equivalent behaviour to an unmonitored but well-behaved network, i.e. in which all local processes are well-behaved against the same local protocols.

Session fidelity states that, as all message flows of a network satisfy global specifications, whenever the network changes because some local processes take actions, all message flows continue to satisfy global specifications.

4.7 Concluding Remarks

We have presented a runtime verification framework for Python programs based on Scribble protocols. We discuss the core design elements of the implemention of a conversation-based API in a dynamically typed language,

Python. Through a runtime layer of protocol management Scribble protocols are loaded and translated to CFSMs such that during a program execution, messages emitted by the program are checked against a corresponding CFSM. We also introduce a construct for expressing exception-like patterns in Scribble, which syntactically splits the protocol into sub-regions, allowing certain messaging to act on the regions as a whole and thus permitting controllable races, traditionally disallowed by the theory of session types.

Acknowledgements We thank the anonymous reviewers for their insightful comments, which helped us to improve the article. This work is partially supported by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1; by EU FP7 612985 (UP-SCALE).

References

[1] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65, 2013.

[2] Tzu-Chun Chen. *Theories for Session-based Governance for Large-scale Distributed Systems*. PhD thesis, Queen Mary, University of London, 2013.

[3] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC'11*, volume 7173 of *LNCS*, pages 25–45, 2012.

[4] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verication with multiparty session types and python. *FMSD*, pages 1–29, 2015.

[5] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[6] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63, 2016.

[7] Ocean Observatories Initiative. http://www.oceanobservatories.org/
[8] OOIExamples. http://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI
[9] Scribble project home page. http://www.scribble.org