Integrating Session Types into Software Development: A Repository of Use-Cases

Dimitrios Kouzapas[1], Tihana Galinac Grbac[4], Rumyana Neykova[2], Raymond Hu[2], Simon J. Gay[1], Ornela Dardha[1], Nicholas Ng[2], Simon Fowler[3], Sam Lindley[3], J. Garrett Morris[3], Roly Perera[1], Florian Weber[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. By incorporating session types into programming languages and tools, 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. [6] 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.

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 satisfactory level of understanding of session types. This situation is an obstacle for the adoption of session types as part of practical software development.

More generally, the problem of weak industrial adoption of formal methods has been extensively studied. The main barriers are lack of information, lack of tool support, and cost [4]. Even if these barriers are removed, there is still the need for a well-defined strategy for managers within a software organisation to follow while adopting formal methods [14]. It is therefore necessary to broaden the engagement of software engineers with formal methods and formal methods experts.

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.

Our approach to meeting 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 computing. Use-cases can bridge the communication gap between researchers and can help the wider promotion of the principles of session types.

Furthermore, current research directions in session types are pushing towards a dialectic between session types and disciplines such as software engineering and system design and implementation. Evidence for this trend comes from the research experience gained from the project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (ABCD for short) [2]. The goals of the ABCD project include (i) working with industrial partners towards integrating session types in real world use-cases; and (ii) compiling a set of use-cases 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 2.4). The paper includes a discussion on their possible adoption by software engineers in the software development process (Section 2.4 and Section 4).
3. Demonstrate the robustness, functionality, and overall applicability of session types through a diverse overview of use-cases (Section 3) that cover: (i) a range of domains that exhibit different computational needs; (ii) interpretations of session types in several programming paradigms; and (iii) current tools and technologies that integrate session types into the software development process.

This paper is intended to 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.

2 Session Types

This section introduces session types through a discussion of the need for them in software engineering. This section takes the approach to identify and define a clear and distinctive terminology for transferring the basic ideas of session types to a wider audience. It also presents the Scribble [15] protocol description language, which is a formal language for describing protocols using the principles of session types.

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

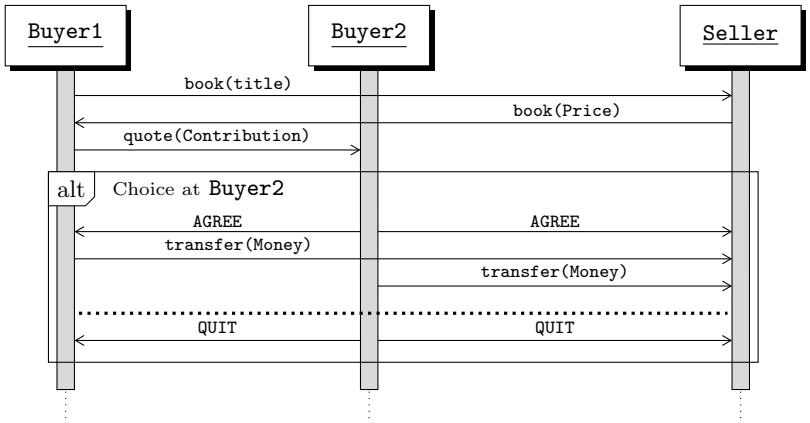

Fig. 1. The Bookstore Protocol: Sequence Diagram

diagrams Sequence diagrams 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 bookstore protocol, which is a use-case in the ABCD online repository [1]. The bookstore 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 diagram, each role is represented by a vertical line and communication messages are represented by horizontal arrows indicating the direction of communication flow. Intuitively, the communication progress proceeds downwards. Thus the protocol's behaviour is represented visually within the diagram. However, sequence diagrams 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 diagrams. Each software module that participates in the protocol implements its *local* software role within the protocol. A limitation of sequence diagrams 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.*

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 operation has a corresponding receive operation.
  - *Message 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*: If the protocol has not yet terminated, then there is a pair of components that can communicate safely, leading to a system state that again satisfies the safety and progress properties.

In addition, the structured nature of session types enables a methodology for integrating session types in software development.

Furthermore, the Scribble tool-chain [15] 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 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 Types 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

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

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

Fig. 2. Scribble: Session Type Protocol for the Bookstore Use-Case

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. Local protocols can also be seen as syntactic representations of state machines, where each state machine edge depicts a communication operation.

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.

2.3 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. 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, a message is defined as a list of types annotated with a label.

In line 5 of the protocol there is a choice interaction, 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 `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.

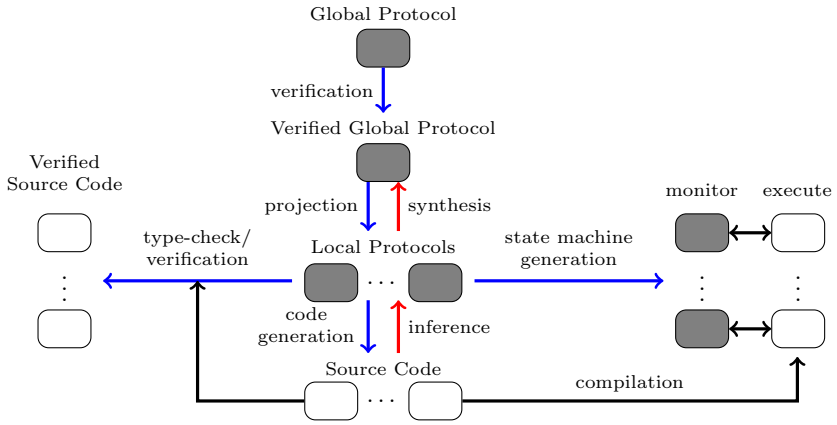

Fig. 3. Use of Session Types in Software Development

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

Line 1 of the lower protocol defines the local protocol for `Buyer1` of the Bookstore 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.

2.4 Integrating Session Types into Software Development

Figure 3 illustrates a holistic approach to the integration of session types in software engineering, that follows procedures derived out of the rigorous nature of 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.

Furthermore and in addition to a protocol description language, Scribble offers a tool-chain that implements most of the procedures described in Figure 3.

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 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's correctness: for example, checking that choices are communicated to all roles whose subsequent behaviour depends on them. 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.

Type-check/Verification: 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 typechecking procedure. The typechecker 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 typechecker 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 *communication safety* and *communication progress* properties of session types.

Code generation: 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 program templates that implement communication behaviour and need manual work by the developers to add computational logic. Scribble supports the generation of a Java API for each one of 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 method call 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 exactly once.

Monitoring: A third 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, a local monitor runs a state machine that is derived from 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 supports the procedure of deriving state machines from local protocol syntax.

Inference/Synthesis: Figure 3 also shows a bottom-up approach, where the communicating software source codes are developed independently. Then an inference procedure identifies the communication behaviour of each component, resulting in a set of local protocols. After that, an automatic synthesis procedure can be used, where the derived local protocols are synthesised to create a global protocol. Successful synthesis of the global protocol guarantees that the software has the communication safety and progress properties.

In any of the above methodologies, session types can be used as requirements 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. In 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 reasoning techniques. Protocols can also be expressed as diagram. For example, a local protocol is a syntactic form of a state machine. Global protocols can be expressed as petri-nets [**?**], communicating finite state machines [**?**] and BPMN choreographies [**?**].

Session fidelity: 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 of developing the usecases for the ABCD online repository. The Scribble platform and its procedures have an independent status next to the plethora of technologies and programming paradigms used in the case studies.

3 Session Types Tools and Use-Cases

This section firstly gives an overview of a selection of languages and tools based on session types. We then summarise the use-cases in the ABCD online repository [1] that feature session types. ... protocol specispecified . It also gives more detailed presentations of two particular use-cases: (1) the bookstore, which was introduced in Section 2 and is a standard example of session types; (2) Simple Mail Transfer Protocol (SMTP), which is a standard network protocol.

Table 2 lists the name of each use-case, its original source, and the language(s) and/or tool(s) that have been used to implement it. Full source code, running examples and and detailed descriptiona of all of the use-cases can be found in the repository [1]. The use-cases are organised into application domains and are intended to be representative examples for each domain.

3.1 Technologies based on Session Types

Several technologies and tools based on session types have been used to implement the use-cases that we describe in Section 3.

1. Session Java [9] is a Java extension that statically checks binary session types for communication channels that are implemented as an API.
2. Eventful Session Java [8] supports asynchronous, event-driven programming, using session types to track progress through individual sessions.
3. Multiparty Session C [13] supports programming in C with the MPI library. The compiler statically checks session types.
4. SPY [11] uses session types for runtime monitoring of communication protocols in Python.
5. Two technologies apply session types to the actors paradigm:
   - SPY is used to monitor Python threads that simulate actors [10].
   - Session type monitoring is also used for Erlang actors [5].
6. Pabble [12] extends Scribble to express communication structures that are parametrised by the number of roles. Pabble protocols are implemented and typechecked for the C+MPI framework.

Use Case Name	Source	Implementation	Remark
Internet Application Protocols			
SMTP	[18]	Java API, Mungo, Links	
HTTP	[?]	Java API	
DNS	[?]	Erlang	
POP3	[?]	Mungo	
Web Service Applications			
Book Strore	[?]	SJ, Mungo, Java API	interaction logic
Travel Agency	[?]	SJ	
Chat Application	[?]	Erlang	
Network Topologies and Parallel Algorithms			
Ring	[?]	MPI	
Butterfly	[?]	MPI	
All to All	[?]	MPI	
Stencil	[?]	MPI	
Farm (Master-Worker)	[?]	MPI	
Map Reduce	[?]	MPI	
Classic Concurrency Problems			
Dining Philosophers	[?]	SPython	synchronisation
Sleeping Barber	[?]	SPython, SScala	
Cigarette Smoker	[?]	SPython	race conditions
Peano Numbers	[]	GV, Links	
Data Structures			
Queue	[?]	Sill	
Collections	[?]	Mungo	a stack client
File Access	[?]	Mungo	file access client

Table 1. Overview of the use-case repository (Add papers summarising the results)

7. Mungo [3] is a tool that integrates session types into the object-oriented paradigm through the notion of typestate. Communication operations on a channel are accessed via a state-dependent interface. The Scribble to Mungo (StMungo) tool transforms Scribble protocols into Mungo interfaces.
8. GV: A functional programming implementation of binary session types [19].
9. The web programming language LINKS [**?**] uses the linear logic interpretation of session types [19] to statically type-check protocols.
10. The Scribble API generator, automatically creates an API for protocols, where it describes each protocol state as a class equipped with communication methods. Runtime checks ensure state linearity.
11. SILL: Functional programming based on the linear dual intuitionistic interpretation of session types. Separate this from the repository — say something elsewhere.

3.2 Application Domains Covered by the Use-Cases

The use-cases are drawn from a wide range of application domains, in order to demonstrate that session types can capture a broad area of communication specifications. Covering a wide variety of patterns which allows comparison of features available in the session-based frameworks being evaluated.

RN For each domain explain (1) What are the challenges when implementing the use cases in that particular domain? (2) How Session Types overcome these challenges (currently this is not well explained).

Internet Application protocols This set of use cases is the first step towards more realistic (real-world) applications. Implementing Internet application protocols The motivation to apply session types verification techniques to Internet Application Protocol is three-fold.

First, an Internet Application protocol should conform to a semi-informal RFC (request for comments specification). RFCs are written in an english prose. As a result specifications are tedious and error-prone to implement. Second, the aim of the specification is to enable interaction between heterogeneous systems. Third, implementations often involve untrusted components.

Session types remove a burden from the developer because they automatically verify that the state of the code follows the state of the protocol. Moreover, applying runtime monitor helps protecting the implemented components from interactions with untrusted parties. For example, the SMTP client implementations implemented in Java can interact with any (none session-based) SMTP server implementation.

Web Service Application Internet Application protocols are often binary. The aim of the use cases presented in this domain is to demonstrate the usage of multi-party protocols. In web services, applications make an extensive use of communications among multiple components and services through a standardised format. Business transactions using web services are often termed *business protocols* because each of them obeys an agreed-upon conversation structure. Therefore, a protocol expresses an execution logic for the application. The first

two use cases are taken from the specification repository of an official W3C working group set up to establish a domain-specific language for specifying business protocols.

This set of use cases is a practical indication that session types can be used to describe the behavioural logic of an application. The current development of the usecases follows a top-down design approach, where the projection of the global protocol is used by different implementing technologies.

Network Topologies and Parallel Algorithms The highlights of this domain are: (1) computation and communication are often separated, (2) topologies are scalable and thus parametrised and (3) topologies are reusable, comprising limited set of patterns.

Issues with High Performance Computing (HPC) programmers productivity and programs correctness has long been recognised. A survey among MPI (the de facto standard programming model for HPC) programmers [**?**] ranks *communication mismatches* as the most common errors. Communication mismatches include: (1) Send/receive inconsistency caused by an error in program execution flow) and (2) Send/receive inconsistency caused by incorrect sender and/or receiver specified for a message.

So far we have examined use cases, where the number of interacting parties is fixed. Network Topologies, however, often involve unknown number of participants. Pabble is an extension of Scribble with parameterised roles, thus enabling expressing flexible topologies with unknown number of participants. We evaluate the applicability and usability of session types on a well-known set of patterns for parallel programming. Pabble can express all structured patterns in the HPC (High Performance Computing) Dwarf benchmark suit. Dwarf suit captures common patterns of communication and computation. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future.

Applying session types workflow, in particular generating MPI code form well-established patterns, has shown to improve developer productivity. It saves development and debugging efforts for MPI parallel applications, especially for novice parallel programmers.

Classic Concurrency Problems

Coordination of processes is a challenging problem when implementing concurrent systems. A study in [**?**] points out that "the property of no shared space and asynchronous communication can make implementing coordination protocols harder and providing a language for coordinating protocols is needed". The problem is best exemplified when implementing classic concurrency problems.

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 be-

haviour and nested choice branches. Often precise message sequence should be followed. Moreover, it is not obvious if the components can be safely composed. 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 interactions between components with session types ensures correct synchronisation and prevents deadlocks, unexpected termination and orphan messages.

3.3 The Simple Mail Transfer Protocol

The SMTP is an internet standard for electronic mail transfer. It originally followed RFC 821 [17] and was later extended in RFC 5321 [16], which is the version consider here. SMTP is an application layer protocol, that typically runs on a TCP/IP connection.

Motivation: The session typed implementation of the STMP has to offer a number of motivations for using session types for software development. Firstly, it demonstrates how *natural* it is to describe of a widely used standard protocol in session types. Secondly, and in contrast with the HTTP, it is a rather state-full protocol (see state machine diagram below), i.e. it requires implementation of a sequence of communication states. The complexity of states adds to the effort of developing SMTP. With the use of session typed support that effort can be reduced. Thirdly, the SMTP implies streaming a tree data-structure that corresponds to the structure of an email. It is thus, shown how session types can describe the streaming of tree data-structures.

Session Type Specification The implementation of the SMTP starts with the definition a Scribble global protocol [7] that describes the interaction between a client and the server. Part of the global protocol can be found in Appendix A.4. The global specification involves a number of interactions that include nested recursion and choices, thus the SMTP is a rather state-full protocol. An SMTP interaction is an exchange of text-based commands between a client and the server. For example, in line 4 the client may send the `EHLO` command to identify itself and open a server connection. Responses from the server have follow the format: three digits followed by an optional dash "-", and then some text, e.g. in line 8 the server might reply to `EHLO` with `_250 <text>` check here. Lines 18-23 there is a loop that allows the client to construct and stream a tree data structure that contains the email information to the server. Specifically, in line 19 the client streams the subject of the email to the server and then performs iteration to stream email data lines, as in line 20. The streaming finishes in line 21 and 22 when the `atad` message is sent and the server replies to the client.

The protocol is then projected to the local specifications of the roles server and client. Part of the local protocol of the client can be found in Appendix A.5. Local protocols can be represented as state machines. The state machine for the client protocol can be found in Appendix A.6.

Implementation The above session protocol drives the implementations: (i) of an SMTP client in Mungo via StMungo, (ii) the SMTP client and server in

LINKS, and (iii) the generation of a Java API that can be used for implementing SMTP servers and clients. In all the above technologies, the implementation of the SMTP protocol was aided by automated procedures that can handle the communication structure of a program. Furthermore, the implementation need not to be validated for communication correctness, since if the session protocol is correct and session fidelity is ensured then the implementation is correct.

The Mungo implementation of the protocol implements an interaction between an SMTP client and the `gmail` server. It uses the StMungo tool to translate the client local protocols into Mungo typestate specifications. StMungo also generates template code that implements the communication functionality of the client. The final implementation is type-checked by the Mungo tool against the generated typestate specification.

LINKS implements SMTP server and client by first translating the local protocols into LINK channel types. The programmer then develops the client (resp., server) as a session typed channel, used to exchange information with the server (resp., client).

Finally, Scribble offers the possibility of generating a Java API; each protocol state corresponds to a generated class, with each class implementing as methods the communication operations available in each state. Method calls return an object corresponding to the next state. A runtime check ensures that each object/state is used only once. The Java API can guide the developer in order to implement a variety of SMTP clients and servers.

4 Session types and Software Engineering

Write here about the main parts of software life-cycle (requirements, specification, design, implementation, verification, testing, maintenance), and software management and organisation.

Support the text later with the discussion of section 2 and the results of Section 3.

Session types is a formal method that belongs to the class of type systems. On the one hand, types is a static analysis technique applied directly on the source code in order to verify a set of predefined properties. In the case of session types these properties derive from the fundamentals of *good* communication. On the other hand, types are subject to a syntax, and possibly a semantics, that describes an implementation structure. These two facts allow for the integration of session types both in-depth and in-breadth in the software development process. Furthermore, this integration is supported by automated procedures. Consequently, session types can drive the entire process of software development. The latter claim is supported by the experience gained from the repository presented in Section 3.

Hint how session types can be used in the software life-cycle. e.g. in a top-down approach: specify communication requirements and build prototypes, use session types to describe software structure, type-checker/code generators/mon-

itors can be used in implementation. bottom-up approach: design structures, infer specification, synthesis as a verification technique, etc...

In contrast other formal methods, have restricted applications in specific software development phases. For example, model checking narrows down to verification/testing phase of software development. It requires specific expertise, to define of a complete model that approximates software and furthermore, perform semantic analysis based on non-predefined properties. Often in the case of error detection, results are hard to analyse and developers need to re-iterate through previous phases to fix errors. Another example is the usage of formal specification languages that do not find any usage after the specification phase.

References

1. ABCD: Session Types Usecase Repository. https://github.com/epsrc-abcd/session-types-use-cases.
2. From Data Types to Session Types: A Basis for Concurrency and Distribution. http://groups.inf.ed.ac.uk/abcd/.
3. Mungo Webpage. http://www.dcs.gla.ac.uk/research/mungo/.
4. J. A. Davis, M. Clark, D. Cofer, A. Fifarek, J. Hinchman, J. Hoffman, B. Hulbert, S. P. Miller, and L. Wagner. *Formal Methods for Industrial Critical Systems: 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings*, chapter Study on the Barriers to the Industrial Adoption of Formal Methods, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
5. S. Fowler. Monitoring erlang/otp applications using multiparty session types. Master's thesis, University of Edinburgh, 2015.
6. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
7. R. Hu. Monitoring SMTP with Scribble and Java. Unpublished presentations, 2015.
8. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
9. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
10. R. Neykova and N. Yoshida. Multiparty session actors. In *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 131–146, 2014.
11. R. Neykova, N. Yoshida, and R. Hu. SPY: local verification of global protocols. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 358–363, 2013.
12. N. Ng and N. Yoshida. Pabble: Parameterised scribble for parallel programming. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 707–714, 2014.

13. N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS*, volume 7304 of *LNCS*, pages 202–218. Springer, 2012.
14. C. Ponsard, J.-C. Deprez, and R. Landtsheer. *Formal Methods for Industrial Critical Systems: 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings*, chapter High-Level Guidance for Managers Deploying Formal Methods in Their Organisation, pages 139–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
15. Scribble Project homepage. www.scribble.org.
16. Extended simple mail transfer protocol, RFC 5321. https://tools.ietf.org/html/rfc5321.
17. Simple mail transfer protocol, RFC 821. https://tools.ietf.org/html/rfc821.
18. The Simple Mail Transfer Protocol. http://tools.ietf.org/html/rfc5321.
19. P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

A Appendix: Use-Cases

A.1 Use-Cases (original)

This section summarises the use-cases in the ABCD online repository [1]. It also gives more detailed presentations of two particular use-cases: (1) the bookstore, which was introduced in Section 2 and is a standard example of session types; (2) Simple Mail Transfer Protocol (SMTP), which is a standard network protocol.

Table 2 lists the name of each use-case, its original source, and the language(s) and/or tool(s) that have been used to implement it. Full source code, running examples and and detailed descriptiona of all of the use-cases can be found in the repository [1]. The use-cases are organised into application domains and are intended to be representative examples for each domain.

A.2 Technologies based on Session Types

Several technologies and tools based on session types have been used to implement the use-cases that we describe in Section 3.

1. Session Java [9] is an extension of Java with binary session types for communication channels, supported by a runtime library. The compiler statically checks session types.
2. Eventful Session Java [8] supports asynchronous, event-driven programming, using session types to track progress through individual sessions.
3. Multiparty Session C [13] supports programming in C with the MPI library. The compiler statically checks session types.
4. SPY [11] uses session types as the basis for runtime monitoring of communication protocols in Python.
5. Two technologies apply session types to the actors paradigm:
   – An implementation that uses SPY to monitor python threads that simulate actors [10].

Use Case Name	Source	Implementation	Remark
Internet Application Protocols			
SMTP	[18]	Java API, Mungo, Links	
HTTP	[?]	Java API	
DNS	[?]	Erlang	
POP3	[?]	Mungo	
Web Service Applications			
Book Strore	[?]	SJ, Mungo, Java API	interaction logic
Travel Agency	[?]	SJ	
Chat Application	[?]	Erlang	
Network Topologies and Parallel Algorithms			
Ring	[?]	MPI	
Butterfly	[?]	MPI	
All to All	[?]	MPI	
Stencil	[?]	MPI	
Farm (Master-Worker)	[?]	MPI	
Map Reduce	[?]	MPI	
Classic Concurrency Problems			
Dining Philosophers	[?]	SPython	synchronisation
Sleeping Barber	[?]	SPython, SScala	
Cigarette Smoker	[?]	SPython	race conditions
Peano Numbers	[]	GV, Links	
Data Structures			
Queue	[?]	Sill	
Collections	[?]	Mungo	a stack client
File Access	[?]	Mungo	file access client

Table 2. Overview of the use-case repository (Add papers summarising the results)

  - A similar approach on monitoring session types is also used for the Erlang programming language. Is this Erlang system one of ours? Simon Fowler's MSc
6. Pabble [12] extends Scribble to express communication structures that are parameterised by the number of roles. Pabble protocols are implemented and typechecked for the C+MPI framework.
7. Mungo [3] is a tool that integrates session types into the object-oriented programming paradigm through the notion of typestate. Communication operations on a channel are accessed via a state-dependent interface. The Scribble to Mungo (StMungo) tool transforms Scribble protocols into Mungo interfaces.
8. GV: A functional programming implementation of binary session types. Refer to Phil's paper.
9. The web programming language Links [] includes static typechecking of protocols, using the linear logic interpretation of session types [].
10. SILL: Functional programming based on the linear dual intuitionistic interpretation of session types. Separate this from the repository — say something elsewhere.
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.

A.3 Application Domains Covered by the Use-Cases

The use-cases are drawn from a wide range of application domains, in order to demonstrate that session types can capture a broad area of communication specifications.

1. *Network Application / Business Logic.* The first two applications in Table 2 come from the domain of network applications.
   The first is the online bookstore, which was discussed in Section 2. It is a standard use-case demonstrating the basic features of session types. The communication protocol represents the execution logic of the application. The implementation of this use-case follows a top-down design approach, in which the global protocol is projected to local protocols. There are several versions of the bookstore, using different implementation languages: Session Java
   The bookstore 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

need LINKS and the hybrid implementation

- 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.

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 composed. 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. Pattern-based structured parallel programming. Pabble can express all structured patterns in the HPC (High Performance Computing) Dwarf benchmark suit which capture common pattern of communication and computation. A dwarf is an algorithmic method that captures a pattern of computation and communication. The Seven Dwarfs, constitute equivalence classes where membership in a class is defined by similarity in computation and data movement. The dwarfs are specified at a high level of abstraction to allow reasoning about their behavior across a broad range of applications. Programs that are members of a particular class can be implemented differently and the underlying numerical methods may change over time, but the claim is that the underlying patterns have persisted through generations of changes and will remain important into the future. The dwarfs present a method for capturing the common requirements of classes of applications while being reasonably divorced from individual implementations. The dwarfs present a method for capturing the common requirements of classes of applications while being reasonably divorced from individual implementations.Helps productivity, less LOC, clear description. Results show that our workflow saves development and debugging efforts for MPI parallel applications, especially for novice parallel programmers)

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.

A.4 Global Protocol for SMTP

```
1  global protocol SMTP(role S, role C) {
2    _220(String) from S to C;
3    choice at C {
4      ehlo(String) from C to S;
5      rec X {
6        choice at S { _250dash(String) from S to C; continue X; }
7        or {
8          _250(String) from S to C;
9          choice at C {
10           ...
11           rec X1 {
12             ...
13             choice at S { _250dash(String) from S to C; continue X1; }
14             or {
15               250(String) from S to C; ...
16               rec Z1 {
17                 ... data(String) to S; ...
18                 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; }
}
```

A.5 SMTP Client Local Protocol

```
local protocol SMTP_C(role S, self C) {
  _220(String) from S;
  choice at C{
    ehlo(String) to S; ...
    rec Z1 {
      ... data(String) to S; ...
      rec Z3 {
        choice at C { subject(String) to S; continue Z3;  }
        or          { dataline(String) to S; continue Z3; }
        or          { atad(String) to S; _250(String) from S; continue Z1; }}}
      ...
  } or { quit(String) to S; }
}
```

A.6 SMTP Client State Machine