Practical interruptible conversations

Distributed dynamic verification with session types and Python

Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon

Imperial College London

Abstract. The rigorous and comprehensive verification of communication-based software is an important engineering challenge in distributed systems. Drawn from our industrial collaborations [32,35,26] on Scribble, a choreography description language based on multiparty session types, this paper proposes a dynamic verification framework for structured interruptible conversation programming.

We first present our extension of Scribble to support asynchronously interruptible conversations. We then implement a concise API for conversation programming with interrupts in Python that enables distributed processes to be dynamically verified. Our framework ensures the global safety of a system in the presence of asynchronous interrupts through runtime monitoring of each endpoint, checking the conformance of each local execution trace against the specified protocol. The usability of our framework for describing and verifying choreographic communications has been tested by integration into the large scientific cyberinfrastructure developed by the Ocean Observatories Initiative. Asynchronous interrupts have proven expressive enough to represent and verify their main classes of communication patterns, including asynchronous streaming and various timeout-based protocols, without requiring additional synchronisation mechanisms. Benchmarks show conversation programming and monitoring can be realised with little overhead.

1 Introduction

The main engineering challenges in distributed systems include finding suitable specifications that model the range of states exhibited by a system, and ensuring that these specifications are followed by the implementation. In message passing applications, rigorous specification and verification of communication protocols is particularly crucial: a protocol is the interface to which concurrent components should be independently implementable while ensuring their composition will form a correct system as a whole. Multiparty Session Types (MPST) [15,4] is a type theory for distributed programs that originates from works on the types of the π-calculus towards tackling this challenge. In the original MPST setting, protocols are expressed as types and static type checking verifies that the system of processes engaged in a communication session (also referred to as a *conversation*) conforms to a globally agreed protocol. The properties enjoyed by well-typed processes are communication safety (no unexpected messages or races during the execution of the conversation) and deadlock-freedom.

In this paper, we present the design and implementation of a framework for dynamic verification of protocols based on MPST, developed from our collaboration with industry partners [32,35,26] on the application of MPST theory. In this ongoing partnership, we are motivated to adapt MPST to dynamic verification for several reasons. First, session type checking is typically designed for languages with first-class communication

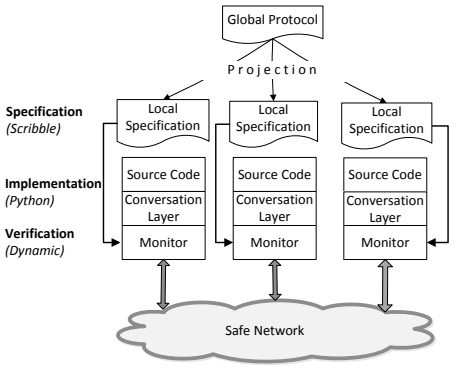


Fig. 1: Scribble methodology from global specification to local runtime verification

and concurrency primitives, whereas our collaborations use mainstream engineering languages, such as Python and Java, that lack the features required to make static session typing tractable. Distributed systems are also often heterogeneous in nature, meaning that different languages and techniques (e.g. the control flow of an event-driven program is tricky to verify statically) may be used in the implementation of one system. Dynamic verification by communication monitoring allows us to verify MPST safety properties directly for mainstream languages in a more scalable way. Second, a system may use third-party components or services for which the source code is unavailable for type checking. Third, certain protocol specification features, such as assertions on specific message values, can be precisely evaluated at run-time, while static treatments would usually be more conservative.

Framework overview. Figure 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 [34,35], an engineering incarnation of the formal MPST type language. The core features of Scribble include multicast message passing and constructs for branching, recursive and parallel conversations. These features support the specification of a wide range of protocols, from domains such as standard Internet applications [16], parallel algorithms [25] and Web services [10].

Our toolchain validates that the global protocol satisfies certain well-formedness properties, such as coherent branches (no ambiguity between participants about which branch to follow) and deadlock-freedom (between parallel flows). From a well-formed global protocol, the toolchain mechanically generates (projects) Scribble *local* protocols for each participant (role) defined in the protocol. A local protocol is essentially a view of the global protocol from the perspective of one role, and provides a more direct specification for endpoint implementation than the global protocol.

When a conversation is initiated at run-time, the monitor at each endpoint generates a finite state machine (FSM) representation of the local communication behaviour from the local protocol for its role. In our implementation, the FSM generation is an extension of the correspondence between MPST and communication automata in [11] to support interruptible sessions (discussed below) and optimised to avoid parallel state

explosion. The monitor tracks the communication actions performed by the endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the FSM. Each monitor works to protect both the endpoint from illegal actions by the environment, and to protect the network from bad endpoints. In this way, our framework is able to ensure from the local verification of each endpoint that the global progress of the system as a whole conforms to the original global protocol [5], and that illegal actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints.

This MPST monitoring framework has been integrated into the Python-based run-time platforms developed by the Ocean Observatories Initiative (OOI) [26]. 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. Their architecture relies on the combination of high-level protocol specifications of network services (expressed as Scribble protocols [27]) and distributed run-time monitoring to regulate the behaviour of third-party applications within the system [29]. Although this work is in collaboration with the OOI, our implementation can be used orthogonally as a standalone monitoring framework for distributed Python applications.

Contributions and summary. This paper demonstrates the application of multiparty session types, through the Scribble protocol language, to industrial practice by presenting (1) the first implementation of MPST-based dynamic protocol verification (as outlined above) that offers the same safety guarantees as static session type checking, and (2) use cases motivated extension of Scribble to support the first construct for the verification of asynchronous communication interrupts in multiparty sessions.

We developed the extension of Scribble with asynchronous interrupts to support a range of OOI use cases that feature protocol structures where one flow of interactions can be asynchronously interrupted by another. Examples include various service calls (request-reply) with timeout, and publish-subscribe applications where the consumer can request to pause and resume externally controlled feeds. Although the existing features of Scribble (i.e. those previously established in MPST theory) are sufficiently expressive for many communication patterns, we observed that this important structure could not be directly or naturally represented without interrupts.

We outline the structure of this paper, summarising the contributions of each part:

§ 2 presents a use case for the extension of Scribble with asynchronous interrupts. This is a new feature in MPST, giving the first general mechanism for nested, multiparty session interrupts. We explain why implementing this feature is a challenge in session types. The previous works on exceptions in session types are purely theoretical, and are either restricted to binary session types (i.e. not multiparty) [9], do not support nesting and continuations [9,8], or rely on additional implicit synchronisation [7]. A formal proof of the correctness of our design is given in § 5.1.

§ 3 discusses the Python implementation of our MPST monitoring framework that we have integrated into the OOI project, and demonstrates the global-to-local projection of Scribble protocols, endpoint implementation, and local FSM generation. § 3.1 describes a concise API for conversation programming in Python that supports standard socket-like operations and event-driven interfaces. The API decorates conversation messages with the run-time session information required by the monitors

to perform the dynamic verification. § 3.2 discusses the monitor implementation, how asynchronous interrupts are handled, and the other architectural requirements of our framework.

§ 4 evaluates the performance of our monitor implementation through a collection of benchmarks. The results show that conversation programming and monitoring can be realised with low overhead.

The source code for our Scribble toolchain, conversation runtime and monitor, performance benchmarks and further resources are available from the project page [36].

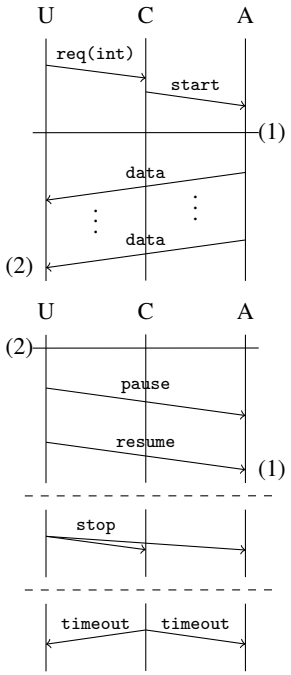
2 Communication protocols with asynchronous interrupts

This section expands on why and 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. Using this example, we outline the technical challenges of extending Scribble with interrupts.

Resource Access Control (RAC) use case. As is common practice in industry, the cyber-infrastructure team of the OOI project [26] manages communication protocol specifications through a combination of informal sequence diagrams and prose descriptions. Figure 2 (left) gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control use case [27], regarding access control of users to sensor devices in the ION Cyber-infrastucture for data acquisition. In the ION setting, a User interacts with a sensor device via its Agent proxy (which interacts with the device via a separate protocol outside of this example). ION 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 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, however, has the option at any point in phase (1) to move the protocol to the phase labelled (2) in the right-hand diagram.

Phase (2) comprises three cases, separated by the dashed lines. In the top 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. In this diagram, note that `stop` and `timeout` can appear anytime.




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

Fig. 2: Sequence diagram (left) and Scribble protocol (right) for the RAC use case

Interruptible multiparty session types. Figure 2 lists a multiparty session type that uses our new extension for asynchronous interrupts to formally capture the communication protocol for the Resource Access Control use case. For readability, and to introduce the language, we give the protocol here using Scribble. Besides the formal foundations, we find the Scribble specification is 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 (demonstrated in § 3.1).

A Scribble protocol starts with a header declaring the protocol name (in Figure 2, `ResourceAccessControl`) and role names for the participants (three roles, aliased in the scope this protocol definition as `U`, `C` and `A`). Lines 3 and 5 straightforwardly correspond to the first two communications in the sequence diagram. The Scribble syntax for message signatures, e.g. `req(duration:int)`, means a message with *operator* (i.e. header, or label) `req`, and *payload* `int` annotated as `duration`. The `start()` message signature means operator `start` with an empty payload.

We now come to "phase" (1) of the sequence diagram. The new `interruptible` construct captures the informal usage of protocol phases in disciplined manner, making explicit the interrupt messages and the *scope* in which they apply. Although the syntax has been designed to be readable and familiar to programmers, `interruptible` is an advanced construct that encapsulates several aspects of asynchronous interaction, which we discuss at the end of this section.

The intended communication protocol of our example is clarified in Scribble as two nested `interruptible` statements. The outer statement, on lines 6–23, corresponds to

the options for User and Controller in order to end the protocol via the `stop` and `timeout` interrupts. An `interruptible` consists of a main body of protocol actions, here Lines 7–18, and a set of interrupt message signatures, Lines 19–23. 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 `stop()` message, and `C` by a `timeout()`.

The body of the outer `interruptible` is a standard labelled recursion statement with label `X`. The `continue X;` inside the recursion (Line 17) causes the flow of the protocol to return to the top of the recursion (Line 7). This recursion corresponds to the loop implied by the sequence diagram for 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` repeatedly sends `U` `data()` messages. The inner `interruptible` specifies that `U` may interrupt the `Y`-recursion by a `pause()` message, and is followed by the `resume()` message from `U` before the protocol returns to the top of the `X`-recursion.

Challenges of asynchronous interrupts in MPST. We outline some observations from our extension and usage of MPST with asynchronous interrupts. We find the basic operational meaning of `interruptible`, as illustrated in the above example, is readily understood by architects and developers, which is a primary consideration in the design of Scribble. The challenges in this extension are in the design of the supporting runtime and verification techniques to preserve the desired safety properties in the presence of `interruptible`. The challenges stem from the fact that `interruptible` combines several tricky, from a session typing view, aspects of communication behaviours that session type systems traditionally aim to prohibit to prevent communication races and thereby ensure the safety properties.

One aspect is that of mixed choice in the protocol, in terms of both communication direction (e.g. `U` may choose to either receive the next `data` or send a `stop`), and between different roles (e.g. `U` and `C` independently, and possibly concurrently, interrupt the pro-

```
1  // The choice is not well-formed
2  choice at A {
3  // A makes the choice
4      rec Y {
5          data() from A to U;
6          continue Y;}
7  } or {
8  // U makes the choice
9      pause() from U to A;}
10 resume() from U to A;
```

```
1  // Well-formed, but different
       semantics
2  // The recursion cannot be stopped
3  par {
4      rec Y {
5          data() from A to U;
6          continue Y;}
7  } and {
8  // Does not stop the recursion
9      pause() from U to A;}
10 resume() from U to A;
```

Fig. 3: Naive interrupt encoding with choice (left) and parallel (right)

Conversation API operation	*Purpose*
`create(protocol_name,invitation_config.yml)`	Session initiation and invitation sending
`join(self, role,principal_name)`	Accept an invitation
`send(role, op, payload)`	Send operation and payload (`conv_msg`)
`recv(role)`	Receive message
`recv_async(self, role, callback)`	Receive asynchronously
`scope(msg)`	Create a scope
`close()`	Close the connection

Fig. 4: Basic Python Conversation API operations

tocol) due to multiparty (see Figure 3 (left), which breaks the unique sender condition in [11]). In addition, the interrupt choice is truly optional in the sense that it may never be selected at run-time. The basic choice in standard MPST (e.g. as defined in [15,11]) is inadequate because it is designed to identify a single role as the decision maker, who communicates exactly one of a set of message choices unambiguously to all relevant roles.

Another aspect, due to asynchrony, is that an interrupt may occur in parallel to the actions of the roles being interrupted (e.g. `pause` by `U` to `A` while `A` is streaming `data` to `U`, see Figure 3 (right)). Although standard MPST (and Scribble) support parallel protocol flows, the interesting point here is that an interrupt is sent to preclude further actions in (i.e. interfere with) another parallel flow under the control of a different role. Due to the asynchronous setting, it is important that `interruptible` does not introduce additional synchronisation, still preserving a safety property. These mechanisms are formalised and the correctness of our interruptible extension is proved in § 5.1.

3 Runtime Verification

This work is the first implementation of the theory in [5] to practice. Although the Scribble language is directly inspired by the multiparty session types formalism, this is the first work (theory or practice) to feature a general, asynchronous interrupt mechanism in MPST, and the first presentation of our Python API for MPST programming.

As an outline, the verification methodology in our framework is as follows. Endpoint programs communicate via Conversation API calls to the local conversation runtime. When a conversation is initiated, the monitor at each endpoint (inlined into the runtime) observes the initiation messages containing the local protocols, which are then translated into FSMs. Bookkeeping by the monitor associates the FSM to the new conversation instance. As the conversation proceeds, the header of each message contains the conversation identifier and the source and destination roles. From this information, the monitor can determine the relevant FSM, and verify whether the message is permitted by an available transition.

3.1 Conversation API

Our Python conversation API offers a high-level interface for safe conversation programming. It maps the core communication primitives of session types to lower-level communication actions on concrete transports. Our current implementation is built on top of the PIKA [30] AMQP client library for Python, an event-based library that adopts

```
local protocol ResourceAccessControl
     at User as U(
     role Controller as C,
     role Agent as 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;
    } }
```

```
class UserApp(BaseApp):
  user, controller, agent = ['user', '
       controller', 'agent']

  def start(self):
    c = Conversation.create(
          'RACProtocol', 'config.yml')
    c.join(user, 'alice')
    # request 1 hour access
    c.send(controller, 'req', 1)
    with c.scope('timeout','stop') as c1:
      while self.limit_reached():
        with c.scope('pause') as c2
          while not self.buffer.full
            resource = c1.recv(device)
            buffer.append(resource)
            c2.send_interrupt('pause')
        # sleep before resume
        c1.send('resume', resource.id)
      if self.should_stop():
        c1.send_interrupt('stop')
    if c1.interrupt: # handle interrupt
      c.close()
```

Fig. 5: Scribble local protocol (left) and Python implementation (right) for the User role

continuation-passing style for synchronous (blocking) calls. In summary, the API provides functionality for (1) session initiation and joining, (2) basic send/receive and (3) *conversation scope* management for handling interrupt messages. Figure 4 lists the basic API operations. Although the invitation operations are not captured in standard MPST types, the formal counterparts of `create` and `join` appear in the literature in formalisms such as [9].

We demonstrate two different implementations of the User process: sequential (single-threaded) (Figure 5) and event-driven (Figure 6). An advantage of run-time monitoring is that both programs can be checked by the same monitor against the same specification, whereas type checking each would involve quite different approaches. The former follows the local protocol shape and handles the interruptible control flow in a structured manner, hinting that static checking could be possible with additional (orthogonal) reasoning to deal with linear variable usage and potential object aliasing. The latter uses asynchronous message handling, which offers better scalability in a concurrent environment, but is less intractable for static validation due to the obfuscated control flow. Below we explain each of the primitives following the implementation in Figure 5.

Conversation initiation. First, the `create` method of the Conversation API initiates a new conversation (session) following the Resource Access Control protocol, and returns a conversation token that can be used for joining the created conversation (Line 6). The config.yml file specifies which network principals will play which roles in this session and send invitation messages that are routed to principals. The `join` method then conforms to join this session as the principal `alice` playing the role `user`. `Conversation.join` returns a conversation channel to be used for communication operations. Once the invitations are sent and accepted (via the `Conversation.join`), a session is established and

```
1  class UserApp(BaseApp):
2    def start(self):
3      c = Conversation.create(
4            'RACProtocol', ...)
5      c.join(user, 'alice')
6      # request 1 hour access
7      c.send(controller, 'Request', 1)
8      c1 = c.new_scope('Timeout', 'Stop')
9      c.receive_async(agent, on_data_recv)
10
11   def on_data_recv(self, c,
12       op, payload):
13     if c.interrupt:
14       log(c.exception.info)
15       if  # timeout do clearing
16     elif # want to stop:
17       c.send_interrupt('Stop')
18       c.close()
19     elif # want to pause:
20       c.send('Resume'm resource.id)
21       # sleep
22       c = c.new_scope('Pause')
23       c.receive_async(agent, on_data_recv)
24     elif # just received
25       self.buffer.append(payload)
26       c.receive_async(agent, on_data_recv)
```

Fig. 6: Event-driven Python implementation for the User role

the intended message exchange can start. As a result of the initiation, each participant stores a mapping (conversation table) that associate AMQP addresses and role.

Conversation message passing. Then, following the local protocol, the `user` sends a request to the `controller` passing the time duration for which he requires an access to the `agent`. The `send` method called on the conversation channel `c` takes, in this order, the destination role, message operator and payload values as arguments. This information is encapsulated in the message payload as part of a conversation header and is later used for checking by the runtime verification module. The receive method (`recv`) can take the source role as a single argument, or additionally the operator of the desired message. Send is asynchronous, meaning that a basic send does not block on the corresponding receive; however, the basic receive does block until the complete message has been received. An asynchronous receive `recv_async` is non-blocking and provides a support for event-driven usage of the conversation API. In Figure 6 the stream is handled by the user through the callback function (`on_data_recv`) passed to the `recv_async` method. The callback executions are linked to the protocol by having a conversation channel as an argument. Note that the API does not mandate how the message operator field (for example `'req'`) 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.

Handling interrupts via scopes. The stream from the `agent` to the `user` can be interrupted either permanently (if a `timeout` is received from the `agent`) or temporarily if the `user` explicitly interrupts the stream by a `pause` interrupt message (Line 16) and resumes it later. An interruptible block is treated through `c.scope()` (as in Line 10). A conversation channel returned by `c.scope()` is a wrapper of the default channel (`c1` is a wrapper around `c`). It (1) ensures that every send and receive operation is guarded by a check on the interrupt queue and (2) can be used as a context for `with` statements (enhanced try-finally construct in Python). The combination of `with` and channels allows convenient handling of interrupt messages. For example, the interruptible block associated with `c1` spawns across its associated `with` statement (Line 10–4), which guarantees that if a `'timeout'` message is received, the control flow will jump out the block to Line 21. When an interrupt messages is thrown (`send_interrupt` in Line 16), the scope

field in the conversation header is set accordingly. Then each receiver identifies the interrupted scope and throws internal `ConversationException`. Each `with` statement handles only the exception associated to its scope.

The try-finally mechanism is used for convenience, but is not mandatory. In the case of an the event-driven implementation (Figure 6) it is not appropriate and the interrupts are handled manually by checking the interrupt attribute (Line 13) of the wrapped channel and acting accordingly.

3.2 Monitoring Architecture

Inline and outline monitors. 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. This principle requires that all outgoing messages from a principal before reaching the destination, and all incoming messages before reaching the principal, are routed through the monitor.

The monitor implementation (and the accompanying theory [5]) is compatible with a range of monitor configurations. At one end of the spectrum is *inline monitoring*, where an endpoint monitor is embedded into a component's code. Then there are various configurations for *outline monitoring*, i.e. monitors external to their components. OOI architecture realises interceptor stack, the monitor is a part of the interceptor stack and is embedded inside the application endpoint.

Monitor implementation. Figure 7 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 [11]. Our implementation differs from [11] in the treatment of parallel sub-protocols (i.e. unordered message sequences) and interrupt. 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 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: $(\mathit{current\_state},\ \mathit{transition}) \mapsto (\mathit{next\_state},\ \mathit{assertion},\ \mathit{var})$ where *transition* is a triple $(\mathit{label}, \mathit{sender}, \mathit{receiver})$, and *var* is the variable binder for the message payload.

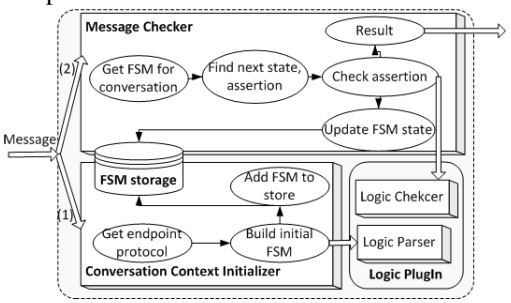


Fig. 7: Monitor workflow for (1) *invitation* (2) *in-session messages*

The upper part of Figure 7 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

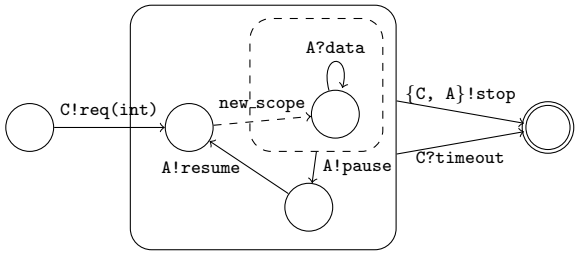


Fig. 8: FSM generated from the User local protocol

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.

Handling interrupts. To handle interruptible local types, our FSM generation algorithm extends [11] to *the nested FSMs*. Each interruptible block induces a new nested FSM produced by parsing the constructs in the interruptible block. The FSM generated by the local type for the User role is shown in Figure 8. Each nested FSM is augmented with an additional interruptible table and a scopeID, obtained by the name of the interruptible block. The interruptible table stores all interruptible messages that are expected to be thrown/received for this scope. Interrupt messages are handled in the same way as the normal messages with the difference that checking is done against the interruptible table. Messages are explicitly marked (via the message type field in the conversation header) as interrupt, in-session or initialising. If a received interrupt trace does not have a match in the interruptible table of the current active FSM, the check is done against its parent FSM. This FSM unfolding continues until the interrupt is matched or the topmost FSM is reached. If the interrupt match is not found, the monitor marks the message as faulted.

4 Evaluations

Our dynamic verification framework has been implemented and integrated into the Ocean Observatories cyber-infrastructure prototype [28]. This section reports our integration effort and discusses the performance of our framework.

4.1 Experience: OOI integration

OOI is using a Service-Oriented Architecture, with all of the distributed system services accessible by RPC. As a part of their effort to move to agent-based communication patterns and to enable distributed governance for more than just individual RPC calls, we engineered a step-by-step transition. The first step was to add our Scribble monitor to the stack of interceptors present in the middleware layer of the implementation. The second step was to propose our conversation programming interface to developers. To facilitate the use of session types without obstructing the existing application code we used the interface of the RPC libraries, but replace the underlying machinery it with

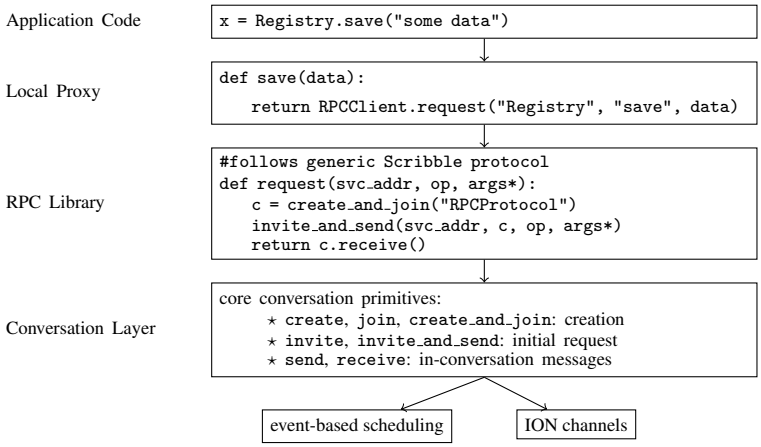


Fig. 9: Translation of an RPC command into lower-level conversation calls

the distributed runtime for session types. The RPC library is now realised on top of the Conversation Layer (as shown in Figure 9). Thus using the session primitives, it is automatically verified by the internal monitors. This conversion is feasible because not even a single line of application code needs to be changed. As a result developers can use the same interface (conversation API) with a formal foundation automatically assuring correctness for more complex interactions.

The final step in our integration efforts is ongoing. It consists in leveraging the present conversation layer providing advanced protocols such as a complicated negotiation between distributed agents [27].

4.2 Benchmarks

A main concern during the integration efforts was the potential performance overhead that a conversation layer and monitoring can introduce to the system. However, the benchmarks show it is reasonable. Table 1 presents the execution time comparing RPC calls using the original OOI RPC library implementation and the conversation-based RPC without and with monitor verification. 13% of overhead is recorded, which is due to the FSM generation time. The cost of the checking is negligible. Further optimisations are still possible, such as caching state machine to reduce the monitor initialisation time.

We complete this benchmark with numbers showing how well our framework scales beyond RPC. Figure 2 shows that our overall verification architecture (conversation layer and inline monitor) scales reasonably (as prescribed by the theory) and does not induce performance concerns when stress tested. We present two corner case protocol examples, each aimed at testing a particular performance concern: increasing session length (number of messages) and increasing parallel states (FSM size). We measure the time to complete a session between client and server endpoints connected to a single-broker AMQP network. Two benchmark cases are compared. The main case (Monitor, Mon) is fully monitored, i.e. FSM generation and message checking are enabled for both the client and server. The base case for comparison (No Monitor, NoM) has the

	10 RPC calls (s)
RPC Lib	0.103
No Monitor	0.108 +4
Monitor	0.122 +13

Table 1: OOI library vs conversation-based RPC

Rec States	NoM (s)	Mon (s)	
10	0.92	0.95	+3.2%
100	8.13	8.22	+1.1%
1000	80.31	80.53	+0.8%

Par States	NoM (s)	Mon (s)	
10	0.45	0.49	+8%
100	4.05	4.22	+4.1%
1000	40.16	41.24	+2.7%

Table 2: Microbenchmarks comparing end-to-end monitor performance

client and server in the same configuration, but monitors are disabled (messages do not go through the interceptor stack).

Each table gives the mean time for the client and server to complete one session after repeating the benchmark 100 times for each parameter configuration. The Python processes and their monitors run on separate machines (Intel Core2 Duo 2.80 GHz, Linux). Latency between each node is measured to be 0.24 ms on average (ping 64 bytes).

The tables emphasise the linear growth that is introduced by the monitor in the various cases. Tests shows that this grow is due to the cost of the initial FSM construction. As expected, checking a large, but flat FSM (1000 states) gives a negligible overhead (0.02 ms on average). Most importantly, the relative overhead decreases as the session length increases, because the one-time FSM generation cost becomes less prominent. For the dense FSM the worse case scenario results in linear overhead growth w.r.t. the number of parallel branches.

4.3 Use Cases

Finally, we conclude our evaluation with some remarks on the use-cases we have investigated. Table 3 features a list of protocols, coming both from the research community and from our industrial use cases, that we have written in Scribble. We also fed them to our monitor implementation in order to measure how well it behaves on realistic protocol descriptions. A natural question for protocol-based validation is the overhead imposed by the developers for formally writing the protocols. A primary motivation for the use of Scribble is that it allows clearer, more precise and concise specifications, thereby reducing design and testing effort for distributed systems.

The average scribble protocol is about 10 LOC with the longest one being 20 LOC of Scribble. This suggests that Scribble is reasonably concise. The main factors that depend on the choice of protocol and which might affect the capacity and performance of the validation framework are the (i) time required for the generation of states machines and the (ii) memory overhead that may be induced by the generation of nested state machines in case of parallel blocks and interrupts. Table 3 addresses these concerns over practical use cases and demonstrates the applicability of our verification mechanism. The time required for FSM generation remains under 20 ms, measuring on average to be around 10 ms. As shown in Figure 2 such relative overhead decreases with the increase of the transaction length and the latency between the nodes. The memory overhead also remains in reasonable boundaries (under 1.5 KB) which hints that catching the FSM is a feasible approach for further optimisations.

Use Cases from research papers	Global Scribble (LOC)	FSM Memory (B)	Generation Time (s)
a vehicle subsystem protocol [19]	8	840	0.006
map web-service protocol [13]	10	1040	0.010
a bidding protocol [22]	26	1544	0.020
Amazon search service [14]	12	1088	0.010
SQL service [31]	8	1936	0.009
online shopping system [12]	10	1024	0.008
travel booking system [12]	16	1440	0.013
Use Cases from OOI and Savara			
a purchasing protocol [33]	11	1088	0.010
a banking example [27]	16	1564	0.013
negotiation protocol [27]	20	1320	0.014
RPC with timeout [27]	11	1016	0.013
Resource Access Control [27]	21	1854	0.018

Table 3: Implemented use cases

Overall, considering all the numbers given in this section and our experience of running our conversation framework within the OOI system, we predict that, in any large distributed system, the monitoring cost is decentralised and would be negligible compared with all the other services (routing, QOS, logging, etc) that are running at the same time. The important benefits in terms of safety and management of high-level applications therefore come at a very reasonable cost that should be considered in such future distributed systems. The full source code and raw results can be obtained from the project homepage [36].

5 Interruptible session type theory and related work

5.1 Session type theory for interrupts

In this subsection, we sketch the underlying session type theory with interrupts and its correctness result, *session fidelity*, justifying our design choice. We build over the multiparty session theory [15], adding syntax and semantics for interrupts. In our theory, global types correspond to session specifications whereas local types are used to express monitored behaviours of processes [5]. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that realises, through an explicit identifier, the domain of interrupts. Our scope-based session types can handle nested interrupts and multiparty continuations to interruptible blocks, allowing us to model truly asynchronous exceptions implemented in this paper (these features have not been modelled in existing MPST theories for exceptions [9,8,7]). The full definitions and proofs are available in Appendix A.

Global types (G) in Figure 12 corresponds to Scribble protocol. In types, scopes are made explicit by the use of scope variables S, corresponding to the dynamic scope generation present in the implementation in § 3.1. Roles in types are denoted by $\mathtt{r}$, and labels with l.

The main primitive is the interaction with directed choice: $\mathtt{r}\to\mathtt{r}' : \{l_i.G_i\}_{i\in I}$ is a communication between the sender $\mathtt{r}$ and the receiver $\mathtt{r}'$ which involves a choice be-

$$
\begin{array}{lll}
G ::= & \mathtt{r}\to\mathtt{r}' : \{l_i.G_i\}_{i\in I} & \mid G|G \mid \{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G' \mid \mu\mathbf{x}.G \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend} \\
T ::= & \mathtt{r}!\{l_i.T_i\}_{i\in I} & \mid \mathtt{r}?\{l_i.T_i\}_{i\in I} \\
\mid & T|T \mid \{|T|\}^S \triangleleft \langle \mathtt{r}!l\rangle; T' & \mid \{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T' \mid \mu\mathbf{x}.T \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend}
\end{array}
$$

Fig. 10: Global and local types

tween several labels l_i, the corresponding continuations are denoted by the G_i. Parallel composition $G_1|G_2$ allows the execution of interactions not linked by causality.

Our types feature a new interrupt mechanism by explicit interruptible scopes: we write $\{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G'$ to denote a creation of an interruptible block identified by scope S, containing protocol G, that can be interrupt by a message l from $\mathtt{r}$ and continued after completion (either normal or exceptional) with protocol G'. This construct corresponds to the `interruptible` of Scribble, presented in § 2. Note that we allow interruptible scopes to be nested. This syntax - and the related properties - can be easily extended to multiple messages from different roles. We use `Eend` (resp. `end`) to denote the exceptional (resp. normal) termination of a scope.

The local type syntax (T) in Figure 12 follows the same pattern, but the main difference is that the interruptible operation is divided into two side, one $\triangleleft$ side for the role which can send an interrupt $\{|T|\}^S \triangleleft \langle \mathtt{r}!l\rangle; T'$, and the $\triangleright$ side for the roles which should expect to receive an interrupt message $\{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T'$.

In Figure 14, we describe a formal global type which corresponds to the Scribble protocol in Figure 2. The explicit naming of the scopes, S_1 and S_2, correspond to the dynamic scope generations in § 3.1, and are formally required to formalise the semantics of local types.

$$
\begin{array}{l}
G_{\mathtt{ResCont}} = \mathtt{U}\to\mathtt{C} : \mathrm{req}; \mathtt{C}\to\mathtt{A} : \mathrm{start} \\
\quad \{|\mu X. \\
\quad\quad \{|\mu Y.\mathtt{A}\to\mathtt{U} : \mathrm{data}; Y|\}^{S_2}\langle \text{pause by } \mathtt{U}\rangle; \\
\quad\quad \mathtt{U}\to\mathtt{A} : \mathrm{resume}; X \\
\quad |\}^{S_1}\langle \text{stop by } \mathtt{U}, \text{timeout by } \mathtt{C}\rangle; \mathtt{end}
\end{array}
$$

Fig. 11: Global type for Figure 2

We define the relation $G \rightsquigarrow G'$ as:

$$
\mathtt{r}\to\mathtt{r}' : \{l_i.G_i\}_{i\in I} \rightsquigarrow G_i \qquad \{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \rightsquigarrow \{|\mathtt{Eend}|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0
$$

$$
G \rightsquigarrow G' \text{ implies } \{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \rightsquigarrow \{|G'|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \quad G \rightsquigarrow G' \text{ implies } G \mid G_0 \rightsquigarrow G' \mid G_0
$$

and say G' is a *derivative* of G if $G \rightsquigarrow^* G'$. We define *configurations* Δ, Σ as a pair of a mapping from a session channel to a local type and a collection of queues (a mapping from a session channel to a vector of the values). Configurations model the behaviour of a network of monitored agents. We say a configuration Δ, Σ corresponds to a collection of global types $G_1, \ldots, G_l$ whenever Σ is empty and the environment Δ is a projection of $G_1, \ldots, G_l$. The reduction semantics of the configulation $(\Delta, \Sigma \to \Delta', \Sigma')$ is defined using the contexts with the scopes. We leave the formal definition in Appendix A.

The correctness of our theory is ensured by Theorem 1, which states a local enforcement implies global correctness: if a network of monitored agents (modelled as a configuration) corresponds to a collection of well-formed specifications and makes some steps by firing messages, then the network can perform reductions (consuming these messages) and reaches a state that corresponds to a collection of well-formed specifications, obtained from the previous one. This property guarantees that the network is always linked to the specification, and proves, with the previous dynamic mon-

itoring process theory [5], that the introduction of interruptible blocks to the syntax and semantics yields a sound theory. See the proofs in Appendix A.

Session fidelity If Δ corresponds to $G_1, \ldots, G_n$ and $\Delta_0, \varepsilon \rightarrow^* \Delta, \Sigma$, there exists $\Delta, \Sigma \rightarrow^* \Delta', \varepsilon$ such that Δ' corresponds to $G'_1, \ldots, G'_n$ which is a derivative of $G_1, \ldots, G_n$.

5.2 Related works

Distributed run-time verification. The work [2] explores run-time monitoring based on session types as a test framework for multi-agent systems (MAS). A global session type is specified as cyclic Prolog terms in Jason (a MAS development platform). 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 and session interrupt). Their monitor is centralised (thus no projection facilities are discussed), and neither formalisation, global safety properties, proof of correctness nor compositional reasoning methodology are given in [2].

Other works, notably from the multi-agent community, have studied distributed enforcement of global properties through monitoring. A distributed architecture for local enforcement of global laws is presented by Zhang et al. [37], where monitors enforce *laws* expressed as event-condition-action. In [24], monitors may trigger sanctions if agents do not fulfil their obligations within given deadlines. Unlike such frameworks, where all agents belonging to a group obey the same set of laws, our approach asks agents to follow personalised laws based on the role they play in each session.

In run-time verification for Web services, the work [22,23] proposes FSM-based monitoring using a rule-based declarative language for specifications. These systems typically position monitors to protect the safety of service interfaces, but do not aim to enforce global network properties. Cambronero et al. [6] transform a subset of Web Services Choreography Description Language into timed-automata and prove their transformation is correct with respect to timed traces. Their approach is model-based, static and centralised, and does not treat either the runtime verification or interrupts. Baresi et al. [3] develop a run-time monitoring tool for BPEL with assertions. 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. Kruger et al. [20] 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 [12] follows a similar but centralised approach of [20]. As a language for protocol specification, a main advantage of Scribble (i.e. MPST) over alternatives, such as message sequence charts (MSC), CDL and BPML, is that MPST has both a formal basis and an in-built mechanism (projection) for decentralisation, and is easily integrated with the language framework as demonstrated for Python in this paper.

Language-based monitor tools. Jass [17] is a precompiler tool that monitors dynamic behaviour of sequential objects, the ordering of method invocations and calls. It annotates Java programs with specifications that can be checked at run-time. Another popular language-based approach is aspect-oriented programming [21]. In comparison,

our approach is language-independent and interoperable. Moreover, our safety properties are backed up by a solid process theory, well-suited to the analysis of distributed systems. Mace [18] is a language designed to write clean specifications for the systems layer of distributed systems targeted to C++. Programming of service objects in Mace is based on a state-event-transition model, using aspects for taking actions when a certain condition is satisfied. They use model-checking for error detection for the behaviour of service objects. Their frameworks do not treat protocol descriptions for communication-centred development, nor do they consider specifications of the global behaviour of distributed systems.

6 Conclusion

We have first implemented the dynamic verification of distributed communications based on multiparty session types and shown that a new interrupt mechanism is effective for the run-time verification of message exchanges over a large cyber-infrastructure [26] and Web services [32,35,34]. Our implementation automates distributed monitoring by generating FSMs from local protocol projections. We sketched the formulation of asynchronous interruptions with session scopes, and proved the correctness of our design through the session fidelity theorem. Future work includes the incorporation of a more elaborate handling of error cases into monitors and the automatic generation of services stubs. Although our implementation work is ongoing through industry collaborations, the results already 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.

References

1. Advanced Message Queuing protocols (AMQP) homepage. `http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`.
2. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in Jason. In *DALT'12*. Springer, 2012.
3. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04*, pages 193–202, 2004.
4. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
5. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. (to appear), 2013.
6. M.-E. Cambronero et al. Validation and verification of web services choreographies by using timed automata. *J. Log. Algebr. Program.*, 80(1):25–49, 2011.
7. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *FSTTCS'10*, volume 8 of *LIPICS*, pages 338–351, 2010.
8. M. Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.
9. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
10. W3C WS-CDL. `http://www.w3.org/2002/ws/chor/`.
11. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, LNCS. Springer, 2012.

12. Y. Gan et al. Runtime monitoring of web service conversations. In *CASCON '07*, pages 42–57. ACM, 2007.
13. C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.
14. S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire. Runtime verification of web service interface contracts. *Computer*, 43(3):59–66, Mar. 2010.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
16. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
17. Jass Home Page. `http://modernjass.sourceforge.net/`.
18. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *PLDI*, pages 179–188, 2007.
19. I. H. Krüger, M. Meisinger, and M. Menarini. Runtime verification of interactions: from mscs to aspects. In *RV'07*, RV'07, pages 63–74, Berlin, Heidelberg, 2007. Springer-Verlag.
20. I. H. Krüger, M. Meisinger, and M. Menarini. Interaction-based runtime verification for systems of systems integration. *J. Log. Comput.*, 20(3):725–742, 2010.
21. LAVANA project. `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`.
22. Z. Li, J. Han, and Y. Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC'05*, pages 73–86, 2005.
23. Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC'06*, pages 70–79, Washington, DC, USA, 2006. IEEE Computer Society.
24. N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM*, 9:273–305, July 2000.
25. N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.
26. OOI. `http://www.oceanobservatories.org/`.
27. OOI. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.
28. OOI codebase. `https://github.com/ooici/pyon`.
29. OOI COI Governance Framework. `https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Framework`.
30. AMQP for Python (PIKA). `https://github.com/pika/pika`.
31. G. Salaün. Analysis and verification of service interaction protocols - a brief survey. In *TAV-WEB*, volume 35 of *EPTCS*, pages 75–86, 2010.
32. Savara JBoss Project. `http://www.jboss.org/savara`.
33. Savara examples. `http://www.jboss.org/savara/downloads`.
34. Scribble Project homepage. `http://www.scribble.org`.
35. Scribble JBoss homepage. `http://www.jboss.org/scribble`.
36. Full version of this paper. `http://www.doc.ic.ac.uk/~rn710/mon`.
37. W. Zhang, C. Serban, and N. Minsky. Establishing global properties of multi-agent systems via local laws. In *E4MAS'06*, pages 170–183, 2007.

A Session types with interrupts

This appendix presents the underlying session type theory with interrupts and its correctness result, *session fidelity*, justifying our choice for the implementation of interrupt messages. We build over an existing multiparty session theory [15], adding syntax and semantics for interrupts. In our theory, global types correspond to session specifications

whereas local types are used to express monitored behaviours of processes [5]. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that realises, through an explicit identifier, the domain of interrupts.

A.1 Global and local types

Syntax. Global types (G) in Figure 12 describe role-based global scenarios between multiple participants as a type signature, projectable into local types. In types, scopes are made explicit by the use of scope variables S. We assume there is an infinite set of such variables and that no two variables are the same inside global types. This is crucial as our syntax contains recursion: recursive types are treated as equi-recursive terms, meaning that the lazy unfolding of the types is implicit; thus, when a scope variable appears inside of a recursion loop, it actually stands for an infinite number of fresh variables. We consider that this is an appropriate abstraction of the dynamic scope generation present in the implementation in § 3.1. Roles in types are denoted by $\mathtt{r}$, and labels (appearing in directed choices) with l.

$$
\begin{array}{rcl}
G & ::= & \mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I} \;\mid G|G \;\mid \{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G' \;\mid \mu\mathbf{x}.G \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend} \\
T & ::= & \mathtt{r}!\{l_i.T_i\}_{i\in I} \;\mid \mathtt{r}?\{l_i.T_i\}_{i\in I} \\
 & \mid & T|T \;\mid \{|T|\}^S \triangleleft \langle \mathtt{r}!l\rangle; T' \;\mid \{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T' \;\mid \mu\mathbf{x}.T \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend}
\end{array}
$$

Fig. 12: Global and local types

The main primitive is the interaction with directed choice: $\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}$ is a communication between the sender $\mathtt{r}$ and the receiver $\mathtt{r}'$ which involves a choice between several labels l_i, the corresponding continuations are denoted by the G_i. Parallel composition $G_1|G_2$ allows the execution of interactions not linked by causality.

Our types feature a new interrupt mechanism by explicit interruptible scopes: we write $\{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G'$ to denote a creation of an interruptible block identified by scope S, containing protocol G (called *inner protocol*), that can be interrupt by a message l from $\mathtt{r}$ and continued after completion (either normal or exceptional) with protocol G' (called *continuation protocol*). This construct corresponds to the `interruptible` of Scribble, presented in § 2. For the sake of clarity, we suppose there is only one possible interrupt message (from one particular role) for each scope, but extending it to multiple interrupt messages (possibly from different roles) is not difficult. Note that we allow interruptible scopes to be nested. We use `Eend` (resp. `end`) to denote the exceptional (resp. normal) termination of a scope.

The local type syntax (T) in Figure 12 follows the same pattern, but the main difference is that the interruptible operation is divided into two side, one $\triangleleft$ side for the role which can send an interrupt $\{|T|\}^S \triangleleft \langle \mathtt{r}!l\rangle; T'$, and the $\triangleright$ side for the roles which should expect to receive an interrupt message $\{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T'$.

Global types are subject to some well-formedness conditions [15], which constrain the type syntax. This enforces causality in an asynchronous framework (preventing $\mathtt{r}_1 \to \mathtt{r}_2; \mathtt{r}_3 \to \mathtt{r}_4$ to be viable). We assume every global type G is well-formed according to the conditions from [15], and adding interruptible scopes does not introduce new conditions.

Projection. We define the projection operation $\upharpoonright \mathtt{r}$, which, for any participant playing a role $\mathtt{r}$ in a session G, specifies its local type.

$$
\begin{aligned}
(\mathtt{r}\to\mathtt{r}':\{l_i.G_i\}_{i\in I})\upharpoonright\mathtt{r} &= \mathtt{r}'!\{l_i.(G_i\upharpoonright\mathtt{r})\}_{i\in I}\\
(\mathtt{r}\to\mathtt{r}':\{l_i.G_i\}_{i\in I})\upharpoonright\mathtt{r}' &= \mathtt{r}?\{l_i.(G_i\upharpoonright\mathtt{r}')\}_{i\in I}\\
(\mathtt{r}\to\mathtt{r}':\{l_i.G_i\}_{i\in I})\upharpoonright\mathtt{r}_0 &= G_1\upharpoonright\mathtt{r}_0\\
(\mu\mathbf{x}.G)\upharpoonright\mathtt{r}_0\in G &= \mu\mathbf{x}.(G\upharpoonright\mathtt{r}_0)\\
(\mu\mathbf{x}.G)\upharpoonright\mathtt{r}_0\notin G &= \mathtt{end}\\
\mathbf{x}\upharpoonright\mathtt{r}_0 &= \mathbf{x}\\
\mathtt{end}\upharpoonright\mathtt{r}_0 &= \mathtt{end}\\
\{|G|\}^S\langle l\text{ by }\mathtt{r}\rangle;G'\upharpoonright\mathtt{r} &= \{|G\upharpoonright\mathtt{r}|\}^S\triangleright\langle\mathtt{r}?l\rangle;G'\upharpoonright\mathtt{r}\\
\{|G|\}^S\langle l\text{ by }\mathtt{r}'\rangle;G'\upharpoonright\mathtt{r}\in G &= \{|G\upharpoonright\mathtt{r}|\}^S\triangleleft\langle\mathtt{r}'!l\rangle;G'\upharpoonright\mathtt{r}\\
\{|G|\}^S\langle l\text{ by }\mathtt{r}'\rangle;G'\upharpoonright\mathtt{r}\notin G &= G'\upharpoonright\mathtt{r}
\end{aligned}
$$

We assume $\mathtt{r}$, $\mathtt{r}'$ and $\mathtt{r}_0$ are pairwise distinct.

The projection rules themselves are straightforward, and similar to the one in [15]: an interaction is projected as a send action $\mathtt{r}'!$ of the sender side, a receive $\mathtt{r}'?$ action on the receiver side and does not appear to other roles (the well-formedness conditions from [15] ensure that every branch is the same to these roles). When it comes to interruptible constructs, the projection on role $\mathtt{r}$ works as follows: if $\mathtt{r}$ does not appear in the inside protocol, the projection ignores the construct and amounts to the projection on the continuation. If role $\mathtt{r}$ is the role responsible for the interrupt, the projection is a $\triangleright$ local type; and if the role $\mathtt{r}$ is not responsible for the interrupt, but appears inside the inner scope, the projection is a $\triangleleft$ local type.

A.2 Configuration and semantics

In order to justify our framework, we need to introduce a semantics for local types. This will be defined through the use of configurations, which are meant to represent the situation of an on-going network of monitored principals.

We use Δ to denote *session environments* which are collections of local types $s_1[\mathtt{r}_1]:T_1,\ldots,s_n[\mathtt{r}_n]:T_n$ and abstract monitored principals, more precisely $s_1[\mathtt{r}_1]:T_1$ is the status of participant $\mathtt{r}_1$ in session s_1 which is expected to behave as T_1. *Standard messages* are explained as follows: $S[\mathtt{r},\mathtt{r}']\langle l\rangle$ meaning it appears inside scope S, is sent from $\mathtt{r}$ to $\mathtt{r}'$ and contains label l. We also annotate messages for interrupts as in $S^{\mathtt{I}}[\mathtt{r},\mathtt{r}']\langle l\rangle$. A *queue* $s[\mathtt{r}]:h$ is a sequence of messages waiting to be consumed by a particular role $\mathtt{r}$ in session s. Queues are ordered, but we allow permutations of two messages in the same queue if they have different receivers (as in [15]). For the sake of clarity, we do not describe here the relaxing of conditions on permutability induced by the use of scope (we could allow two messages to the same receiver to be permuted if they are not tagged with the same scope).

Configurations Δ,Σ are pairs composed of a session environment and a *transport* Σ which is a collection of queues. Configurations model the behaviour of a network of monitored agents.

We define a reduction semantics for configurations in Figure 13. In order to treat a message with its corresponding scope, we need to remember from which scope the message was sent. To this purpose, we enrich the definition of scopes with ε the empty

$$
\begin{array}{ll}
(\mathsf{Out}) & s[\mathtt{r}] : E^S[\mathtt{r}'!\{l_i.T_i\}]; s[\mathtt{r}'] : h \to s[\mathtt{r}] : E^S[T_i]; s[\mathtt{r}'] : h.S[\mathtt{r},\mathtt{r}']\langle l_i\rangle \\
(\mathsf{In}) & s[\mathtt{r}] : E^S[\mathtt{r}'?\{l_i.T_i\}]; s[\mathtt{r}] : S[\mathtt{r}',\mathtt{r}]\langle l_i\rangle.h \to s[\mathtt{r}] : E^S[T_i]; s[\mathtt{r}] : h \\
(\mathsf{EOut}) & s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T']; s[\mathtt{r}_1] : h, \ldots, s[\mathtt{r}_n] : h \\
 & \qquad \to s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}?l\rangle; T']; s[\mathtt{r}_1] : S^{\mathrm{I}}[\mathtt{r},\mathtt{r}_1]\langle l\rangle.h, \ldots, s[\mathtt{r}_n] : S^{\mathrm{I}}[\mathtt{r},\mathtt{r}_n]\langle l\rangle.h \\
(\mathsf{EIn}) & s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : S^{\mathrm{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : h \\
(\mathsf{Disc}) & s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : S_1[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : h \\
(\mathsf{EDisc}) & s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : S_1^{\mathrm{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; T']; s[\mathtt{r}] : h \\
(\mathsf{Par}) & \Delta, \Delta_0; \Sigma, \Sigma_0 \to \Delta', \Delta_0; \Sigma', \Sigma_0 \quad \text{if } \Delta; \Sigma \to \Delta'; \Sigma'
\end{array}
$$

In (EOut), we assume $\Gamma(S) = \{\mathtt{r}, \mathtt{r}_1, \ldots, \mathtt{r}_n\}$; and in (Disc, EDisc), we assume $\Gamma \vdash S \mathrel{\mathscr{R}} S_1$.

Fig. 13: Reduction semantics for a specification

scope and add a scope annotation on contexts. Thus evaluation contexts are defined by:

$$
\begin{array}{l}
E^{\mathcal{E}} = [\,] \mid (E^{\mathcal{E}}|T) \mid (T|E^{\mathcal{E}}) \\
E^S = \{|E^S|\}^{S' \neq S} \triangleright \langle \mathtt{r}?l\rangle; T' \quad \mid \{|E^S|\}^{S' \neq S} \triangleleft \langle \mathtt{r}!l\rangle; T' \quad \mid \{|E^{\mathcal{E}}|\}^S \triangleright \langle \mathtt{r}?l\rangle; T' \\
\qquad \mid \{|E^{\mathcal{E}}|\}^S \triangleleft \langle \mathtt{r}!l\rangle; T' \quad \mid \{|\mathtt{Eend}|\}^{S' \neq S} \triangleright \langle \mathtt{r}?l\rangle; E^S \mid \{|\mathtt{Eend}|\}^{S' \neq S} \triangleleft \langle \mathtt{r}!l\rangle; E^S \\
\qquad \mid \{|\mathtt{end}|\}^{S' \neq S} \triangleright \langle \mathtt{r}?l\rangle; E^S \mid \{|\mathtt{end}|\}^{S' \neq S} \triangleleft \langle \mathtt{r}!l\rangle; E^S \quad \mid E^S|T \quad \mid \quad T|E^S
\end{array}
$$

Evaluation contexts are indexed by scope S; our definition ensures that the evaluation actually happens inside S (i.e. S is the innermost scope in which the hole appears). Evaluation can proceed from inside the inner scope of an interruptible (either $\triangleright$ or $\triangleleft$) construct, or from inside the continuation scope of a interruptible, but only when the inner scope has ended (normally or exceptionally).

The reduction semantics is defined w.r.t. a *scope environment* $\Gamma = \mathscr{T}, \mathscr{R}$ composed of a *scope table* $\mathscr{T} ::= \varepsilon \mid S : \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}, \mathscr{T}$ and a *scope order* which is the reflexive and transitive closure of the relation given by: $S_1 \mathrel{\mathscr{R}} S_2$ whenever a global type contains $E^{S_1}[\{|G|\}^{S_2}\langle l \text{ by } \mathtt{r}\rangle; G']$. The scope table keeps a track of every participant in a scope and the scope order keeps track of scope nesting (when $S_1 \mathrel{\mathscr{R}} S_2$ it means that scope S_2 is inside scope S_1). We note $\Gamma(S) = \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$ whenever $\Gamma = \mathscr{T}, \mathscr{R}$ and $\mathscr{T}$ contains $S : \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$. The environment is omitted when not necessary.

Semantics rules in Figure 13 are as follows: in (Out), an output from $\mathtt{r}$ to $\mathtt{r}'$ appearing inside the scope S of the type of role $\mathtt{r}$ in session s is played and a message is placed in the queue $s[\mathtt{r}']$, tagged with S. Conversely in (In), a message in queue $s[\mathtt{r}']$ can be consumed by $\mathtt{r}'$ inside a matching scope. In rule (EOut), a type T inside scope S is interrupted by $\mathtt{r}$, which replaces T by Eend and places an interrupt message in the queues of each participant of scope S (we need the table from Γ). Conversely in rule (EIn), an interrupt message for scope S is consumed to exceptionally terminates the type T inside scope S. Rule (Disc) discards an incoming message to scope S_1 nested inside scope S if the latter has already been exceptionally terminated (we need the scope order from Γ). Rule (EDisc) performs the same thing for exceptional messages.

Regarding to the semantics, we have two remarks. Most of existing theoretical works such as [15] consider session creations, through the use of auxiliary actions. Also the garbage collection can be handled by adding completion annotation to types and additional rules to control broadcasts of special messages: when a participant receives a completion message it can assume its sender is finished, and when every other partic-

ipants of a scope are finished the whole interrupt construct can be garbage collected. Both these facilities can be integrated into the current semantics.

A.3 Session fidelity proof

The correctness of our theory is ensured by Theorem 1, which states a local enforcement implies global correctness: if a network of monitored agents (modelled as a configuration) corresponds to a collection of well-formed specifications and makes some steps by firing messages, then the network can perform reductions (consuming these messages) and reaches a state that corresponds to a collection of well-formed specifications, obtained from the previous one. This property guarantees that the network is always linked to the specification, and proves, with the previous dynamic monitoring process theory [5], that the introduction of interruptible blocks to the syntax and semantics yields a sound theory.

First, we define configuration correspondence: a configuration Δ, Σ *corresponds to* a collection of global types $G_1, \ldots, G_l$ whenever Σ is empty and $\Delta = \{G_i \upharpoonright \mathtt{r} \mid \mathtt{r} \in G_l,\ 1 \leq i \leq l\}$. That is, the environment is a projection of existing well-formed global types. We use $\rightarrow^*$ to denote the reflexive-transitive closure of $\rightarrow$.

We say that a global type G' is a *derivative* of G whenever G' can be obtained from G by progressing in the types. The formal definition is given by taking the reflexive and transitive closure of the $\rightsquigarrow$-relation:

$$\mathtt{r} \rightarrow \mathtt{r}' : \{l_i.G_i\}_{i \in I} \rightsquigarrow G_i \qquad \{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \rightsquigarrow \{|\mathtt{Eend}|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0$$

$$\{|G|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \rightsquigarrow \{|G'|\}^S\langle l \text{ by } \mathtt{r}\rangle; G_0 \text{ if } G \rightsquigarrow G' \qquad G \mid G_0 \rightsquigarrow G' \mid G_0 \text{ if } G \rightsquigarrow G'$$

A.4 Type memory

We say that a queue *has an ongoing exception on S*, written $\varphi(\Sigma, S)$ whenever Σ contains at least one message $S_1^{\mathtt{I}}[\mathtt{r}', \mathtt{r}]\langle l \rangle$ and $S \mathscr{R} S_1$.

We use a special annotation, called *memory* to remember what has been discarded by exceptions. The syntax of memory types is the same as the one for standard local types except we add $E^S[\![T]\!]$. We define the recursive operator $Erase(\cdot)$ which removes memory annotations from types: $Erase(s[\mathtt{r}] : E^S[\![T]\!]) = s[\mathtt{r}] : E^S[\mathtt{Eend}]$.

From the definition of the correspondence relation between global types and Δ we build the *intermediate correspondence* between global types and configurations Δ, Σ containing type with memories using the following updates:

- $\Delta, s[\mathtt{r}] : E^S[T]; \Sigma, s[\mathtt{r}'] : h.S[\mathtt{r}', \mathtt{r}]\langle l_j \rangle$ becomes $\Delta, s[\mathtt{r}] : E^S[\mathtt{r}!\{l_i.T_i\}]; \Sigma$ for some $(T_i)_{i \neq j}$
- If the participants of S are $\mathtt{r}, \mathtt{r}_1, \ldots, \mathtt{r}_n, \Delta, s[\mathtt{r}] : E^S[\![\mathtt{T}]\!] \prod_{1 \leq i \leq k} s[\mathtt{r}_i] : {E_i}^S[\![T_i]\!], \prod_{k+1 \leq j \leq n} s[\mathtt{r}_j] : {E_i}^S[T_j]; \Sigma, \prod_{k+1 \leq j \leq n} s[\mathtt{r}_j] : S^{\mathtt{I}}[\mathtt{r}, \mathtt{r}_j]\langle l \rangle.h_j$ is treated as $\Delta, s[\mathtt{r}] : E^S[T], \prod_{1 \leq i \leq n} s[\mathtt{r}_i] : {E_i}^S[T_i]; \Sigma$.

This definition ensures first that ongoing outputs are treated as if they were not yet emited, and that ongoing exceptions are treated as if the exceptions were not yet triggered.

Special semantics for type with memory annotations is obtained by giving memories the same semantics as `Eend` w.r.t. contexts and using the following rules for annotated types (replacing (Disc), (EDisc) and (EIn)):

(Disc′) assuming $s[\mathtt{r}] : E^{S_0}[T]; s[\mathtt{r}] : S_1[\mathtt{r}', \mathtt{r}]\langle l \rangle.h \to s[\mathtt{r}] : E^{S_0}[T']; s[\mathtt{r}] : h$
$s[\mathtt{r}] : E^{S_0}[\{\!|\!|T|\!|\!|\}^S \triangleright \langle \mathtt{r}'?l \rangle; \_]; s[\mathtt{r}] : S_1[\mathtt{r}', \mathtt{r}]\langle l \rangle.h \to E^{S_0}[\{\!|\!|T'|\!|\!|\}^S \triangleright \langle \mathtt{r}'?l \rangle; \_]; s[\mathtt{r}] : h$

(EIn1) assuming $\varphi(\Sigma, S)$
$s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l \rangle; T']; s[\mathtt{r}] : S^{\mathtt{I}}[\mathtt{r}', \mathtt{r}]\langle l \rangle.h \to E^{S_0}[\{\!|\!|T|\!|\!|\}^S \triangleright \langle \mathtt{r}'?l \rangle; T']; s[\mathtt{r}] : h$

(EIn2) assuming $\neg\varphi(\Sigma, S)$,
$\prod_{1 \le i \le n} s[\mathtt{r}_i] : E_i^{S_i}[\{|\!|T_i|\!|\!|\}^{\text{-}} \triangleright \langle \mathtt{r}'?l \rangle; \_], s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l \rangle; T']; \Sigma, s[\mathtt{r}] : S^{\mathtt{I}}[\mathtt{r}', \mathtt{r}]\langle l \rangle.h$
$\to \prod_{1 \le i \le n} s[\mathtt{r}_i] : E_i^{S_i}[\{|\mathtt{Eend}|\}^{\text{-}} \triangleright \langle \mathtt{r}'?l \rangle; \_], s[\mathtt{r}] : E^{S_0}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l \rangle; T']; \Sigma, s[\mathtt{r}] : h$

(EDisc1) assuming $\varphi(\Sigma, k_1)$
$s[\mathtt{r}] : E^{S_0}[\{\!|\!|E^{\text{-}}[\{|T|\}^{k_1} \triangleright \langle \mathtt{r}'?l \rangle; T']|\!|\!|\}^S \triangleright \langle \mathtt{r}''?l'' \rangle; \_]; \Sigma, s[\mathtt{r}] : k_1^{\mathtt{I}}[\mathtt{r}', \mathtt{r}]\langle l \rangle.h$
$\to s[\mathtt{r}] : E^{S_0}[\{\!|\!|E^{\text{-}}[\{\!|\!|T|\!|\!|\}^{k_1} \triangleright \langle \mathtt{r}'?l \rangle; T']|\!|\!|\}^S \triangleright \langle \mathtt{r}''?l'' \rangle; \_]; \Sigma, s[\mathtt{r}] : h$

(EDisc2) assuming $\neg\varphi(\Sigma, k_1)$
$s[\mathtt{r}] : E^{S_0}[\{\!|\!|E^S[\{|T|\}^{k_1} \triangleright \langle \mathtt{r}'?l \rangle; T']|\!|\!|\}^S \triangleright \langle \mathtt{r}''?l'' \rangle; \_],$
$\quad \prod_{1 \le i \le n} s[\mathtt{r}_i] : E_1{}^S[\{\!|\!|T_i|\!|\!|\}^{k_1} \triangleright \langle \mathtt{r}'?l \rangle; T_i']; \Sigma, s[\mathtt{r}] : S_i^{\mathtt{I}}[\mathtt{r}', \mathtt{r}]\langle l \rangle.h$
$\to \prod_{1 \le i \le n} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{|\mathtt{Eend}|\}^{S_i} \triangleright \langle \mathtt{r}'?l \rangle; T_i],$
$\quad s[\mathtt{r}] : E^{S_0}[\{\!|\!|E^S[\{|\mathtt{Eend}|\}^{k_1} \triangleright \langle \mathtt{r}'?l \rangle; T']|\!|\!|\}^S \triangleright \langle \mathtt{r}''?l'' \rangle; \_]; \Sigma, s[\mathtt{r}] : h$

For rule corresponding to (Disc), we reduce the memory instead of discarding the message. For rules corresponding to (EIn) and (EDisc), in both cases, we do a discussion on whether the exception corresponding to the message is "ongoing" or not. If it is the case, it means other exception messages for the same scope still exist in queue, thus we annotate the type in the scope (which would have been discarded) as a memory type, in order to remember it. If the exception message was the last one from its scope, then we remove the whole memory for this exception by replacing every corresponding memory (in every types) with `Eend`.

It is easy to see that Δ, Σ simulates $\mathit{Erase}(\Delta), \Sigma$ and that $\mathit{Erase}()$ preserves the intermediate correspondence w.r.t. $G_1, \ldots, G_n$. Thus in the following we will work with memory annotated configurations, which are useful because they remember what local type has been discarded by an exception as long as the type has not need discarded for every participant of the scope.

Main proof Thus Theorem 1 states that if a configuration corresponds to $G_1, \ldots, G_n$ and makes some reduction steps, we can let it make other steps to reach a configuration that corresponds to some derivatives of $G_1, \ldots, G_n$. The intermediate configurations correspond to the situation where messages are exchanged through queues.

Theorem 1 (Session fidelity).

If Δ corresponds to $G_1, \ldots, G_n$ and $\Delta, \varepsilon \to^* \Delta', \Sigma'$, there exists $\Delta', \Sigma' \to^* \Delta'', \varepsilon$ such that Δ'' corresponds to $G_1'', \ldots, G_n''$ which is a derivative of $G_1, \ldots, G_n$.

Proof. We prove that if there is an intermediate correspondence between Δ, Σ and $G_1, \ldots, G_n$ and if $\Delta, \varepsilon \to \Delta', \Sigma'$, then there is an intermediate correspondence Δ'' and $G_1'', \ldots, G_n''$ which is a derivative of $G_1, \ldots, G_n$.

We use Ω, Θ alongside Δ to denote session environment. According to the derivative definition above, we extend the notion of evaluation contexts to global types.

Case (Out) is trivial from the first rule of intermediate correspondence.

Case (EOut). We have $\Delta = \Theta, s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T']$. Correspondence gives $\Delta = \Theta_0, s[\mathtt{r}] : E^{S_0}[\{|T|\}^S \triangleright \langle \mathtt{r}?l\rangle; T'], \prod_{1\leq i\leq n} s[\mathtt{r}_i] : E_i^{S_i}[\{|T_i|\}^S \triangleright \langle \mathtt{r}?l\rangle; T_i'] \; \Sigma = \Sigma_1, \Sigma_0$ with $\Theta_0; \Sigma_0$ corresponding to $G_2, \ldots, G_n$ and $(\Delta - \Theta'); \Sigma_1$ corresponding to G_1. We know that $\Sigma' = \Sigma_0, \prod_{1\leq i\leq n} s[\mathtt{r}_i] : h_i.S^{\mathrm{I}}[\mathtt{r}, \mathtt{r}_i]\langle l\rangle$. Concluding is easy using the second rule of intermediate correspondence with $k = 0$.

Case (In). We assume $\Delta = \Theta, s[\mathtt{r}'] : E^S[\mathtt{r}?\{l_i.T_i\}]$ and $\Sigma = \Sigma_0, s[\mathtt{r}'] : h.S[\mathtt{r}, \mathtt{r}']\langle l_j\rangle$. We know there exists $G_1, \ldots G_n$ and Δ_0 such that $\Delta_0 = \Theta_1, \ldots, \Theta_n$ with $\Theta_i = \bigcup_{\mathtt{r}\in G_i} G_i \upharpoonright^{\mathtt{r}}$. Without loss of generality we have $\Theta_1 = s[\mathtt{r}'] : E^S[\mathtt{r}?\{l_i.T_i\}], \Theta_1'$. By the rules of projection, it means $G_1 = \mathtt{r}\to\mathtt{r}' : \{l_j.G_j\}_{j\in J}$, implying $\Theta_1' = s[\mathtt{r}'] : E^{S'}[\mathtt{r}?\{l_i.T_i\}], \Theta_1''$. So we have $\Delta' = \Omega, s[RR] : E^S[T_j], s[\mathtt{r}'] : E^{S'}[\mathtt{r}?\{l_i.T_i\}], \Theta_1'$ and $\Sigma' = \Sigma_0, s[\mathtt{r}'] : h.S[\mathtt{r}, \mathtt{r}']\langle l_j\rangle$. We apply use (In) to conclude, using the projection rule on G_j.

Case (EIn_1). We pose $\mathtt{r} = \mathtt{r}_{k+1}$. We have $\Sigma = \Sigma', S^{\mathrm{I}}[\mathtt{r}_0, \mathtt{r}_{k+1}]\langle l\rangle.h$ and $\varphi(\Sigma', S)$. Then let us define $\Delta = \Theta, s[\mathtt{r}_{k+1}] : E^{S_0}[\{|T_{k+1}|\}^S \triangleright \langle \mathtt{r}_0?l\rangle; T']$ and $\Delta' = \Theta$, $s[\mathtt{r}_{k+1}]E^{S_0}[\{\|T\|\}^S \triangleright \langle \mathtt{r}_0?l\rangle; T']$. Without loss of generality we suppose $\Delta; \Sigma$ corresponds to G. We deduce that $G = F\{|G_0|\}^S\langle l \text{ by } \mathtt{r}\rangle; G'_{\_}$ and $\Delta = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{1\leq i\leq k} s[\mathtt{r}_i] : E_i^{\text{-}}[\{\|T_i\|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{k+1\leq j\leq n} s[\mathtt{r}_j] : E_i^{\text{-}}[\{|T_j|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_]$ and $\Sigma = \Sigma_0, \prod_{k+1\leq j\leq n} s[\mathtt{r}_j] : S^{\mathrm{I}}[\mathtt{r}, \mathtt{r}_j]\langle l\rangle.h_j$. Thus we have $\Delta' = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{1\leq i\leq k+1} s[\mathtt{r}_i] : E_i^{\text{-}}[\{\|T_i\|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{k+2\leq j\leq n} s[\mathtt{r}_j] : E_i^{\text{-}}[\{|T_j|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_]$ and $\Sigma' = \Sigma_0, \prod_{k+2\leq j\leq n} s[\mathtt{r}_j] : S^{\mathrm{I}}[\mathtt{r}, \mathtt{r}_j]\langle l\rangle.h_j$. We conclude using the second definition of intermediate correspondence with $k = k+1$.

Case (EIn_2). We pose $\mathtt{r}_n = \mathtt{r}$, we have $\Sigma = \Sigma', S^{\mathrm{I}}[\mathtt{r}_0, \mathtt{r}_n]\langle l\rangle.h$ and $\neg\varphi(\Sigma', S)$. Then $\Delta = \Theta, s[\mathtt{r}_n] : E^{S_0}[\{|T_n|\}^S \triangleright \langle \mathtt{r}_0?l\rangle; T']$ and $\Delta' = \Theta, s[\mathtt{r}_{k+1}]E^{S_0}[\{\|T\|\}^S \triangleright \langle \mathtt{r}_0?l\rangle; T']$. Without loss of generality we suppose $\Delta; \Sigma$ corresponds to G. We deduce that $G = F\{|G_0|\}^S\langle l \text{ by } \mathtt{r}\rangle; G'_{\_}$, $\Delta = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{1\leq i\leq n-1} s[\mathtt{r}_i] : E_i^{\text{-}}[\{\|T_i\|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], s[\mathtt{r}_n] : E_n^{\text{-}}[\{|T_n|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_]$ and $\Sigma = \Sigma_0, s[\mathtt{r}_n] : S^{\mathrm{I}}[\mathtt{r}, \mathtt{r}_n]\langle l\rangle.h$. From the semantics, we also have $\Delta' = s[\mathtt{r}_0] : E^{\text{-}}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_], \prod_{1\leq i\leq n} s[\mathtt{r}_i] : E_i^{\text{-}}[\{|\mathtt{Eend}|\}^S \triangleright \langle \mathtt{r}'?l\rangle; \_]$ and $\Sigma = \Sigma_0, s[\mathtt{r}_n] : h$. We use the hypothesis and the intermediate correspondence rule to prove that $\Delta'; \Sigma'$ corresponds to $F\{|\mathtt{Eend}|\}^S\langle l \text{ by } \mathtt{r}\rangle; G'_{\_}$ which is a derivative of G. We conclude.

Case (Disc') is easy using the first definition of the intermediate correspondence.

Case (EDisc_1) is very similar to (EIn_1).

Case (EDisc_2): is very similar to (EIn_2).

We prove the following progress property: if Δ, Σ is in intermediate correspondence with $G_1, \ldots, G_n$, and $\Sigma \neq \varepsilon$ then there exist Δ', Σ' with Σ' strictly smaller than Σ. We prove it as follows:

- if Σ contains $S[\mathtt{r}, \mathtt{r}']\langle l\rangle$, we use the weak projection definitions to prove that Δ contains either $s[\mathtt{r}'] : E^S[\mathtt{r}?\{l_i.T_i\}]$ or $s[\mathtt{r}'] : E^{S'}[\{\|T\|\}^]\triangleright \langle \mathtt{r}?l\rangle; \_$ with T containing $\mathtt{r}?\{l_i.T_i\}$. We conclude by applying (In) or (Disc').
- otherwise Σ contains $S^{\mathrm{I}}[\mathtt{r}, \mathtt{r}']\langle l\rangle$ and we use the intermediate correspondence definition to discuss whether S is inside an interrupted scope or not and then whether $\varphi(\Sigma_0,)$ or not, the we conclude by applying ($\mathsf{EIn1}$), ($\mathsf{EIn2}$), ($\mathsf{EDisc1}$) or ($\mathsf{EDisc2}$).

We use both properties to conclude.

A.5 Example

In Figure 14, we describe a formal global type which corresponds to the Scribble protocol in Figure 2. It requires to enrich the syntax with interruptible constructs accepting two interrupt messages, which can be performed either by slightly updating the semantics or by encoding the example into two nested interruptible constructs. The formal global type G_{ResCont} is very closed to its Scribble counterpart in Figure 2. The main difference comes from the explicit naming of the scopes (here, S_1 and S_2). Remember, as stated above, that our types are equi-recursive and every scope annotation has to be different, so in this representation S_2 actually stands for an infinite set of scopes $(S_2^i)_{i\geq 0}$, one for every unfolding of the recursion of X.

$$
\begin{aligned}
G_{\texttt{ResCont}} = {} & \texttt{U}\to\texttt{C}:\text{req};\texttt{C}\to\texttt{A}:\text{start} \\
& \{|\mu X. \\
& \quad \{|\mu Y.\texttt{A}\to\texttt{U}:\text{data};Y|\}^{S_2}\langle\text{pause by }\texttt{U}\rangle; \\
& \quad \texttt{U}\to\texttt{A}:\text{resume};X \\
& |\}^{S_1}\langle\text{stop by }\texttt{U},\text{timeout by }\texttt{C}\rangle;\texttt{end}
\end{aligned}
$$

Fig. 14: Global type for Figure 2