The Scribble Protocol Language

Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng

Imperial College London

Abstract. This paper describes a brief history of how Kohei Honda initiated the Scribble project, and summarises the current status of Scribble.

1 Introduction

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other [35, 39]. Scribble was born in Paris in December 2006 when Kohei Honda took his six month sabbatical. He started writing a seventy-page document of the first version of Scribble [17], based on his experiences as an invited expert for the W3C Web Services Choreography Description (WS-CDL) Working Group [8]. Since 2003, Kohei and the first author (Nobuko Yoshida) had been working for formalising WS-CDL in the π-calculus to guarantee deadlock-free communications by session types. Later, Marco Carbone joined the academic team of WS-CDL. Unexpectedly, it took more than five years for us to understand and formalise their core technologies due to complexity of the description: for example, to describe just a "hello world" protocol, WS-CDL requires the definition of Participant Types, Role Types, Relationship Types, Channel Types, Information Types, Tokens, Token Locators and finally Sequences with an Interaction and Exchange. During this work, Kohei proposed a much simpler, abstract version of choreography, which only focuses on signatures (or types) of CDLs. This is the origin of Scribble. He sent his first seventy-page draft to his close industry colleagues by e-mail together with his motivation:

> *Scribbling is necessary for architects, either physical or computing, since all great ideas of architectural construction come from that unconscious moment, when you do not realise what it is, when there is no concrete shape, only a whisper which is not a whisper, an image which is not an image, somehow it starts to urge you in your mind, in so small a voice but how persistent it is, at that point you start scribbling.*

This draft encouraged two of the members of WS-CDL WG, Gary Brown and Steve-Ross Talbot, to design and implement Scribble through Pi4 Technologies Foundations [33], collaborating with Kohei. The second version of Scribble document was written in collaboration with Brown in October 2007.

Interestingly, Scribble gave clues to solving the main theoretical open problem of the session type theory repeatedly posed by researchers and industry

partners at that time: that is whether original *binary* sessions [19, 37] can be extended to *multiparty sessions*. This is a natural question since most business protocols and parallel computations in practice involve multiparty communications. Honda, Yoshida and Carbone formalised the essence of Scribble as the multiparty session type theory (MPST) in the π-calculus, and published in [21]. Since then Kohei has worked with several standardisation bodies [2, 40] and open source communities [32, 36]. Red Hat opened a new JBoss Project, Scribble [35]. More details about a history of his collaborations with the industry partners can be found in [18, 20]. His last paper, which was mostly written by himself, is about Scribble [16].

The aims of this paper are to record his first draft [17] and to show the current status of Scribble project. Section 2 summarises the first version of Scribble draft; Section 3 outlines Scribble framework and its Python implementation; Section 4 discusses an extension of Scribble for subprotocols and interrupts; Section 5 shows another extension of Scribble for high-performance computations; Section 6 gives future works and Section 7 concludes.

2 Preamble of the first Scribble document

This section presents extracts from the preamble of the first Scribble document as originally written in [17], and remarks how these initial ideas have been carried out.

2.1 Conversations and Protocols (from [17, § 1.1])

This document presents concrete description examples of various interaction scenarios written in the first layer of Scribble. Scribble is a language for describing the structures and behaviours of communicating processes at a high level of abstraction, offering an intuitive and expressive syntax built on a rigorous mathematical basis. While the language can potentially be used for many purposes, our initial primary application area is description, validation and execution of the whole class of financial protocols and applications which use them.

Our central philosophy in designing Scribble, as a high-level language for describing communication-centred applications, is to enable description which is free from implementation details but which allows efficient and flexible implementation. The key idea to achieve these seemingly contradictory requirements is the use of the unit of abstraction called "conversation," known as *session* in the literature on theories of processes and programming languages.

A conversation in the present context means a series of interactions among two or more participants which follow a prescribed scenario of interactions. This scenario is the type (signature) of that conversation which we call *protocol*. A protocol is a minimal structure which guarantees type-safety of conversations, and has been known as *session type* [7, 13, 19, 24, 41] in theories of processes which in turn is based on theories of types for programming languages [34]. At runtime,

a conversation is established among its participants, and the participants get engaged in communications in its context following a stipulated protocol.

A single distributed application may be engaged in two or more conversations, even simultaneously. For example, during a commercial transaction, an application running for a merchant may be engaged in two conversations at the same time, one for a credit transfer and another for a debit transfer protocol. Another example is a travel agency who interacts with its customer electronically following a certain protocol and, to meet the demands of the customer, interacts with other service providers (for example airline companies), each following a distinct protocol. The agency's conversation with its customer and those with other services will interleave.

We specify a protocol using a type language of Scribble (just as types in ML are specified using a type language of ML). This type language constitutes the most abstract level of the description layers in Scribble. On its basis, the immediately upper layer of description defines what we call *conversation models* (which correspond to class models in UML). Conversation models serve many purposes including a foundation for a design-by-contract (DBC) framework, which starts from augmenting conversation models with assertions written in a logical language. Further we have languages for describing detailed behaviour, reaching executable descriptions, some of which may as well take the form of integration with existing programming languages. These languages as a whole contribute to flexible and comprehensive descriptions of the structure of message exchange (choreography) among communicating agents. Example descriptions in some of these languages will be treated in the sequels to the present note.

The language for protocols is the most abstract and terse: at the same time, it is also a rich description language for conversation scenarios, as well as offering a basis for the remaining layers. Protocols are also a basis of diverse forms of static and dynamic validation. Thus understanding this language is the key to understanding the present description framework as a whole.

2.2 Applications (from [17, § 1.2])

The first and foremost objectives of Scribble is to allow scribbling of structures of interactions intuitively and unambiguously. Just like we are sure what is the intended behaviour of our programs and models for sequential computation, we want to be sure what our description for interactional applications means in a simple and intuitive syntax.

Scribble is based on theories of processes, in particular the π-calculus [26–28]. This is not a place to discuss the nature of this theoretical basis but it is worth noting that this theory enables us to mathematically identify what is the (interactional) "behaviour" embodied in a given description. Thus we can rigorously stipulate what each description means. While the meaning of sequential programs is relatively intuitive to capture, this may not be so for interactional software: thus this theory pins down the tenet of descriptions of interactional behaviour, bootstrapping all endeavours in the present enterprise.

Another theoretical underpinning of the design of Scribble is the study on session types [7, 13, 19, 24, 41] mentioned already, which present in-depth study of type languages for conversations and their use in static validation, abstraction concerns and runtime architecture.

Starting from clarity and precision in description, Scribble (together with its theoretical basis) is intended to be used for several purposes, some of which we summarise in the following.

- Describe protocols of conversations for applications clearly, intuitively and precisely; statically validate if the resulting descriptions are consistent; with unambiguous shared understanding on the meaning of resulting descriptions.
- Generate code prototypes and associated runtime parameters (e.g. an FSA (Finite State Machines) for monitoring) from stipulated protocols, with a formal guarantee that code/data exactly conform to the protocols.
- Describe conversation scenarios of a distributed application which use these protocols, as conversation models. Statically validate if the resulting models use protocols correctly, as well as other significant properties.
- Elaborate protocols and conversation models with assertions (logical formulae) to specify their properties, for enriched behavioural constraints/models.
- Develop (and debug) endpoint applications which realise given conversation models with incremental validation that the resulting programs conform to the stipulated protocols and conversation models.
- Statically validate if the applications have specific desirable properties (such as deadlock-freedom) leveraging high-level conversation structures.
- Dynamically validate (monitor) if runtime message exchanges of an application precisely follow the stipulated protocols/models: with a formal guarantee that all and only violations of the stipulated scenario are detected; automatically generate such a monitor from protocols/conversation models.
- Offer a tractable and unambiguous specification of software tools and infrastructure needed for achieving these goals.

We note that the central point of having a theoretical basis in Scribble is first of all to allow these ideas themselves (for example validation) to "make sense": we can share clearly what they mean and what they do not mean. And all of this should be built on the clarity of the behavioural description in the first place.

2.3 Remarks on the Preamble

The preamble ends with a "Caution" subsection. Kohei explicitly noted that "this compilation only lists *signatures* (or *types*) for conversations, *not* direct behavioural description. While it may look we are describing dynamic behaviour, what is indeed described is the static structure underlying dynamic behaviour, just as signature in class models extracts the static core of dynamic behaviour of objects." This became the basis for establishing a theory of multiparty session

types [21]. In the rest of the document, the presentation is organised centring on concrete examples (use cases) described in Scribble. There are 29 examples divided into 11 sections: the last section treats fairly complex examples from real world financial protocols. Many examples were obtained from his industry partners working in financial IT, which became valuable sources to not only implement Scribble but also extend the original theory [21]. For example, the work on exceptions [6], subsessions [10], dynamic multiroles [**?**] and asynchronous messaging optimisation [**?**] directly tackled the examples in [17]. Their results are reflected in the subsequent designs and updates of Scribble, as discussed in the next section. From the list of the applications in § 2.2, we can observe that Kohei had a clear vision how Scribble should be used in future: in 2007, Kohei had even not known the Ocean Observatories Initiative [32] (cf. § 3), but he had already an idea to apply Scribble for dynamic verification via generations of FSAs. About code generation, the Scribble team is currently working for generating type-safe, deadlock-free parallel algorithm implementations from Scribble (cf. § 5). A conversation model mentioned in § 2.1 is formalised as the DBC of MPSTs in [5] and its application to Scribble is on-going (cf. Logical Annotations in § 6). The rest of the paper explains how the Scribble team has been working and developing Scribble, following his initial predictions.

3 Scribble

This section first describes the stages of the Scribble framework, explaining the design challenges of applying session types to practice and recent research threads motivated by this work. We then illustrate an example protocol specification in the Scribble language, and list a couple of extensions.

3.1 The Scribble Framework

The Scribble project [16, 18, 35, 39] is a collaboration between session types researchers and architects and engineers from industry [25, 36] towards the application of session types principles and techniques to current engineering practices. Building on the theory of multiparty session types [3, 21] (MPST), this ongoing work tackles the challenges of adapting and implementing session types to meet real-world usage requirements. This section gives an overview of the current version of the Scribble framework for the MPST-based development of distributed software. In the context of Scribble, we use the terms *session* and *conversation* interchangeably.

The main elements of the Scribble framework, outlined in Figure 1, are as follows.

The Scribble language is a platform-independent description language for the *specification* of asynchronous, multiparty message passing protocols [16, 18, 38]. Scribble may be used to specify protocols from both the global (neutral) perspective and the local perspective of a particular participant (abstracted

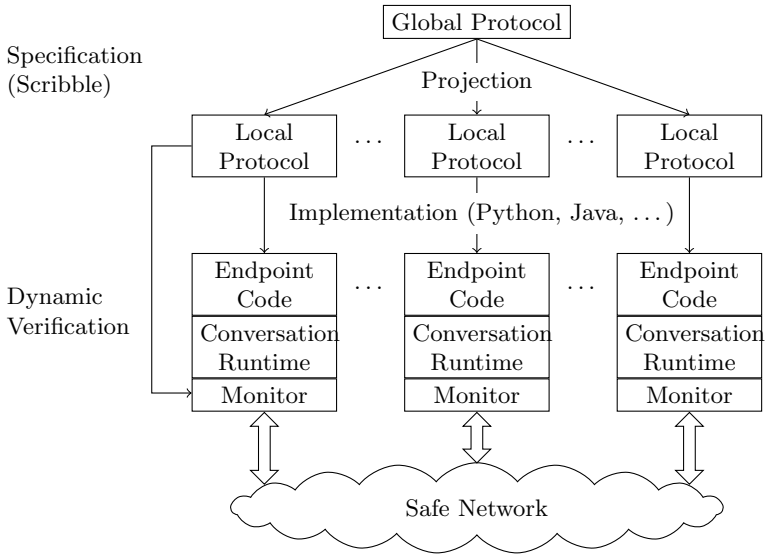

Fig. 1. The Scribble framework for distributed software development Scribble methodology from global specification to local runtime verification

as a role); at heart, the Scribble language is an engineering incarnation of the notation for global and local types in formal MPST systems and their correctness conditions.

The Scribble Conversation API provides the local communication operations for *implementing* the endpoint programs for each role natively in various mainstream languages. The current version of Scribble supports Java [35] and Python [25] Conversation APIs with both standard socket-like and event-driven interfaces for initiating and conducting conversations.

The Scribble Runtime is a local platform library for executing Scribble endpoint programs written using the Conversation API. The Runtime includes a conversation monitoring service for *dynamically verifying* [4, 23, 29] the interactions performed by the endpoint against the local protocol for its role in the conversation. In addition to internal monitors at the endpoints, Scribble also supports the deployment of external conversation monitors within the network [9].

3.2 Development Challenges of Scribble

The Scribble development workflow starts from the explicit specification of the required global protocols, similarly to the existing, informally applied approaches based on prose documentation, such as Internet protocol RFCs, and common graphical notations, such as UML and sequence diagrams. Designing an engineering language from the formal basis of MPST types faces the following challenges.

– To developers, Scribble is a new language to be learned and understood, particularly since most developers are not accustomed to formal protocol specification in this manner. For this reason, we have worked closely with our collaborators towards making Scribble protocols easy to read, write and maintain. Aside from the core interaction constructs that are grounded in the original theory, Scribble features extensions for the practical engineering and maintenance of protocol specifications, such as subprotocol abstraction and parameterised protocols [16] (demonstrated in the examples below).
– As a development step (as opposed to a higher-level documentation step), developers face similar coding challenges in writing formal protocol descriptions as in the subsequent implementation steps. IDE support for Scribble and integration with other development tools, such as the Java-based tooling in [35], are thus important for developer uptake.
– Although session types have proven to be sufficiently expressive for the specification of protocols in a variety of domains, including standard Internet applications [22], parallel algorithms [31] and Web services [8], the evaluation of Scribble through our collaboration use cases has motivated the development of new multiparty session type constructs, such as asynchronous conversation interrupts [23] (demonstrated below) and subsession nesting [10], which were not supported by the pre-existing theory.

The Scribble framework combines the elements discussed before to promote the MPST-based methodology for distributed software development depicted in Figure 1. Scribble resources are available from the project home pages [35, 39].

3.3 Online Travel Agency example

To demonstrate Scribble as a multiparty session types language, Figure 2 lists the Scribble specification of the global protocol for an Online Travel Agency example (a use case from [1]).

In this example, there are three interacting roles, named Customer, Agency and Service, that establish a session.

1. Customer is planning a trip through a Travel Agency. Each query from Customer includes the journey details, abstracted as a message of type *String*, to which the Agency answers with the price of the journey, abstracted as a message of type *Int*. This query is repeated until Customer decides either `ACCEPT` or `REJECT` the quote.
2. If Customer decides to `ACCEPT` a travel quote from Agency, Agency relays a confirmation to Service, which represents the transport service being brokered by Agency. Then Customer and Service exchanges the address details (a message of type *String*) and the ticket dispatch date (a message of type *Date*).
3. If Customer decides to `REJECT` a travel quote from Agency, Agency sends a termination signal to Service to end the interaction.

The Scribble is read as follows:

```
1  module TravelAgency;
2
3  type <py> "types.IntType" from "types.py" as Int;
4  type <py> "types.StringType" from "types.py" as String;
5  type <py> "travelagency.Date" from "Date.py" as Date;
6
7  global protocol BookJourney(role Customer as C,
8       role Agency as A, role Service as S) {
9    rec LOOP {
10     choice at C {
11       query(journey:String) from C to A;
12       price(Int) from A to C;
13       info(String) from A to S;
14       continue LOOP;
15     } or {
16       choice at C {
17         ACCEPT() from C to A;
18         ACCEPT() from A to S;
19         Address(String) from C to S;
20         (Date) from S to C;
21       } or {
22         REJECT() from C to A;
23         REJECT() from A to S;
24 } } } }
```

Fig. 2. A Scribble specification of a global protocol for the Online Travel Agency use case

– The first line declares the Scribble module name. Although this example is self-contained within a single module, Scribble code may be organised into a conventional hierarchy of packages and modules. Importing payload type and protocol declarations between modules is useful for factoring out libraries of common payload types and subprotocols.

– The design of the Scribble language focuses on the specification of protocol *structures*. With regards to the payload data that may be carried in the exchanged messages, Scribble is designed to work orthogonally with external message format specifications and data types from other languages. The `type` declaration on Line 3 declares a payload type using the Python data format, specifically the `IntType` definition from the file `types.py`, aliased as `Int` within this Scribble module. Data type formats from other languages, as well as XML or various IDL based message formats, may be used similarly. A single protocol definition may feature a mixture of message types defined by different formats.

– Lines 7–8 declare the signature of a global protocol called `BookJourney`. This protocol involves three roles, `Customer`, `Agency` and `Service`, aliased as `C`, `A` and `S`, respectively.
– Lines 9–24 define the interaction structure of the protocol. Line 11 specifies a basic message passing action. `query(journey:String)` is a message signature for a message with header (label) `journey`, carrying one payload element within the parentheses. A payload element is an (optional) annotation followed by a colon and the payload type, e.g. `journey` details are recorded in a `String`. This message is to be dispatched by `C` to be received by `A`.
– The outermost construct of the protocol body is the `rec` block with label `Loop`. Similarly to labelled blocks in e.g. Java, the occurrence of a `continue` for the same label within the block causes the flow of the protocol to return to the start of the block. The first `choice` within the `rec`, decided by `C`, is to obtain another quote (lines 11–14: send `A` the `query` details, receive a `price`, and `continue` back to the start), or to accept/reject a quote. The latter is given by the inner `choice`, with `C` sending `ACCEPT` to `A` in the first case and `REJECT` in the second. In the case of `ACCEPT` (lines 17–20), `A` forwards the confirmation to `S` before `C` and `S` exchange `Address` and `Date` messages; otherwise, `A` forwards the `REJECT` to `S` instead.

3.4 Scribble Projection and Verification

After the specification of the global protocols, the next step of the Scribble framework (Figure 1) is the *projection* of local protocols from the global protocol for each role. In comparison to languages implemented from binary session types, such as Sing# [15] and SJ [22], this additional step is required to derive local specifications for the endpoint implementation of each role process from the central global protocol specification. Scribble projection follows the standard MPST algorithmic projections, with extensions for the additional features of Scribble, such as the subprotocols and conversation interrupts mentioned above [38].

Figure 3 lists the local protocol generated by the Scribble tools [39] as the projection of the `BookJourney` for the `Customer` role, as identified in the local protocol signature. Projection preserves the dependencies of the global protocol, such as the payload types used, and the core interaction structures in which the target role is involved, e.g. the `rec` and `choice` blocks, as well as payload annotations and similar protocol details. The well-formedness conditions on global protocols allow the projection to safely discard all message actions not involving `C` (i.e. messages between `A` and `S`).

As for the binary session languages cited above, it is possible to statically type check role implementations written in endpoint languages with appropriate MPST programming primitives against the local protocols following the standard MPST theory: if the endpoint program for every role is correct, then the correctness of the whole multiparty system is guaranteed. The endpoint languages used in the Scribble industry projects, however, are mainstream engineering languages like Java and Python that lack the features, such as first-class communication channels with linear resource typing or object alias restriction, required

```
module TravelAgency_BookJourney_Customer;

type <py> "types.IntType" from "types.py" as Int;
type <py> "types.StringType" from "types.py" as String;
type <py> "travelagency.Date" from "Date.py" as Date;

local protocol BookJourney_Customer at Customer
      (role Customer as C, role Agency as A,
       role Service as S) {
  rec LOOP {
    choice at C {
      query(journey:String) to A;
      price(Int) from A;
      continue LOOP;
    } or {
      choice at C {
        ACCEPT() to A;
        Address(String) to S;
        (Date) from S;
      } or {
        REJECT() to A;
} } } }
```

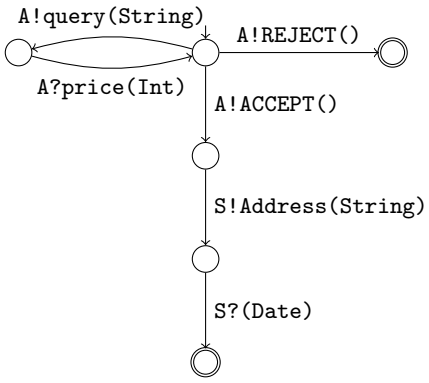

Fig. 3. (a) Scribble local protocol for `Customer` projected from the `BookJourney` global protocol, and (b) the FSA generated from the local protocol by the Scribble conversation monitor

to make static session typing feasible. In Scribble practice, the Conversation API (see § 3.5) is used to perform the relevant conversation operations natively in these languages, making static MPST type checking intractable. In general, distributed systems are often implemented in a mixture of languages, including dynamically typed languages (e.g. Python), and techniques such as event-driven programming, for which the static verification of strong safety properties is acknowledged to be difficult.

For these reasons, the Scribble framework, differently to the above session languages, is designed to focus on *dynamic verification* of endpoint behaviour [23]. Endpoint monitoring by the local Conversation Runtime is performed by converting local protocols to communicating finite state automata, for which the accepted languages correspond to the I/O action traces permitted by the protocol. The conversion from syntactic Scribble local protocols to FSA extends the algorithm in [11] to support subprotocols and interrupts, and to use nested FSM (Finite State Machine) for parallel conversation threads to avoid the potential state explosion from constructing their product. Figure 3 depicts the FSA generated by the monitor from the `Customer` local protocol. The FSA encodes the control flow of the protocol, with transitions corresponding to the valid I/O actions that `C` may perform at each state of the protocol.

Analogously to the static typing scenario, if every endpoint is monitored to be correct, the same communication-safety property is guaranteed [4]. In addition, since the monitor verifies both messages dispatched by the endpoint into the network and the messages inbound to the endpoint from the network, each conversation monitor is able to protect the local endpoint within an untrusted network and vice versa. The internal monitors embedded into each Conversation runtime function perform synchronous monitoring (the actions of the endpoint are verified synchronously as they are performed); Scribble supports mixed configurations between internal endpoint monitors and asynchronous, external monitors deployed within the network (as well as statically verified endpoints, where possible) [9].

3.5 Conversation API

This subsection describes Python endpoint implementation of Scribble. The Python conversation API offers a high level interface for safe conversation programming and maps basic session calculus primitives to lower-level communication actions on a concrete transport. In short, the API provides functionality for (1) session initiation and joining and (2) basic send/receive. Figure 4 illustrates the conversation API by presenting an implementation in Python of the Customer role.

```
class Customer:
  customer, A, S = ['customer', 'agency', 'service']

  def book_journey(self):
    conv = Conversation.create('BookJourney', 'config.yml')
    with conv.join(customer, "\\address...") as c:
      for place in self.destinations:
        c.send(A, 'query', place)
        msg = c.recv(A)

        if msg.value<=self.budget()
          c.send(A, 'ACCEPT')
          c.send(A, 'Address', 'SE2 6UF')
          date = c.recv(S)
          self.save_the_day(date)
          return

      c.send(A, 'REJECT')
```

Fig. 4. Python implementation of Customer role

Conversation initiation Line 5 initialises a new session, using the class named `Conversation`. When creating a session, we specify the protocol name `BookJourney` and a configuration file, holding the network addresses for all roles.

`Conversation.create` creates a fresh conversation id and sends an invitation message for each role specified in the protocol. The invitation mechanism is needed to map the role names to concrete addressable entities on the network and to propagate this mapping to all participants. In Line 6, after initialisation, the process joins (`joins`) the session as Customer role. By `conv.join`, it returns a communication channel `c` to be used for the message exchange during the session. The explicit use of a conversation channel `c` in the program makes it possible to build the application logic with a clear understanding on the session control flow.

The next part of the code iterates over a list of travel destinations, following the interaction flow specified in the `BookJourney` protocol in Figure 3. In each iteration Customer sends a message to `A` (line 8) and then it receives a reply (line 9) from `A` with the price for the booking. Then Customer can end the session in two ways: (1) tf the price for a `place` (`msg.value`) is acceptable (line 11), Customer completes the booking by sending an `ACCEPT` message (line 12) to `A`; (2) if none of the prices are good, Customer sends `REJECT` message (line 18) to `A` and the session ends.

Conversation message passing The primitives for sending and receiving specify the name of the sender and receiver role respectively. All messages are sent or received as a tuple of an operation and a payload, accessible via the message attributes `op` and `value`. The API does not mandate how the operation field should be treated and allows the runtime freedom to interpret the operation name in various ways, e.g. as a plain message label, an RMI method name, etc. A syntactic sugar such as an automatic dispatch on method calls based on the message operation is possible. The sending operation is asynchronous, meaning that a basic send does not block on the corresponding receive; however, the basic receive does block until the complete message has been received.

4 Extensions of Scribble: Subprotocols and Interrupts

The following gives two further examples to demonstrate additional features of Scribble motivated by application in practice.

The first example demonstrates the abstraction of protocol declarations as *subprotocols*, and the related feature of protocol declarations *parameterised* on payload types and message signatures. Figure 5 gives an alternative specification for the Travel Agency example that is decomposed into four smaller global protocols.

`ServiceCall` specifies a generic call-return pattern between a `Client` and a `Server`.
The message signatures of the two communications are abstracted by the `Arg` and `Res` parameters, declared by the `sig` keyword inside the angle brackets of the protocol signature.

```
1  global protocol CustomerOptions
2       (role Customer as C, role Agency as A, role Service as S) {
3    choice at C {
4      do GetQuote(C as Customer, A as Agency);
5    } or {
6      do Forward<ACCEPT()>(C as X, A as Y, S as Z);
7      do ServiceCall<Address(String), (Date)>(C as Client, S as Server);
8    } or {
9      do Forward<REJECT()>(C as X, A as Y, S as Z);
10 } }
11
12 global protocol GetQuote
13      (role Customer as C, role Agency as A, role Service as S) {
14   do ServiceCall<query(String), price(Int)>
15                       (C as Client, A as Server);
16   info(String) from A to S;
17   do CustomerOptions(C as Customer, A as Agency, S as Service);
18 }
19
20 global protocol ServiceCall<sig Arg, sig Res>
21      (role Client as C, role Server as S) {
22   Arg from C to S;
23   Res from S to C;
24 }
25
26 global protocol Forward<sig M>(role X, role Y, role Z) {
27   M from X to Y;
28   M from Y to Z;
29 }
```

Fig. 5. Decomposition of the `BookJourney` global protocol using subprotocols with message signature parameters

`Forward` specifies a generic forwarding pattern between three roles, from `X` to `Y` and then `Y` to `Z`. The intent is for `Y` to forward a copy of the same message, so the signatures of the two communications are abstracted by the same `M` parameter.

`CustomerOptions` is the main protocol in this version of the Travel Agency specification, with the same signature as `BookJourney` in Figure 2. It starts with the `choice` of `C` to get another quote, to accept a quote or reject. The main interactions are now built by composing instances of the `Forward` and `ServiceCall` subprotocols. For example, `do Forward<ACCEPT()>(C as X, A as Y, S as Z)` on Line 6 states that the `Forward` protocol should be performed with the target roles `X`, `Y` and `Z` played by `C`, `A` and `S`, respectively, and `ACCEPT()` as the concrete message signature in place of the `M` parameter; `C` sends `ACCEPT` to `A`,

```
1 global protocol InterruptServiceCall(role Client as C, role Server as S) {
2   Arg from C to S;
3   interruptible {
4     Res from S to C;
5   } with {
6     cancel() by C;
7 } }
```

Fig. 6. Revision of the `ServiceCall` global protocol with a request cancel interrupt

who forwards it to `S`. After this, `C` and `S` engage in a `ServiceCall` subprotocol to exchange the `Address` and `Date` messages.

`GetQuote` performs the quote query case of the choice between `C` and `A`, and loops back to the overall start of the protocol. The quote exchange is specified by instantiating the `ServiceCall` with the appropriate role and message signature parameters. To return to the start of the protocol, we recursively `do` the main protocol `CustomerOptions`. The loop is thus specified by the mutual recursion between these two protocol declarations.

The final example demonstrates the Scribble feature for asynchronously interruptible conversations. Unlike the previous features, which involve the integration of session types with useful, general programming language features (code abstraction and parameterisation), conversation interrupts require extensions to the core design of session types [23]. The motivation for interrupts comes from our collaboration use cases, featuring patterns such as asynchronously interruptible streams and interaction timeouts [25], which could not be directly expressed in the standard MPST formulations.

Figure 6 gives a very simple revision of the `ServiceCall` protocol that allows the `Client` to `cancel` the call by interrupting the `Server`'s reply. A key design point is that interruptible conversation segments do not incur any additional synchronisation over the explicit messaging actions (i.e. interrupts are themselves communicated as regular messages). Due to asynchrony between `C` and `S`, the interrupt can cause various communication race conditions to arise, e.g. `C` sending `cancel` before `S` processes the initial `Arg` or after `S` has already dispatched the `Res`. The Scribble Runtime is designed to handle these issues by tracking the progress of the local endpoint through the protocol (as part of the monitoring service). This allows the Runtime to resolve the communication races by discarding messages that are no longer relevant due to the local role raising an interrupt or receiving an interrupt message from another role.

5 Extensions of Scribble: Parameterised Scribble

This section presents Parameterised Scribble (*Pabble*) [30]. Pabble extends Scribble roles with indices, such that each role can represent multiple Scribble partic-

```
1 global protocol MapReduce(role Worker[0..N], group Workers={Worker[0..N]}){
2   rec MOREDATA {
3     Map(int) from Worker[0] to Workers;
4     Sum(int) from Workers to Worker[0];
5     continue MOREDATA;
6 } }
```

Fig. 7. MapReduce protocol in Pabble.

ipants, and each of the participants can be addressed by its index. This extension is a result of applying Scribble to parallel programming, where programs are designed in a way that they can be scaled up to any number of participants, depending on parameters supplied at execution time. Figure 7 shows a simple Map-Reduce protocol in Pabble. This protocol distributes data from one participant (`Worker[0]`) to all other participants (`Workers`, which is a group role shorthand for `Worker[0..N]`), followed by a parallel reduction on the `Sum` operation. The results are sent to `Worker[0]`.

Parallel programming with Pabble starts by defining the global protocol. The global protocol is projected into endpoint protocols. However, in contrast to Scribble endpoint protocols, where a single global protocol will be projected to the same number of endpoint protocols as the number of participants, a Pabble global protocol will convert to a single endpoint protocol. The endpoint protocol represents multiple endpoints grouped together. The details of the projection algorithm are explained in [30].

Then endpoint protocols are used to generate MPI (Message-Passing Interface) code, which makes up communication parts of the parallel application. An example MPI backbone code generated from the `MapReduce` protocol is given in Figure 8. In Figure 8, `Workers_COMM` is a custom MPI communicator, which groups together all processes from process id (called *rank*) `0` to `N`. This is declared in the Pabble protocol on Line 1, `group Workers={Worker[0..N]}`. Line 9 of Figure 8 corresponds to a map operation, which distributes data from the process with rank `0` (7^{th} parameter of `MPI_Scatter`) to all other processes in `Workers_COMM`. Similarly, Line 12 of Figure 8 is the reduction operation, collecting results of applying `MPI_SUM` to pairs of participants to the process with rank `0` (6^{th} parameter of `MPI_Reduce`).

The significance of Pabble lies in the ability to represent scalable protocols, thus it is very useful in representing protocols used in high performance computing, involving hundreds of thousands process units (or participants) with relatively little effort. In Figure 7, the protocol is designed such that `Worker` can be an arbitrary number of participants (`N`). In the generated implementation in Figure 8, `size` on Line 5 represents the total number of processes, and this number is given at execution time by the MPI environment via a command line argument. The rest of the program body adapts based on `size` during the ex-

```
int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size); /* = N+1 */
  // ... Setting up of data and custom communicators ...
  MPI_Init(&argc, &argv);
  while (/* moreData() */) {
    MPI_Scatter(sndbuf0, sndcnt0, MPI_INT,
                rcvbuf0, rcvcnt0, MPI_INT,
                0/*Worker[0]*/, Workers_COMM);
    MPI_Reduce(sndbuf1, rcvbuf1, count1,
               MPI_INT, MPI_SUM, 0/*Worker[0]*/, Workers_COMM);
  }
  MPI_Finalize();
  // ... Freeing memory and destroying custom communicators ...
  return EXIT_SUCCESS;
}
```

Fig. 8. MapReduce protocol in MPI.

ecution, for example, `MPI_Scatter` distributes data to all `N` processes, whatever the value of `N` is.

6 Future Work

The development of the Scribble framework and its application in real-world use cases is ongoing work. The two main use case projects mentioned above are:

Savara [36] is JBoss project developed by Red Hat and employed in a commercial setting by a Cognizant business unit [14]. Savara relies on Scribble as an intermediate language for representing protocols, to which high-level notations, such as BPMN2, are translated to perform endpoint projections and various refactoring tasks. Savara provides a suite of tools for testing of service specifications against the initial project requirements. The testing is based on simulations between the former, represented in Scribble, and the latter, expressed as sequence diagram traces.

The Ocean Observatories Initiative [32] is an NSF-funded project to develop the infrastructure for the remote, real-time acquisition and delivery of data from a large sensor network deployed in ocean environments to users at research institutions. The Scribble framework, including Conversation Runtime monitoring, has been integrated into the Python-based OOI platform. So far, the OOI cyberinfrastructure is mainly running on an RPC-based architecture. The current Scribble integration is accordingly primarily used for the specification of RPC service and application protocols, and the dynamic verification of the Python client/server endpoints.

Below, we summarise some of the active threads in regards to these projects.

Expressivity The Savara project is examining formal encodings between the specification languages commonly used in practice and Scribble (the current translation by Savara is not yet formalised), which is motivating further extensions to Scribble, such as dynamic introduction of roles during a conversation and fork-join conversation patterns. In general, adapting MPST and Scribble to graphical representations will increase the expressiveness of the protocol specification language. Using the native semantics of formal graphical formats for concurrency, such as communication automata [11, 12] and Petri nets, to provide global execution models of conversations is an interesting direction for integrating Scribble protocol specifications with specifications of other system aspects, such as internal endpoint workflows.

Logical Annotations The current phase of the OOI project includes the development of a framework for actor-based interactions over the existing service infrastructure. To support the specification and verification of higher-level application properties above the core message passing protocol, Scribble is being extended with a framework for annotating protocols with assertions and policies in third-party languages. Annotations may be associated to individual messages, interaction steps, control flow structures, roles or protocols as a whole; examples range from basic constraints on specific message values and control flow (e.g. recursion bounds) to more complicated logics for security or contractual obligations of roles. The Scribble framework will accept plugins for parsing and projecting the annotation language, and evaluating the annotations at run-time. This allows the Scribble tools and monitors to be extended modularly with application- and domain-specific annotations, and the dynamic verification approach enables the enforcement of properties that would be difficult or impossible to verify statically without conservative restrictions.

Endpoint Implementations The Savara and OOI use cases implement the Scribble language, meaning the syntax, well-formedness (valid protocol) conditions and projections, as defined by the central language reference [39]. Both implementations also necessarily conform to baseline communication model of Scribble, namely asynchronous but reliable and role-to-role ordered messaging. The Scribble project is currently working on defining an accompanying Conversation Runtime specification. This will provide the reference for Scribble runtime libraries and platforms, including the specification of the key system protocols for conversation initiation, message formats (conversation and monitoring message meta data) and more advanced features such as conversation delegations [24]. This work is towards full interoperability of Scribble endpoints running on different platforms, such as the Java and Python platforms of the above use cases, supported by platform-independent monitoring. This interoperability will also extend to safely combining dynamically and statically verified endpoints within conversations.

7 Conclusion

While the Scribble project is actively proceeding with our collaborators, it is hardly believable that Kohei Honda cannot work anymore in this project. We conclude our paper with some words from Ocean Observatories Initiative Cyber Infrastructure Team (OOI-CI) [32] to Kohei:

A rare cluster of qualities: *Kohei has lead us deep into the nature of communication and processing. His esthetics, precision and enthusiasm for our mutual pursuit of formal Session (Conversation) Types and specifically for our OOI collaboration to realize this vision in very concrete terms were, as penned by Henry James, lessons in seeing the nuances of both beauty and craft, through a rare cluster of qualities – curiosity, patience and perception; all at the perfect pitch of passion and expression.*

Acknowledgements We thank Gary Brown for his comments. The work has been partially sponsored by the Ocean Observatories Initiative, VMware, EPSRC KTS under Cognizant, and EPSRC EP/K011715/1, EP/K034413/1 and EP/G015635/1.

References

1. Web Services Choreography Description Language: Primer 1.0. `http://www.w3.org/TR/ws-cdl-10-primer/`.
2. Advanced Message Queueing Protocols. `www.amqp.org/confluence/display/AMQP/Advanced+Message+Queuing+Protocol`.
3. Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR '08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
4. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FMOODS*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.
5. Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
6. Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global Escape in Multiparty Sessions. In *FSTTCS 2010*, volume 8 of *LIPIcs*, pages 338–351. Schloss Dagstuhl, 2010.
7. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.
8. W3C Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
9. Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *LNCS*, pages 25–45, 2011.

10. Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012.
11. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
12. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013.
13. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
14. Qualit e Cognizant business unit. Zero Deviation Life Cycle. `http://0deviation.com/`.
15. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of EuroSys'06*, pages 177–190. ACM, 2006.
16. Kohei Honda, , Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, , and Nobuko Yoshida. Structuring communication with session types. In *COB'12*, LNCS. Springer. To appear.
17. Kohei Honda. Scribble Examples: (1) Protocols, 2007.
18. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
19. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
20. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Web Services, Mobile Processes and Types. *EATCS Bulletin*, 91:160–188, 2007.
21. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL '08*, pages 273–284. ACM, 2008.
22. Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.
23. Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *RV'13*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.
24. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *Proceedings of ECOOP'08*, volume LNCS 5142, pages 516–541, 2008.
25. Ocean Observatories Initiative. Scribble OOI derivalables. `https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI`.
26. Robin Milner. The polyadic π-calculus: A tutorial. In *Proceedings of the International Summer School on Logic Algebra of Specification*. Marktoberdorf, 1992.
27. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
28. Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Info. & Comp.*, 100(1), 1992.

29. Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *RV'13*, volume 8174 of *LNCS*, pages 358–363. Springer, 2013.
30. Nicholas Ng and Nobuko Yoshida. Pabble: Parameterised Scribble for Parallel Programming. In *PDP*. IEEE, 2014. To appear.
31. Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.
32. OOI. The Ocean Observatories Initiative. `http://oceanobservatories.org/`.
33. Pi4tech home page. `http://www.pi4tech.com`.
34. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
35. Red Hat JBoss. JBoss Community Scribble homepage. `http://www.jboss.org/scribble`.
36. JBoss Savara. JBoss Savara Project homepage. `http://www.jboss.org/savara`.
37. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
38. Scribble Team. Scribble Language Reference. `https://github.com/scribble/scribble-spec`.
39. Scribble Team. Scribble Project github homepage. `http://www.scribble.org`.
40. UNIFI. International Organization for Standardization 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.
41. Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.