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. [8] 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 [3]. 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 [19]. 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) [1]. 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 [21] 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

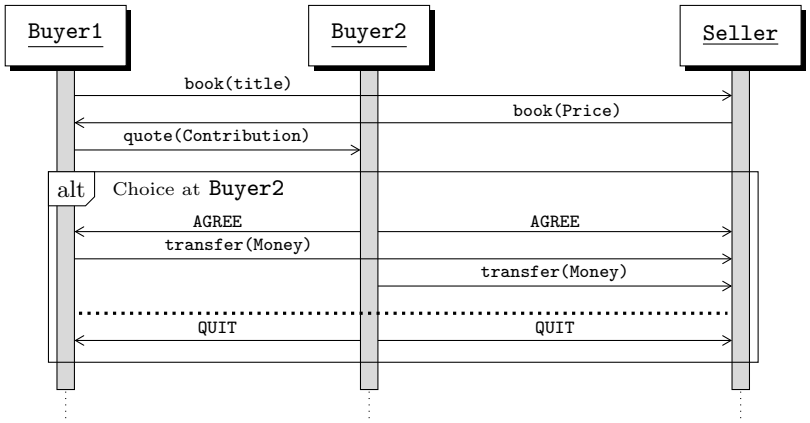

Fig. 1. The Bookstore Protocol: Sequence Diagram

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 [25]. 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 [21] 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.

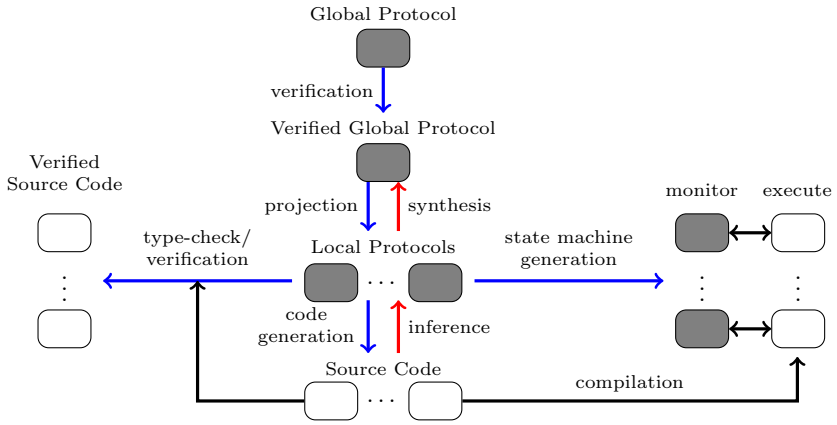

Fig. 3. Use of Session Types in Software Development

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. Blue arrows represent a top-down view of the diagram while red arrows take the bottom-up approach. 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 [6], communicating finite state machines [4] and BPMN choreographies [13].

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.

2.5 Technologies based on Session Types

We describe a selection of practical languages, frameworks and tools, based on the range of session types engineering methodologies described above, that we have used to implement the use-cases (describe in Section 3).

1. Session Java [11] is a Java extension that statically checks binary session types for communication channels that are implemented as an API.
2. Eventful Session Java [10] supports asynchronous, event-driven programming, using session types to track progress through individual sessions.
3. Multiparty Session C [18] supports programming in C with the MPI library. The compiler statically checks session types.
4. SPY [16] 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 [15].
   - Session type monitoring is also used for Erlang actors [7].
6. Pabble [17] 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.
7. Mungo [14] 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 [27].
9. The web programming language LINKS [**?**] uses the linear logic interpretation of session types [27] 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 Use-Cases

This section summarises the ABCD on-line repository [25] use-cases. The description focuses on the solutions session types offers in the different use-case domains. During the discussion the outline details of a selection of individual use-case is given. The discussion goes into more depth on two particular canonical use-cases: (1) the bookstore, a standard session types example introduced in Section 2; and (2) Simple Mail Transfer Protocol (SMTP), a standardised Internet application protocol.

Table 1 lists the name of each use-case, its original source, and the language(s) and tool(s) that have been used to implement it. Full source code, running examples and detailed description of all of the use-cases can be found in [25]. Use-cases are organised into application domains and are intended to be representative examples for each domain. Some of the use-cases were presented independently in the bibliography.

3.1 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 and a wide variety of patterns. On one hand, applying session types to diversity of use-cases demonstrates a range of features and extensions in the tools. On the other hand, each domain offers appreciation of session types from a different perspective.

CUT : We explain the specific challenges tackled by session types when applied to a specific domain. Communication safety guarantees are essential when programming in an error-prone environment (as in High Performance Computing). Session fidelity guarantee is essential when protocols are specified in a semi-formal format (as in Internet Application Protocols). Ensuring progress is challenging when a complex coordination of multiple parties is required (as is Web Services) and compositionality is not apparent (as in Classic Concurrency Problems). 1

1 do we need this long paragraph

Web Service Application The aim of the use cases presented in this domain is to demonstrate the usage of multi-party protocols, as descriptions of the communication behaviour of network components. In web services, applications make an extensive use of communications among multiple components and services through a standardised format and agreed-upon protocol structure. The protocol stipulates the business logic to be implemented by the application.

To show the applicability of session types to this domain we implemented use cases (Book Store, Travel Agency) designed by the WS-CDL Working Group [26], intended to represent general concepts common to many applications of web services

The book-store is a canonical example for demonstrating business logic interaction. Its global and local protocol for `Buyer1` can be found in Figure 2. Two Buyer roles, `Buyer1` and `Buyer2`, interact with a `Seller` role to buy a book. `Buyer1` requests a quote for a book. The `Seller` replies with a price and then

Use Case Name	Source	Implementation	Remark
Web Service Applications			
Book Strore	[26]	SJ, Mungo, Java API	interaction logic
Travel Agency	[26]	SJ	
Chat Application	[7]	Erlang	
Internet Application Protocols			
SMTP	[20]	Java API, Mungo, Links	stateful interaction
HTTP	[20]	Java API	stateless request/response
DNS	[20, 7]	Erlang	
POP3	[20]	Mungo	stateful interaction
Classic Concurrency Problems			
Dining Philosophers	[12]	SPython	synchronisation
Sleeping Barber	[12]	SPython, SScala	
Cigarette Smoker	[12]	SPython	race conditions
Parallel Algorithms			
Concurrent Fibonacci		Mungo	parallel algorithm
Monte Carlo	[**?**]	MPI	parallel algorithm
Jacoby	[**?**]	MPI	parallel algorithm
Network Topologies			
Ring	[2]	MPI	parametrised
Butterfly	[2]	MPI	parametrised
All to All	[2]	MPI	parametrised
Stencil	[2]	MPI	parametrised
Farm (Master-Worker)	[2]	MPI	parametrised
Map Reduce	[2]	MPI	parametrised
Data Structures			
Collections	[14]	Mungo	a stack client
File Access	[14]	Mungo	file access client
Peano Numbers	[**?**]	GV, Links	

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

`Buyer1` asks `Buyer2` for a contribution. `Buyer2` then makes a choice: i) `Buyer2` can agree to contribute and informs both `Buyer1` and the `Seller`, with the price transferring interactions taking place afterwards; ii) `Buyer2` does not agree and ends the interaction informing both `Buyer1` and `Seller`. It is implemented in Mungo using the automatic code generation of StMungo. The effort of the programmer was focused on filling the decision logic for the values exchanged on the message interactions. 2

2 put the other implementations

CUT : The web-services 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 use-cases follows a top-down design approach, where the projection of the global protocol is used by different implementing technologies.

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

First, an Internet Application Protocol should conform to a RFC (request for comments specification), which is an error prone procedure due to: (1) the semi-formal nature of the protocol description; and (2) the rich interaction structures described (as observed in SMTP and POP3 use-cases). Session protocols are expressive enough to capture the communication patterns required for the popular use-cases in this domain. Second, the aim of the specification is to enable interactions between heterogeneous systems. The software development method allows for component verification in different technologies Third, implementations often involve untrusted components. Using session-based runtime monitors both incoming and outgoing messages of a component are verified, thus protecting the implemented components from interactions with untrusted parties. For example, the DNS implementation in the repository is implemented in monitored Erlang but it can interact with any (none session-based) server implementations.
CUT : Monitoring ensures correct communication behaviour with untrusted or erroneous components.

Another widely used internet protocol implemented in the repository is HTTP. Despite, the fact that HTTP is a basic request-response protocol, it has a number of different commands that require attention. HTTP's global specification can be defined in many different ways: e.g. a request/response pattern with large message types. The approach taken here is to construct a request (resp., response) header as a stream of simpler messages passed that construct the overall request (resp., response) header. The implementation in the online repository first describes the HTTP in Scribble and then automatically generates a Java API.

The SMTP is also implemented as a use-case in the repository. A detailed description of SMTP can be found in Section 3.2.

CUT : **Ray** ...binary (thin abstraction layer over TCP session), can have "rich" interaction structure (e.g. SMTP, POP3 conversations), simple data types (ASCII text)...

Classic Concurrency Problems Coordination of processes is a challenging problem when implementing concurrent systems. A study in [24] 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. They require correct coordination among multiple components to avoid starvation and deadlock. Preserving the causalities between interactions is challenging since the communication often involves complex patterns, that combine long sequence of interactions with recursive behaviour and nested choice branches. Moreover, it is not obvious if the components can be safely composed. Sending a wrong message type, or 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 as demonstrated by the implemented use cases. The repository includes correct Scribble protocols and verified endpoint implementations in Python for classic problems of dining philosophers, sleeping barber, and cigarette smoker. The sleeping barber problem is also implemented in Session Scala. The problem is modelled with three components: the customer, the shop and the barber, using session types to formalise their expected interactions such that the customer and/or the barber are always synchronised on the number of waiting customers in the shop. The implementation shows how multiple concurrent sessions (one per customer) can be handled by single-threaded programs (the shop and the barber). It also shows how to address the problem with session delegation, by leveraging higher-order session types.

Both Scala and Python implementations rely on a custom session types libraries and do not require language extensions. The Scala library provides static verification, while the Python library uses runtime monitors.

Parallel Algorithms Issues with High Performance Computing (HPC) programmers productivity and programs correctness have long been recognised. A survey among MPI (the de facto standard programming model for HPC) programmers [5] 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. These are precisely the type of errors prevented with session types verification.

The Jacobi and Monte Carlo algorithms are classic parallel algorithms that require session types to ensure no errors in their concurrent implementation. In addition, concurrent Fibonacci is an example that shows the underlying communication pattern needed by a parallel algorithm. The algorithm assumes two nodes that store consecutive Fibonacci numbers. An exchange in numbers allows for the calculation of the next Fibonacci number.

Network Topologies The previously presented use-cases in the repository involved a fixed number of interacting parties. Parallel patterns, however, often involve unknown number of participants. To overcome this problem, the protocol description language Pabble is used, which is an extension of the Scribble language that features parametrised roles, thus enabling the expression of flexible topologies with unknown number of participants. Pabble can express all structured patterns in the HPC (High Performance Computing) Dwarf benchmark suit [2], which is a popular benchmark suit that captures common parametrised patterns of communication. The dwarfs are specified at a high level of abstraction to allow reasoning about their behaviour across a broad range of applications.

The benchmark suit was implemented using automatic code generation from Pabble specifications. Furthermore, it is shown that the practise of code generation from Pabble specifications improves developers productivity in terms of development and debugging efforts [**?**].

Data Structures The last domain of use-cases describes the behaviour of clients accessing data-structures that are implemented concurrently. The main feature here are: (1) a session protocol can be used to restrict a client to a data-structure in using specific operations; and (2) often data-structures require an ordering in the operations, where such ordering is described using session protocols.

For example, the collection use-case features a client that has access to a stack data-structure. The client is allowed to arbitrary push elements to the stack, but it may need to check if the stack is empty before it pops an element from the stack. A second example is the file access use-case, where: (i) a file needs to be opened before it can be processed; (ii) there is an empty file check before a read; (iii) the close file operation ends the protocol; and (iv) the client is not allowed any other file operations then the operations in (i), (ii), and (iii). Both use-cases are implemented in Mungo.

Travel Agency. ...W3C CDL working group Web services use case, initially specified as binary sessions between multiparty roles (not a single multiparty session), features delegation...

Parallel algorithms. ...shared memory as transport, integrating linearity for session typing into language as general linearity/aliasing control for shared memory message passing optimisation...

..OOI python use cass... ...TODO: add use cases? (e.g., negotiation, resource access control)... ...python: run-time monitoring motivation...

3.2 The Simple Mail Transfer Protocol

The SMTP is an internet standard for electronic mail transfer. It originally followed RFC 821 [23] and was later extended in RFC 5321 [22], 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 SMTP 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 (Appendix **??**), 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 [9] that describes the interaction between a client and the server. Part of the global protocol can be found in Appendix A.1. 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.2. Local protocols can be represented as state machines. The state machine for the client protocol can be found in Appendix A.3.

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 Use-Case Repository Analysis and Future Work

This section does an overall discussion on the results, experience gained and possible future work derived from the application of session types for the implementation of the repository presented in Section 3. The discussion focuses on the

technical lessons learned, as well as an informal discussion on the engineering perspective of the software developed.

subsectionTechnical Lessons Learned Session types exhibit robustness in a wide range of use-cases and domains. All the scenarios demonstrated are structured on the message passing, choice interactions and recursion. However, there are communication patterns in use-case scenarios that remain as open questions, such as the interaction of a role with two other roles in parallel, or the case where a signal can interrupt the communication between two roles.

- types features: parallel ("race conditions"), interruptible, ...more
- type system and endpoint language features: event-driven sessions, CPS actors (Akka/Scala), ...more
- runtime: transport-independent sessions, ST-cFSM correspondence from monitoring, implementing asynchronous distributed delegation, ...more

4.1 Software Engineering

As already discussed in Section 2.4, due to their formal and structured nature, session types can provide support throughout the entire process of software development. The results in Section 3 come to support the ability of session types in offering solutions to software development. Despite the diversity on domains, paradigms, and technologies, common engineering patterns can be found in the software developed in Section 3.

A first observation is the use of session protocols as means for understanding the software communication *requirements*. For example, the book-store session protocol (Figure 2) can give the reader a clear understanding of the interaction logic of the problem. Furthermore, the fact that session protocols can be expressed in diagrammatic languages, help the process of transferring the requirements of a software, e.g. the SMTP client state machine in Appendix A.3.

A second observation is that the main effort for each use-case is focused on the development of a protocol. Session specifications need to be capture the exact and valid communication *requirements* of a system. For example, the SMTP global protocol (Appendix A.1) (resp., HTTP) had a significant amount of time invested in its development, due to the study of RFC 5321 (resp., RFC ????). During the development process the Scribble validating tool help the programmer develop a valid protocol for SMTP.

Thirdly, a session protocol can be seen as an approximation of the software under development that can be used to drive and lessen the burden for the *design* and *implementation* process. For example, the automated code generation in the case of SMTP and HTTP decreased the development time. Furthermore, automatic type-checkers removed the burden of further *validating* the software.

Finally monitoring is not used only for ensuring correct runtime communication. It also be used to *test* the communication against test cases that resemble arbitrary software clients. This approach was used in the development of the DNS and the chat Server. Errors in the monitoring phase lead to an iteration for redevelopment of the session protocol.

A problem with the above discussion is the lack of metrics for an accurate understanding on the cost and effectiveness of the application of session types in software engineering. The only set of use-cases that have such support is network topologies [**?**]. A possible new area of research in session types could be to quantify their effectiveness when it comes to software development, in terms of bugs, line of code, performance, development time and cost.

The use-case repository lacks a use-case developed using the bottom up technique presented in Figure 3, due to the current lack of technologies. Nevertheless, this work is a strong indicator to drive similar research towards that direction as well.

"engineering" ...

5 Conclusion

This paper draws motivation from the need of integrating session types in the software development process. The main problem towards this direction is the difficulty of accessing session types, in order to initiate a process of knowledge exchange between session types experts and a wider academic and industrial audience. This paper, uses as common ground the experience gained by implementing the on-line session types use-case repository [**?**], developed for the requirements of the ABCD project [1], to bridge the above gap. For the purposes of the presentation of the repository the session terminology and method is explained in using non-mathematical definitions.

The ABCD repository contains a diverse set of use-cases, in terms of domains, programming paradigms, and technologies. In the domain of web-services session types demonstrate how to specify and implement the interaction logic between network components. Session types can be used to implement internet application protocols, where a session protocol is used to specify a standard RFC. Network topologies require a definition which is parametric on the number of participants. Classic Concurrency problem demonstrate how session types can offer solutions to problems that control access to resources. The final domain of data structures and parallel algorithms demonstrates solutions on data structure clients and furthermore, how session types can cope with the underlying communication requirements of parallel algorithms.

The formal and structured nature of session types can offer a well studied method for applying session types in the software development method. The Scribble platform is a tool based on the theory of session types and it is used as a core tool for the implementation of the uses-cases in the ABCD repository.

The main conclusions derived in this paper are: the feasibility, the robustness and the diversity of session type solutions. The integration of session types in the software development process goes beyond the theory and can now be presented in practical terms.

Furthermore, a wide opportunity of future research arises in the wider area of applying session types in real software. The immediate objective is to continue

developing technology, and enriching the repository with a more diverse set of use-cases. Furthermore, a new area of research may arise that has to do with: i) measure the cost/effectiveness of the application of session types in the software development process, in order to provide more accurate ; ii) research session type solutions for further needs of in the software development area, especially in the areas that need formal support in order to reduce complexity and effort.

References

1. From Data Types to Session Types: A Basis for Concurrency and Distribution. http://groups.inf.ed.ac.uk/abcd/.
2. K. Asanovic, J. Wawrzynek, D. Wessel, K. Yelick, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, and K. Sen. A view of the parallel computing landscape. *CACM*, 52, 2009.
3. 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: FMICS 2013*, chapter Study on the Barriers to the Industrial Adoption of Formal Methods, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
4. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP*, ESOP'12, pages 194–213. Springer-Verlag, 2012.
5. J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of mpi programs with intel message checker. In *SE-HPCS*, SE-HPCS '05, pages 78–82. ACM, 2005.
6. L. Fossati, R. Hu, and N. Yoshida. Multiparty session nets. In *TGC*, pages 112–127, 2014.
7. S. Fowler. Monitoring erlang/otp applications using multiparty session types. Master's thesis, University of Edinburgh, 2015.
8. 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.
9. R. Hu. Monitoring SMTP with Scribble and Java. Unpublished presentations, 2015.
10. 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.
11. 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.
12. S. M. Imam and V. Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *AGERE!*, pages 67–80. ACM, 2014.
13. J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232. ACM, 2015.
14. Mungo Webpage. http://www.dcs.gla.ac.uk/research/mungo/.
15. R. Neykova and N. Yoshida. Multiparty session actors. In *Coordination*, pages 131–146, 2014.
16. R. Neykova, N. Yoshida, and R. Hu. SPY: local verification of global protocols. In *RV*, pages 358–363, 2013.
17. N. Ng and N. Yoshida. Pabble: Parameterised scribble for parallel programming. In *PDP*, pages 707–714, 2014.

18. 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.
19. C. Ponsard, J.-C. Deprez, and R. Landtsheer. *FMICS*, chapter High-Level Guidance for Managers Deploying Formal Methods in Their Organisation, pages 139–153. Springer Berlin Heidelberg, 2013.
20. The Internet Engineering Task Force (IETF) official webpage. https://www.ietf.org/.
21. Scribble Project homepage. www.scribble.org.
22. Extended simple mail transfer protocol, RFC 5321. https://tools.ietf.org/html/rfc5321.
23. Simple mail transfer protocol, RFC 821. https://tools.ietf.org/html/rfc821.
24. S. Tasharofi, P. Dinges, and R. E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920, pages 302–326. Springer, 2013.
25. ABCD: Session Types Usecase Repository. https://github.com/epsrc-abcd/session-types-use-cases.
26. W3C Working group. http://www.w3.org/2002/ws/chor/.
27. P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

A Appendix: Use-Cases

A.1 Global Protocol for SMTP

```
global protocol SMTP(role S, role C) {
  _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; }
}
```

A.2 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.3 SMTP Client State Machine