Structuring Communication with Session Types

Kohei Honda[1], Raymond Hu[2], Rumyana Neykova[2], Tzu-Chun Chen[1], Romain Demangeon[1], Pierre-Malo Deniélou[2,3], and Nobuko Yoshida[2]

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

Abstract. Session types are types for distributed communicating processes. They were born from process encodings of data structures and typical interaction scenarios in an asynchronous version of the π-calculus, and are being studied and developed as a potential basis for structuring concurrent and distributed computing, as well as in their own right. In this paper, we introduce basic ideas of sessions and session types, outline their key technical elements, and discuss how they may be usable for programming, drawing from our experience and comparing with existing paradigms, especially concurrent objects such as actors. We discuss how session types can offer a programming framework in which communications are structured both in program text and at run-time.

1 Introduction

This paper illustrates a structuring method for distributed computing based on session types [19, 20, 29]. We take the standpoint that communication is an essential building block for concurrent and distributed computation and that there is a strong prospect that both software and hardware engineers need to position this notion as a foundation of their design activities. Under this assumption, we seek a general principle for structuring communications as a basis to facilitate the development of correct and efficient programs. Computation based on communication is so rich – it certainly includes the whole of sequential and shared variable computation – that it looks hopeless to identify a principle which may apply to its different realisations. There is also a difficulty inherent in communication as we discuss in the next section. Given these potential difficulties, instead of looking for general principles, we may be content with having different techniques depending on different classes of use cases and different levels of expertise. But we believe this difficulty should not deter us from our quest towards a unifying foundation since only with such a foundation we can start to harness the richness of the large class of behaviours realisable through communication and concurrency, providing a guide for individual problems and giving a basis upon which different techniques can be positioned and integrated with greater benefits than isolated solutions.

A central idea for structuring communications in session types is to divide them into chunks of inter-related interactions forming logical units, called sessions. Each session, in its own temporal-spatial confine, consists of messages

which are clearly identifiable as belonging to that session. The term "session" comes from the networking community where such a classification has been practised for a long time, albeit informally. Each session is associated with its protocol, specifying how its participants may interact with each other, which gives a type for the session in the sense that it classifies interaction structures, and that they are directly linked to programming primitives as a formal specification, just as types for functions and methods are directly linked to their underlying primitive. This is how protocols arise as types when programming with sessions. We illustrate this framework more concretely in Section 2.

The study of session types over the past two decades has extensive interactions with other threads of research. Session types were born from a desire to articulate the abstract structures arising from idioms that repeatedly occur when we encode high-level data types and programs in the asynchronous version of the π-calculus [23], which in turn was influenced by actor model. Theories of concurrency, in particular process algebras such as ACP [3], CCS [21] and CSP [16], offered mathematical foundations of session types: the research on concurrent languages based on actors and concurrent objects also played an important role in the inception of session types. These languages include the ABCL family of programming languages starting from [32], developed by Akinori Yonezawa and his team, which is one of the prominent accomplishments in the study of concurrent languages and formed a cultural background of the initial introduction of session types.

This paper is intended for a concise presentation of key ideas as well as some of the open topics. We also provide comparisons with related programming and software development methodologies. For technical details, we hope the reader can consult citations in each section. Section 2 gives the background of session types. Section 3 introduces its programming methodology informally through examples. Section 4 discusses one of its application examples. Section 5 compares our approach with other framework for concurrent programming with a focus on concurrent objects and actors, and concludes.

2 Background

2.1 Structuring Sequential Programs

Computing in its modern sense started from the discoveries in 1930s and 1940s of abstract and concrete machines which are in nature sequential. Among them, the abstract machine by Turing and its crystallisation as an engineering design by Von Neumann offered the combination of striking simplicity and universality with a finite state automaton as the processing unit and a linear array of memory cells as the workspace for the automata (designated as a "tape" containing many squares in Turing's model: symbols are read from and written to these squares by an automata). This simple machine model was to be explored extensively by generations of engineers, developing faster processing units and larger memories with high-bandwidth for reading and writing. By Turing's result, engineers know

that, just by focusing on these two key elements (the processor and the memory) and enlarging their capabilities, the machine can simply get better.

It is on this stable hardware model that the fundamental programming abstractions for sequential computing were developed, from assemblers to a simple notion of control flows and data types, to procedures and the structured programming discipline, to dynamically created data structures with multiple operations (objects), to higher-order procedures. The stable and universal hardware model makes it possible, assisted by other fundamental theories including, among others, the λ-calculus and its type theories, to incrementally build up layers of abstractions that assist designers and programmers to describe the intended behaviour with clear structures understandable by the programmer and his/her fellow colleagues, as well as by compilers which perform static checking of programs. Without good structures, it is hard for both humans and machines to understand programs' semantics.

This point is well-articulated by Dijkstra, when he advocates the structures programming discipline in his famous communication [13]:

> Our intellectual powers are rather geared to master static relations. [...] For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text) and the process (spread out in time) as trivial as possible.

Implicit in this observation is that the dynamic process realisable by the structured presentation of programs has the same expressive power as the "unstructured" method. Another observation underlying Dijkstra's remark is the fact that a formal basis for the structuring method, such as Hoare logic for structured programming constructs, can pinpoints the status of the method.

The quote above also indicates a crucial element for any effective structuring method for programming: we obtain abstraction and good structure so that we can map the resulting program text tractably into efficient code, since without the existence of such a mapping, it is hardly expected that we can make the correspondence between program text and how it will be executed "as trivial as possible."[1] And for this correspondence to be judged to be effective for a high-level programming language, we needed a stable machine model which not only underlies the existing hardware products but also would underlie for potential ones.

2.2 Communication and Concurrency

Communicating processes are at the heart of computing since early days of computing. While, as we have just discussed, computing has been based on the

[1] Note that this correspondence is preserved, albeit not too trivially, even for dynamic data structures such as objects, by a stable compilation strategy based on class tables. Such a basic correspondence is a basis for individual optimisations for architectures.

most effective sequential model, scientists and engineers quickly found the use of networking in combination with computing machinery, especially in the shape of packet-switching networks that deliver digital data throughout networks with effective use of the capacity of wires and flexibility which is not possible through circuit-based networks. This is done through the help of intermediate nodes which act as exchanges of data packets.

On this basis, at the network engineering level, we saw the emergence of the idea of inter-networking, which links multiple networks, born and crystallised as the TCP/IP combination of protocols [7]. This protocol was later split into the two components as we know now based on the understanding on the end-to-end principle [26], leading to the scalable inter-network infrastructure now known as Internet, which was eventually to span the globe. Around the time when TCP/IP was being engendered and incorporated as part of the then nascent Internet, many studies on communicating processes, in abstract models, programming languages and verifications were initiated, on which we shall discuss later.

In Internet, after several notable applications had been developed such as electronic mails based on corresponding application-layer protocols, we saw an invention of a simple but useful idea to implement hyperlinks over Internet, embodied in the document format HTML and the application-layer protocol HTTP. HTTP, a simple protocol based on server-client interactions performed in a TCP-connection, has turned out to be a great medium for providing services to users, by which the user base of Internet has undergone an explosive growth. Later we found other applications of Internet, such as Internet Telephony as well as social networking, leading to the proliferation of web services, where many businesses become Internet-based and have global presence, be they bookshops, music or flower delivery. The resulting socio-technical complex is to be called World-Wide Web.

Global services in the World-Wide Web need to cope with a large number of clients. This in turn necessitated the development of server technologies, to be used for the backend of these web services. Combined with virtualisation technologies of OSes and networks, this has led to a set of technologies by which multiple users can share a gigantic interconnected network of commodity hosts as if each has its own network and computing resources, leading to cloud computing. Cloud computing is giving at least three impacts. First it allows every user an opportunity to use large amount of computing resources economically. Second, it allows diverse networking technologies to be experimented without interfering with other users. Thirdly, it offers users an economical platform where an embarrassing amount of concurrency and distribution are the norm rather than a marginal concern.

The cloud computing has become prominent in the first decade of the 21st century. Not neglecting other factors, an insight which the cloud computing may give us is that, to share computation, that computation had better be distributed. This is a physical problem, having the same root as the following observation by Hoare on multi-processor architecture several decades ago [16]:

> [...] Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer [...]. However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

In brief, there is a limit to share a large amount of computing power in the sequential form (or, in Hoare's words, to "disguise" it to be sequential), due to the existence of latency. In spite of all the engineering efforts to achieve the contrary, we see a clear slowing-down of sequential performance of representative CPUs at the beginning of the 21st century, fulfilling Hoare's prediction in a globally aware form. Since then, the architectural development centres on increasing parallel performance through many cores. This is also in line with the architectural evolution of super computers, which, after prominent instances such as Blue-gene showed their performance merits, have turned into communication-centred designs.

2.3 Structuring Communication

Thus, in all scales of computing, communication is becoming one of the major elements. And theoretical results such as Milner's embedding of the λ-calculus into the π-calculus [22] confirms their status as an expressive computing primitive. However, as Hoare himself observed when he introduced CSP, communication is hard to harness, in both design and formal verifications, which is one of the reasons why Hoare and Milner have chosen synchronous interaction. Can we find a tractable way to specify and manage communication in program texts and runtime?

There is however subtlety in this question itself: what is it to which we aim to give a good structure? Sequential computation has a stable execution basis, in the abstract models and in concrete machine instructions. But communication is different. Either inside a chip, among different machines in a cluster or across continents, communication is always mediated by intermediate infrastructure, be it on-chip interconnect and buffering facilities in a manycore chip, Ethernet bus and drivers, or IP routers. Communication is not a hardware primitive at the same level of assignment, and never will be. Thus it is hard to determine and agree on what would count as basic *primitives* for communication. And if we cannot identify primitives, how can we think of the structuring method for them?

Answering this problem is hard because communication is useful, after all, because it is between two computing machines: there may not be only one way to realise it. Session types started from a theory of processes based on asynchronous communication based on the π-calculus which is also close to the actor model. This theory, introduced in [5, 18], is interesting in that it is a sub-set of the π-calculus, which itself is based on synchronous, handshake computation, but just

by taking its subset, now represents asynchrony. This also suggests all theories developed in CSP, CCS and the π-calculus are now applicable to asynchronous theory. These theories show that, at the foundational level, we can indeed have a rigorous theory of asynchronously communicating processes, with an exact notion of behaviours, their equivalence, and logical specifications. But having a general theory does not dispel the theoretical intractability, and accompanying mental intractability, of asynchrony: assuming we use large or infinite buffering in communication, it looks hard to reason about behaviours (consider model checking interactional behaviours with infinite buffering).

It is here that the notion of protocols and session types comes in, on which we shall discuss in the next section.

3 Multiparty Protocols and Sessions

3.1 Session Types

One of the outcomes of using very large-scale integration for implementing a central processor of a computer is that, to link "remote" areas of a single chip (since we want different cores to share data), we need to rely on asynchronous communication. This is for the following simple reason: if we need to have two computations to be not too closely synchronised, that is (as we want when two different cores to calculate two parts of computation) if we want their computations to proceed independently unless absolutely necessary, the only way is to link them with a buffered communication medium, which an on-chip interconnect in VLSI readily provides. Note that a relative independence in processing also means that we can overlap computation and communication, which is a major method to make the most of distributed computing resources.

But this very asynchrony also poses a problem in understanding computation: the "dynamic process", as Dijkstra called it, of asynchronously communicating processes looks hard to harness, because, simply put, all different ways in which the messages can be buffered add new states in potential computations, making the reasoning extremely difficult. For example, if a process changes its state on each occasion when a new message is received, and each sending action depends on this state, then unbounded buffering means unbounded states and behaviours.

It is to harness this untenable nature of asynchronous communicating processes that has led to the birth of the structuring method for communications programming based on sessions and session-based primitives (creating sessions and communication through sessions), together with the underlying types which offer a way to specify protocols for sessions as types, drawing from the study of the π-calculus and its type theories as well as the foregoing studies on types in programming languages. By restricting asynchrony by protocols, we can reduce the size of state space to be considered, for each session and interleaved sessions, ensuring safe interactions by static checking and giving a basis for understanding and verifying behaviour. It has the equivalent expressive power as the original primitive of the π-calculus, which is known to possess a universal computing power for interactions in a certain technical sense.

The original session typed π-calculi are based on synchronous communication primitives, assumed to be compiled into asynchronous interaction: later researchers found that, if we assume *ordered* asynchronous communications for binary interactions inside a session, the original synchronous theory of safety can be preserved while directly expressing asynchronous interaction. This safety theory includes the simple fact that the type of a message by a sender coincides with what a receiver expects inside a multiparty dialogue, which is practically important because such an error costs a lot more in asynchronous communicating processes than in sequential computing.

3.2 Writing Protocols

Session types describe a way, or a pattern, in which interactions can take place in sessions. Session types have been called *protocols* for many years in network and other engineering disciplines which need to treat such patterns. For this reason, and because session types are sufficiently different in nature from data types, we know in sequential computing (although the former share the key principle from data types as we shall discuss later), hereafter we often use the term "protocols" instead of "session types" when discussing their use for programming.

One of the key ingredients of session-based programming is the use of protocols as an essential element of design and programming, because a clear understanding of an interaction scenario is an essential ingredient of communications programming. For this reason, one of the key features of programming with sessions is a *protocol description language*, the language with which engineers read and write their protocols. They are close to types in sequential programming: like data and function types, there is a tight linkage to language primitive. Like data and function types, protocols may be inferred from programs or declared by programmers so that programs may be checked against them. A difference is that a protocol describes interactions for a session, and that, for this reason, each session involves a sequence of interactions (which may not necessarily be contiguous, since interactions in other sessions or internal computation may interleave).

A Simple Protocol. To illustrate how we can specify a protocol, we take a simple scenario, and show how the corresponding protocol can be specified using an experimental protocol description language we are developing, called Scribble [17, 27, 28] (the name comes from our desire to create an effective tool for architects, designers and developers alike to quickly and accurately write down protocols).

A key feature of Scribble is that all of its constructs are fully founded on the formal theory of multiparty session types, starting from the core language features for message passing, choice and recursion [4, 20], to more advanced features, such as parallel [10], interrupts [6], sub-sessions [9] and run-time monitoring [8], and studies relating session types to alternatives such as communicating automata [10]. The development of Scribble is a collaboration between researchers and industry partners [24, 27]. Most of the examples presented in this section are

```
1  type <ysd> "ListingFormat" from "ListingFormat.ysd" as Format;
2
3  protocol ListResources(role client as cl, role resource_registry as rr) {
4     request(resource_kind:String) from cl to rr;
5     rec loop {
6        choice at rr {
7           response(element:Format) from rr to cl;
8           continue loop;
9        } or {
10          completed() from rr to cl;
11       }
12    }
13 }
```

Fig. 1. A protocol for the List Resources use case

supported by the current working version of Scribble [28], with a few exceptions that we note as being planned for future release.

The initial scenario we treat is called "List Resources", where a Client obtains a list of resources of some kind from a Resource Registry. This is a basic use case applicable to many environments where a user may be provided with a variety of resources by the infrastructure, e.g. remotely operable instruments or systems resources such as bandwidth. The scenario consists of two steps:

Step 1: Client asks Registry to send her a resource list, specifying the kind of resources it is interested in.

Step 2: Registry responds by sending the list of the resources of the kind specified, until the list is exhausted.

It is a simple elaboration of a remote procedural call. Note, however, that Step 2 involves a repetition of sending actions. This use case may be further elaborated in various ways, but this simple version is sufficient for our first exercise.

Writing down a protocol goes through a natural flow, practised for decades in the networking community. We first list the *message formats*, followed by the participating *actors* (and other parameters). Then we scribble away the structure of the *conversation* between the actors. The result for our mini use case is given in Figure 1.

Line 1 starts from importing an message type `ListingFormat`, specified in YAML (`ysd`), from the external source (file) `ListingFormat.ysd`. This message type can then be referred to in this Scribble protocol specification by the given alias `Format`. (In the coloured presentation of this paper, the `import` and `as` are coloured **blue**, signifying they are keywords.) Message type imports allow Scribble to be used in conjunction and orthogonally with externally defined message formats: here we are using a YAML schema, but any data format given in a well-defined schema/type language may be used as far as the protocol validator is notified. Data format is of course fundamental in protocols to ensure interacting parties understand what the other is saying.

In Line 3, we give the name to the protocol, `ListResources`, followed by its parameters. The parameters consist of the names of the two actors roles which

participants can play, `client` and `resource_registry`, aliased as `cl` and `rr` (short names are often good for scribbling away protocols). This completes the header of the protocol.

The remaining lines (Lines 4–13) constitute the *protocol body*, which describes the structured flow of the conversation in a session. We have the first interaction described in Line 4, which reads:

> *A* `request` *message whose content, annotated as* `resource_kind` *and typed as* `String`*, is sent from* `cl` *to* `rr` *asynchronously.*

In Line 4, `request` is the message *operator*; `String` (which is a built-in type for strings) is a message *payload* type, and `resource_kind` is the payload annotation (a simple name). Finally `from` and `to` specify the source and destination, respectively.

Line 4 is reminiscent of a method/function declaration found in APIs and modules of high-level sequential programming languages: an interaction signature is a symmetric, peer-to-peer version of the familiar notion of "interface" of functions and objects. As such, Line 4 does *not* specify constraints on *concrete values* a message may carry, but specifies only the *type* of an interaction. For this reason, we call the description in Line 4 as a whole, an *interaction signature*.

Registry now responds through a sequence of one or more messages: in the protocol, we use a light form of labelled recursion for such repetition. Line 5 declares the *recursion label* `loop` that names the *recursion body* starting from Line 6 and reaching Line 12. The recursion body consists of a single *choice* statement.

The choice construct starts from Line 6, which first declares the choice: `at rr` says that it is the Registry who will be the deciding party of this choice, through a subsequent send action.

Lines 7–8 and Line 10 are respectively two distinct *branch*es of the choice, separated by `or` on Line 9. In the first branch, Line 7 says that Registry sends a `response` message to Client, with message content annotated as (list) `element` and typed as `Format`. Again we specify only a sender, a receiver and a message signature. This is followed by Line 8, a *recurrence* denoted by the `continue` keyword, which says that the protocol flow at this point returns to the start of the recursion body labelled by `loop`, i.e. to Line 5.

The other branch consists of a single interaction, Line 10, where a `completed` message with an empty payload is sent from Registry to Client, indicating the end of the list, i.e. the end of the recursion – since there is no recurrence, the loop terminates if this branch is chosen. As described in Step 2 above, at the level of the application logic, the repetition should terminate only when all the resource data for the specified kind has been sent by Registry: our protocol description again abstracts from exactly how this may be determined in the program logic (although the protocol assertions we discuss later can constrain this behaviour in some way or another). After this action, the flow exits the choice and the recursion, and (since no further interactions are specified) the session terminates.

```
1  protocol ListResources<type ListingFormat as Format>
2       (role client as cl, role resource_registry as rr) {
3     request(resource_kind:String) from cl to rr;
4     rec loop {
5         choice at rr {
6             response(element:Format) from rr to cl;
7             continue loop;
8         } or {
9             completed() from rr to cl;
10        }
11    }
12 }
```

Fig. 2. A refined List Resources protocol (1)

Nature of Protocols. We have seen a simple but self-contained protocol (session type), `ListResources`. Even from this simple example, we can find unique features of protocols. First, a protocol is like an API in that it defines a *contract*, but this contract is not just between a function and its user, but among conversing agents. Further, a protocol describes a *series* of interactions, with conditional and repeated segments, because conversations among distributed agents will often involve more elaborate structures than call-return. Like APIs, a protocol only offers a bare minimal behavioural specification, without constraining values nor conditions for actions. This paucity has a practical merit: minimal notations are needed for reading and writing basic protocols; they are amenable for efficient validation at both compilation time and at runtime; and they can serve as a minimal sufficient basis for elaborating them with refined behavioural constraints through, among others, assertions.

Elaborating Protocols. For protocols to assist computer software development, be it a newly built system or an upgrade of an existing system, they had better be *reusable*, i.e. once you author a protocol, it should be able to be used for many concrete applications. From this viewpoint, the `ListResources` protocol in Figure 1 may not be fully satisfactory. In particular, it works only for the message type defined in the specification by the concrete `ListingFormat` YAML schema. Even if only one listing format is known now, new formats may arise later. Why should we write different protocols for all different formats, given the structure of interactions is identical? We use a basic technique from programming theory, *parametrisation*, to solve the problem.

There are at least two different, and natural, ways we may employ parametericity in the protocol of this this example. The first approach, supported in the current version of Scribble, is given in Figure 2. Here, we directly abstract the message type as a parameter to the protocol. In Line 1, the protocol has gotten an additional parameter, `<type ListingFormat>`, as well as dispensing with the "import" statement. This additional parameter means, with the keyword `<type>`, that `ListingFormat` (again aliased as `Format`) is now a type name to be instantiated each time this protocol is instantiated as a whole into a run-time session. Later, in the `response` interaction in Line 6, Registry is obliged to send the list

```
1  protocol ListResources(role client as cl, role resource_registry as rr) {
2      request(resource_kind:String, type ListingFormat) from cl to rr;
3      rec loop {
4          choice at rr {
5              response(element:ListingFormat) from rr to cl;
6              continue loop;
7          } or {
8              completed() from rr to cl;
9          }
10     }
11 }
```

Fig. 3. A refined List Resources protocol (2)

elements according to the concrete type known at run-time, while the Client should be ready to receive them. The protocol again gives a contract among participants, while now flexibly catering for arbitrary data formats.

A second approach, based on a dynamic form of parametrisation [25, 30], is presented in Figure 3. This time, we elaborate the initial `request` interaction, in Line 2 (the `import` clause is again dispensed with), so that the type `ListingFormat` is now explicitly communicated from the Client to the Registry as the *value* of a message, signifying its kind as `type`. This communicated type is then used in Line 5, specifying that Registry should send the datum using the format it has received from Client in Line 2. Scribble may be extended to support this alternative technique for achieving the necessary parametrisation in a future release, as the underlying theory is already well established.

Nested protocols. Consider the protocol given in Figure 4. It has two actors, a Requester and an Authority. In Lines 2–3, Requester sends a `check` message to query on whether a subject is permitted to do an operation on a resource, carrying the identities of a subject and a resource, the name of an operation, and the certificate of Requester (for authentication, possibly validated via a separate protocol) in its payload. In Lines 4–10, Authority responds, saying the operation is allowed or not, or else by saying `other`, to deal with cases when the answer cannot be delivered for some reason, such as an unqualified Requester.

Now consider the following elaboration of our original "List Resources" use case:

Step 1: Client asks Resource Registry to send a resource list (as before).
Step 2: Registry checks if Client has sufficient privileges.
Step 3: If everything is fine, the Registry replies by a sequence of data for resources of the specified kind to Client.

This use case incorporates a privilege check as part of the protocol, as an extension to the original use case. Note this use case composes two previous use cases, by nesting a protocol inside another protocol. Can we realise such composite use cases as a protocol?

In Figure 5, we show how such a composition is done in Scribble, by combining the previously specified `CheckPrivileges` and `ListResources` (the Figure 1

```
1  protocol CheckPrivileges(role requester as req, role authority as au) {
2     check(subject:URI, resource:URI, operation:String, certificate:String)
3        from req to au;
4     choice at au {
5        allowed() from au to req;
6     } or {
7        not_allowed(reason:String) from au to req;
8     } or {
9        other(reason:String) from au to req;
10    }
11 }
```

Fig. 4. A protocol for the Check Privileges use case

version). In Line 5, we use the `introduces` keyword to indicate that Registry will "introduce" a new actor, `authority`. After this preparation, the `CheckPrivileges` protocol is launched (`spawn`) by Registry (`at rr`) in Line 6. Note the arguments include Authority which has just been introduced, as well as Registry (who will play the `requester` role in the spawned session). We call the nested `CheckPrivileges` session spawned during the execution of the `ListResources` protocol a *child session*, or a *sub-session*, of the *parent* `ListResources` session. The lifetime of a child session is, in the standard run-time semantics [9], dependent on its parent (e.g. if a parent session aborts, its child session(s) should also abort). Where such causal dependency is not desired, these unrelated protocols may well be specified separately, to be instantiated into distinct sessions at run-time.

Returning to Figure 5, after the `CheckPrivileges` sub-session is carried out, Registry, now knowing the qualification of Client for this query, responds to Client with either an `ok` or an `error` message with the reason (a `String` payload). When `ok`, the remainder of the protocol is the same as in Figure 1 (and also Figures 2 and 3). Note that the result of running `CheckPrivileges` is likely to related to whether `ok` or `error` is selected at the application logic but, at this type level, we do not specify such detailed constraints.

As mentioned earlier, there are other constructs in Scribble, and in session types in general. Among them are parallel composition, where two concurrent threads of conversations can occur; interrupts, where a participant can asynchronously interrupt an ongoing session using a message with one of the declared signatures; and other modes of interactions beyond simple unicast. Additional features supported by Scribble, and founded on formal theory, include nested protocols, which is based on recent work introduced in [9] studying a general form of nesting and instantiating session types.

3.3 Writing Programs with Sessions

We next take a brief look at how we can use the proposed concept of protocols and sessions to implement clear and understandable communication programs, taking a Python implementation of the List Resources protocol from Figure 1 as an example. We cannot give a full implementation in its entirety here, but we hope the reader can get the flavour.

```
1  import Authentication.CheckPrivilege as CheckPrivilege;
2
3  protocol ListResources(role client as cl, role resource_registry as rr) {
4     request(resource_kind:String, type ListingFormat) from cl to rr;
5     rr introduces au;
6     spawn CheckPrivilege(rr as requester, au as authority) at rr;
7     choice at rr {
8        ok() from rr to cl;
9        rec loop {
10          choice at rr {
11             response(list:ListingFormat) from rr to cl;
12             continue loop;
13          } or {
14             completed() from rr to cl;
15          }
16       }
17    } or {
18       error(reason:String) from rr to cl;
19    }
20 }
```

Fig. 5. A refined List Resources protocol (3)

Preliminaries. A protocol describes interactions among two or more agents. While the running agents are often distributed in terms of run-time locality, the implementation of the agent programs is also often "distributed" in terms of development. Indeed, one of the primary purposes of protocols is to provide a minimal interface against which each agent program may be independently implemented, by different parties using different languages and techniques, while ensuring full interoperability when global application is executed as a whole. Therefore, the basic but general protocol- and session-oriented methodology for developing programs is based on designing and implementing one program for each endpoint. These programs interact with each other inside run-time conversations via asynchronous messages following the specified protocols.

At run-time, a multiparty session functions like a network of TCP connections between the multiple endpoints, enabling them to communicate with each other following the stipulated protocol. However, the concept of session also insulates interactions among its participants from the underlying concrete transport mechanisms, so that developers can (mostly) stay unaware of the particular networking technologies that may be employed at run-time. Our session-oriented programs are constructed using "socket" abstractions that can be seen as standard TCP sockets generalised for multiparty messaging. Explicit structuring of conversation flows makes the description of multiple flows of interactions within an endpoint implementation clear with regards to the dependencies within each flow and between flows. Since interactions in a session are ensured to never violate the underlying protocol, either by static checking [4, 20] or through run-time monitoring (by protocol machines) [8], each endpoint knows what kinds of messages are coming from which other participants at each stage of a conversation.

To demonstrate the description of multiple conversation flows, our example implementation shall integrate the List Resources protocol with a sepa-

```
1  protocol RequestResponse(role Client as cl, role Server as sr) {
2    choice at cl {
3      GET() from cl to sr;
4      choice at sr {
5        sc200(s:String) from sr to cl;
6      } or {
7        sc500(reason:String) from sr to cl;
8      ...
9    } or {
10     POST() from cl to sr;
11     ...
12   ...
13 }
```

Fig. 6. A HTTP-like request-response protocol (extract)

rately specified HTTP-like request-response protocol (simply called Request-Response). We first give the relevant part of the Scribble for Request-Response in Figure 6 before proceeding to the code. In the figure, "sc" in e.g. `sc200` stands for the "status code" of a message.

Program. We now consider a Python program that uses the `ListResources` and `RequestResponse` protocols (the latter for transparently receiving user requests) in combination. The program is an implementation of a service proxy that obtains data from the Registry on behalf of the User. We call this endpoint program simply "Proxy" from now on. Proxy needs to carry out two kinds of conversations:

1. As a Request-Response server, it will engage in sessions with Users, accepting the User query and returning the results from the Registry.
2. As a List Resources client, it will engage in sessions with the Registry, passing on the User query and receiving the list of resources following Figure 1.

Proxy will return the results to User in HTML format, in a similar manner to a standard CGI application. The main Python code for Proxy related to implementing these sessions is given in Figure 7 (in the version with colours, the **blue** and **red** indicate Python keywords and conversation programming constructs, respectively).

Line 2 declares a `try` block for handling exceptions that may arise during session execution. In Line 3, Proxy (receives and) accepts an invitation to interact in the Request-Response session with User. The `proxy_uri` object represents Proxy as a network *principal*, and may roughly be considered as a conversation programming counterpart to a TCP server socket. Proxy can then `accept` an invitation through this interface, with respect to the `RequestResponse` protocol, playing the role of `Server` to User. Specifying the protocol and role for this endpoint prescribes the local programming interface for `c1`, by which Proxy will interact with User.

In Line 4, through `c1`, Proxy receives from User (denoted by its role name `Client` in the protocol), a message `msg`. The basic attributes of a session message

```
1  c1 = None
2  try:
3      c1 = proxy_uri.accept("RequestResponse", "Server")
4      msg = c1.receive("Client")
5      if msg.op == "GET":
6          resource_kind = parse_query(msg.value) # fun def omitted
7          c2 = None
8          try:
9              c2 = Conversation("ListResources")
10             c2.join("client")
11             registry_uri.invite(c2, "resource_registry")
12             c2.send("resource_registry", "request", resource_kind)
13             html_str = ""
14             def loop():
15                 msg = c2.receive("resource_registry")
16                 if msg.operator == "response":
17                     html_str = html_str + yaml2html(msg.value)
18                     loop()
19                 elif msg.operator == "completed":
20                     return
21             loop()
22             c1.send("HTTPClient", "sc200", html_str) # All went well
23         except Exception as e:
24             if c1.alive():
25                 c1.send("HTTPClient", "sc500", "internal error")
26             raise e
27         finally:
28           if c2 != None:
29              c2.close()
30     else:
31         c1.send("HTTPClient", "sc501", "internal error")
32 except ConversationException as e:
33     print("Error({0})@{1}:{2}".format(e.errno, e.cid, e.strerror))
34 except:
35     print("Error({0}):{1}".format(e.errno, e.strerror))
36 finally:
37     if c1 != None:
38         c1.close();
```

Fig. 7. Conversation endpoint program for a service proxy program in Python (extract)

include `op`, the operation name for the message (i.e. the message label or header), and the `value` array, the message payload. In Line 5, we check if the operation of `msg` is `GET`. We assume the kind of resources is specified by the message value, parsed by the `parse_query` function and the result stored in `resource_kind`.

This example demonstrates the interleaving of multiple sessions in a single application. Here we introduce a second session in which Proxy now acts as `client` according to `ListResources`. Line 8 declares a nested try block for this session. In Line 9, we initialises a new session, using the class named `Conversation`. When creating a session, we specify the protocol name `ListResources` (taken to be simplest version presented earlier, in Figure 1). In Line 10, after initialisation, Proxy "joins" the session as the `client` role specified in the protocol.

In Line 11, Proxy *invites* the remote `registry_uri` principal to this newly created session (to play the role `resource_registry`). The method returns when an acknowledgement is returned by the principal to accept the invitation. Now that both roles have joined, in Line 12, Proxy sends to Registry (role name

`resource_registry`), a message with the `request` operation and the kind of resources it is interested in. Note the message format precisely follows the protocol.

The next part of the code gives a tail recursive routine for repeated data delivery, whose flow exactly matches that in the `ListResources` protocol. Lines 14–20 define a function `loop`. In its body, first in Line 15, the client receives a `msg` from Registry. Then we have two cases, depending on the operation of the message:

- If the operation is `response` (Line 16), a HTML-formatted version of the original message (which was specified in the protocol to have a YAML format) is appended to the string (Line 17), and the recursion is enacted (Line 18).
- If the operation is `completed`, the recursion is terminated (Line 19).

Line 21 executes this recursive function, and Line 22 returns the HTML request to User, concluding the inner try-block.

Line 23 catches exceptions. Line 24 checks if `c1` is alive (i.e. if it can still send a message), and if so, sends the Request-Response status code for an internal error before re-raising the exception. In this simple example, Line 30 handles the case when the request is not a `GET` by returning another error message. Finally Line 32 catches exceptions specific to sessions, signified by the exception class named `ConversationException`, whose content is printed in Line 33 (when an interrupt signature is specified in a protocol level, an endpoint program can use this signature to raise the interrupt, which can be caught in the same way). Line 33 shows that session exceptions contain the `cid` field, not present in standard exceptions (Line 35). Finally, either in the normal completion or not, Lines 27 and 36 clean up the sessions upon exiting their respective try blocks.

Discussions. We have illustrated above a simple use of sessions in communications programming. The use of sessions in programs makes it possible to build the application logic with a clear understanding on explicit conversation flows. These flows are clearly visible: by going through how conversation channels are mentioned in a given program (the **red** part in Figure 7), one can clearly capture these flows.

Having distinct flows of interactions explicitly expressed in your program help modular development, in the sense that one flow can be tweaked because e.g. we wish to offer better user experience, while keeping other flows intact. For example, we may consider a variation of the client in Figure 7, with more asynchronous interactions with the web server following more programmatic (e.g. Javascript-based) user-level interactions at the browser. A client can send data incrementally to the web server following repeated messages from the registry, which will be sent and displayed in the browser. We may also enrich the Request-Response protocol in Figure 7 to reflect the interactions at the user interface level. These refinements however do not affect the other protocol, for interactions between the client and the resource registry: so, in the program, we may only refine the interactions at `c1`, keeping those at `c2` intact.

Our purpose in our introduction to session programming in this section was to illustrate the core ideas of session programming, to see how it looks like

to structure communications with (typed) sessions. There are other basic constructs for sessions, such as those for sending interrupts; creating a sub-session; inviting participants from the parent session in a child session; and others. Further, in practice, we often naturally wish to combine two or more consecutive sending actions, such as the invitation to join a session as a role and the initial sending action to that role. However the central idea is the same: to clearly present, in a communication program, how a flow of interactions – a session – proceeds through a sequence of program actions and their composition, possibly interleaved with actions in other sessions.

The resulting organisation of communication actions enable not only programs with a clear presentation of interaction structures, but also static validation of conformance to the underlying protocols through type checking; and its dynamic counterpart through finite state machine based protocols monitors. In the latter (dynamic) validation, it is assumed that we can identify the underlying session by inspecting a message, if that message belongs to a session. In this way the runtime messages also get organised, dividing numerous message exchanges in distributed computing environments into different chunks with a binding to underlying protocols. This is how sessions structure communication-centred computing.

4 Using Session Types

4.1 Session Types in Distributed Systems

Unlike in sequential computing, where a piece of software can often be regarded as a self-contained mathematical function, software in distributed computing environments evolve over long periods of time, interacting with other applications and services with disparate origins and histories. A piece of software interacts with other pieces of software, and their mutual interactions critically affect their behaviour to e.g. users. Because different endpoints should communicate with each other to realise a certain function, we need an infrastructure by which all this software can interact with each other. The global Internet is a typical and prominent example, which provides an infrastructure for communications in the shape of the TCP/IP protocol suite and, building on the end-to-end principle, enables diverse software and services to evolve and inter-relate with each other, creating the web of mutually dependent and evolving services. Partly overlapping with Internet but forming their own networks, we see many distributed computing environments designed and evolve, with different geographic expanses, shapes and functions, such as the corporate backbone networks, the backends of popular web services, and networked infrastructures for sciences and engineering.

Session types were introduced to structure distributed communicating processes. By different endpoints communicating with typed sessions, their interactions follow the stipulated scenarios, without inducing communication error: when a sender sends a message, the receiver can understand what it is, and in turn will send messages in an expected way. That is, we expect all parties to behave properly in their interactions following the protocols of sessions. It is then

a natural question how we ensure proper communication behaviours of systems at run-time.

For example, we may realise typed sessions without having session information at run-time (for example, we may use a set of TCP connections to realise a session), with each program being type checked statically and whose session primitives invokes actions on such connections. In this case, there is no explicit session (wrt. the concept of session being proposed here – i.e. beyond that implicit to TCP) at run-time – except in our mind's eye. The freedom to realise sessions in this way is certainly the merit of having high-level abstraction in the shape of sessions assisted by static verification made possible by that very abstraction.

Another method is having sessions and protocols explicitly incorporated as part of the infrastructure in a distributed computing environment: a web of distributed runtimes, by which we can create and use sessions, become part of the infrastructure and applications use these runtimes to communicate with each other. Some of the practical motivations to use such a configuration include to track errors, to dynamically share protocols, and to optimise communication paths on the fly using information on sessions and their protocols. But the most prominent reason to choose this explicit approach is to insulate the specification, design and runtime behaviour of software systems in a distributed computing environment from low-level transport details. It leads to an environment where all or most communication behaviours in that environment are governed by explicitly declared protocols, and messages exchanged at runtime are marked by distinct sessions so that they can be multiplexed over communication channels and are checked against state machines induced by the underlying protocols (just as TCP and other transport and higher protocols are checked at endpoint network stacks). On this basis, we may build a machinery to assure high-level behavioural constraints such as conformance to security policies.

4.2 Using Session Types for End-to-end Cyberinfrastructure

Ocean Observatories Initiative [24], often abbreviated as OOI, is a large-scale NSF-funded project to build a cyberinfrastructure for observing oceans in the United States and beyond, with usage span of 30 years. It integrates real-time data acquisition, processing and data storage for ocean research (e.g. sensor arrays, underwater gliders, high-resolution under-water cameras), providing access for a wide ranging user community under different administrative domains. It consists of multiple marine networks where we lay cables over a large area under the sea, which are integrated by a distributed cyberinfrastructure. This cyberinfrastructure, called OOI CI (CI for CyberInfrastructure), is itself a network, consisting of distributed infrastructural services whose main sites are two large clouds but whose distributed components in the shape of containers also reside all over its distributed sites residing in hundreds of universities and marine institutions.

One of the central features of the OOI CI is its end-to-end nature, in the sense that its design allows and encourages scientists to register data (which often takes

the form of real-time streaming data from sensors over different time scales) and data products (which are derivatives from raw data by application of models). Just as scientists publish their papers, they may as well publish their data and data products, shared by other scientists, as well as by teachers for educational purposes. In the same spirit, the OOI CI should allow an easy and well-regulated sharing of instruments and other resources, each under a specific administrative control. For example, a seabed camera owned by one institution may be used by a scientist in the other. Thus, in this system, multiple heterogeneous organisations and individuals participate, run their software (such as simulation models for sensor data) inside the system, and we need to ensure a high-level quality of usage including transparency, partly because marine data play a critical role at the time of calamity such as earthquakes and tsunami.

One of the architectural decisions of OOI CI is to regulate the behaviours of heterogeneous participants in the OOI CI by imposing high-level abstractions based on interaction patterns, which are in turn regulated by high-level policies through runtime monitors. The catalogue of interaction patterns will in turn assist developers to implement their distributed services with ease and clarity. Thus we need a descriptive means to write down these interaction patterns clearly and without ambiguity, use them for software development, and regulate communications behaviour of participating endpoints at runtime through induced protocol machines, augmented with regulation by policies on their basis. For the description of interaction patterns, the use of session types (and Scribble) is considered, building a framework to regulate interaction behaviour based on policies on its basis. This policy-based regulation is called "governance" by the OOI CI architects, conceived by Munindar Singh and the OOI CI architects, centring on the notion of commitments [11]. To use session types as a basis of regulating behaviour in this distributed computing platform, several technical challenges were identified, which include (restricted to those proper to session types):

- Can we accurately describe interaction patterns which are and will potentially be used in distributed applications in OOI CI?
- Can we ground them to programming? Can we help developers to build safe and robust systems with ease?
- Can we have a simple and efficient execution framework for these programs?
- Can we guarantee their communication safety at runtime? What would be a simplest mechanism?

The research team on session types in Imperial College London and Queen Mary, including the present authors, are contributing to the OOI CI development through, among others, the following technical elements:

- A protocol description language, *Scribble*, and development/execution environments centring on this language.
- A tool chain for protocol validation, endpoint projection, FSM translations, APIs and runtimes.
- Part of the monitor architecture based on the protocol machines (FSM) translated from protocols.

As we have already observed, Scribble is fully based on research on session types. The FSM translation is a direct application of the theory which links automata theory (communication automata) and session types, recently introduced in [10], where a session type can be directly translated into a communication automaton.

The development efforts are producing several interesting findings. For example, one of the methods for facilitating the use of session types for developers who are not accustomed to session types is to use the interface of the standard communication APIs such as RPC. These libraries were independently developed in the OOI CI to support application development based on traditional technologies: the idea is to replace them with distributed runtimes for session types. What we found is that this approach, where we implement libraries using session primitives, has rewarding practical merits in the tractability and transparency in engineering. For instance, each library is now a short scripting code by using the underlying session machinery, automatically monitored by the corresponding protocol. As one example, RPCs with diverse signatures are now based on a single parametrised protocol, and its interactions are checked by a generic monitor for general session types. This conversion is feasible because not even a single line of application code needs be changed: the resulting behaviour is the same, we can use the same interface file, with a formal foundation automatically assuring correctness of interactions. The layer for typed sessions is called Conversation Layer in OOI CI. As well as the extensive experiments on Conversation Layer itself, our development efforts are focusing on the governance functions to be realised on top of Conversation Layer.

5 Conclusion

In this work we have examined the motivations and backgrounds of the introduction of session types and associated programming methodology, together with illustration of how we may design and implement a program centring on session types. For organising communicating processes, there are other approaches which address different aspects of abstractions for communicating processes.

One basic approach centres on the notion of concurrent objects [33], where objects communicate with each other by sending messages to their object identities, starting from the actor model [1, 15], which also gives one of the simplest forms of this paradigm. In concurrent objects in general, there is a strong integration of the idea of objects and concurrency, where concurrency is considered to be a default rather than an exception. While programming languages based on concurrent objects may not have treated sessions and session type beyond request-response patterns, the use of constraints on interaction patterns in such languages should certainly be feasible, as a recent work shows [14]. Similarly, the identities as found in actors and concurrent objects may as well be part of the session-based programming (for example, distributed infrastructures such as the OOI CI demand the use of identities for principals which act as endpoints of communications). Different experiments in such integrations will deepen our understanding on the relationships between these two paradigms. How the pursuit

towards flexible programming abstraction in concurrent objects (e.g. reflection) may interact with the type-based approach in session types is another interesting future topic.

Concurrency and communication are a rich realm for which many different approaches exist. Occam-Pi [31] is a highly efficient systems-level concurrent programming language centring on synchronous communication channels, based on CSP and the π-calculus. Erlang [2] is a communication-centred programming language with emphasis on reliability whose central programming and execution paradigm is based on actors. Session types are an approach to structuring communications programs based on session abstraction and protocol description, with its formal basis in the π-calculus and its type theories. Protocols, arising as types for dialogue among endpoints, are used to constrain behaviour so that the resulting programs and runtime configurations are easy to understand and reason. For fully identifying its possibilities and limitations, we need to explore the use of typed sessions in various stages of software development, ranging from high-level modelling to execution, as well as formal specifications and verifications. Not restricted to session types, we need to identify a wide range of concrete methods usable to address these problems, as well as a unifying foundation for them, to reach a truly effective methodology for distributed computing systems.

We refer the reader to [12] for more detailed comparison of session type theory and session-based implementations against other related works.

Acknowledgements. We thank the reviewers for their comments, Dr Gary Brown for his collaborations on the Scribble project, and our colleagues in the Mobility Reading Group for discussions. This work is partially supported by the Ocean Observatories Initiative, EPSRC grants EP/F002114/1, EP/G015481/1 and EP/G015635/1, and EPSRC KTS.

References

1. Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
3. Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes. *Theoretical Computer Science*, 37:77–121, 1985.
4. Lorenzo Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
5. Gerard Boudol. Asynchrony and the pi-calculus. Technical Report 1702, INRIA, 1992.
6. Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. In *FSTTCS*, volume 8 of *LIPIcs*, pages 338–351, 2010.
7. Vinton G. Cerf and Robert E. Khan. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22:637–648, 1974.
8. Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7454 of *LNCS*, pages 25–45, 2011.

9. Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *LNCS*, pages 272–286, 2012.
10. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *LNCS*, pages 194–213, 2012.
11. Nirmit Desai, Amit K. Chopra, Matthew Arrott, Bill Specht, and Munindar P. Singh. Engineering foreign exchange processes via commitment protocols. In *IEEE SCC'07*, pages 514–521, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
12. Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In *WS-FM*, volume 6194 of *LNCS*, pages 1–28, 2009.
13. Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.
14. Peter Dinges and Gul Agha. Scoped synchronization constraints for large scale actor systems. In *COORDINATION*, volume 7274 of *LNCS*, pages 89–103, 2012.
15. Carl Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.
16. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
17. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75, 2011.
18. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.
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, 1998.
20. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
21. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, Berlin, 1980.
22. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
23. Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.*, 100(1), 1992.
24. Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.
25. Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of ACM*, 47(3):531–584, 2000.
26. Jerome Saltzer, David Reed, and David Clark. End-to-end arguments in system design. *ACM Transactions in Computer Systems*, 2(4):277–288, 1984.
27. Scribble development tool site. `http://www.jboss.org/scribble`.
28. Scribble github project. `https://github.com/scribble`.
29. 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, 1994.
30. David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.
31. Peter Welch and Fred Barnes. Communicating Mobile Processes: introducing Occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.
32. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.

33. Akinori Yonezawa and Makoto Tokoro. Object-oriented concurrent programming: An introduction. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 1–7. MIT Press, Cambridge, MA, 1987.