Practical interruptible conversations
Distributed dynamic verification with session types and Python

Raymond Hu[1], Rumyana Neykova[1], Nobuko Yoshida[1]
Romain Demangeon[1], and Kohei Honda[2]

[1] Imperial College London
[2] Queen Mary, University of London

Abstract. The rigorous and comprehensive verification of communication-based software is an important engineering challenge in distributed systems. Drawn from our industrial collaborations [33,28] on Scribble, a choreography description language based on multiparty session types, this paper proposes a dynamic verification framework for structured interruptible conversation programming. We first present our extension of Scribble to support 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. 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-based protocols, without requiring additional synchronisation mechanisms. Benchmarks show conversation programming and monitoring can be realised with little overhead.

1 Introduction

The main engineering challenges in distributed systems include finding suitable specifications that model the range of states exhibited by a system, and ensuring that these specifications are followed by the implementation. In message passing applications, rigorous specification and verification of communication protocols is particularly crucial: a protocol is the interface to which concurrent components should be independently implementable while ensuring their composition will form a correct system as a whole. Multiparty Session Types (MPST) [17,6] is a type theory for communication-oriented programming, originating from works on types for the π-calculus, towards tackling this challenge. In the original MPST setting, protocols are expressed as types and static type checking verifies that the system of processes engaged in a communication session (also referred to as a *conversation*) conforms to a globally agreed protocol. The properties enjoyed by well-typed processes are communication safety (no unexpected messages or races during the execution of the conversation) and deadlock-freedom.

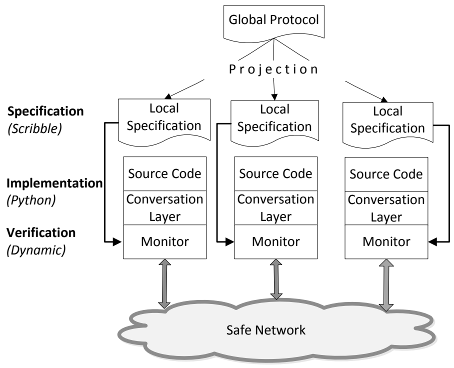


Fig. 1. Scribble methodology from global specification to local runtime verification

In this paper, we present the design and implementation of a framework for dynamic verification of protocols based on MPST, developed from our collaboration with industry partners [33,28] on the application of MPST theory. In this ongoing work, we are motivated to adapt MPST to dynamic verification for several reasons. First, session type checking is typically designed for languages with first-class communication and concurrency primitives, whereas our collaborations use mainstream engineering languages, such as Python and Java, that lack the features required to make static session typing tractable. Distributed systems are also often heterogeneous in nature, meaning that different languages and techniques (e.g. the control flow of an event-driven program is tricky to verify statically) may be used in the implementation of one system. Dynamic verification by communication monitoring allows us to verify MPST safety properties directly for mainstream languages in a more scalable way. Second, a system may use third-party components or services for which the source code is unavailable for type checking. Third, certain protocol specification features, such as assertions on specific message values, can be precisely evaluated at runtime, while static treatments would usually be more conservative.

Framework overview. Figure 1 illustrates the methodology of our framework. The development of a communication-oriented application starts with the specification of the intended interactions (the choreography) as a *global* protocol using the Scribble protocol description language [34], an engineering incarnation of the formal MPST type language. The core features of Scribble include multicast message passing and constructs for branching, recursive and parallel conversations. These features support the specification of a wide range of protocols, from domains such as standard Internet applications [18], parallel algorithms [27] and Web services [12].

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

When a conversation is initiated at runtime, the monitor at each endpoint generates a finite state machine (FSM) representation of the local communication behaviour from the local protocol for its role. In our implementation, the FSM generation is an extension of the correspondence between MPST and communication automata in [13] to support *interruptible* sessions (discussed below) and optimised to avoid parallel state explosion. The monitor tracks the communication actions performed by the endpoint, and the messages that arrive from the other endpoints, against the transitions permitted by the FSM. Each monitor thus works to protect both the endpoint from illegal actions by the environment, and the network from bad endpoints. In this way, our framework is able to ensure from the local verification of each endpoint that the global progress of the system as a whole conforms to the original global protocol [7], and that unsafe actions by a bad endpoint cannot corrupt the protocol state of other compliant endpoints.

This MPST monitoring framework has been integrated into the Python-based runtime platform developed by the Ocean Observatories Initiative (OOI) [28]. 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 [29]) and distributed runtime monitoring to regulate the behaviour of third-party applications within the system [31]. Although this work is in collaboration with the OOI, our implementation can be used orthogonally as a standalone monitoring framework for distributed Python applications.

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

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

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

§ 2 presents a use case for the extension of Scribble with asynchronous interrupts. This is a new feature in MPST, giving the first general mechanism for nested, multiparty session interrupts. We explain why implementing this feature is a challenge in session types. The previous works on exceptions in session types are purely theoretical, and are either restricted to binary session types (i.e. not multiparty) [11], do not support nesting and continuations [11,10], or rely on additional implicit synchronisation [9]. A formal proof of the correctness of our design is given in § 5.

§ 3 discusses the Python implementation of our MPST monitoring framework that we have integrated into the OOI project, and demonstrates the global-to-local projection of Scribble protocols, endpoint implementation, and local FSM generation. § 3.1 describes a concise API for conversation programming in Python. The API decorates conversation messages with the runtime session information required by the monitors to perform the dynamic verification. § 3.2 discusses the monitor implementation, how asynchronous interrupts are handled, and the key architectural requirements of our framework.

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

The source code of our Scribble toolchain, conversation runtime and monitor, performance benchmarks and further resources are available from the project page [35].

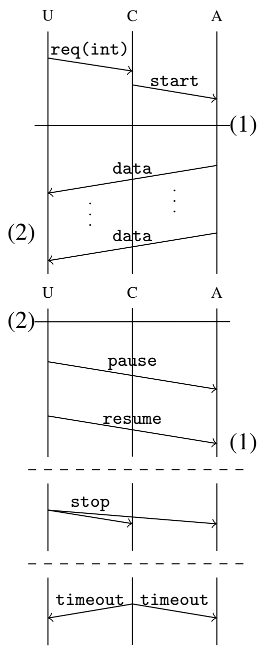
Acknowledgements. We thank Gary Brown and the Scribble team for discussions and collaborations. This work has been partially sponsored by the Ocean Observatories Initiative, VMWare, Cognizant and EPSRC EP/K034413/1, EP/K011715/1, EP/G015635/1 and EP/G015481/1.

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 [28] manages communication protocol specifications through a combination of informal sequence diagrams and prose descriptions. Figure 2 (left) gives an abridged version of a sequence diagram given in the OOI documentation for the Resource Access Control use case [29], 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.



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

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

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

```
// 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)

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

the basic operational meaning of `interruptible`, as illustrated in the above example, is readily understood by architects and developers, which is a primary consideration in the design of Scribble. The challenges in this extension are in the design of the supporting runtime and verification techniques to preserve the desired safety properties in the presence of `interruptible`. The challenges stem from the fact that `interruptible` combines several tricky, from a session typing view, aspects of communication behaviours that session type systems traditionally aim to prohibit, 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 [17,13]) 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 [13]).

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.

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 [11].

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 5, right) initiates a new conversation instance of the `ResourceAccessControl` (Figure 2) protocol, 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.

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

```
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      conv = Conversation.create(
6          ’RACProtocol’, ’config.yml’)
7      c = conv.join(user, ’alice’)
8      # request 1 hour access
9      c.send(controller, ’req’, 123)
10     with c.scope(’timeout’, ’stop’)
11         as c1:
12       while not self.limit_reached():
13         with c1.scope(’pause’) as c2:
14           while not buffer.full:
15             resource = c2.recv(controller)
16             buffer.append(resource)
17           c2.send_interrupt(’pause’)
18         # sleep before resume
19         c1.send(agent, ’resume’)
20       if self.should_stop():
21         c1.send_interrupt(’stop’)
22     c.close()
```

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

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. For asynchronous, non-blocking receives, the API provides `recv_async` to be used in an event-driven style.

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 10, to create and enter the scope of an `interruptible` Scribble block (here, the outer interruptible of the RAC protocol). The `timeout` and `stop` arguments associate these message signatures as interrupts to this scope. The conversation channel `c1` 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 `c1` is given by the enclosing `with` (lines 10–21); 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

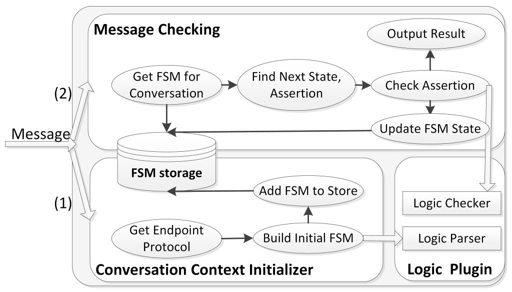


Fig. 6. Monitor workflow for (1) invitation and (2) in-conversation messages

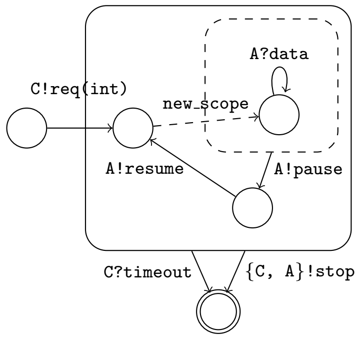


Fig. 7. Nested FSM generated from the User local protocol

protocol, thus allows 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 `c1` (outside its parent scope) after a `timeout` is received 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 resource in the payload of the `resume` (in line 21). The API can also make the discarded data available to the programmer through secondary (non-monitored) operations.

An alternative event-driven implementation using `receive_asyc` and callbacks (that can, however, be safely monitored against the same local protocol) is given in [35].

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 [31].

Monitor implementation. Figure 6 depicts the main components and internal workflow of our prototype monitor. The lower part relates to conversation initiation. The *invitation* message carries (a reference to) the local protocol for the invitee and the conversation id (global protocols can also be exchanged if the monitor has the facility for projection.) The monitor generates the FSM from the local protocol following [13]. Our implementation differs from [13] in the treatment of parallel subprotocols (i.e. unordered message sequences), and additionally supports interrupts. For efficiency, we extend [13] 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. Transition functions are similarly hashed, each entry having the shape: $(\mathit{current\_state},\ \mathit{transition}) \mapsto (\mathit{next\_state},\ \mathit{assertion},\ \mathit{var})$, where *transition* is a triple $(\mathit{label}, \mathit{sender}, \mathit{receiver})$ and *var* is the variable binder for the message payload. 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 6 relates to *in-conversation* messages, which carry the conversation id (matching an entry in the FSM hash table), sender and receiver fields, and the message label and payload. This information 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.

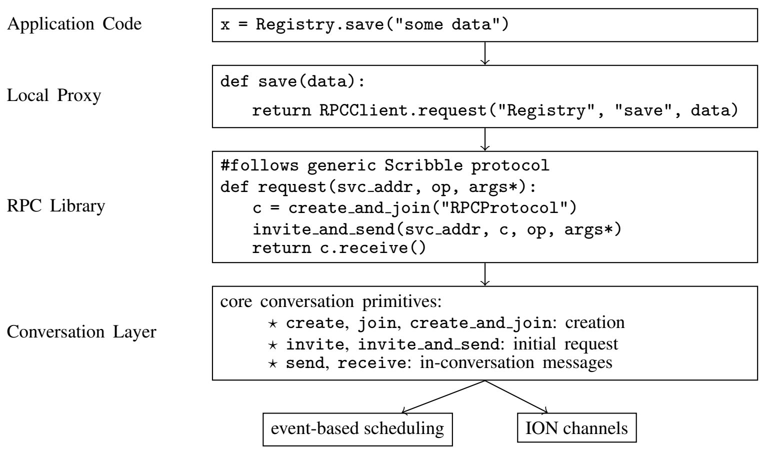
Monitoring interrupts. FSM generation for interruptible local protocols again makes use of nested FSMs. Each `interruptible` induces a nested FSM given by the main interruptible block, as illustrated in Figure 7 for the User local protocol. The monitor internally augments the nested FSM with a scope 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 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 infrastructure [30]. 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



```
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. 8. Translation of an RPC command into lower-level conversation calls

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 [31]. 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 runtime for session types (as shown in Figure 8, 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 [29], 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 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.

	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

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

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

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

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): "Rec 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" (NoM) 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 [35].

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.

Use Cases from research papers	Global Scribble (LOC)	FSM Memory (B)	Generation Time (s)
A vehicle subsystem protocol [21]	8	840	0.006
Map web-service protocol [15]	10	1040	0.010
A bidding protocol [24]	26	1544	0.020
Amazon search service [16]	12	1088	0.010
SQL service [32]	8	1936	0.009
Online shopping system [14]	10	1024	0.008
Travel booking system [14]	16	1440	0.013
Use Cases from OOI and Savara			
A purchasing protocol [20]	11	1088	0.010
A banking example [29]	16	1564	0.013
Negotiation protocol [29]	20	1320	0.014
RPC with timeout [29]	11	1016	0.013
Resource Access Control [29]	21	1854	0.018

Table 3. Use case protocols implemented in Scribble

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 1.5 KB), indicating that FSM caching is a feasible optimisation approach. The full Scribble protocols can be found at [35].

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 Interruptible session type theory and related work

5.1 Session type theory for interrupts

In this subsection, we sketch the underlying session type theory with interrupts and its correctness result, *session fidelity*, justifying our design choices. We build over the multiparty session theory [17], adding syntax and semantics for interrupts. In our theory, global types correspond to session specifications whereas local types are used to express monitored behaviours of processes [7]. We show that interruptible blocks can be treated through the use of *scopes*, a new formal construct that realises, through an explicit identifier, the domain of interrupts. Our scope-based session types can handle nested interrupts and multiparty continuations to interruptible blocks, allowing us to model truly asynchronous exceptions implemented in this paper (these features have not been modelled in existing MPST theories for exceptions [11,10,9]). The full definitions and proofs are available from [35].

Global types (G) below correspond to Scribble protocols. Scopes are made explicit by the use of scope variables S, corresponding to the dynamic scope generation present in the implementation in § 3.1. Roles in types are denoted by $\mathtt{r}$, and labels with l.

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

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

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

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

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

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

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

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

and say G' is a *derivative* of G if $G \rightsquigarrow^* G'$. We define *configurations* Δ, Σ as a pair of a mapping from a session channel to a local type and a collection of queues (a mapping from a session channel to a vector of the values). Configurations model the behaviour of a network of monitored agents. We say a configuration Δ, Σ corresponds to a collection of global types $G_1, \ldots, G_l$ whenever Σ is empty and the environment Δ is a projection of $G_1, \ldots, G_l$. The reduction semantics of the configuration ($\Delta, \Sigma \to \Delta', \Sigma'$) is defined using the contexts with the scopes. Formal definitions can be found in [35].

The correctness of our theory is ensured by Theorem 1, which states a local enforcement implies global correctness: if a network of monitored agents (modelled as a configuration) corresponds to a collection of well-formed specifications and makes some steps by firing messages, then the network can perform reductions (consuming these messages) and eventually 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. The proofs can be found in [35].

Theorem 1 (Session fidelity). *If Δ corresponds to $G_1, \ldots, G_n$ and $\Delta_0, \varepsilon \to^* \Delta, \Sigma$, there exists $\Delta, \Sigma \to^* \Delta', \varepsilon$ such that Δ' corresponds to $G'_1, \ldots, G'_n$ which are derivatives of $G_1, \ldots, G_n$.*

5.2 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. [36], where monitors enforce *laws* expressed as event-condition-action. In [26], 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 [24,25] 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. [8] 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. [22] 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 [14] follows a similar but centralised approach of [22]. 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 [19] 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 [23]; other works that use process calculi formalisms, such as CSP [19]; 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.

6 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 [28] and Web services [33,34]. 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.

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'12*. 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 '04*, pages 193–202, 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. 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.
9. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *FSTTCS'10*, volume 8 of *LIPICS*, pages 338–351, 2010.
10. M. Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.
11. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
12. W3C WS-CDL. `http://www.w3.org/2002/ws/chor/`.
13. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, LNCS. Springer, 2012.
14. Y. Gan et al. Runtime monitoring of web service conversations. In *CASCON '07*, pages 42–57. ACM, 2007.
15. C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264. Springer, 2007.
16. 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.
17. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
18. 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.
19. Jass Home Page. `http://modernjass.sourceforge.net/`.
20. Jboss Savara project. `http://www.jboss.org/savara/downloads`.
21. I. H. Krüger, M. Meisinger, and M. Menarini. Runtime verification of interactions: from mscs to aspects. In *RV'07*, RV'07, pages 63–74, Berlin, Heidelberg, 2007. Springer-Verlag.
22. 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.
23. LAVANA project. `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`.
24. Z. Li, J. Han, and Y. Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC'05*, pages 73–86, 2005.
25. Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC'06*. IEEE, 2006.
26. N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM*, 9:273–305, July 2000.
27. 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.
28. Ocean Observatories Initative. `http://www.oceanobservatories.org/`.
29. OOI. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.
30. OOI codebase. `https://github.com/ooici/pyon`.
31. OOI COI governance framework. `https://confluence.oceanobservatories.org/display/syseng/CIAD+COI+OV+Governance+Framework`.
32. G. Salaün. Analysis and verification of service interaction protocols - a brief survey. In *TAV-WEB*, volume 35 of *EPTCS*, pages 75–86, 2010.
33. JBoss Savara Project. `http://www.jboss.org/savara`.
34. Scribble Project homepage. `http://www.scribble.org`.
35. Full version of this paper. `http://www.doc.ic.ac.uk/~rn710/mon`.
36. W. Zhang, C. Serban, and N. Minsky. Establishing global properties of multi-agent systems via local laws. In *E4MAS'06*, pages 170–183, 2007.