Noname manuscript No.
(will be inserted by the editor)

Practical interruptible conversations

Distributed dynamic verification with multiparty session types and Python

Romain Demangeon · Kohei Honda · Raymond Hu · Rumyana Neykova · Nobuko Yoshida

the date of receipt and acceptance should be inserted later

Abstract The rigorous and comprehensive verification of communication-based software is an important engineering challenge in distributed systems. Drawn from our industrial collaborations [40, 34] on Scribble, a choreography description language based on multiparty session types, and its theoretical foundations [20], this article proposes a dynamic verification framework for structured interruptible conversation programming. We first present our extension of Scribble to support the specification of asynchronously interruptible conversations. We then implement a concise API for conversation programming with interrupts in Python that enables session types properties to be dynamically verified for distributed processes. Finally, we expose the underlying theory of our interrupt mechanism, studying its syntax and semantics, its integration in MPST theory and proving the correctness of our design. Our framework ensures the global safety of a system in the presence of asynchronous interrupts through independent runtime monitoring of each endpoint, checking the conformance of the local execution trace to 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-

R. Demangeon
Sorbonne Universités, UPMC, Univ Paris 06, UMR 7606, LIP6 F-75005, Paris, France,
E-mail: romain.demangeon@lip6.fr

K.Honda
Queen Mary, University of London

R. Hu
Imperial College London,
E-mail: raymond.hu05@imperial.ac.uk

R. Neykova
Imperial College London
E-mail: rumyana.neykova.10@imperial.ac.uk

N. Yoshida
Imperial College London,
E-mail: n.yoshida@imperial.ac.uk

based protocols, without introducing any implicit synchronisations. Benchmarks show conversation programming and monitoring can be realised with little overhead.

Keywords session types, runtime monitoring, python, distributed systems

1 Introduction

Two of the most important elements of software development are finding suitable specifications to model the range of states exhibited by a system, and ensuring that these specifications are followed by the implementation. In distributed systems, the rigorous specification and verification of communication protocols is particularly crucial: a protocol is the interface to which the components should be separately implementable, such that their composition as a concurrent, asynchronous system is still ensured to be correct as a whole. Multiparty session types (MPST) [20,6] is a type theory for communication-oriented programming, originating from works on types for the π-calculus, towards tackling these challenges. 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.

This article presents two main contributions towards the application of MPST theory to current engineering practices, developed from our collaborations with industry partners [40, 34]. The first is the design and implementation of a framework for *dynamic, distributed verification of MPST* message passing protocols using session endpoint monitors. We have been motivated to adapt MPST to dynamic verification for several reasons:

- Session type checking is typically designed for languages with first-class communication and concurrency primitives, whereas our collaborations use mainstream engineering languages such as Python and Java. These languages either lack the features required to make static session typing tractable, or, in the case of dynamically typed languages like Python, may simply be unsuited to this approach. Certain programming techniques can further complicate static analysis; for example, the obfuscation of control flow in event-driven programming, a common paradigm in distributed systems.
- Distributed systems are often heterogeneous in nature, meaning that a range of languages and platforms may be involved in the implementation of a given system, as well as third-party components or services for which the source code is unavailable for static type checking. Dynamic verification by communications monitoring allows us to verify MPST safety properties directly for mainstream languages in a more scalable way.
- Certain protocol specification features, such as assertions on specific message values, can be evaluated precisely at runtime, whereas static treatments would often be more conservative.

This article secondly presents the implementation and formalisation of a new construct for verifying *asynchronous multiparty session interrupts*, motivated by use cases from our collaborations. 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. Previous attempts at incorporating exception-like constructs into session type theory have been limited in their practical application and cannot express our use case patterns: [13] is

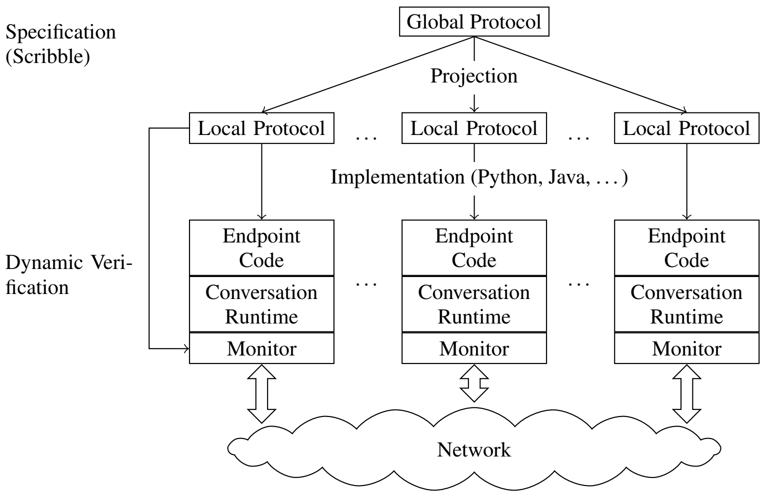


Fig. 1: Scribble methodology from global specification to local runtime verification

restricted to binary session types, [13, 12] do not support nested interrupts or continuations, and [11], although multiparty, relies on synchronous exception flags which are not feasible in general distributed systems.

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 article 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 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. Note that tracking the local protocol state is a core function of the session monitors for dynamic MPST, forming an underlying technical connection between these two topics.

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 [41], an engineering incarnation of formal MPST types. The core features of Scribble include multicast message passing and constructs for branching (choice), recursive and parallel conversations. These features support the specification of a wide range of protocols, from domains such as standard Internet applications [21], parallel algorithms [33] and Web services [14].

Our toolchain validates that the global protocol satisfies MPST 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 a Scribble *local protocol* (called a *projection*) for each participant (abstracted as a *role*) that is involved. A local protocol is essentially a view of the global protocol from the perspective of one role, and provides a more direct and focused specification for endpoint implementation than the global protocol.

As a session is conducted at run-time, the monitor at each endpoint uses a finite state machine (FSM) representation of the local communication behaviour, generated from the local protocol for its role, to track its progress in the session. In our implementation, the FSM generation is an extension of the correspondence between MPST and communication automata in [16] to support interruptible protocol scopes and optimised to avoid parallel state explosion. The monitor validates the communication actions performed by the local endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the FSM. Each monitor thus works to protect both the endpoint from invalid actions by the network environment, and the network from bad endpoints. Interestingly, we treat both interruptible scopes and parallel subprotocols 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) corresponding to the generation of fresh scope names in the syntactic model.

Our dynamic MPST framework is designed in this way to ensure, from the decentralised monitoring of each local endpoint, that the progress of the session as a whole conforms to the original global protocol [7], and that unsafe actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints. We have integrated our framework into the Python-based runtime platform developed by the Ocean Observatories Initiative (OOI) [34]. 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 [36]) and distributed runtime monitoring to regulate the behaviour of third-party applications within the system [37]. 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. In summary, this article demonstrates the application of multiparty session types, through the Scribble protocol language, to industry practice by presenting:

- the implementation of an MPST-based dynamic, distrubuted protocol verification that offers the same global safety guarantees as static session type checking;
- and a use case motivated extension of Scribble to support a new MPST construct for the verification of asynchronous multiparty session interrupts.

This article is an expansion of the initial presentation of our dynamic MPST framework and implementation of monitors for interruptible sessions in [22]. Apart from the expanded introduction and related work, this article presents the full formalisation and proof of correctness of our extension of MPST with the new interruptible construct and scope mechanism for resolving the protocol race conditions that arise from asynchronous interrupts. We also give an additional event-driven implementation of the main example, to demonstrate our Python Conversation API for event-driven sessions and the benefit of dynamic MPST verification to support flexible implementations that would otherwise be difficult to statically verify.

The OOI use cases motivating this work include a variety of RPC-based service calls (request-reply) with timeout interrupts, and publish-subscribe applications where the consumer or other parties can interrupt to pause, resume and stop remotely driven sensor feeds; we use the latter for the main running example in this article. Although the existing features of Scribble (i.e. those previously established in MPST theory) are sufficiently expressive for many practical protocols, we observed that these important patterns could not be directly or naturally represented without interrupts.

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

§ 2 explains an OOI use case for the extension of Scribble with asynchronous session interrupts. This is a new feature for MPST, giving the first general mechanism for nested, multiparty interrupts. We discuss why adding this feature is a challenge in session types.

§ 3 discusses the Python implementation of our dynamic MPST framework that we have integrated into the OOI project, and demonstrates the global-to-local projection of interruptible Scribble protocols, endpoint implementations, and local FSM generation for monitoring. § 3.1 demonstrates the Python API for conversation programming in Python, including event-driven conversations. The API decorates conversation messages with the run-time MPST information required by the monitors to perform the dynamic verification. § 3.2 discusses the monitor implementation, focusing on the key architectural requirements of our framework and the treatment of asynchronous interrupts.

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

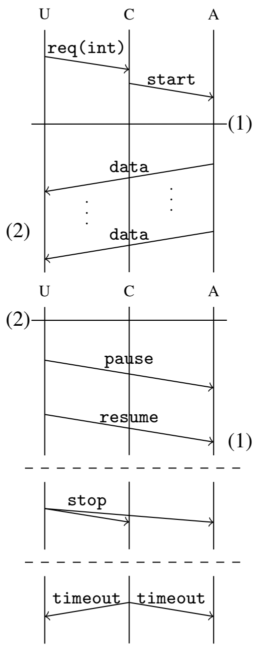
§ 5 presents the supporting theory for asynchronous session interrupts. We show the soundness of our framework by proving session fidelity, asserting that the decentralised verification of a system always conforms to its global specification.

The source code of our Scribble toolchain, conversation runtime and monitor, performance benchmarks and further resources are available from the project page [42].

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 cyberinfrastructure team of the OOI project [34] manages communication protocol specifications through a combination of informal sequence diagrams and prose descriptions. Figure 2 (left)



```
global protocol ResourceAccessControl(role User as U,
            role Controller as C, role Agent as A) {
    req(duration:int) from U to C;
        // U requests the device for some duration
    start() from C to A;
    interruptible { // U, C and A in scope
        rec X {
            interruptible { // U and A in scope
                rec Y {
                    data() from A to U;
                    continue Y;
                }
            } with { // Interrupts A in Y
                pause() by U;
            }
            resume() from U to A;
            continue X;
        }
    } with { // Interrupts A and C/U in X
        stop() by U; // Any time within the duration
        timeout() by C; // Duration is up
    }
}
```

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

gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control use case [36], regarding access control of users to sensor devices in the ION cyberinfrastucture 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 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).

Interruptible multiparty session types. Figure 2 (right) shows a Scribble protocol that formally captures the structure of interaction in the Resource Access Control use case and demonstrates the uses of our new extension for asynchronous interrupts. 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 of 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`, carrying a *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 in our example is clarified in Scribble as two nested `interruptible` statements. The outer statement, on lines 6–22, corresponds to the options for User and Controller 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 19–22. 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 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 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.

Challenges of asynchronous interrupts in MPST. The following summarises our observations from the extension and usage of MPST with asynchronous interrupts. We find the

```
// Well-formed, but incorrect semantics:
// the recursion cannot be stopped
par {
    rec Y {
        data() from A to U;
        continue Y; }
} and {
    // Does not stop the recursion
    pause() from U to A;
}
resume() from U to A;
```

```
// Naive mixed-choice is not well-formed
choice at A {
   // A should make the choice..
   rec Y {
        data() from A to U;
        continue Y; }
} or {
    // ..not U
    pause() from U to A;
}
resume() from U to A;
```

Fig. 3: Naive, incorrect interruptible encoding attempts using parallel (left) and choice (right)

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, in order to prevent communication races and thereby ensure the desired safety properties.

A key 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`). Although standard MPST (and Scribble) support parallel protocol flows, the interesting point here is that the nature of an interrupt is to preclude further actions in another parallel flow under the control of a different role, whereas the basic MPST parallel does not permit such interference. Figure 3 (left) is a naively incorrect attempt to express this aspect without interruptible: the second parallel path is never able to intefere with the first to actually stop the recursion.

Another 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 protocol) due to multiparty. Moreover, the implicit interrupt choice is truly optional in the sense that it may never be selected at runtime. The basic choice in standard MPST (e.g. as defined in [20, 16]) is inadequate because it is designed to safely identify a single role as the decision maker, who communicates exactly one of a set of message choices unambiguously to all relevant roles. Figure 3 (right) demonstrates a naive mixed choice that is not well-formed (it breaks the unique sender condition in [16]).

Due to the asynchronous setting, it is also important that `interruptible` does not require implicit synchronisations to preserve communication safety. The underlying mechanisms are formalised and the correctness of our extension is proved in § 5.

3 Runtime verification

This section discusses implementation details of our monitoring framework and the accompanying Python API (Conversation API) for writing monitorable, distributed MPST programs. This work is the first implementation of the theory in [7] in practice, and is the first (theory or practice) to support a general, asynchronous MPST interrupt mechanism in the protocol language and API for endpoint implementation.

Conversation API operation	Purpose
`create(protocol_name, invitation_config.yml)`	Initiate conversation, send invitations
`join(self, role, principal_name)`	Accept invitation
`send(role, op, payload)`	Send message with operation and payload
`recv(role)`	Receive message from role
`recv_async(self, role, callback)`	Asynchronous receive
`scope(msg)`	Create a conversation scope
`close()`	Close the connection to the conversation

Fig. 4: The core Python Conversation API operations

We first outline the verification methodology of our framework to clarify the purpose of the main components. Developers write endpoint programs in native Python using the Conversation API, an MPST-based message passing library that supports the core MPST primitives for communication programming. The execution of these operations at each endpoint is performed by the local conversation library runtime. The full runtime includes infrastructure for inline monitoring of conversation actions, while the lightweight version is used with an outline (i.e. externally hosted) monitor. In both cases, the API enables MPST verification of message exchanges by the monitor by embedding a small amount of MPST meta data (e.g. conversation identifier, message kind and operator, source and destination roles), based on the actions and current state of the endpoint, into the message payload. For each conversation initiated or joined by an endpoint, the monitor generates an FSM from the local protocol for the role of the endpoint. The monitor uses the FSM to track the progress of this conversation according to the protocol, validating each message (via the meta data) as it is sent or received.

3.1 Conversation API

The Python Conversation API offers a high-level interface for safe conversation programming, mapping the interaction primitives of session types to lower-level communication actions on concrete transports. Our current implementation is built over an AMQP [2] transport. In summary, the API provides the functionality for (1) session initiation and joining, (2) basic send/receive and (3) *conversation scope* management for handling interrupt messages. Figure 4 lists the core API operations. The invitation operations (`create` and `join`) have not been captured in standard MPST systems, but have formal counterparts in the literature in formalisms such as [13].

We demonstrate the usage of the API in a Python implementation of the local protocol projected for the User role. Figure 5 gives the local protocol and its implementation.

Conversation initiation. First, the `create` method of the Conversation API (line 6, right) initiates a new conversation instance of the `ResourceAccessControl` protocol (Figure 2), and returns a token that can be 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. The `join` method confirms that the endpoint is joining the conversation as the principal `alice` playing the role `User`, and returns a conversation channel object for performing the subsequent communication operations. Once the invitations are sent and accepted (via `Conversation.join`), the conversation is established and the intended message exchanges 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.

```
1  local protocol ResourceAccessControl
2      at User as U (role Controller as C,
3          role Agent as A) {
4    req(duration:int) to C;
5    interruptible {
6      rec X {
7        interruptible {
8          rec Y {
9            data() from A;
10           continue Y;
11         }
12       } with {
13         pause() by U;
14       }
15       resume() to A;
16       continue X;
17     }
18   } with {
19     stop() by U;
20     timeout() by C;
21   }
22 }
```

```
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      # request 1 hour access
10     c.send(controller, 'req', 3600)
11     with c.scope('timeout', 'stop') as c_x:
12       while not self.should_stop():
13         with c_x.scope('pause') as c_y:
14           while not self.buffer.is_full():
15             data = c_y.recv(agent)
16             self.buffer.append(data)
17           c_y.send_interrupt('pause')
18         use_data(self.buffer)
19         self.buffer.clear()
20         c_x.send(agent, 'resume')
21       c_x.send_interrupt('stop')
22     c.close()
```

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

Conversation message passing. Following its local protocol, the User program sends a request to the `controller`, stating the duration for which it requires access to `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 embedded into the message payload as part of the conversation meta data, and is later used by the monitor in the runtime verification. The `recv` method can take the source role as a single argument, or additionally the operator of the desired message. Send is asynchronous, meaning that the operation does not block on the corresponding receive; however, the basic receive does block until the complete message has been received.

Interrupt handling via conversation scopes. This example demonstrates a way of handling conversation interrupts by combining conversation scopes 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 11, 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 to 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 11–21); if, e.g., a `timeout` is received within this scope, the control flow will exit the `with` to line 22. The inner `with` (lines 13–17), corresponding to the inner interruptible block, is associated with the `pause` interrupt. When an interrupt, e.g. `pause` in line 17, 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

```
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')
```

```
16     elif self.buffer.is_full():
17       self.process_buffer(c, payload)
18     else:
19       self.buffer.append(payload)
20       c.recv_async(agent, recv_handler)
21
22   def process_buffer(self, c, payload):
23     with c:
24       c_x = c.send_interrupt('pause')
25       use_data(self.buffer, payload)
26       self.buffer.clear()
27       c_x.send(agent, 'resume')
28       c_y = c_x.scope('pause')
29       c_y.resv_async(agent, recv_handler)
```

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

interrupted control flows to be coordinated between distributed participants in a structured manner.

The scope wrapper channels are closed (via the `with`) after throwing or handling an interrupt message. For example, using `c_x` after a `timeout` is received (i.e. outside its parent scope) will be flagged as an error. By identifying the scope of every message from its meta data, the conversation runtime (and monitor) is able to compensate for the inherent discrepancies in protocol synchronisation, due to asynchronous interrupts between distributed endpoints, by safely discarding out-of-scope messages. In our example, the User runtime discards `data` messages that arrive after `pause` is thrown. To prevent the loss of such messages in the application logic when the stream is resumed, we could extend the protocol to simply carry the id of the last received data element in the payload of the `resume` (in line 20). The API can also make the discarded data available to the programmer through secondary (non-monitored) operations.

Event-driven conversations For asynchronous, non-blocking receives, the Conversation API provides `recv_async` to be used in an event-driven style. Figure 6 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 (`recv_handler`) 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 24 would be considered invalid by the monitor. When the buffer is full (Line 16), 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 27) and a fresh scope is created and again registered for receiving data events (Line 28). Any occurence of the `timeout` interrupt is handled implicitly by the `with` statements, which will close the conversation and clean up the associated resources. Although the event-driven API promotes a notably different programming style, our framework monitors both this implementation and that in Figure 5 transparently without any modifications.

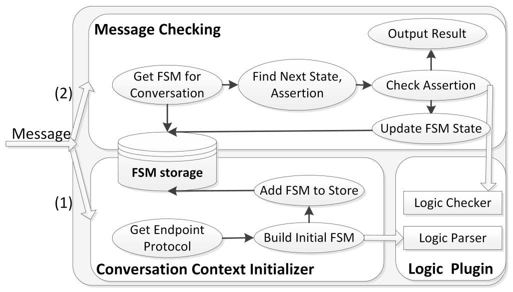


Fig. 7: Monitor workflow for (1) invitation and (2) in-conversation messages

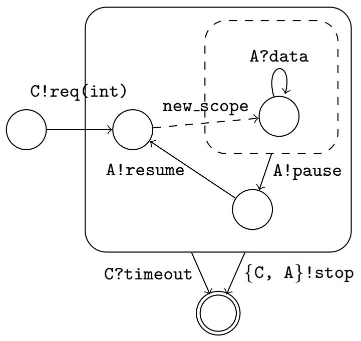


Fig. 8: Nested FSM generated from the User local protocol

3.2 Monitoring architecture

Inline and outline monitoring. In order 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 [7]) is compatible with a range of monitor configurations. At one end of the spectrum is *inline monitoring*, where the monitor is embedded into the endpoint code. Then there are various configurations for *outline monitoring*, where the monitor is positioned externally to its component. In the OOI project, our focus has been to integrate our framework for inline monitoring due to the architecture of the OOI message interceptor stack [37].

Monitor implementation. Figure 7 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 (global protocols can also be exchanged if the monitor has the facility for projection.)

We use a parser generator (ANTLR) 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 an infinitive integer generator.

When a *send/receive* node is visited, two states are generated, and a linking transition of type send or receive is added to the transition table. Dummy states are created on entering a *choice* subtree and then again for each of the choice branches. An empty transition connects a dummy *initial choice* state and *branch* states. A recursion is handled by keeping a mapping between a *recursion label* and its *recursion start* state. Processing a node *continue* node results in an empty transition between a current state and a corresponding recursion start state.

The algorithm for generating FSM from a MPST protocol is presented formally in [16]. Our implementation differs from [16] in the treatment of parallel sub-protocols (i.e. unordered message sequences), and additionally supports interrupts. For efficiency, we extend [16] to generate a nested FSM for each conversation thread, avoiding the potential state explosion that comes from constructing their product. This allows FSM generation in polynomial time and space in the length of the local protocol. The (nested) FSMs are stored in a hash table with conversation id as the key. Due to standard MPST well-formedness (message label distinction), any nested FSM is uniquely identifiable from any unordered message, i.e. message-to-transition matching in a conversation FSM is deterministic.

The upper part of Figure 7 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 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 an external logic engine; a monitor can be configured to use various logic engines, such as for the validation of assertions, automata-based specifications (e.g. security automata), or other stateful properties. Our current implementation uses a basic Python predicate evaluator, which is sufficient for the use case protocols we have developed so far.

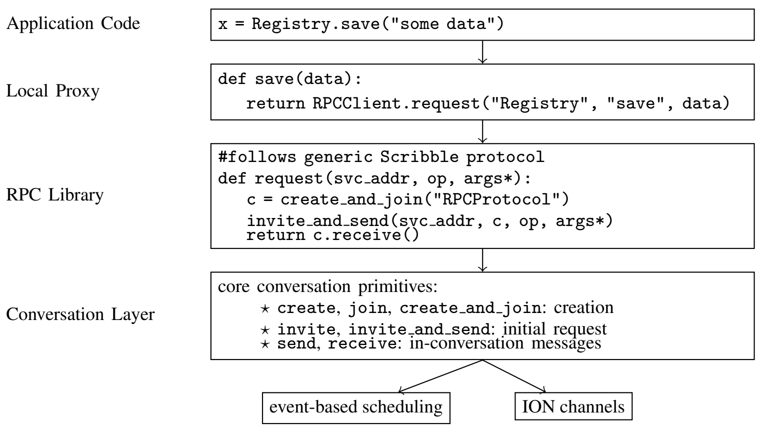
Monitoring interrupts. FSM generation for interruptible local protocols again makes use of nested FSMs. Each `interruptible` induces a nested FSM given by the main interruptible block, as illustrated in Figure 8 for the User local protocol. The monitor internally augments the nested FSM with a scope 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 is reached.

4 Evaluation

Our dynamic MPST verification framework has been implemented and integrated into the current release of the Ocean Observatories platform [35]. This section reports on our integration efforts and the performance of our framework.

4.1 Experience: OOI integration

The current release of OOI is based on a Service-Oriented Architecture, with all of the distributed system services accessible by RPC. As part of their efforts to move to agent-based systems in the next release, and to support distributed governance for more than just individual RPC calls, we engineered the following step-by-step transition. The first step was to add our Scribble monitor to the message interceptor stack of their middleware [37]. The second was to propose our conversation programming interface to the OOI developers. To facilitate the use of session types without obstructing the existing application code, we preserved the interface of the RPC libraries but replaced the underlying machinery with the distributed



```
x = Registry.save("some data")
```

```
def save(data):
   return RPCClient.request("Registry", "save", data)
```

```
#follows generic Scribble protocol
def request(svc_addr, op, args*):
   c = create_and_join("RPCProtocol")
   invite_and_send(svc_addr, c, op, args*)
   return c.receive()
```

```
core conversation primitives:
     ⋆ create, join, create_and_join: creation
     ⋆ invite, invite_and_send: initial request
     ⋆ send, receive: in-conversation messages
```

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

runtime for session types (as shown in Figure 9, the RPC library is now realised on top of the Conversation Layer). As wrappers to the conversation primitives, all RPC calls are now automatically verified by the inline MPST monitors. This approach was feasible because no changes were required to existing application code, but at the same time, developers now have the option to use the Conversation API directly for conversations more complex than RPC. The next step in this ongoing integration work involves porting higher-level and more complex OOI application protocols, such as distributed agent negotiation [36], to Scribble specifications and Conversation API implementations.

4.2 Benchmarks

The potential performance overhead that the Conversation Layer and monitoring could introduce to the system is an important consideration. The following performance measurements for the current prototype show that our framework can be realised at reasonable cost. Table 1 presents the execution time comparing RPC calls using the original OOI RPC library implementation and the conversation-based RPC with and without monitor verification. On

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

Table 1: Original OOI RPC vs. conversation-based RPC with monitoring disabled/enabled

Seq States	No-Mon (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	No-Mon (s)	Mon	
10	0.45	0.49	+8%
100	4.05	4.22	+4.1%
1000	40.16	41.24	+2.7%

Table 2: Conversation execution time for an increasing number of sequential and parallel states

Use Cases from research papers	Global Scribble (LOC)	FSM Memory (B)	Generation Time (s)
A vehicle subsystem protocol [26]	8	840	0.006
Map web-service protocol [18]	10	1040	0.010
A bidding protocol [30]	26	1544	0.020
Amazon search service [19]	12	1088	0.010
SQL service [39]	8	1936	0.009
Online shopping system [17]	10	1024	0.008
Travel booking system [17]	16	1440	0.013
Use Cases from OOI and Savara			
A purchasing protocol [25]	11	1088	0.010
A banking example [36]	16	1564	0.013
Negotiation protocol [36]	20	1320	0.014
RPC with timeout [36]	11	1016	0.013
Resource Access Control [36]	21	1854	0.018

Table 3: Use case protocols implemented in Scribble

average, 13% overhead is recorded for conversations of 10 consecutive RPCs, mostly due to the FSM generation from the textual local Scribble protocol (our implementation currently uses Python ANTLR); the cost of message validation itself is negligible in comparison. We plan to experiment with optimisations such as pre-generating or caching FSMs to reduce the monitor initialisation time.

The second benchmark gives an idea of how well our framework scales beyond basic RPC patterns. Table 2 shows that the overall verification architecture (Conversation Layer and inline monitor) scales reasonably with increasing session length (number of message exchanges) and increasing parallel states (nested FSM size): “Seq States” is the number of states passed through sequentially by a simple recursive protocol (used to parameterise the length of the conversation), and “Par States” the number of parallel states in a parallel protocol. Two benchmark cases are compared. The main case “Monitor” (Mon) is fully monitored, i.e. FSM generation and message validation are enabled for both the client and server. The base case for comparison “No Monitor” (No-Mon) has the client and server in the same configuration, but monitors are disabled (messages do not go through the interceptor stack). As above, we found that the overhead introduced by the monitor when executing conversations of increasing number of recursive and parallel states is again mostly due to the cost of the initial FSM generation. We also note that the relative overhead decreases as the session length increases, because the one-time FSM generation cost becomes less prominent. For dense FSMs, the worse case scenario results in linear overhead growth wrt. the number of parallel branches.

In both of the above tables, the presented figures are the mean time for the client and server, connected by a single-broker AMQP network, to complete one conversation after repeating the benchmark 100 times for each parameter configuration. The client and server Python processes (including the conversation runtime and monitor) and the AMQP broker were each run on separate machines (Intel Core2 Duo 2.80 GHz, 4 GB memory, 64-bit Ubuntu 11.04, kernel 2.6.38). Latency between each node was measured to be 0.24 ms on average (ping 64 bytes). The full source code of the benchmark protocols and applications and the raw data are available from the project page [42].

$$
\begin{aligned}
G_{\texttt{ResCont}} = {} & \texttt{U}\to\texttt{C}:\text{req};\texttt{C}\to\texttt{A}:\text{start}\\
& \{|\mu X.\\
& \quad \{|\mu Y.\texttt{A}\to\texttt{U}:\text{data};Y|\}^{\texttt{c}_2}\langle\text{pause by }\texttt{U}\rangle;\\
& \quad \texttt{U}\to\texttt{A}:\text{resume};X\\
& |\}^{\texttt{c}_1}\langle\text{stop by }\texttt{U},\text{timeout by }\texttt{C}\rangle;\texttt{end}
\end{aligned}
$$

Fig. 10: Global type for the Resource Access Control protocol in Figure 2

4.3 Use cases

We conclude our evaluation with some remarks on use cases we have examined. Table 3 features a list of protocols, sourced from both the research community and our industry use cases, that we have written in Scribble and used to test our monitor implementation on more realistic protocol specifications. A natural question for our methodology, being based on explicit specification of protocols, is the overhead imposed on developers wrt. writing protocols, given that a primary motivation for the development of Scribble is to reduce the design and testing effort for distributed systems. Among these use cases, we found the average Scribble global protocol is roughly 10 LOC, with the longest one at 26 LOC, suggesting that Scribble is reasonably concise.

The main factors that may affect the performance and scalability of our monitor implementation, and which depend on the shape of a protocol, are (i) the time required for the generation of FSMs and (ii) the memory overhead that may be induced by the generation of nested FSMs in case of parallel blocks and interrupts. Table 3 measures these factors for each of the listed protocols. The time required for FSM generation remains under 20 ms, measuring on average to be around 10 ms. The memory overhead also remains within reasonable boundaries (under 2.0 KB), indicating that FSM caching is a feasible optimisation approach. The full Scribble protocols can be found at [42].

From our experience of running our conversation monitoring framework within the OOI system, we expect that, in many large distributed systems, the cost of a decentralised monitoring infrastructure would be largely overshadowed by the raw cost of communication (latency, routing) and other services running at the same time. Considering the presented results, we thus believe the important benefits in terms of safety and management of high-level applications come at a reasonable cost and would be a realistic mechanism in many distributed systems.

5 Multiparty session types with asynchronous interrupts

This section 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 [20], adding syntax and semantics for interrupts.

In our theory, we manipulate global types, which correspond to session specifications and local types, which are used to express monitored behaviours of processes [7]. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that makes explicit, through an explicit identifier, the domain of interrupts.

Global session type for RAC use case. In Figure 10, to introduce the syntax of global types informally, we first show a global type which corresponds to the Scribble protocol in Figure 2. The formal syntax will be given in the next subsection. The first line denotes the two interactions at the start between the three participants. The outer loop is embedded inside a scope construct, explicitly by $\mathtt{c}_1$. The inner loop is embedded inside another scope $\mathtt{c}_2$. The information directly after the scope describes how it can be interrupted.

We insist on the fact that the formal global type G_{ResCont} is very close to its Scribble counterpart in Figure 2. The main difference comes from the explicit naming of the scopes (here, $\mathtt{c}_1$ and $\mathtt{c}_2$). Note that:

- Our types are equi-recursive and every scope annotation has to be different, so in this representation $\mathtt{c}_2$ actually stands for an infinite set of scopes $(\mathtt{c}_2^i)_{i\geq 0}$, one for every unfolding of the recursion of X.
- This example requires to enrich the syntax presented below 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.

5.1 Global and local types

We introduce a *session type* theory with: global types G which are protocols involving several participants abstracted into *roles* $\mathtt{r}$. Global types can be projected into a set of local types $T_{\mathtt{r}}$, which are considered as fragments of the global protocol seen from the point of view of a role.

Basic syntax. The inductive definition of G in the upper half of Figure 11 describe *global types*, role-based global scenarios between multiple participants as a type signature. The basic type construct is the interaction $\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}$ which stands for a message from $\mathtt{r}$ to $\mathtt{r}'$ containing a label chosen by $\mathtt{r}$ between the l_i. Each label l_i corresponds to a specific continuation G_i. Notice that, in this theory section, we do not specify the content of messages (or their type), as it is irrelevant in the semantics. $G_1 \mid G_2$ denotes the parallel composition of two protocols, $\mu\mathbf{x}.G$ and $\mathbf{x}$ are, respectively, the recursion construct and the recursion variable. $\mathtt{end}$ is the end of a protocol.

Scopes. We add to this classical framework two new constructs to handle interrupts. We use *scopes* to delimit interruptible blocks inside protocols. In types, scopes are made explicit by the use of scope variables $\mathtt{c}$. 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 § 3.2.

Interrupts. Our types feature a new interrupt mechanism by explicit interruptible scopes: we write $\{|G|\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle; G'$ to denote a creation of an interruptible block identified by scope $\mathtt{c}$, 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

$$
\begin{array}{rcl}
G & ::= & \mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I} \mid G|G \mid \{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle ; G' \mid \mu\mathbf{x}.G \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend} \\[1em]
T & ::= & \mathtt{r}!\{l_i.T_i\}_{i\in I} \mid \mathtt{r}?\{l_i.T_i\}_{i\in I} \mid T|T \mid \{\!|T|\!\}^{\mathtt{c}} \lhd \langle \mathtt{r}!l\rangle ; T' \mid \{\!|T|\!\}^{\mathtt{c}} \rhd \langle \mathtt{r}?l\rangle ; T' \\
 & \mid & \mu\mathbf{x}.T \mid \mathbf{x} \mid \mathtt{end} \mid \mathtt{Eend}
\end{array}
$$

Fig. 11: Global and local types

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

$$
\begin{array}{rcl}
(\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}) \uparrow \mathtt{r} & = & \mathtt{r}'!\{l_i.(G_i \uparrow \mathtt{r})\}_{i\in I} \\
(\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}) \uparrow \mathtt{r}' & = & \mathtt{r}?\{l_i.(G_i \uparrow \mathtt{r}')\}_{i\in I} \\
(\mathtt{r}{\to}\mathtt{r}' : \{l_i.G_i\}_{i\in I}) \uparrow \mathtt{r}_0 & = & G_1 \uparrow \mathtt{r}_0 \\
(\mu\mathbf{x}.G) \uparrow \mathtt{r}_0 & = & \mu\mathbf{x}.(G \uparrow \mathtt{r}_0) \text{ when } \mathtt{r}_0 \in G \\
(\mu\mathbf{x}.G) \uparrow \mathtt{r}_0 & = & \mathtt{end} \text{ otherwise} \\
\mathbf{x} \uparrow \mathtt{r}_0 & = & \mathbf{x} \\
\mathtt{end} \uparrow \mathtt{r}_0 & = & \mathtt{end}
\end{array}
\quad\Bigg|\quad
\begin{array}{rcl}
\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle ; G' \uparrow \mathtt{r} & = & \{\!|G \uparrow \mathtt{r}|\!\}^{\mathtt{c}} \rhd \langle \mathtt{r}?l\rangle ; G' \uparrow \mathtt{r} \\
\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}'\rangle ; G' \uparrow \mathtt{r} & = & \{\!|G \uparrow \mathtt{r}|\!\}^{\mathtt{c}} \lhd \langle \mathtt{r}'!l\rangle ; G' \uparrow \mathtt{r} \\
 & & \text{when } \mathtt{r} \in G \\
\{\!|G|\!\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}'\rangle ; G' \uparrow \mathtt{r} & = & G' \uparrow \mathtt{r} \\
 & & \text{otherwise}
\end{array}
$$

Fig. 12: Projection algorithm

(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 to denote the exceptional termination of a scope. As a result, Eend is not present in a specification and will appear at runtime, to denote that a block has been interrupted.

Local types. To represent sessions at the level of participants, we use local type T. Their syntax is presented in the lower half of Figure 11 and follows a pattern similar to the global ones: the interaction is divided into two sides: one for emitting a message $\mathtt{r}!\{l_i.T_i\}_{i\in I}$, the other for receiving a message $\mathtt{r}?\{l_i.T_i\}_{i\in I}$. Parallel composition $T \mid T$, recursion and ending constructs serve the same purpose as their global type counterparts.

For scopes, the main difference is that the interruptible operation is divided into two sides, one $\lhd$ side for the role which can send an interrupt $\{\!|T|\!\}^{\mathtt{c}} \lhd \langle \mathtt{r}!l\rangle ; T'$, and the $\rhd$ side for the roles which should expect to receive an interrupt message $\{\!|T|\!\}^{\mathtt{c}} \rhd \langle \mathtt{r}?l\rangle ; T'$.

Well-formedness. Global types are subject to some well-formedness conditions [20], 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 [20], and handling interruptible blocks introduces a unique condition: *uniqueness of scope names*, meaning that in a (equi-recursive) well-formed type, a scope name appears only once in an interruptible construct (note that, as explained above, scope names inside recursions are considered as name generators).

Projection. Figure 12 defines the projection operation $\uparrow \mathtt{r}$, which, for any participant playing a role $\mathtt{r}$ in a session G, specifies its local type. We write $\mathtt{r} \in G$ when role $\mathtt{r}$ appears in global type G either as an endpoint in an interaction (sender or receiver) or as role allowed to send an interrupt message in a scope construct.

The projection rules themselves are identical to the ones in [20] except the interrupts: an interaction is projected as a send action $\mathtt{r}'!$ of the sender side, a receive $\mathtt{r}'?$ action on the receiver side and is transparent to other roles (the well-formedness conditions from [20] allows us to do as such by ensuring that every branch is the same to these roles). When projecting types embedded inside a recursion on a role that does not appear inside the body of the recursion, we project on the end type $\mathtt{end}$.

When it comes to interruptible constructs, the projection on role $\mathtt{r}$ works as follows: 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. If $\mathtt{r}$ does not appear in the inside protocol, the projection ignores the construct and amounts to the projection on the continuation.

As an example, we give projections of our global type. As stated above, we restrict ourselves in the formal section to interruptible scopes accepting only one interrupt message (this can be encoded by two nested scopes), so we omit the timeout interruption. On role $\mathtt{U}$, projection gives:

$G_{\mathtt{ResCont}} \uparrow \mathtt{U} = \mathtt{C}!\mathrm{req}; \{|\mu X.\{|\mu Y.\mathtt{A}?\mathrm{data}.Y|\}^{\mathtt{c}_2} \triangleright \langle \mathtt{U}?\mathrm{pause}\rangle; \mathtt{A}!\mathrm{resume}.X|\}^{\mathtt{c}_1} \triangleright \langle \mathtt{U}?\mathrm{resume}\rangle;$

The two nested scopes can be interrupted by $\mathtt{U}$ (hence the $\triangleright$ symbol). Projection of the same global type on $\mathtt{A}$ would yield:

$G_{\mathtt{ResCont}} \uparrow \mathtt{A} = \mathtt{C}?\mathrm{start}; \{|\mu X.\{|\mu Y.\mathtt{U}!\mathrm{data}.Y|\}^{\mathtt{c}_2} \triangleright \langle \mathtt{U}?\mathrm{pause}\rangle; \mathtt{U}?\mathrm{resume}.X|\}^{\mathtt{c}_1} \triangleleft \langle \mathtt{U}!\mathrm{resume}\rangle;$

As $\mathtt{C}$ does not appear inside the loops (and we omit the `timeout` interrupt), the projection of $\mathtt{C}$ is:

$G_{\mathtt{ResCont}} \uparrow \mathtt{C} = \mathtt{U}?\mathrm{req}; \mathtt{A}!\mathrm{start}; \mathtt{end}$

5.2 Configurations 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.

Environments. We identify multiple sessions – possibly instances of the same global type – taking place simultaneously by a unique *session channel* $(s, k, \dots)$, mimicking the conversation id in our implementation. We use Δ to denote *session environments* which are mappings from session channels to local types, i.e.$s_1[\mathtt{r}_1] : T_1, \dots, s_n[\mathtt{r}_n] : T_n$. The session environments 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.

Messages and queues. Standard messages are explained as follows: $\mathtt{c}[\mathtt{r}, \mathtt{r}']\langle l\rangle$ meaning it appears inside scope $\mathtt{c}$, is sent from $\mathtt{r}$ to $\mathtt{r}'$ and contains label l. We also annotate messages for interrupts as in $\mathtt{c}^{\mathtt{I}}[\mathtt{r}, \mathtt{r}']\langle l\rangle$. 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 [20,15]). 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. $\Delta; \Sigma$ 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 14. In order to treat a message with its corresponding scope, we need to remember from which scope the message

$$
\begin{array}{lllll}
E^{\varepsilon} = [\,] \mid (E^{\varepsilon}|T) \mid (T|E^{\varepsilon}) \\
E^{\mathsf{c}} = \{\!|E^{\mathsf{c}}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;T' & \mid \{\!|E^{\mathsf{c}}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \lhd \langle \mathtt{r}!l\rangle;T' & \mid \{\!|E^{\varepsilon}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;T' \\
\quad\mid \{\!|E^{\varepsilon}|\!\}^{\mathsf{c}} \lhd \langle \mathtt{r}!l\rangle;T' & \mid \{\!|\mathtt{Eend}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;E^{\mathsf{c}} & \mid \{\!|\mathtt{Eend}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \lhd \langle \mathtt{r}!l\rangle;E^{\mathsf{c}} \\
\quad\mid \{\!|\mathsf{end}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;E^{\mathsf{c}} & \mid \{\!|\mathsf{end}|\!\}^{\mathsf{c}'\neq\mathsf{c}} \lhd \langle \mathtt{r}!l\rangle;E^{\mathsf{c}} & \mid E^{\mathsf{c}}|T \quad \mid \quad T|E^{\mathsf{c}}
\end{array}
$$

Fig. 13: Evaluation contexts

$$
\begin{array}{lll}
(\mathsf{Out}) & s[\mathtt{r}] : E^{\mathsf{c}}[\mathtt{r}'!\{l_i.T_i\}];s[\mathtt{r}'] : h & \to s[\mathtt{r}] : E^{\mathsf{c}}[T_i];s[\mathtt{r}'] : h.\mathsf{c}[\mathtt{r},\mathtt{r}']\langle l_i\rangle \\
(\mathsf{In}) & s[\mathtt{r}] : E^{\mathsf{c}}[\mathtt{r}'?\{l_i.T_i\}];s[\mathtt{r}] : \mathsf{c}[\mathtt{r}',\mathtt{r}]\langle l_i\rangle.h & \to s[\mathtt{r}] : E^{\mathsf{c}}[T_i];s[\mathtt{r}] : h \\
(\mathsf{EOut}) & s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|T|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;T'];s[\mathtt{r}_1] : h,\ldots,s[\mathtt{r}_n] : h & \\
 & \quad \to s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}?l\rangle;T'];s[\mathtt{r}_1] : \mathsf{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_1]\langle l\rangle.h,\ldots,s[\mathtt{r}_n] : \mathsf{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_n]\langle l\rangle.h & \\
(\mathsf{EIn}) & s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|T|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : h.\mathsf{c}^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h & \\
 & \quad \to s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : h & \\
(\mathsf{Disc}) & s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : \mathsf{c}_1[\mathtt{r}',\mathtt{r}]\langle l\rangle.h & \\
 & \quad \to s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : h & \\
(\mathsf{EDisc}) & s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : \mathsf{c}_1^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h & \\
 & \quad \to s[\mathtt{r}] : E^{\mathsf{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathsf{c}} \rhd \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 & \text{if } \Delta;\Sigma \to \Delta';\Sigma'
\end{array}
$$

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

Fig. 14: Reduction semantics for a specification

was sent. To this purpose, we enrich the definition of scopes with ε the empty scope and add a scope annotation on contexts. Evaluation contexts are defined in Figure 13.

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

Semantics. The reduction semantics is defined w.r.t. a *scope environment* $\Gamma = \mathscr{T},\mathscr{R}$ composed of a *scope table*

$$\mathscr{T} ::= \varepsilon \ \mid\ \mathsf{c} : \{\mathtt{r}_1,\ldots,\mathtt{r}_n\},\mathscr{T}$$

and a *scope order* which is the reflexive and transitive closure of the relation given by: $\mathsf{c}_1\ \mathscr{R}\ \mathsf{c}_2$ whenever a global type contains $E^{\mathsf{c}_1}[\{\!|G|\!\}^{\mathsf{c}_2}\langle l \text{ by } \mathtt{r}\rangle;G']$. The scope table keeps a track of every participant in a scope and the scope order keeps track of scope nesting (when $\mathsf{c}_1\ \mathscr{R}\ \mathsf{c}_2$ it means that scope c_2 is inside scope c_1). We note $\Gamma(\mathsf{c}) = \{\mathtt{r}_1,\ldots,\mathtt{r}_n\}$ whenever $\Gamma = \mathscr{T},\mathscr{R}$ and $\mathscr{T}$ contains $\mathsf{c} : \{\mathtt{r}_1,\ldots,\mathtt{r}_n\}$. The environment is omitted when not necessary.

Semantics rules in Figure 14 are as follows: in (Out), an output from $\mathtt{r}$ to $\mathtt{r}'$ appearing inside the scope c 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 c. 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 c is interrupted by $\mathtt{r}$, which replaces T by $\mathtt{Eend}$ and places an interrupt message in the queues of each participant of scope c (we need the table from Γ). Conversely in rule (EIn), an interrupt message for scope c is consumed to exceptionally terminates the type T inside scope c. Rule (Disc) discards an incoming message to scope c_1 nested inside scope c if the latter has already been exceptionally terminated (we need the scope order from Γ). Rule (EDisc) performs the same thing for exceptional messages. The three points highlighted in the implementation 3.1 are treated in the theory as follows: (1) scopes are explicitly present in messages (2) interrupt messages can be fired at any time from the global queue (see rule (EIn)), however, using a single-queue system prevent us from giving them priority (it would otherwise lead to) and (3) scope nesting is handled by the definition of evaluation contexts (which depends on scopes and includes scope nesting in their structures).

Remarks on semantics. Regarding to the semantics, we have two remarks. Most of existing theoretical works such as [20] 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 participants of a scope are finished the whole interrupt construct can be garbage collected. Both these facilities can be integrated into the current semantics.

The correctness of our theory is ensured by Theorem 5.3. This theorem states the following property:

> *Session Fidelity:* 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 [7], that the introduction of interruptible blocks to the syntax and semantics yields a sound theory.

Correspondence. First, we define configuration correspondence: a configuration Δ, Σ *corresponds to* a collection of global types $G_1, \dots, G_l$ 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 $\rightarrow$.

Derivative. 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:

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

5.3 Type memory

In order to reconstruct a global type from a configuration, we need to remember what was the type inside a scope at the moment it was interrupted, this is done by using type memories. Type memories are a syntactical construct which works as a simple way to remember interrupted types. It is needed because we want, in order to prove session fidelity, to build a well-formed global type from a configuration. If we use the syntax proposed above, some problems would arise: consider a type whose scope `c` contains a sequence interaction between `A` and `B`, and suppose the scope is interrupted by `B`, but the interrupt message is not yet

(Disc') assuming $s[\mathtt{r}] : E^{\mathtt{c}_0}[T];s[\mathtt{r}] : \mathtt{c}_1[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to s[\mathtt{r}] : E^{\mathtt{c}_0}[T'];s[\mathtt{r}] : h$

$s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_];s[\mathtt{r}] : \mathtt{c}_1[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to E^{\mathtt{c}_0}[\{\!|\,\|T'\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_];s[\mathtt{r}] : h$

$(\mathsf{EIn1})$ assuming $\quad \varphi(\Sigma,\mathtt{c})$

$s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : \mathtt{c}^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h \to E^{\mathtt{c}_0}[\{\!|\,\|T\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;T'];s[\mathtt{r}] : h$

$(\mathsf{EIn2})$ assuming $\quad \neg\varphi(\Sigma,\mathtt{c})$,

$\prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\mathtt{c}_i}[\{\!|\,\|T_i\|\,|\!\}^{\text{-}} \triangleright \langle \mathtt{r}'?l\rangle;\_],s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|T|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;T'];\Sigma,s[\mathtt{r}] : \mathtt{c}^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h$
$\to \prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\mathtt{c}_i}[\{\!|\mathtt{Eend}|\!\}^{\text{-}} \triangleright \langle \mathtt{r}'?l\rangle;\_],s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\mathtt{Eend}|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;T'];\Sigma,s[\mathtt{r}] : h$

$(\mathsf{EDisc1})$ assuming $\quad \varphi(\Sigma,k_1)$
$s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\text{-}}[\{\!|T|\!\}^{k_1} \triangleright \langle \mathtt{r}'?l\rangle;T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}''?l''\rangle;\_];\Sigma,s[\mathtt{r}] : k_1^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h$
$\to s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\text{-}}[\{\!|\,\|T\|\,|\!\}^{k_1} \triangleright \langle \mathtt{r}'?l\rangle;T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}''?l''\rangle;\_];\Sigma,s[\mathtt{r}] : h$

$(\mathsf{EDisc2})$ assuming $\quad \neg\varphi(\Sigma,k_1)$

$s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\mathtt{c}}[\{\!|T|\!\}^{k_1} \triangleright \langle \mathtt{r}'?l\rangle;T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}''?l''\rangle;\_]$,
$\qquad \prod_{1\le i\le n} s[\mathtt{r}_i] : E_1{}^{\mathtt{c}}[\{\!|\,\|T_i\|\,|\!\}^{k_1} \triangleright \langle \mathtt{r}'?l\rangle;T_i'];\Sigma,s[\mathtt{r}] : \mathtt{c}_i^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle.h$
$\to \prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{\!|\mathtt{Eend}|\!\}^{\mathtt{c}_i} \triangleright \langle \mathtt{r}'?l\rangle;T_i]$,
$\qquad s[\mathtt{r}] : E^{\mathtt{c}_0}[\{\!|\,\|E^{\mathtt{c}}[\{\!|\mathtt{Eend}|\!\}^{k_1} \triangleright \langle \mathtt{r}'?l\rangle;T']\|\,|\!\}^{\mathtt{c}} \triangleright \langle \mathtt{r}''?l''\rangle;\_];\Sigma,s[\mathtt{r}] : h$

Fig. 15: Semantics for types with memories

received by `A`: we cannot obtain a session fidelity result, i.e. we cannot build a well-formed session types from the configuration, the reason being there is no counterpart to the type currently included in scope `c` in `A`, as it as been discarded when `B` exceptionally terminated. As a way to remember what `B` was supposed to do before the interruption, we use a type memory, that is, a syntactic annotation that remember the type in `c` in `B` when the interrupt was raised.

Memories. 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 $E^{\mathtt{c}}[\|T\|]$. We define the erase operator $\mathit{Erase}(\cdot)$ which removes memory annotations from types: $\mathit{Erase}(s[\mathtt{r}] : E^{\mathtt{c}}[\|T\|]) = s[\mathtt{r}] : E^{\mathtt{c}}[\mathtt{Eend}]$.

We say that a queue *has an ongoing exception on* `c`, written $\varphi(\Sigma,\mathtt{c})$ whenever Σ contains at least one message $\mathtt{c}_1^{\mathtt{I}}[\mathtt{r}',\mathtt{r}]\langle l\rangle$ and $\mathtt{c}\mathscr{R}\mathtt{c}_1$.

Intermediate correspondence. 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 types with memories using the following updates:

- $\Delta,s[\mathtt{r}] : E^{\mathtt{c}}[T];\Sigma,s[\mathtt{r}'] : h.\mathtt{c}[\mathtt{r}',\mathtt{r}]\langle l_j\rangle$ becomes $\Delta,s[\mathtt{r}] : E^{\mathtt{c}}[\mathtt{r}!\{l_i.T_i\}];\Sigma$ for some $(T_i)_{i\neq j}$
- If the participants of `c` are $\mathtt{r},\mathtt{r}_1,\ldots,\mathtt{r}_n$, then

$$\Delta,s[\mathtt{r}] : E^{\mathtt{c}}[\|T\|] \prod_{1\le i\le k} s[\mathtt{r}_i] : E_i{}^{\mathtt{c}}[\|T_i\|], \prod_{k+1\le j\le n} s[\mathtt{r}_j] : E_i{}^{\mathtt{c}}[T_j];\Sigma, \prod_{k+1\le j\le n} s[\mathtt{r}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_j]\langle l\rangle.h_j$$

  is treated as $\Delta,s[\mathtt{r}] : E^{\mathtt{c}}[T],\prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\mathtt{c}}[T_i];\Sigma$.

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

Special semantics for types 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)) and presented in Figure 15.

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 $Erase(\Delta),\Sigma$ and that $Erase()$ preserves the intermediate correspondence w.r.t. $G_1,\dots,G_n$. 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.

Results. Theorem 5.3 states that if a configuration corresponds to $G_1,\dots,G_n$ and makes some reduction steps, we can let it make other steps to reach a configuration that corresponds to some derivatives of $G_1,\dots,G_n$. The intermediate configurations correspond to the situation where messages are exchanged through queues.

Theorem 5.3 (Session fidelity) *If Δ corresponds to $G_1,\dots,G_n$ and $\Delta,\varepsilon \rightarrow^* \Delta',\Sigma'$, there exists $\Delta',\Sigma' \rightarrow^* \Delta'',\varepsilon$ such that Δ'' corresponds to $G''_1,\dots,G''_n$ which is a derivative of $G_1,\dots,G_n$.*

Proof We prove that if there is an intermediate correspondence between Δ,Σ and $G_1,\dots,G_n$ and if $\Delta,\varepsilon \rightarrow \Delta',\Sigma'$, then there is an intermediate correspondence Δ'' and $G''_1,\dots,G''_n$ which is a derivative of $G_1,\dots,G_n$ by induction on $\rightarrow$. A full proof can be found in Appendix A.

6 Related work

Distributed runtime verification. The work in [3] explores runtime 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 session interrupts). Their monitor is centralised (thus no projection facilities are discussed), and neither formalisation, global safety property nor proof of correctness is given in [3].

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. [46], where monitors enforce *laws* expressed as event-condition-action. In [32], 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 runtime verification for Web services, the works [30,31] propose 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. [10] 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. [5] develop a runtime 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. [27] propose a runtime 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 [17] follows a similar but centralised approach of [27]. 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 monitoring tools. Jass [24] is a precompiler tool for monitoring the dynamic behaviour of sequential objects and the ordering of method invocations by annotating Java programs with specifications that can be checked at runtime. Other approaches to runtime verification of program execution by monitors generated from language-based specifications include: aspect-oriented programming [29]; other works that use process calculi formalisms, such as CSP [24]; monitors based on FSM skeletons associated to various forms of underlying patterns [1,4]; and the analysis of dynamic parametric traces [4]. Our monitor framework has been influenced by these works and shares similarities with some of the presented RV techniques. However, the target program domain and focus of our work are different. Our framework is specifically designed for decentralised monitoring of distributed programs with diverse participants and interleaving sessions, as opposed to monitoring the execution of a single program and verifying its local properties. The basis of our design and implementation is the theory of multiparty session types, over which we have developed practically motivated extensions to the type language and the methodology for runtime verification.

Interrupt in session types. Our theoretical work is new, as existing works for distributed system do not support at the same time nested interrupt, multiparty protocols and continuations to interruptible blocks. [13,12] contained *interactional* exceptions for *binary only* sessions and for web service choreographies. This approach is different as try-catch blocks are build upon session-connections—for a single session, exactly two different behaviours are described—which constrains the shape of the protocols. [11] implements nested exceptions with queues and a central synchronisation through the use levels. Yet, the interruptible construct is blocking w.r.t. continuations and thus not truly asynchronous.

Exception handling is common in distributed object-oriented programming [38,45]. Composition Actions have been adapted in [43] to model fault tolerant Web services but these works do not address the same models of protocols. [23] presents a copyless message passing model with exceptions and delegation with a common heap. They do not address distributed systems and use transaction to restored to a consistent configuration. Service-oriented calculi (e.g. [8,28,44]) include compensation or termination handling, but do not coordinate participant behaviour after an exception is raised. CaSPiS [9] models binary sessions (and handle nesting with *pipes*); a session can be explicitly terminated by one participant and a termination handler is subsequently activated at the other endpoint.

7 Conclusion

We have implemented the first dynamic verification of distributed communications based on multiparty session types and shown that a new feature for interruptible conversations is effective in the runtime verification of message exchanges in a large cyberinfrastructure [34] and Web services [40,41]. Our implementation automates distributed monitoring by generating FSMs from local protocol projections. We sketched the formalisation of asynchronous interruptions with conversation scopes, and proved the correctness of our design through the session fidelity theorem. Future work includes the incorporation of more elaborate handling of error cases into monitors and automatic generation of service code stubs. Although our implementation work is ongoing through industry collaborations, the results already confirm the feasibility of our approach. We believe this work contributes towards methodologies for better specification and more rigorous governance of network conversations in distributed systems.

Acknowledgements We thank Gary Brown, Matthew Arrot and the Scribble team for discussions and collaborations. This work has been partially sponsored by the Ocean Observatories Initiative, VMWare, Pivotal, Cognizant, EPSRC EP/K034413/1, EP/K011715/1, EP/G015635/1, EP/G015481/1 and ICT COST Action IC1201 BETTY.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. *SIGPLAN Not.*, 40(10):345–364, Oct. 2005.
2. Advanced Message Queuing protocols (AMQP) homepage. `http://jira.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`.
3. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in Jason. In *DALT*. Springer, 2012.
4. P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, Oct. 2007.
5. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC*, pages 193–202. ACM, 2004.
6. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
7. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.
8. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.
9. M. Boreale, R. Bruni, R. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F. Boer, editors, *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer Berlin Heidelberg, 2008.
10. 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.
11. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *FSTTCS*, volume 8 of *LIPICS*, pages 338–351, 2010.
12. M. Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.
13. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
14. W3C WS-CDL. `http://www.w3.org/2002/ws/chor/`.
15. T.-C. Chen. *Theories for Session-based Governance for Large-scale Distributed Systems*. PhD thesis, Queen Mary, University of London, 2013.
16. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, LNCS, pages 194–213. Springer, 2012.

17. Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime monitoring of web service conversations. In *CASCON*, pages 42–57. ACM, 2007.
18. C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.
19. 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.
20. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
21. 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.
22. R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations - distributed dynamic verification with session types and python. In *RV*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.
23. S. Jakšić and L. Padovani. Exception handling for copyless messaging. In *PPDP*, pages 151–162. ACM, 2012.
24. Jass Home Page. `http://modernjass.sourceforge.net/`.
25. Jboss Savara project. `http://www.jboss.org/savara/downloads`.
26. I. H. Krüger, M. Meisinger, and M. Menarini. Runtime verification of interactions: from mscs to aspects. In *RV*, RV, pages 63–74, Berlin, Heidelberg, 2007. Springer-Verlag.
27. 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.
28. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
29. LAVANA project. `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`.
30. Z. Li, J. Han, and Y. Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC'05*, pages 73–86, 2005.
31. Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC*. IEEE, 2006.
32. N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM*, 9:273–305, July 2000.
33. 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.
34. Ocean Observatories Initative. `http://www.oceanobservatories.org/`.
35. OOI codebase. `https://github.com/ooici/pyon`.
36. Scribble extensions for OOI integration. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.
37. OOI COI governance framework. `https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Framework`.
38. C. M. F. Rubira and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95*, page 499, Washington, DC, USA, 1995. IEEE Computer Society.
39. G. Salaün. Analysis and verification of service interaction protocols - a brief survey. In *TAV-WEB*, volume 35 of *EPTCS*, pages 75–86, 2010.
40. JBoss Savara Project. `http://www.jboss.org/savara`.
41. Scribble Project homepage. `http://www.scribble.org`.
42. Full version of this paper. `http://www.doc.ic.ac.uk/~rn710/mon`.
43. F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for compositeweb services. *Reliable Distributed Systems, IEEE Symposium on*, 0:167, 2003.
44. H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
45. J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *ICDCS*, pages 12–21, Washington, DC, USA, 1998. IEEE Computer Society.
46. W. Zhang, C. Serban, and N. Minsky. Establishing global properties of multi-agent systems via local laws. In *E4MAS*, pages 170–183, 2007.

A Proofs

This appendix includes a full proof of Theorem 5.3.

Theorem 5.3 (Session fidelity) *If Δ corresponds to $G_1,\dots,G_n$ and $\Delta,\varepsilon \to^* \Delta',\Sigma'$, there exists $\Delta',\Sigma' \to^* \Delta'',\varepsilon$ such that Δ'' corresponds to $G''_1,\dots,G''_n$ which is a derivative of $G_1,\dots,G_n$.*

Proof We prove that if there is an intermediate correspondence between Δ,Σ and $G_1,\dots,G_n$ and if $\Delta,\varepsilon \to \Delta',\Sigma'$, then there is an intermediate correspondence Δ'' and $G''_1,\dots,G''_n$ which is a derivative of $G_1,\dots,G_n$.

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 $\Delta = \Theta, s[\mathtt{r}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}?l\rangle;T']$. Correspondence gives

- $\Delta = \Theta_0, s[\mathtt{r}] : E^{\mathtt{c}_0}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}?l\rangle;T'], \prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\mathtt{c}_i}[\{|T_i|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}?l\rangle;T'_i]$ and
- $\Sigma = \Sigma_1,\Sigma_0$ with $\Theta_0;\Sigma_0$

corresponding to $G_2,\dots,G_n$ and $(\Delta-\Theta');\Sigma_1$ corresponding to G_1. We know that $\Sigma' = \Sigma_0, \prod_{1\le i\le n} s[\mathtt{r}_i] : h_i.\mathtt{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_i]\langle l\rangle$. Concluding is easy using the second rule of intermediate correspondence with $k=0$.

Case (In)**.** We assume $\Delta = \Theta, s[\mathtt{r}'] : E^{\mathtt{c}}[\mathtt{r}?\{l_i.T_i\}]$ and $\Sigma = \Sigma_0, s[\mathtt{r}'] : h.\mathtt{c}[\mathtt{r},\mathtt{r}']\langle l_j\rangle$. We know there exists $G_1,\dots G_n$ and Δ_0 such that $\Delta_0 = \Theta_1,\dots,\Theta_n$ with $\Theta_i = \bigcup_{\mathtt{r}\in G_i} G_i \upharpoonright^{\mathtt{r}}$.

Without loss of generality we have $\Theta_1 = s[\mathtt{r}'] : E^{\mathtt{c}}[\mathtt{r}?\{l_i.T_i\}],\Theta'_1$. By the rules of projection, it means $G_1 = \mathtt{r}{\to}\mathtt{r}' : \{l_j.G_j\}_{j\in J}$, implying $\Theta'_1 = s[\mathtt{r}'] : E^{\mathtt{c}'}[\mathtt{r}?\{l_i.T_i\}],\Theta''_1$. So we have

$$\Delta' = \Omega, s[\mathtt{r}] : E^{\mathtt{c}}[T_j], s[\mathtt{r}'] : E^{\mathtt{c}'}[\mathtt{r}?\{l_i.T_i\}],\Theta'_1 \text{ and } \Sigma' = \Sigma_0, s[\mathtt{r}'] : h.\mathtt{c}[\mathtt{r},\mathtt{r}']\langle l_j\rangle.$$

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 $\Sigma = \Sigma', \mathtt{c}^{\mathtt{I}}[\mathtt{r}_0,\mathtt{r}_{k+1}]\langle l\rangle.h$ and $\varphi(\Sigma',\mathtt{c})$. Then let us define $\Delta = \Theta, s[\mathtt{r}_{k+1}] : E^{\mathtt{c}_0}[\{|T_{k+1}|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}_0?l\rangle;T']$ and $\Delta' = \Theta,\ s[\mathtt{r}_{k+1}]E^{\mathtt{c}_0}[\{|\|T\||\}^{\mathtt{c}} \triangleright \langle \mathtt{r}_0?l\rangle;T']$. Without loss of generality we suppose $\Delta;\Sigma$ corresponds to G. We deduce that

- $G = F\{|G_0|\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle;G'_{\_}$;
  $\Delta = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{1\le i\le k} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{|\|T_i\||\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{k+1\le j\le n} s[\mathtt{r}_j] : E_i{}^{\text{-}}[\{|T_j|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_]$; and
- $\Sigma = \Sigma_0, \prod_{k+1\le j\le n} s[\mathtt{r}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_j]\langle l\rangle.h_j$.

Thus we have

- $\Delta' = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{1\le i\le k+1} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{|\|T_i\||\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{k+2\le j\le n} s[\mathtt{r}_j] : E_i{}^{\text{-}}[\{|T_j|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_]$ and
- $\Sigma' = \Sigma_0, \prod_{k+2\le j\le n} s[\mathtt{r}_j] : \mathtt{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_j]\langle l\rangle.h_j$.

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 $\Sigma = \Sigma', \mathtt{c}^{\mathtt{I}}[\mathtt{r}_0,\mathtt{r}_n]\langle l\rangle.h$ and $\neg\varphi(\Sigma',\mathtt{c})$. Then $\Delta = \Theta, s[\mathtt{r}_n] : E^{\mathtt{c}_0}[\{|T_n|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}_0?l\rangle;T']$ and $\Delta' = \Theta, s[\mathtt{r}_{k+1}]E^{\mathtt{c}_0}[\{|\|T\||\}^{\mathtt{c}} \triangleright \langle \mathtt{r}_0?l\rangle;T']$. Without loss of generality we suppose $\Delta;\Sigma$ corresponds to G. We deduce that

- $G = F\{|G_0|\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle;G'_{\_}$,
- $\Delta = s[\mathtt{r}_0] : E^{\text{-}}[\{|T|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{1\le i\le n-1} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{|\|T_i\||\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], s[\mathtt{r}_n] : E_n{}^{\text{-}}[\{|T_n|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_]$ and
- $\Sigma = \Sigma_0, s[\mathtt{r}_n] : \mathtt{c}^{\mathtt{I}}[\mathtt{r},\mathtt{r}_n]\langle l\rangle.h$.

From the semantics, we also have

- $\Delta' = s[\mathtt{r}_0] : E^{\text{-}}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_], \prod_{1\le i\le n} s[\mathtt{r}_i] : E_i{}^{\text{-}}[\{|\mathtt{Eend}|\}^{\mathtt{c}} \triangleright \langle \mathtt{r}'?l\rangle;\_]$ 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 $F\{|\mathtt{Eend}|\}^{\mathtt{c}}\langle l \text{ by } \mathtt{r}\rangle;G'_{\_}$ which is a derivative of G. We conclude.

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

Case (EDisc_1) is similar to (EIn_1).

Case (EDisc_2) is similar to (EIn_2).

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

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

  By using these properties, we conclude the proof.