Session Types for Software Engineering

Dimitrios Kouzapas[1], Raymond Hu[2], Rumyana Neykova[2], Simon J. Gay[1], Tihana Galinac Grbac[4], Ornela Dardha[1], Simon Fowler[3], Nicholas Ng[2], Sam Lindley[3], J. Garrett Morris[3], Roly Perera[1], Philip Wadler[3], and Nobuko Yoshida[2]

[1] University of Glasgow, UK
[2] Imperial College London, UK
[3] University of Edinburgh, UK
[4] University of Rijeka, Croatia

Abstract. Session types are formal descriptions of communication protocols. The formal nature of session types enables their incorporation into programming languages and tools so they can be used to drive the software development process. Research on session types has moved from theoretical studies towards practical applications, and the field is maturing to the point where session types can be integrated into software engineering methodologies. However, most of the literature is not sufficiently accessible to a wider academic and industrial audience; more systematic technology transfer is required. The present paper provides an entry point to a substantial repository of practical use-cases for session types in a range of application domains. It aims to be an accessible introduction to session types as a practical foundation for the development of communication-oriented software, and to encourage the adoption of session-type-based tools.

1 Introduction

Session types are formal descriptions of communication protocols, which can be incorporated into programming languages and tools so that the technology of typechecking can be used to verify the dynamic structure of communication as well as the static structure of data. Since the introduction of session types by Honda et al. [**?**] more than twenty years ago, there has been a broad research effort to study their properties and apply their principles to a range of computational models and programming paradigms. Current research is converging towards the development of tools and technologies that use session types as part of the software development process. In order to take full advantage of the possibilities of session types, it is important to start a programme of technology transfer so that they can be integrated into broader approaches to software engineering.

There are numerous formal approaches (for example?) proposed for describing communication protocols, in software development. Because of their ability to define clear and verifiable solutions, formal method for the description of

communication are implemented within programming languages. (What does this mean? Are we saying that programming languages already include formal methods for the description of communication?) Some programming languages are even standardised for specifying telecommunication systems, e.g. the set of languages standardised for telecommunication systems within the ISO ITU-T Z.100 series. In the telecommunication (complex system) domain, many of the requirements usually limit the use of formal methods. Furthermore, there is an unexplored area on how to balance the use of formal methods and the complexity in industrial software systems. (SG: I think we need to clarify what this paragraph is saying.) Research in session types exhibits some promising results that can help explore and fill in this gap. In addition, the problem of weak adoption of formal methods by the industry has been extensively studied. The main barriers recognised can be found in the lack of education, tool support and the cost of formal methods [**?**]. But even if these barriers are removed, there comes the importance of a well defined improvement strategy, that managers within a software organisation have to follow while adopting formal methods [**?**]. It is therefore necessary for a wider engagement of software engineers with formal method and formal method experts.

Research on session types is mainly driven by the need to demonstrate that realistic, general patterns of concurrency and communication can be described. A typical research paper identifies a problem in terms of a real-world use-case and proposes a session-typed framework, expressed in a strict mathematical form, that applies the principles of session types to the solution of the problem. The development of session-typed tools and technologies is a consequence of this research process: tools are developed following the theory and are applied to real-world use-cases.

This approach to presenting results about session types, although successful within the session types community, presents difficulties for the process of knowledge exchange between experts and interested practitioners.

1. There is a huge volume of research information about session types. Interested individuals need to invest substantial effort to find and study a large and diverse literature.
2. There is no uniform approach or terminology in the presentation of research results. Due to the diversity of session types, there are many terms that derive from different disciplines and are extended to refer to the same concepts within session types. This may lead to confusion among less expert readers. An example is the use of the terms *typestate*, *session type* and *protocol* for essentially the same concept in the context of object-oriented programming.
3. Individual results about session types are often presented in isolation and not placed in the context of the whole field. Obtaining a broader understanding of the subject requires a systematic literature search, which may be off-putting for non-experts.
4. The majority of results are tightly coupled with a high level of formal technicality, making the comprehension of session types by people without a formal background even more difficult.

Summarising this discussion, we conclude that a wider audience would need to study a large series of partial results and at the same time filter out a huge amount of unnecessary technical detail in order to achieve a satisfying level of understanding session types. This situation is an obstacle for the adoption of session types as part of practical software development.

Motivated by the goal of integrating session types into the software development process, this paper identifies the problem of accessibility of session types to a wider academic and industrial audience. Our aims are: (i) to demonstrate session types in a comprehensive way and in terms that are easier to understand by non-experts; and (ii) to enable a process that will integrate session types into a broader context in computing.

A way to meet these objectives is to establish a common ground that can be used for knowledge exchange between researchers. This paper identifies as a common means of communication the demonstration of practical scenarios from a broad area of application in computer science. Use-cases can bridge the communication gap between researchers and can help the wider promotion of the principles of session types.

Furthermore, the current research directions on session types are pushing towards a dialectic between session types and disciplines like software engineering, and system design and implementation. Evidence on this direction comes from the he research experience gained by working on the project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (ABCD for short) [**?**]. Part of the requirements of the ABCD project is to (i) work with industrial partners towards integrating session types in real world use-cases; and (ii) compile a set of usecases for session types that can be used for current and future research on the design and development of frameworks and technologies. More concretely, the aims of this paper are to:

1. Describe in non-technical language, and develop common terminology for, the mathematical terms currently used in the theory of session types (Section 2.1 and Section 2.2).
2. Demonstrate the methods that are currently used to present, analyse, and express an application in terms of session types (Section 2.3). The paper includes a discussion on their possible adoption by software engineers in the software development process (Section 2.3 and Section **??**).
3. Demonstrate the robustness, functionality, and overall applicability of session types through a diverse overview of use-cases (Section 3) that cover: (i) different domains that exhibit different computational needs; (ii) different interpretations of session types in different programming paradigms that are used to implement a usecase; and (iii) current tools and technologies that integrate session types in the software development process.

Overall this paper can be used by experts and non-experts to exchange knowledge about the overall discipline of session types, and to enable future work on the application of session types.

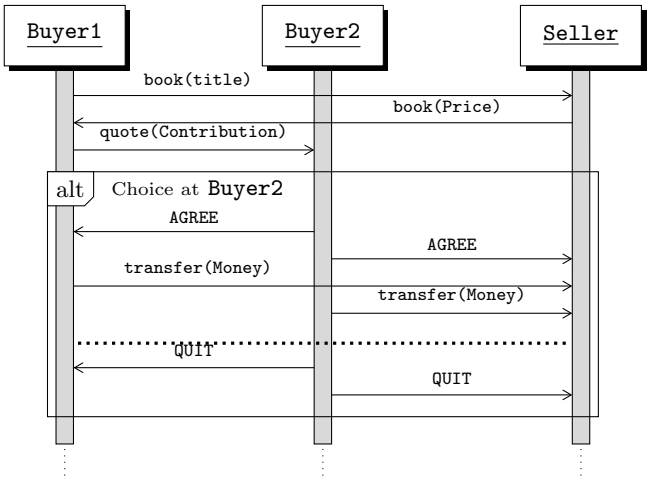


Fig. 1. The Book-store protocol: Sequence Diagram

2 Session Types

This section introduces session types through a a discussion of the need for session types in software engineering. It also presents the Scribble [**?**] protocol description language, which is a formal language for describing protocols using the principles of session types. The overall discussion in this section identifies and defines a clear and distinctive terminology for transferring the basic ideas of session types to a wider audience.

2.1 Session Types and Software Engineering

Communication *protocols* in software engineering are usually described and communicated among humans by using the most intuitive graphical form of sequence charts Use consistent terminology: sequence diagram, or sequence chart? (Figure 1). Sequence charts are used to depict a communication flow as an ordered sequence of *message passing* among communicating participants that represent particular *roles* in a communication protocol.

Figure 1 illustrates the Book-store protocol, which is a use-case in the ABCD online repository [**?**]. The Book-store protocol specifies an interaction between two online Buyer roles that invoke a process of buying a book from an online Seller role, while sharing the expense. In the sequence chart, each role is represented by a vertical line and communication messages are represented by horizontal directed arrows indicating the direction of communication flow. Intuitively, the communication progress proceeds downwards. Thus the protocol's behaviour is represented visually within the chart. However, sequence charts are not considered as formal tools because protocols described in this way cannot be verified as being complete or unambiguous.

In industrial practice, communication protocols for distributed software are usually described in natural language and discussed with the help of sequence charts. Each software module that participates in the protocol implements its *local* software role within the protocol. Usually the process of implementing the local specifications may follow different possible solutions that are not directly visible or are even omitted from the sequence chart, due to the complexity they will introduce. I'm not sure what this means. Another limitation of sequence charts is the inability to deal with complex communication protocols, such as the case where one role can make a choice that affects the subsequent communication behaviour of the entire protocol.

Communication complexity can be managed by introducing *structure* within the communication protocol. Session types describe communication in a structured and rigorous fashion, and aim to define concepts for structured communication that should enable better control of communication complexity.

Typically, a type is broadly understood as a meta-information concept, that is used to describe a class of data. Structured types are used to describe data structures. When types are structured as input/output requirements they are used to describe functionality. The combination of functionality and data structures give rise to the *object oriented* concepts of classes and interfaces.A session type extends the notion of a type to capture the communication behaviour in concurrent systems. *A session type is a type structure that describes communication behaviour in terms of a series of send and receive interactions between a set of communicating entities, and it is used to characterise data structures that are used to implement communication.* I'm not sure what we mean by characterising data structures. Is it just that the data type of each message is defined?

The formal nature of session types enables rigorous reasoning about the communication properties of a system. Session types particularly focus on the following properties:

- *Communication Safety*: Every execution state of the system has *safe* communication behaviour:
  - *Communication interaction match*: Every send (resp., select) operation has a corresponding receive (resp., branch) operation.
  - *Message (Payload?) type match*: The type of every message being sent matches the type of the message expected to be received.
  - *Deadlock-freedom*: Consequently, every message sent will eventually be received.
- *Communication Progress*: Every state of the system either represents a terminated protocol, or will *progress* to a state with safe communication behaviour. Maybe this can be clearer?

In addition, the structured nature of session types enables a methodology for applying session types in the software engineering process.

The Scribble tool-chain [**?**] is a platform that intends to serve as a core element in the process of integrating the practical aspects of session types in software engineering. The design of Scribble draws directly from the principles and properties of session types, as identified through rigorous study. The rest

make clear that it is a syntactic/static technique - in contrast to model checking

of this section introduces the main terminology for session types and, whenever appropriate, it presents these notions using the Scribble tool-chain.

2.2 Sessions as Protocols

Practically, a session type can be seen as a formal *specification* of a communication *protocol* among communicating *roles*. The syntax for session types is defined on pairs of operations: (i) the *send* and *receive* operation give rise to the *message passing* interaction; and (ii) the *select* and *branch* operators enable the *choice* interaction.

Message passing is used for sending data from one role to another. The data that are passed are described in terms of their type. Session types can also support the case of *session delegation* where a session-typed value, i.e. a structure that implements communication, is passed as a message. Choice requires a role to use a value called a *label* to select a communication behaviour among a set of behaviours, called *branch*, offered by another role. The choice interaction enables formal specifications that can cover a more complex structure of communication and describe all possible cases that may happen, in contrast to the limiting structure of a sequence diagram.

Session types assume a *global* protocol that describes the communication interaction of all the roles inside a concurrent system. A global type enables a view of a system as a whole, instead of the approach that a concurrent system is a set of communicating modules. The perspective of a global protocol through a single role is called *local* protocol. The local protocol describes the communication interaction of a single role with all other roles in the system.

The relation between a global protocol and the local protocols of its roles, is expressed through the *projection* procedure; given the global protocol and a role, projection returns as a local protocol only the interactions of the global protocol that are concerned with the role.

The reverse procedure to projection is called *synthesis*, where a set of local protocols are composed together in a global protocol.

The Scribble Protocol Description Language The notions explained above are supported by the Scribble protocol description language, which is a syntax used to express session type specifications. As an example, consider part of the session type protocol for the Book-store use-case as in Figure 2.

Line 1 of the upper protocol in Figure 2 shows the Scribble definition of the global protocol for the Book-store use-case. The communication takes place among the roles `Buyer1`, `Buyer2` and `Seller`. Line 2 demonstrates the syntax of a message passing interaction; a message of type `title` labelled with label `book` is sent `from` role `Buyer1` `to` role `Seller`. In Scribble, messages are defined as a list of types annotated with a label.

In line 5 of the protocol the usage of a choice interaction is presented, where role `Buyer2` makes an internal decision and selects from two possibilities, that are expressed with the labels `agree` in line 5 or `quit` in line 6. In both cases roles

```
1 global protocol Bookstore(role Buyer1, role Buyer2, role Seller) {
2   book(title) from Buyer1 to Seller;
3   book(price) from Seller to Buyer1;
4   quote(contribution) from Buyer1 to Buyer2;
5   choice at Buyer2 {agree() fom Buyer2 to Buyer1, Seller;
6                     transfer(money) from Buyer1 to Seller;
7                     transfer(money) from Buyer2 to Seller;}
8   or               {quit() from Buyer2 to Buyer1, Seller;}
9 }
```

```
1 local protocol Bookstore at Buyer1(role Buyer1, role Buyer2,
2                                    role Seller) {
3   book(title) to Seller;
4   book(price) from Seller;
5   quote(contribution) to Buyer2;
6   choice at Buyer2 {agree() fom Buyer2; transfer(money) to Seller;}
7   or               {quit() from Buyer2;}
8 }
```

Fig. 2. Scribble: Session Type Protocol for the Book-store Use-case

`Buyer1` and `Seller` respectively offer alternative branches. The choice interaction requires that Role `Buyer2` informs roles `Buyer1` and role `Seller`. All three roles will then synchronise on the same choice.

Scribble also provides a looping construct, which is needed for more realistic protocols but is not used in this simple example.

describe the recursion construct?

Line 1 of the lower protocol defines the local protocol for `Buyer1` of the Book-store protocol. Role `Buyer1` interacts with roles `Buyer2` and `Seller`. In essence, the local protocol for role `Buyer1` is derived automatically by stripping out all the irrelevant information with respect to `Buyer1` from the global protocol.

2.3 Sessions as Software Development Method

In addition to a protocol description language, Scribble offers a tool-chain that integrates session types into the software development process.

The formal nature of session types protocols allow for rigorous manipulation of the communication behaviour.

Figure 3 clarifies the terminology developed previously by showing the typical usage of session types in software engineering. Furthermore, it distinguishes the desirable properties that are offered by session types.

In Figure 3 rectangles represent source code for either the software under development (white coloured) or for the definition of the session type protocols (grey coloured). The arrows represent automated procedures that ensure the desirable properties of session types in the development process. The Scribble tool-chain supports many of these automated procedures. An important assumption is that the software source code in the bottom of Figure 3 uses an interface for communication, where the communication operations can be identified by automated procedures.

In a typical top-down approach, a developer will first develop a global protocol as syntax that expresses session type specifications. This for example can

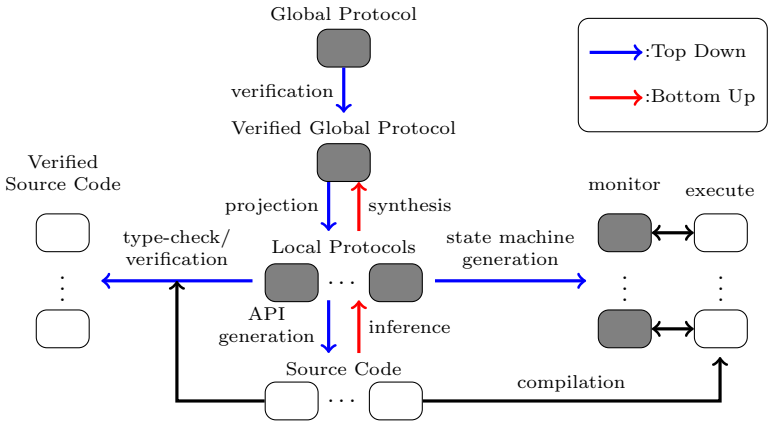


Fig. 3. Use of Session Types in Software Development

what does it verify?

be done using the Scribble protocol description language. The global protocol source code is shown on the top of Figure 3. The next step is to run a procedure to verify the global protocol correctness . After the global protocol verification, a projection procedure takes place where a local protocol for each global protocol role is derived. The obtained local protocols are also expressed syntactically as session type specifications. Verification, projection and local specification syntax are supported by the Scribble tool-chain.

One way to use the set of local protocols is to ensure conformance between the source code of the software under development and the local protocols, by using a type-checker procedure. The type-checker takes as an input source code that corresponds to a communicating process; it identifies its communication operations, and derives the communication behaviour of the source code. The type-checker also takes as an input the corresponding syntax of the local protocol and checks whether the software source code implements the corresponding role of the specified communication behaviour. Upon success, the result is verified software that enjoys the session types *communication safety* and *communication progress* properties.

A second way to use the set of local protocols is for automatic code generation. The generated code might be in the form of APIs that implement communication behaviour or in the form of programming templates that implement communication behaviour and they further need attention by the developers for adding extra computational logic. Scribble supports the generation of a Java API out of every the set of Scribble local protocols. The generated API has a class for each state of the protocol that implements as methods all the available communication operations. A call to method returns an object of the class that corresponds to the next state of the protocol. A dynamic check ensures that each object (i.e. state) is used once.

Another way of using the set of local protocols is for monitoring the execution of the communication interaction inside a concurrent system. The local protocols are used to feed a set of local monitors. Essentially, the local monitors run a state machine that is derived out of the local protocol syntax. It is assumed that the communication interaction of the software under execution can be recognised and monitored by the monitor interface. Scribble does support the procedure of deriving state machines out of local protocol syntax.

Figure 3 presents the bottom-up approach as well, were the communicating software source codes are developed independently. Then an inference procedure identifies the communication behaviour of each communication process that results in a set of local protocols. Then an automatic synthesis procedure can take place, where the derived local protocols are synthesised to create a global protocol. Upon success, the developed software is guaranteed to ensure the communication safety and progress.

In any of the above methodologies for using session types, session types can be used as requirements by humans means to drive the software development process. In the top-down approach local types can be used to give intuition and guide developers when implementing code. Furthermore, the bottom-up approach more agile development teams can analyse and compare the obtained global and/or local protocols against the intended communication behaviour using human reading techniques.

how about diagrammatic languages?

The case where the implementation of all the communication processes inside a system conforms to the corresponding local protocols of all the roles in a global protocol is called *session fidelity*. When session fidelity is ensured then the communication safety and communication progress properties are ensured.

As seen in Figure 3 Scribble and the procedures that manipulate session type protocols lie at the core of the software development process. This is also clarified by the experience in developing the usecases for the ABCD online repository. The Scribble platform and its procedures has an independent status next to the plethora of technologies and programming paradigms used in the case studies.

2.4 Usecase Classification on Technologies used

Just presentation and references

The technologies used to implement the demonstrated usecases are presented below:

1. Session Java [**?**] is the first tool developed as a tool with a library that support binary session types. The tool requires compiler support to perform session type-check.
2. The Eventful Session Java [**?**] is an extension of session types to support asynchronous, event-driven programming.
3. Multiparty Session C [**?**] is the first tool that implemented a multiparty session type library for MPI. The tool requires compiler support to perform session type-check.
4. Session fidelity can also be ensured using runtime monitoring. Modules for monitoring were implemented in python and are presented in [**?**].

5. Session types are embedded in the Actors paradigm in the following technologies:
   - An implementation that uses the python monitoring module for session types to monitor python threads that simulate actors [**?**].
   - An similar approach on monitoring session types is also used for the Erlang programming language, which is a programming language for the Actors model.
6. Pabble [**?**] is an extension to Scribble used to express communication structures that have a parametrised number of roles. Pabble protocols are implemented and type-checked for the MPI framework.
7. Mungo [**?**] is a tool that integrates session types in the object-oriented programming paradigm through the typestate structure; communication interactions can be seen as an object interface with a behaviour. Mungo uses a protocol description language for capturing typestate. The tool Scribble to Mungo (StMungo) is used to transform Scribble protocols into Mungo protocols.
8. GV: A functional programming implementation of binary session types.
9. LINKS: Functional programming based on the linear logic interpretation of session types.
10. SILL: Functional programming based on the linear dual intuitionistic interpretation of session types.
11. A state pattern implementation as a library. Uses Scribble to automatically create an API for state pattern programming and uses runtime checks to cope with linearity requirements.

2.5 Fine/coarse grain representation

this should go as future work

Concurrency patterns e.g. network topology, client/server, race condition, etc. (maybe I am writing nonsense here)

3 Usecases

This section will give a summary of the usecases in the ABCD online repository [**?**]. At the same time it will give a more detail presentation on how session types are used in the implementation of the network application usecase for a book-store, which is a standard example for introducing session types. It will also focus on the implementation of a session type Simple Mail Transfer Protocol, which is a standard network protocol.

RN Table Figure 4 summarises the use case from the ABCD repository. The table presents the name of the use case, its original source, and the language that it is implemented. Full source code, running examples and and detailed description of all use cases can be found in [**?**]. The use cases are organised into domains and for each domain we have chosen representative examples.

No	Use Case Name	Source	Implementation	Remark
	Network Protocols			
1	SMTP	[?]	Java API, Mungo, Links	stateful protocol
2	HTTP	[?]	Java API	request-response
3	DNS	[?]	Erlang	
4	POP3	[?]	Mungo	**RN** Check with D.
	Application Protocols			
4	Book Strore	[?]	SJ, Mungo, Java API	interaction logic
5	Travel Agency	[?]	SJ	
6	Chat Application	[?]	Erlang	
	Concurrency Patterns and Algorithms			
7	Dining Philosophers	[?]	SPython	synchronisation
8	Sleeping Barber	[?]	SPython, SScala	
9	Cigarette Smoker	[?]	SPython	race conditions
11	Peano Numbers	[]	GV, Links	
12	Add Server	[]	GV, Links	
13	Concurrent Fibonacci	[?]	SPython, Mungo	**RN** Check impl.
	Data Structures			
14	Queue	[?]	Sill	
15	Collections	[?]	Mungo	a stack client
16	File Access	[?]	Mungo	file access client
	Network Topologies and Parallel Algorithms			
17	Ring	[?]	MPI	
18	Butterfly	[?]	MPI	
19	All to All	[?]	MPI	
20	Stencil	[?]	MPI	
21	Farm (Master-Worker)	[?]	MPI	
22	Map Reduce	[?]	MPI, SPython	
	Operating Systems			
23	Memory Coherence	[?]	Mungo	
24	Lock	[?]	Eventful Session Java	race conditions

Table 1. List of implemented use cases

	Usecase	Domain	Technologies/Tools	Description
1.	Book Store	Business Application	Mungo, Scribble API	Interaction Logic
2.	Chat Server	Network Application	Erlang	Application Logic
3.	HTTP	Network Protocol	Scribble API	Request/Response
4.	Simple Mail Transfer Protocol	Network Protocol	Mungo, Scribble API, LINKS	Stateful protocol
5.	DNS Server	Network Protocol	Erlang	
6.	Concurrency • Din. Philosophers • Sleeping Barber • Cigarette Smokers	Operating Systems	Python Actors	Race conditions
7.	Lock	Operating Systems	Eventful SJ	Race conditions
8.	Collection	Data Structures	Mungo	Stack client
9.	File Access	Data Structures	Mungo	File Access client
10.	Fibonacci	Parallel Algorithms	Mungo	
11.	Network Topologies • Ring • Ring • Butterfly • All to All • Stencil • Master-Worker • Map Reduce	Parallel Algorithms	Pabble and MPI	Parametrised algorithms
12.	Functional Algor. • Peano Numbers • Add Server	Parallel Algorithms	Links, GV	
13.	Memory Coherence	Systems, Hardware	Mungo	

Fig. 4. Usecases in the ABCD online repository

3.1 Domain Classification of Usecases

The usecases are presented following a wide range of application domains, in order to demonstrate the fact that session types can capture a broad area of communication specifications. For the implementation of the usecases in the online repository different technologies that integrate session types in different programming paradigms were used.

1. *Network Application/Business Logic.* The first two applications in Figure 4 come from the domain of network applications.
   The first network application is the online book-store, where its protocol and specifics are already discussed in the paper. It is a standard usecase able to demonstrate the basic practical features of session types.
   The Book-store usecase is implemented following a communication protocol that embeds an execution logic for the application. This is a practical indication that session types can be used to describe the behavioural logic of an

application. The current development the usecase follows a top-down design approach, where the projection of the global protocol is used by different implementing technologies. The book-store is implemented in Session Java and Mungo.
- Book-store: Standard example, Communication Interaction shows business logic
- Chat-Server: Implement a server/client architecture, design an application protocol for the servers, client implementations should conform to the protocol, Erlang monitoring implementation cite S. Fowler's Master Thesis.

need LINKS and the hybrid implementation

2. *Network Protocols.* Session types can be used to describe standard and non-standard network protocols. Typically a standard network protocol should conform to an informal RFC (request for comments specification. Session types can present a network protocol formally its manipulation easier by both engineers and machines. Non-standard network protocols can also be developed.
   - HTTP: Request response standard RFC protocol, stateless, assume different types for each request/response header (not just a text header) request-response sequences can give a further structure of interaction, it would be nice if we have a client to stream tree-structured data. automatically generated API from the Scribble tool-chain
   - SMTP: Stateful standard RFC protocol. Expresses request/respond, Expresses specific data-structures streaming such as the data-structure of an email, need for translate between RFC text format to/from message/payload types Complex state makes implementation difficult - session types remove a burden from the developer because they automatically verify that the state of the code follows the state of the protocol.
   - Domain Name System:
3. *Systems/Applications.* A session type may be used to describe the communication specifics of an application that uses multiple resources inside a computing machine.
   - Do we have a system to describe
4. *Operating System.* Another domain where session types can be applied to is the description of the communication specifics of operating system algorithms and routines, that co-ordinate the usage of hardware resources.
   - Locks: Basic OS structure basic for achieving resource utilisation. Its usage implies race conditions that in turn imply asynchronous and reactive communication. The implementation of locks in the eventful session Java shows that session types can express deadlocks.
   - Concurrency Algorithms: Classic concurrency problems expressed in Session types. Implementing in the python monitored actors framework. **RN** Classic concurrency problems require correct coordination among multiple components to avoid starvation and deadlock. Preserving the causalities between the interactions is challenging since the communication often involves complex patterns, combining long sequence of interactions with recursive behaviour and nested choice branches. Often precise message sequence should be followed. Moreover, it is not obvious if the components can be safely com-

posed. Sending the wrong message type, sending to the wrong role or not sending in the correct message sequence may lead to deadlocks, errors which initial cause is hard to be identified or wrong computation results. Modelling the interactions between the components with session types ensures correct synchronisation and prevents deadlocks, unexpected termination and orphan messages.

Sleeping barber: A barber is waiting for customers to cut their hair. When there are no customers the barber sleeps. When a customer arrives, he wakes the barber or sits in one of the waiting chairs in the waiting room. If all chairs are occupied, the customer leaves. Customers repeatedly visit the barber shop until they get a haircut. The key element of the solution is to make sure that whenever a customer or a barber checks the state of the waiting room, they always see a valid state. The problem can be implemented using an additional Selector role that decides which is the next customer.

Dining Philosophers: In this use case N philosophers are competing to eat while sitting around a round table. There is one fork between each two philosophers and a philosopher needs two forks to eat. The challenge is to avoid deadlock, where no one can eat, because everyone is possessing exactly one fork. The problem can be implemented with additional role Arbitrator.

Cigarette Smoker: This problem involves N smokers and one arbiter. The arbiter puts resources on the table and smokers pick them. A smoker needs to collect k resources to start smoking. The challenge is to avoid deadlock by disallowing a competition between smokers from picking up resources. This is done by delegating the control to the arbiter, who decides (in a random manner) which smoker to send the resource to. The session types for this use case combines round-robin pattern, sending a random smoker a message to smoke, with multicast, iterating through all the smokers notifying them to exit).

Concurrent Fibonacci: Parallel algorithms require threads that communicate. Session types can use describe the necessary underlying communication between threads that implement parallel algorithms.

5. *Data Structures and Algorithms.* The above layers are using data structures and algorithms. Session types can express the communication concurrent algorithms are using. Furthermore, session types can express the interaction with data structures.
   - Collection: A stack client protocol. Used to control the put, get access to a collection structure - when the stack is empty there is not get. Session types describe the logic/properties that a data structure can have.
   - File Access: Similarly used to control the access on a file. Cannot read from a file if the file is not open first and if the file is empty. The protocols ends by closing the file. Again access to resources can be described using session types.

- Concurrent Fibonacci: Parallel algorithms require threads that communicate. Session types can use describe the necessary underlying communication between threads that implement parallel algorithms.
- Network Topologies: Topologies are scalable and thus parametrised. Pabble is an extension to Scribble that allows us to describe and cope with parametrised protocols.

6. *Hardware*. Hardware mechanisms complete the stratification of domains. The communication of hardware modules may also be expressed using session types.
   - Memory Coherency: Hardware components communicate, here we have two memories that need to be consistent with each other on a hardware level. Session types can describe the hardware (signals/messages) interaction between hardware components.
7. *Security*. Session types can also find applications in the security domain, which is a domain that supports all other domains in the above list.
   - There are some security protocols in the SILL repository. We need to be careful in the description because it is a reference to other people.

3.2 The Simple Mail Transfer Protocol

This section gives a more detailed presentation of the SMTP usecase. SMTP is considered an important implementation for session types since it incorporates challenges that in principle software engineers face.

The SMTP is a protocol that, in contrast with the HTTP, requires handling communication through a number of distinct states (see state machine diagram below), which adds a degree of complexity to the implementation. With the help of session types and the methods and procedures session types support developers can reduce the effort needed to handle the complexity.

Furthermore, SMTP introduces a structure on an email, which is composed by subject, main part and attachments as well as meta-data for the email. Streaming such tree-structured data types over the network is a task that can be rigorously handled and ensured by session types, due to its structured natured. Transforming network data-structures (e.g. text headers) to/from session types structures is an important procedure to understand when applying session types.

SMTP is an application layer protocol. SMTP is an internet standard electronic mail transfer protocol which typically runs over a TCP (Transmission Control Protocol) connection. It was first defined in RFC 821 [**?**] and later extended in RFC 5321 [**?**], which is the version we consider here.

An SMTP interaction consists of an exchange of text-based commands between the client and the server. For example, the client sends the `EHLO` command to identify itself and open the connection with the server. The commands `MAIL FROM` : <address> and `RCPT TO` : <address> specify the e-mail address of the sender and the receiver of the e-mail. The `DATA` command allows the client to specify the text of the e-mail. The `QUIT` command is used to terminate the session and close the connection. The responses from the server have the following format: three digits followed by an optional dash "-", such as `250-`, and then some

text, like OK. The server might reply to `EHLO` with `250` <text> or to `MAIL FROM` or `RCPT TO` with `250` OK.

The implementation of the protocol implements an interaction between a Simple Mail Transfer Protocol (SMTP) client and a `gmail` server.

The global protocol for the SMTP is defined in Scribble, based on Hu's work [**?**].

```
global protocol SMTP(role S, role C) {
 // Global interaction between server and client.
  _220(String) from S to C;
   choice at C {
    ehlo(String) from C to S;
    rec X {
      choice at S {
        _250dash(String) from S to C;
         continue X;
      } or {
       _250(String) from S to C;
        choice at C {
         ...
         rec X1 {
           ...
           choice at S {
             _250dash(String) from S to C;
             continue X1;
          } or {
            250(String) from S to C;
            ...
            rec Z1 {
              ...
              data(String) to S;
              ...
              rec Z3 {
                choice at C {
                  subject(String) from C to S;
                  continue Z3;
                } or {
                  dataline(String) form C to S;
                  continue Z3;
                } or {
                  atad(String) from C to S;
                  _250(String) from S to C;
                  continue Z1; }
                }
              }   ...
           }
           ...
       }
      }
    }
  }
 } or  { quit(String) from C to S; }
}
```

The above Scribble protocol formally specifies the description in Section **??**. The global protocol `SMTP` specifies an interaction between two roles being the

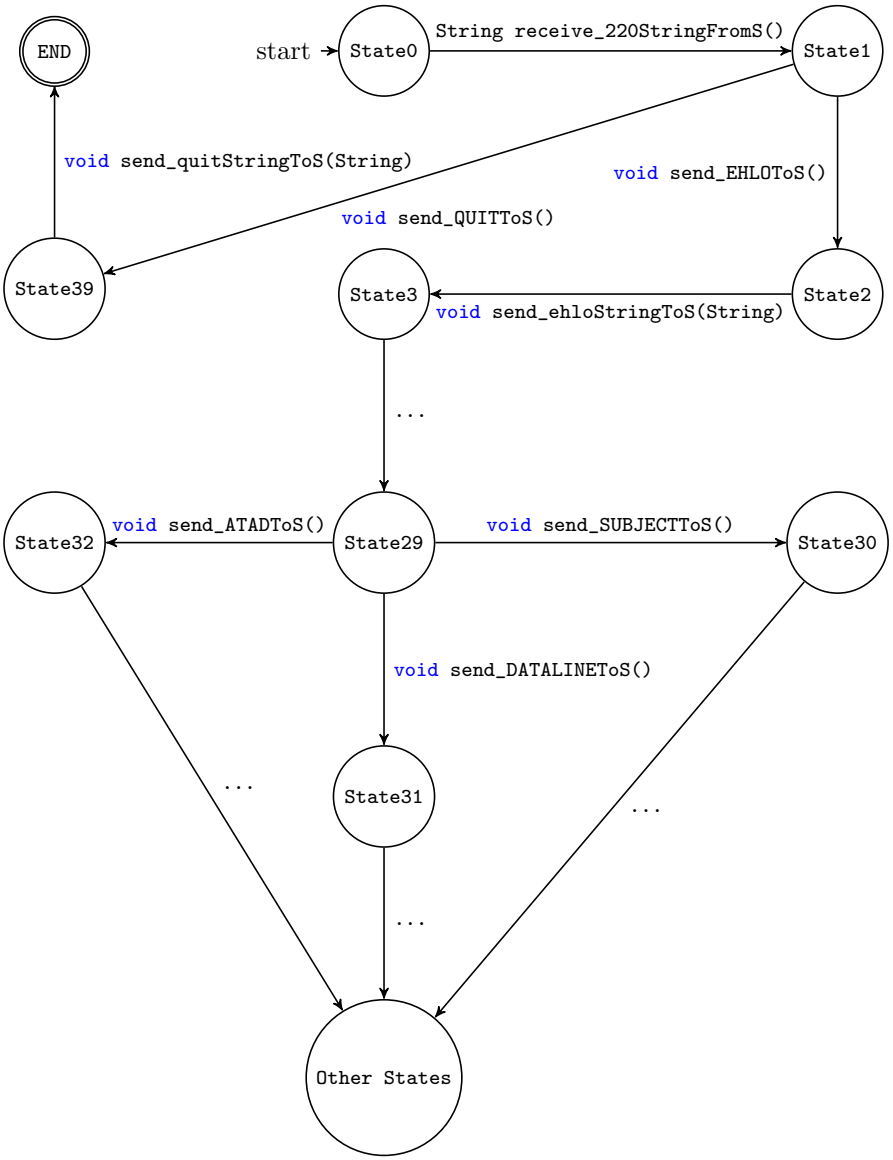
client ant the server of the protocol. The protocol is then projected to the individual communication specifications of each of the participant roles.

The global protocol is used to project the communication behaviour of roles `C` and `S`. The focus is on the projection of the client since the interest is on interacting with a real server.

This part of SMTP describes a loop (`rec Z1`) where the client chooses among the messages `SUBJECT`, to send the subject, `DATALINE`, to send a line of text, or `ATAD` to terminate the e-mail by sending a dot.

```
1  local protocol SMTP_C(role S, self C) {
2    _220(String) from S;
3    choice at C{
4      ehlo(String) to S;
5      ...
6      rec Z1 {
7        ...
8        data(String) to S;
9        ...
10       rec Z3 {
11         choice at C {
12           subject(String) to S;
13           continue Z3;
14         } or {
15           dataline(String) to S;
16           continue Z3;
17         } or {
18           atad(String) to S;
19           _250(String) from S;
20           continue Z1; }}}
21       ...
22   } or { quit(String) to S; }
23 }
```

The local type can be represented as a state machine:



¿¿¿¿¿¿¿ 7ae4313febf389ced7dd794120bf27495cd58faf