Scribbling Interactions with a Formal Foundation*

Kohei Honda[1], Aybek Mukhamedov[1], Gary Brown[2], Tzu-Chun Chen[1], and Nobuko Yoshida[3]

[1]Queen Mary, University of London [2]Red Hat, Inc.
[3]Imperial College London

Abstract. In this paper we discuss our ongoing endeavour to apply notations and algorithms based on the π-calculus and its theories for the development of large-scale distributed systems. The execution of a large-scale distributed system consists of many structured conversations (or sessions) whose protocols can be clearly and accurately specified using a theory of types for the π-calculus, called *session types*. The proposed methodology promotes a formally founded, and highly structured, development framework for modelling and building distributed applications, from high-level models to design and implementation to static checking to runtime validation. At the centre of this methodology is a formal description language for representing protocols for interactions, called Scribble. We illustrate the usage and theoretical basis of this language through use cases from different application domains.

1 Introduction

A fundamental challenge in modern computing is the establishment of an effective and widely applicable development methodologies for distributed applications, comparable in its usability to the traditional methodologies for non-distributed software built on, among others, core UML diagrams and object-oriented programming languages. Though a middle-to-large-scale application is almost always distributed nowadays, and in spite of the presence of an accelerating infrastructural support for portable and reliable distributed components through e.g. clouds, software developers (including architects, designers and programmers alike) are still lacking well-established development methodologies for building systems centring on distributed processes and their interactions. For example, there is no central computational abstractions (comparable to classes and objects) for capturing distributed interactions usable throughout development stages; no UML diagrams are widely in use for modelling distributed applications; no major programming languages offer high-level, type-safe communication primitives, leaving treatment of communications to low-level APIs. In short, we are yet to have a general and tangible framework for developing distributed, communication-centred software systems.

* This work is partially supported by EPSRC EP/F003757, EP/F002114, EP/G015635 and EP/G015481.

We believe that one of the major reasons why it is so hard to even conceive an effective software development framework for distributed systems, is the lack of a core *descriptive framework*, with a uniform conceptual and formal foundation and usable throughout development stages. To illustrate this point, let us briefly examine the descriptive framework in one of the traditional development methodologies for non-distributed software, underpinned by UML diagrams and object-oriented programming. In this framework, the description of computation centres on *objects* (which belong to *classes*) and *operations on objects*, a representative paradigm of sequential computation. Class Diagram in UML and all associated core modelling diagrams such as Sequence Diagrams and State Charts follow this paradigm; and it is supported by many high-level programming languages, including Java, C++, C$\sharp$ and Python.[1]

Types and logics, two linchpins of theories of computing, play fundamental and mutually enriching roles in this traditional descriptive framework. Types, as seen in the now familiar APIs, offer a basic notion of *interface* of subsystems, tightly coupled with the central computational dynamics of this paradigm, i.e. invoking objects and returning results. This dynamics is embodied in high-level programming primitives, which in turn enables cheap and compositional static validation at the compile time [15,29], leading to modular software development. Types are also a basis of logical specifications. In the widely practiced modelling framework known as Design-by-Contracts (DbC) [28], assertions elaborate types with predicates. Assertions are expressive, allowing us to pinpoint practically any property one wishes to specify [21], though automatic validation is not always possible. Assertions offer a refined form of modular software development through compositional behavioural contracts.

In the light of this well-established (and highly successful) engineering framework in the traditional development methodology, a natural question is whether we can build its analogue in the world of distributed processes, centred on common high-level abstraction for modelling and programming, and aiding modular software development on a rigorous theoretical basis.

This paper illustrates our ongoing endeavour to build a core descriptive framework and the associated development environment for large-scale distributed systems based on the π-calculus [30], centring on a simple language for describing interactions, called Scribble. A key insight is that a distributed system can be naturally and effectively articulated as a collection of possibly overlapping structured conversations, and that the structures of these conversations, or *protocols*, can be clearly and accurately described using a type theory of the π-calculus, called *session types* [23, 24, 39]. In Section 2, we discuss how protocols play a fundamental role for modelling and building distributed applications in diverse domains. In Section 3, we introduce Scribble. In Section 4, we present larger description examples in Scribble from real-world use cases. Section 5 outlines a theoretical basis of Scribble. Section 6 discusses a development framework. Section 7 concludes with related and future work.

[1] Having a different origin, functional languages such as Haskell and ML share a common paradigm, data belonging to data types and operations on data.

2 Background: Modelling Interactions through Protocols

Protocols in Interactional Computing. The idea of protocols becomes important for general software development when the shape of software becomes predominantly a collection of numerous distributed processes communicating with each other. Such *interactional computing* is increasingly common in practice, from web services to financial protocols to services in clouds to parallel algorithms to multicore chips. Processes will be engaged in many interleaving conversations, each obeying a distinct protocol: the aggregate of overlapping conversations make up a whole distributed system. Dividing the design into distinct conversations promote tractability because the structure of one conversation in an application are relatively unaffected by other conversations.

A protocol offers an agreement on the ways interactions proceed among two or more participants. Without such an agreement, it is hard to do meaningful interactions: participants cannot communicate effectively, since they do not know when the other parties will send what kind of data and through which channels. This is why the need to describe protocols have been observed in many different contexts in the practice of interactional computing, as we illustrate below.

Needs for Protocols (1): Global Financial Network. ISO TC68, the Technical Committee for Global Financial Services in ISO, recognized the need for a mechanism to register and maintain international financial protocols (FPs) under the auspice of ISO. This has led to the establishment of a working group for FPs, WG4, which is in charge of drafting the evolving global standard for FPs, ISO20022 [41], using high-level models for describing message formats based on UML. The use of high-level models enable flexible engineering, such as compilation to different document format (e.g. XML schemas and ASN.1), semantic matching of message fields, and model-driven development. [20]

However a message format alone cannot describe an FP in its entirety: the flows in which asynchronous messages are exchanged is at the heart of the FPs. In ISO20022, this dynamic aspect of a FP is called its *message choreography* ("Every Business Transaction contains its own Message Choreography" [41]). In spite of its importance, the chair of WG4 observed that the description of the message choreography through the current technology has severe limitations:

1. *It is imprecise:* The descriptions of protocols are unclear, ambiguous and misleading, and legally unusable.
2. *It is incomplete:* It is impossible to describe the structure and constraints of FPs in their entirety up to a suitable abstraction level.
3. *It is informal:* The description cannot be used for formal reasoning about protocols; for checking their internal consistency; for verifying, either by hand or by machine, the conformance of endpoint programs against a given protocol; for code generation; for testing; and for runtime control.

A precise, complete and formal description of message choreography would offer a vital tool for harnessing and governing global FPs. A long-term goal of WG4 is to identify an effective method for describing message choreography of FPs, and use it in future versions of ISO20022.

Needs for Protocols (2): Operating System for Multicore CPUs. We turn our eyes to a basic form of systems software, operating systems. Most commodity computers nowadays are equipped with multi-core processors, which offer an effective way of harnessing high-density transistor circuits without incurring the performance penalties associated with monolithic processors. This trend is expected to continue in future, where many-core processors, whose numerous cores share a high-bandwidth on-chip interconnect, will become a commonplace [5]. As a consequence, computers are increasingly resembling a distributed system, which cannot be effectively utilised by traditional monolithic OS kernels built around shared data structures that suffer from performance and scalability issues in a parallel execution environment.

Barrelfish is a new multi-kernel OS architecture that aims to address the challenge [4]. It is designed to run on heterogeneous multicore machines and is structured as a distributed system of cores that communicate via explicit message passing and share no memory. Early benchmarks on present day multicore computers showed that the performance of Barrelfish is comparable to that of existing commodity operating systems and can scale better to support future many-core hardware [4]. There are clear parallels between Barrelfish OS and a distributed system, and in particular, the importance of having unambiguous specification of communication protocols. However, the current programming development for Barrelfish only offers description of procedural interface, making it hard to ensure compatibility at the level of asynchronous message passing among OS components, a predominant mode in this operating system.

Needs for Protocols (3): Web Services. In web services, applications make an extensive use of communications among components and services through the standardised format and transport technologies (e.g. URI, XML and TCP/HTTP), increasingly combined with other distributed computing technologies such as clouds, messaging and distributed store. Business transactions using web services are often termed *business protocols* because each of them obeys an agreed-upon conversation structure. Web Services Choreography Description Language (WS-CDL) [13] was conceived in W3C as a declarative, XML-based domain-specific language for specifying business protocols. It is also a first standardization effort done in collaboration with the π-calculus experts from academia.

WS-CDL is notable in that its description captures "global" ordering – a *choreography* – of observable behaviour of participants in a channel-based communication. It comprises a rich set of concepts (roles, work units, exceptions, etc.) and general control constructs (sequencing, parallel, conditionals, recursion) for expressing multi-party interaction. At the same time, as a descriptive means for protocols, it has several drawbacks: first, although a subset of WS-CDL has been given a formal semantics using the π-calculus [12], the language as a whole is not equipped with the notion of *projection from a global specification to endpoint specifications* (which is important for deriving communication specification for local participants); and it lacks a clear stratification between specifications and executable programs.

Needs for Protocols (4): Large-scale Cyberinfrastructure. The Ocean Observatories Initiative (OOI) is a large-scale project funded by US National Science Foundation for implementation of a distributed environmental science observatory with persistent and interactive capabilities that have a global physical observatory footprint [14,35]. A key component of the OOI is a comprehensive cyberinfrastructure (CI), whose design is based on loosely coupled distributed services and agents, expected to reside throughout the OOI observatories, from seafloor instruments to on-shore research stations. The CI acts as an integrating element that links the sub-networks of OOI into a coherent system-of-systems and uses a large catalogue of communication protocols among distributed instruments and stakeholders. These protocols are required to be unambiguously specified for the implementation and runtime communication monitoring.

Towards a Descriptive Basis for Protocols. The pervasiveness and complexity of interactional computation in modern and future computing highlight the need for a general and rigorous protocol description framework, usable throughout the software development life cycle, equipped with a clear, transparent semantic basis, and offering foundations for modular software development through computer-aided validation and verification tools. We now illustrate our recent efforts to develop such a framework, centred on a small description language for scribbling protocols.

3 Overview of Scribble

The goal of Scribble is to provide a formal and yet intuitive language and tools for specifying and reasoning about communication protocols and their implementations, based upon the theory of multiparty session types [6,24,43]. Figure 1 gives an overview of this software framework, which we call the *Scribble framework*.

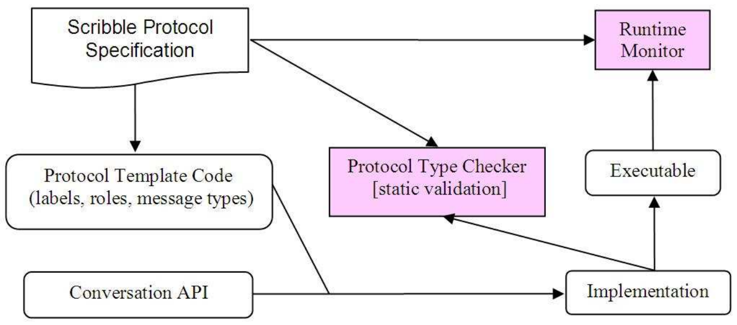
Scribble Protocol Specification
Runtime Monitor
Protocol Template Code (labels, roles, message types)
Protocol Type Checker [static validation]
Executable
Conversation API
Implementation

Fig. 1. Scribble framework overview.

Applications can implement interaction behaviour through the conversation API, a high-level language-independent message-passing interface, to be realized in various high-level programming languages (such as ML, Java, Python, C♯, C++ and others). Static validation is carried out with the aid of a conversation API. In place of the conversation API, we can also use language extensions with intrinsic type checking capability, as studied in [25,26]. The protocol type-checker inspects the application code and decides whether its communication behaviour in a conversation follows the prescribed protocol. Dynamic validation is performed by a monitor that reads in a Scribble protocol specification and inspects runtime communication behaviour of an application. The monitor checks that its interaction follows the behaviour of the corresponding role(s) prescribed by the protocol. For further discussions, see Section 5.

The Scribble framework is currently a work in progress. In the following we present an overview of its underlying protocol specification language, Scribble.

Hello World. We start the overview of Scribble with a customary hello-world example as a protocol, illustrating its basic structure.

```
import Message;

protocol GreetWorld {
  role You, World;
  greet(Message) from You to World;
}
```

The above protocol definition intuitively says:

> The protocol uses `Message` type defined using the `import` statement. Each conversation instance (a run) of the protocol involves two participants – one taking the role `You` and the other taking the role `World`. In each conversation instance, `You` sends a single message to `World`, which consists of the operation name `greet` with a value of type `Message`.

This protocol uses a single interaction (more complex examples will appear later). A *protocol* such as `GreetWorld` gives a global description of interactions among two or more participants. A *session*, or *conversation*, is an instantiation of a protocol that follows the protocol's rules of engagement. *Principals* represent entities, such as corporations and individuals, who are responsible for performing communication actions in distributed applications. When a principal participates in a conversation (i.e. becoming its *participant*), it does so by taking up specific role(s) stipulated in the underlying protocol.

Transport Characteristics. We assume the following properties of the underlying message transport. This is an important assumption to understand the semantics of Scribble.

- *Asynchrony*: send actions are non-blocking.
- *Message order preservation*: the order of messages from the same participant to another participant in a single conversation is preserved.
- *Reliability*: a message is never lost or tampered with during transmission.

These properties may be realised by a transport layer possibly combined with runtime systems at endpoints. They are natural assumptions for many existing transports, be it in Internet, high-performance LAN or on-chip interconnect.

Main Constructs. The top-level grammar of a Scribble description comprises:

(1) At most one *preamble*, which consists of one or more `import` statements: in the `GreetWorld` protocol we have just seen, this is Line 1, importing a message type called `Message`.

(2) A single *protocol definition* (Lines 3–6 in the `GreetWorld` protocol), which consists of the keyword protocol, the name of the protocol (e.g. `GreetWorld`), and the main part – the *protocol body* – enclosed by curly braces.

The protocol body consists of one or more *role declarations* followed by *interaction description*. Roles are placeholders for participating endpoints. When a protocol is instantiated to a concrete conversation, each role, say `You`, is *bound* to a principal, making the latter a participant in that conversation. The behaviour of this participant should follow that of `You` prescribed in the protocol. In the `GreetWorld` protocol, Line 4 gives *role declarations*, which specifies that the protocol description includes two endpoints, `You` and `World`. The grammar of the role declaration is:

```
role role1, ..., roleN;
```

where `roleNamei` is a role name. This is equivalent to:

```
role role1;
...
role roleN;
```

All role names should be distinct and the order does not matter.

The *interaction description* is the main part of the protocol description. There is at most one such description in a protocol specification. It specifies one or more interactions, which belong to a syntactic category called *interaction sentence*.

The grammar of interaction sentences has several forms. Below we describe a few basic types that appear in the current version of Scribble:

(1) Interaction, of the form:

```
msgType from role1 to role2;
```

which defines an *interaction signature* (often simply *interaction*), and reads:

> A participant playing the role `role1` sends a message of type `msgType` to a participant playing the role `role2` and the latter eventually receives the message.

Above the message type `msgType` is imported in an enclosing environment, and can be a base or a composite type. Base type can be a primitive type common to many programming languages, such as `int`, `bool`, or a user-defined type. In the current syntax, composite message types are restricted to an operator name applied to (possibly empty) sequence of base message types: `OpName(ValType1, .., ValTypeN)`. Operator names give clarity to interaction signatures, just as object methods determine its interaction signature in object-oriented programming.

(2) Sequencing, of the form:

```
I1; I2; ...; In
```

represents an interaction sentence, where if the *same role name* appears in both `Ii` and `I(i + k)`, then the interaction actions of that participant take place under a temporal order (thus if none of the role names overlap between `I1` and `In`, then no order is specified). This interpretation is faithful to the asynchronous semantics of communications (formally treated in [24]). For example, in:

```
1   order(Goods) from Buyer to Seller;
2   deliver(Shipment) from Seller to Supplier;
3   confirm(Invoice) from Seller to Buyer;
```

`Seller` sends an invoice to `Buyer` (Line 3) only after it receives an order from `Buyer` and sends a shipment order to Supplier (Lines 1, 2). `Buyer` expects an invoice from `Seller` after it sends an order to `Seller`.

(3) Unordered (also called **Parallel**), of the form:

```
I1 & I2 & ... In
```

represents interleaved interactions that may be observed in any order. We write:

```
msgType from role1 to role2,.., roleK;
```

for a shorthand of:

```
msgType from role1 to role2 & .. & msgType from role1 to roleK;
```

(4) Directed Choice, of the form:

```
choice from role1 to role2,.., roleK {
  msgType1: I1
  ..
  msgTypen: In
}
```

represents interaction flow branching, where role1 makes a choice `msgTypej` to continue interaction following scenario in `Ij`. For example, in :

```
order(Goods) from Buyer to Seller;
choice from Seller to Buyer {
  accept(Invoice):
    payment(CardDetails) from Buyer to Seller;
  decline():
    end;
}
```

After Buyer sends an order to `Seller`, `Seller` makes a choice whether to accepts it or not. If it decides the former, `Seller` returns an invoice to `Buyer` and subsequently waits for a payment in return from him.

(5) Recursion, of the form:

```
rec BlockName { I }
```

where `#BlockName` appears inside `I` at least once, signifying a repetition of the whole block when `#BlockName` is encountered. For example, in

```
rec X {
  order(Goods) from Buyer to Seller;
  choice from Seller to Buyer {
    accept():
      ..
      #X;
    decline():
      end;
  }
}
```

`Seller` can continuously accept orders from `Buyer`, until it decides to decline one. When `Seller` declines an order, the repetition stops.

(6) Nested protocol, of the form:

```
run Protocol(param1,.., paramk, roleInChild1=roleInParent1,.., roleInChildn=
     roleInParentn);
```

represents protocol nesting. When `run` directive is encountered in the interaction flow of a conversation, a new conversation is instantiated and followed as prescribed by the nested `Protocol`. The `Protocol` may require positional arguments, as well as role keyword arguments, by which the roles in the `Protocol` are instantiated with the roles of the enclosing protocol.

Scribble includes other forms of interaction sentences (global escape, delegation, repetition, etc), which we omit for brevity of this presentation, see [37].

4 Scribble Examples

Scribble can be utilised to express communication protocols from a wide range of application domains. In this section we present two examples, taken from web services [13] and from multikernel OS [4, 40].

Web services: Travel Agent. Travel Agent is an interaction scenario designed by the WS-CDL Working Group [13], intended to represent general concepts common to many applications of web services. It comprises multiple participants – a client, a travel agent and a number of service providers – and involves complex branching and repetition in the interaction flow. Figure 2 gives an informal description of the interaction behaviour among the participants.

We present a specification of Travel Agent protocol in Scribble in two parts with: ReserveTravel protocol (Figure 3), by which the client enquires about and reserves travel services with the help of an agent, and PurchaseTravel protocol (Figure 4) for subsequent service booking interaction. PurchaseTravel is parametric in the number of service providers that the agent communicates with in the preceding ReserveTravel protocol. The specification makes use of recursion to represent arbitrary repetition of a series of interactions. In ReserveTravel protocol, lines 6-13 correspond to steps 1 and 2a in Figure 2, lines 15-21 to steps 2b, 3 and 4a. Line 21 uses a nested protocol, `PurchaseTravel`, which describes a successful purchase of travel services by the client (steps 5, 6 and 7 of Figure 2).

1. The client interacts with the travel agent to request information about various services.
2. Prices and availability matching the client requests are returned to the client. The client can then perform one of the following actions:
   (a) The client can refine their request for information, possibly selecting more services from the provider (Repeat step 2). OR
   (b) The client may reserve services based on the response, OR
   (c) The client may quit the interaction with the travel agent.
3. When a customer makes a reservation, the travel agent then checks the availability of the requested services with each service provider.
4. Either
   (a) All services are available, in which case they are reserved. OR
   (b) For those services that are not available, the client is informed.
      – Either
         i. Given alternative options for those services. OR
         ii. Client is advised to restart the search by going back to step 1.
      – Go back to step 3.
5. For every relevant reserved service the travel agent takes a payment for the reservation (credit card can be used as a form of payment)
6. The client is then issued a reservation number to confirm the transaction.
7. Between the reservation and the final date of confirmation, the client may modify the reservation. Modifications may include cancellation of some services or the addition of extra services.

Fig. 2. Travel Agent protocol: informal description.

```
import TravelAgent.messages.*;

protocol ReserveTravel {
  role Client, Agent, Provider[1..num_providers];

  query(Services) from Client to Agent;
  Services_info from Agent to Client;

  rec X {
    choice from Client to Agent {
      more_info():
        query(Services) from Client to Agent;
        Services_info from Agent to Client;
        #X;
      reserve():
        query(Services) from Agent to Provider[1..num_providers];
        Services_info from Provider[1..num_providers] to Agent;
        choice from Agent to Client {
          all_available():
            reserve(Services) from Agent to Provider[1..num_providers];
            run PurchaseTravel(num_providers);
          altern_services():
            Altern_services_info from Agent to Client;
            #X;
          restart(): end;
        }
      quit(): end;
    }
  }
}
```

Fig. 3. Travel Agent: ReserveTravel protocol in Scribble.

```
import TravelAgent.messages.*;

protocol PurchaseTravel(num_providers) {
  rec X {
    choice from Client to Agent {
      cancel():
        cancel(Services) from Client to Agent;
        #X;
      add_services():
        request(Extra_services) from Client to Agent;
        Extra_services_response from Agent to Client;
        #X;
      book():
        Payment from Client to Agent;
        book(Services) from Agent to Provider[1..num_providers];
        confirm(Services) from Provider[1..num_providers] to Agent;
        choice from Agent to Client {
          confirm():
            Receipt from Agent to Client;
          timeout_error():
            Error_details from Agent to Client;
        }
      quit(): end;
    }
  }
}
```

Fig. 4. Travel Agent: PurchaseTravel protocol in Scribble.

Multikernel OS: Distributed USB Manager. Next we present a protocol from a distributed USB manager in Barrelfish multi-kernel OS [4,40], consisting of three primary modules that cooperate via explicit message passing:

– *EHCI host controller driver (HCD).* The host driver manages interaction with the host controller hardware and provides a high-level interface for communicating with the hardware.
– *Client device driver.* The client driver carries out interaction with a USB device and exposes services of the device to applications.
– *USB manager.* The manager is responsible for coordination of the modules and allocation of resources.

The USB manager has the most complex communication logic among these modules, performing orchestration of other components. In Figure 5 we informally describe a USB device `Plug_Unplug` protocol.

Figure 6 presents a specification of `Plug_Unplug` protocol in Scribble. The interaction has a linear structure and its subtlety lies in the correct interleaving of control messages (`ctrl_exe`) with data commands (`dctrl_exe`) between the USB manager and HCD.

– Either, a new device is inserted into one of the USB ports:
  1. HCD notifies the USB manager that a device is inserted.
  2. The manager reads the USB device descriptor (via HCD) that contains the number of configurations, device protocol, class and other information.
  3. The manager reads each configuration reported by the above descriptor, which contain information about power requirements, interfaces and endpoints. The configurations are read twice: at first, to determine the total length of data needed to read interface and endpoint descriptors and subsequently to fetch all interface and endpoint descriptors.
  4. The manager switches the device into addressed mode and crosschecks that by reading device descriptor again.
  5. The manager assigns a configuration and interfaces to the device, which can be later changed by device driver.
  6. The manager locates appropriate client driver by querying System Knowledge Base (SKB). If a match is found, SKB returns the server name running the required driver.
  7. The manager probes the driver if it accepts the new device or not.
     - If the driver accepts the request, it requests the manager to establish a logical connection (pipe) with the device. The pipe is subsequently controlled by HCD.
     - If the driver rejects the request, the manager cleans up its resources.
– OR, a USB device is removed from a port:
  1. HCD notifies USB manager the device is removed.
  2. The manager cleans up its resources and notifies the client driver.

Fig. 5. Distributed USB PlugUnplug protocol from Barrelfush multi-kernel OS.

```
import PlugUnplug.messages.*;

protocol PlugUnplug {
  role HCD, USB_Manager, Driver, SKB;

  choice from HCD to USB_Manager {
    notify_new_device(port):
      // step 2
      dctrl_exe(req,buf,sz,addr,id) from USB_Manager to HCD;
      dctrl_done(id) from HCD to USB_Manager;
      // step 3
      dctrl_exe(req,buf,sz,addr,id) from USB_Manager to HCD;
      dctrl_done(id) from HCD to USB_Manager;
      dctrl_exe(req,buf,sz,addr,id) from USB_Manager to HCD;
      dctrl_done(id) from HCD to USB_Manager;
      // step 4
      ctrl_exe(req,dev,id) from USB_Manager to HCD;
      ctrl_done(id) from HCD to USB_Manager;
      dctrl_exe(req,buf,sz,addr,id) from USB_Manager to HCD;
      dctrl_done(id) from HCD to USB_Manager;
      // step 5
      ctrl_exe(req,dev,id) from USB_Manager to HCD;
      ctrl_done(id) from HCD to USB_Manager;
      // step 6
      get_addr(dev,buf,id) from USB_Manager to SKB;
      get_addr_done(id) from SKB to USB_Manager;
      // step 7
      probe(dev,class,prot) from USB_Manager to Driver;
      choice from Driver to USB_Manager {
        probe_done(ACCEPT,dev):
          pipe_req(dev,type,dir) from Driver to USB_Manager;
          pipe_resp(resp,pipe) from USB_Manager to Driver;
        probe_done(REJECT,dev):
          // clean-up
      }
    notify_device_removal(port):
      disconnect(dev) from USB_Manager to Driver;
  }
}
```

Fig. 6. PlugUnplug protocol in Scribble.

5 Formal Foundations of Scribble

General Ideas. Scribble is formally based on the π-calculus and its type theory called *session types*. Having a general, well-understood theoretical foundation is important since without such a foundation, we cannot establish clear semantics for protocol descriptions, we cannot rigorously analyse how descriptions relate to dynamics, and we cannot accurately state and validate properties of a target system. The π-calculus enjoys full expressiveness for representing interactional behaviours in spite of its tiny syntax: it can mathematically embed a large class of communication-centred software behaviours, including those of existing programming languages, without losing precision. For this reason, the study of session types in the π-calculus, including various validation algorithms, can be directly applicable to real-world programming languages. Below we informally outline the correspondence between the theory of session types and Scribble, including assurance of properties founded on this theory.

Types for Protocols: Session Types. In sequential programming language such as Java and C, a type mainly stipulates the data type of a variable. In particular, in typed languages, all variables should first be declared before they can be used. For example,

$$\mathsf{int\ \ storage} = 1;$$

This program involves stating the type and name of a variable, and telling program that a field named storage exists, holds numerical data, and has an initial value of 1.

Extending this view to interactional computing, session types set the rules for a session (conversation), ensuring safe interactional behaviours for each session. The specification starts from a *global session type* or a *global type* [24], which describes the whole conversation scenario. It gives a specification for the whole protocol from a bird's eyes by giving the rules of conversations for all participants. This global type corresponds to a protocol in Scribble: the Scribble's protocol notation was born from the theories of global types studied in [6], which is the advancement of the theory presented in [24].

A local type represents the type of interactions for each role, played by a principal in a conversation. It is given by projecting the corresponding global type onto a specific role. The following example shows that, in a Buyer-Seller-Broker session s, a Buyer asks Broker for a product, and sends the product's name. Broker refers this request to Seller, then Seller replies to the Broker with the product's price. Broker refers the price to Buyer after receiving it.

$$\begin{aligned} G = {} & \mathsf{Buyer} \rightarrow \mathsf{Broker} : \mathsf{string}. \\ & \mathsf{Broker} \rightarrow \mathsf{Seller} : \mathsf{string}. \\ & \mathsf{Seller} \rightarrow \mathsf{Broker} : \mathsf{int}. \\ & \mathsf{Broker} \rightarrow \mathsf{Buyer} : \mathsf{int}. \\ & \mathsf{end}. \end{aligned}$$

The local types for Buyer are

$$\langle \mathsf{Broker} \rangle ! \langle \mathsf{string} \rangle . \langle \mathsf{Broker} \rangle ? (\mathsf{int}),$$

which are the projection from G onto Buyer. This local types indicate that a Buyer should firstly send a name of type string to Broker, and then wait to receive a price, which is a variable of type int, sent from Broker.

Similarly, the local type for Seller is given as

$$\langle \mathsf{Broker} \rangle ?(\mathsf{string}).\langle \mathsf{Broker} \rangle !\langle \mathsf{int} \rangle,$$

and the local type for Broker is

$$\langle \mathsf{Buyer} \rangle ?(\mathsf{string}).\langle \mathsf{Seller} \rangle !\langle \mathsf{string} \rangle.\langle \mathsf{Seller} \rangle ?(\mathsf{int}).\langle \mathsf{Buyer} \rangle !\langle \mathsf{int} \rangle.$$

Through a global type, we can stipulate the whole set of rules of interactional behaviours participated by all participants; whereas a local type enables the corresponding endpoint (local program) to know the rules of behaviours for a specific role in a conversation.

Safety Assurance by Session Types. The typing system of Scribble for multi-party sessions follows [6], using their global types and projection rules. A well-designed typing system can ensure error-free conversations among multi-party sessions [24], by typing each endpoint with the corresponding local type, which is projected from a stipulated global type. Thus, for a given conversation, we can assure each of its participants plays its role correctly. This assurance can be done effectively at the programming/compilation-time: we can derive a typing algorithm from the tying rules, which can (in)validate that a process, corresponding to an application program at some endpoint, is conforming to a projected local type. Thus session types provide static type-checking at the programming time.

When all endpoints are type-checked and they start interaction, they satisfy several significant properties. First, we have a formal theorem which says that

"well-typed processes never exchange wrong values."

That is, if a process is expecting an integer, it will get an integer; and if it expects a string, it will receive a string. Secondly, inside each session (conversation), there is what is often called *linearity*:

"an output is never shared by more than one inputs, and vice versa."

Finally, we can assure that the interactions through a well-typed conversation follow the initially stipulated protocol:

"interactions inside a session among well-typed processes under a global type, never violate the scenarios given in that type."

This property can be further strengthened under certain conditions that interactions can always proceed in a session, so that they inevitably complete one of the scenarios given in a protocol, assuring an important liveness property. Here by a *liveness* property we mean a property demanding a process can surely do a good thing. In contrast, the preceding three properties are about *safety* since each says that a process never does a stipulated bad thing.

The type-based static checking (including projection) and properties ensured by the typing algorithm give a basis of diverse engineering practice and theories centring on session types. As has been studied in [8], a logical method, which elaborates session types with assertions (just as Design-by-Contract elaborates procedural types with logical formulae), can be built on this basis, which uses precisely the same framework except that it is lifted to logical elaboration of session types. Further, a series of studies show how we can consistently and effectively incorporate session types in the semantics and pragmatics of existing programming languages [19, 25, 26].

6 Development Framework

6.1 General Concepts

Project for Development Environment of Distributed Applications. For Scribble to be useful for development, it should be complemented with associated software tools including programming languages, integrated into a development environment. The present authors, in conversation and collaboration with academic and industry colleagues, started the design and implementation of core development tools centring on Scribble in late 2009, complemented by other activities. Our aim is to reach a simple and effective tool chain for the development of distributed applications which can effectively interface with existing artifacts and tools such as UML and Java. We are still in an early stage of design assessment and prototyping: below we illustrate some of its key ideas.

Modelling with Protocols. The requirement capture phase of the software development life cycle leads to the identification of significant scenarios, or use cases, associated with the target system's usage. For interactional systems, many of such use cases may as well be *conversational* — in the sense that they represent interactions among more than one actor. A use case can then be elaborated into one or more scenarios-as-conversations, each of which will obey a certain protocol: just as an object referred to in a use case scenario belongs to a class.

There are two functions which a tool can provide for this modelling stage: to **edit** protocols (with grammatical checks) and to **validate** their semantic consistency or conformance to other documents. To share protocols with other developers one can also **publish** protocols. One can also **project** a protocol with multiple participants to each endpoint (role), to produce a *local protocol* using the algorithm coming from the underlying theory (see Section 5). This gives a model of conversations from the local viewpoint.

Programming with Protocols. Protocols produced at the modelling stage may be refined into more concrete protocols at the design stage, so that they are eventually usable for implementation. A programmer may also need new protocols just for implementation purposes, as well as using already published protocols. She will then **edit a program**, which uses typed sessions for communications among programs, for example through a Scribble-aware API for communications. She can then statically **check conformance** of her program to stipulated protocols. Protocol descriptions can also be used to **test** programs, where interactions are checked against protocols.

At runtime, each endpoint application is executed through a runtime which links the high-level communication operations for sessions to the underlying messaging infrastructure [25,26]. Multiple such endpoints will converse with each other over multiple conversations, where each endpoint participates in a session taking some role, as specified in the underlying protocol. Communications can be monitored to prevent a conversation from violating a protocol, at each endpoint and/or globally. If some anomaly is detected, a monitor will notify this fact to an entity (a virtual agent) in charge of policy enforcement. The protocol documents also form basic part of the design document of the system.

6.2 Concrete Design

Project for* Scribble*-based Development Environment. Scribble and associated tools are being developed by an open source project hosted at [37], with multiple academic and industry participants. Its purpose is to provide a collaborative environment to support the development of the language Scribble and associated tools. One of the exciting aspects of the project is the collaboration between academia and industry. Within the project, we are aiming to harness the best of these two worlds: leverage the academic results to develop cutting edge capabilities, while providing the stable software development life cycle required to deliver a higher quality and better supported product for use in industry.

Our current design of a tool chain focuses on Eclipse. To enable extensibility in this environment, we leverage the OSGi standard. The tooling includes an Eclipse-based context sensitive editor for Scribble. Since Eclipse is based on the OSGi framework, various Scribble-related functions become automatically available in the editor as OSGi modules become available. Extensibility also facilitates incorporation of new research ideas into existing tool functionalities.

Protocols are parsed by a parser generated by the ANTLR parser generator, producing an internal representation (a Java object model) through the abstract syntax tree. It is this representation which is used and acted upon by various validation modules, discussed next.

Static Validation and Other Algorithms. One of the fruits of the industry-academia collaboration in the Scribble project is the use of the latest research results on validation and other algorithms that are provably correct to give the required results. Each validation module is responsible for processing a Scribble protocol object model to output whether it is valid or not (with respect to a specific criteria). Validations can be arbitrarily chained, and are usually set up so that they are triggered automatically as a protocol description is created or updated in the editor. A new validation function can be added simply by installing an OSGi module implementing the appropriate interface. The main validation functions include:

1. Syntactic consistency (parsability)
2. Semantic consistency of session types (including linearity guarantee, which avoids a race condition in a conversation [24])
3. Conformance checking of a local protocol against a global one [31].

The conformance checking noted above is also used for local protocols extracted from an implementation of an endpoint in an already existing program, written in e.g. BPEL. These validations, together with other functions such as the projection of a global protocol to local protocols (using the latest algorithm from [43]), can be lifted to logical specifications following [8]. Another form of validation is dynamic validation through monitors, using an efficient internal representation of protocols. The design and implementation of these and other validation modules are under way, with a stable release planned in late 2011.

Finally, programming support for Scribble for existing languages comes from two sources: APIs and language extensions. In both cases, a type checker, which validate type correctness of programs against local protocols, plays a key role. Three implementations covering both approaches are under way, based on the latest research on session-based programming and runtime [25, 26].

7 Related Work and Conclusion

In this section we discuss some of the related work, with an emphasis on theoretical studies related to Scribble, and conclude with further topics.

Process Algebra. Process algebras, such as ACP [3], CSP [22] and π-calculus [30], present a semantic framework where interactional behaviour of software systems can be captured on a rigorous mathematical basis through a small set of operators for constructing processes. The fruits from the studies on these and other models of concurrency form an essential engineering foundation of Scribble. For example, behavioural equivalences such as bisimulations, a linchpin of theories of process algebras, offer the mathematical basis of key engineering activities such as optimisations, security, correctness of compiler, correctness of runtime, and various semi-automatic verifications.

Session Types and Other Description Frameworks. Session types [23,39] have been studied over the last decade as a typed foundation for structured communication-centred programming using various programming languages and process calculi. The original binary session types have been generalised to *multiparty session types* [24], in order to guarantee stronger conformance to stipulated session structures when a protocol involves more than two parties. Theories of multiparty session types [6, 24] give the foundations of Scribble, together with their semantic basis given by the π-calculus. Validation and other algorithms from their studies are used as the core elements of its tool chain.

Since [24], the theory of multiparty session types has been extended in different directions, including a theory which ensures the progress property for interleaved multiparty sessions [6] (which also gave the formal basis of the Scribble's syntax); generalised type structures which allow communication optimisation through permutation [31]; and a static analysis for communication buffer overflows [16]. The existing notations for describing protocols include message sequence charts [9,27] and UML sequence diagrams [34] (the latter when method calls are replaced by asynchronous signals). These notations are different in that they are not based on the abstraction of protocols as type signatures. Protocol descriptions from a different viewpoint are studied in [18], where one stipulates possible communication events among endpoints using a logic of commitment. WS-CDL [13] (discussed in Section 2) is one of the first expressive languages which allow description of interactions from a global viewpoint. WS-CDL is also a basis of the preceding validation tool by one of the authors (G.B.), pi4soa [36], on whose experience the design of the Scribble-based development environment is being carried out. In comparison with these descriptive languages, the multiparty generalisation of session types offer, for the first time, a framework of protocol descriptions where they are formally captured as type signature, together with a notion of type conformance through formal projection to endpoints.

Recently the theory of multiparty session types has been applied in different contexts, including protocol optimisation for distributed objects [38]; integrity of session interactions [7, 10]; type-safe asynchronous event programming [25]; safe and efficient parallel programming [32, 43]; multicore programming [44]; and medical guidelines [33]. Many of these studies are inspired by and/or inspire our industrial collaborations.

***Communication-Centred Programming Languages and* Scribble.** Occam-Pi [42] is a highly efficient systems-level concurrent programming language centring on synchronous communication channels, based on CSP and the π-calculus. Hewitt's Actor Model [1] is an influential programming model centring on asynchronous unordered message passing. Erlang [2] is a communication-centred programming language with emphasis on reliability based on actors. Scribble differs from these languages in that it is a protocol description language rather than a programming language, with formal foundations coming from the π-calculus and session types, intended to be used across multiple programming languages through different stages of software development.

Further Topics. To realise the full potential of the proposed approach in general and Scribble in particular, the incorporation of several recent advances of the theory of session types into the description language would be relevant. First, Scribble can be extended to the type-safe multiparty session exceptions recently developed in [11], in order to handle system failure and fault-tolerance in a larger class of distributed protocols, preserving type safety. The incorporation of the parametrised dependent type theory from [43] enables us to directly express more complex communication topologies. More recently, we studied a dynamic multirole session type in [17] where an arbitrary number of participants can dynamically join and leave an active session under a given role. This solved an open problem in the theory of multiparty session types, providing a new framework to handle common distributed communication patterns such as publisher-subscriber or P2P and chat protocols within the theory. The notion of the role with dynamic join capabilities in [17] was motivated by Scribble and protocol descriptions in it, demonstrating an interaction between practice centring on Scribble and academia brings a new theory which can be used to enrich type structures of Scribble. Another significant extension of the theory of multiparty session types, again motivated by a dialogue with practice, is our recent work [8] on an assertion framework built on session types. The incorporation of the assertion-based framework will enrich the expressiveness of Scribble as a tool for description by enabling the specification of fine-grained constraints. The framework generalises the traditional Design-by-Contract, offering a refined modular development framework for distributed communicating processes based on multiparty behavioural contracts.

Acknowledgements. We thank Raymond Hu, Matthew Rawlings and Munindar Singh for their comments. The Scribble project has been enriched by an extensive dialogue with academic and industry colleagues, including, but not limited to, Matthew Arrott, Matthew Rawlings, Steve Ross-Talbot and Olivier Pernet. The first author acknowledges a fruitful discussion with another industry colleague, Antony Alappatt. We thank all of them, and record that this series of a dialogue between practice and theories was made possible by the initiative and leadership of late Robin Milner, who is also the originator of many of the key theoretical foundations on which Scribble is based. For this reason, we dedicate this paper to him, and hope that our work from now on in this domain will contribute to the enrichment of both practice and theories of computing through their deep and honest dialogue, as Robin envisioned in his address in Edinburgh in 1986 and encouraged us in the early days of this endeavour.

References

1. G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
3. J. Baeton and W. Wejland. *Process Algebra.* Cambridge University Press, 1990.
4. A. Baumann et al. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, pages 29–44. ACM, 2009.
5. A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. why isn't your os? In *HotOS'09: Proceedings of the 12th conference on Hot topics in operating systems*, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.
6. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
7. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.

8. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
9. M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.
10. S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session Types for Access and Information Flow Control. In *CONCUR'10*, volume 6269 of *LNCS*, pages 237–252. Springer, 2010.
11. S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty session. In *FSTTCS'10*, 2010. To appear. `http://www.di.unito.it/~capecchi/mpe.pdf`.
12. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
13. W3C Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.
14. A. Chave, M. Arrott, C. Farcas, E. Farcas, I. Krueger, M. Meisinger, J. Orcutt, F. Vernon, C. Peach, O. Schofield, and J. Kleinert. Cyberinfrastructure for the US Ocean Observatories Initiative. In *Proc. IEEE OCEANS'09*. IEEE, 2009.
15. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.
16. P.-M. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010. Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/multianalysis`.
17. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL'11*. ACM, 2011. To appear. `http://www.doc.ic.ac.uk/~malo/dynamic`.
18. N. Desai, A. K. Chopra, M. Arrott, B. Specht, and M. P. Singh. Engineering foreign exchange processes via commitment protocols. In *IEEE SCC*, pages 514–521, 2007.
19. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352, 2006.
20. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, January 2003.
21. T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
22. T. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
23. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
24. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
25. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
26. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
27. International Telecommunication Union. Recommendation Z.120: Message sequence chart, 1996.
28. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
29. R. Milner. Theory of type polymorphism in programming languages. In *TCS*, 1982.

30. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.*, 100(1), 1992.
31. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, number 5502 in LNCS. Springer, 2009.
32. N. Ng. *High performance parallel design based on session programming.* Masters thesis, Department of Computing, Imperial College London, 2010. `http://www.doc.ic.ac.uk/~cn06/individual-project/`.
33. L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sumtypes. Technical Report 8, Department of Computing, Imperial College London, 2009. To appear in Express'10. Apims Project at: `http://www.thelas.dk/index.php/apims`.
34. OMG. Unified Modelling Language, Version 2.0, 2004.
35. Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.
36. pi4soa homepage. `http://pi4soa.sourceforge.net/`.
37. Scribble development tool site. `http://www.jboss.org/scribble`.
38. K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *Coordination'10*, volume 6116 of *LNCS*, pages 152–167. Springer, 2010.
39. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
40. A. Trivedi. Hotplug in a multikernel operating system. Master's thesis, ETH Zurich, 2009.
41. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.
42. P. Welch and F. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.
43. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCs'10*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.
44. N. Yoshida, V. T. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *FMCO'08*, volume 5751 of *LNCS*, pages 226–246. Springer, 2009.