Session Types for Software Engineering

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

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

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

1 Introduction

Session types are formal descriptions of communication protocols, which can be incorporated into programming languages and tools so that the technology of typechecking can be used to verify the dynamic structure of communication as well as the static structure of data. Since the introduction of session types by Honda et al. [5] more than twenty years ago, there has been a broad research effort to study their properties and apply their principles to a range of computational models and programming paradigms. Current research is converging towards the development of tools and technologies that use session types as part of the software development process. In order to take full advantage of the possibilities of session types, it is important to start a programme of technology transfer so that they can be integrated into broader approaches to software engineering.

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

are implemented within programming languages. Some programming languages are even standardised for specifying telecommunication systems, e.g. the set of languages standardised for telecommunication systems within the ISO ITU-T Z.100 series. In the telecommunication (complex system) domain, many of the requirements usually limit the use of formal methods. Furthermore, there is an unexplored area on how to balance the use of formal methods and the complexity in industrial software systems. Research in session types exhibits some promising results that can help explore and fill in this gap. In addition, the problem of weak adoption of formal methods in by the industry has been extensively studied. The main barriers recognised can be found in the lack of education, tool support and the cost of formal methods [4]. But even if these barriers are removed, there comes the importance of a well defined improvement strategy, that managers within a software organisation have to follow while adopting formal methods [12]. It is therefore necessary for a wider engagement of software engineers with formal method and formal method experts.

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

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

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

Summarising this discussion, we conclude that a wider audience would need to study a large series of partial results and at the same time filter out a huge

amount of unnecessary technical detail in order to achieve a satisfying level of understanding session types. This situation is an obstacle for the adoption of session types as part of practical software development.

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

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

Furthermore, the current research directions on session types are pushing towards a dialectic between session types and disciplines like software engineering, and system design and implementation. This is justified by the research experience gained by working on the project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (ABCD for short) [2]. Part of the requirement of the ABCD project is to compile a set of usecases for session types that can be used for current and future research on the design and development of frameworks and technologies. More concretely, the aims of this paper are to:

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

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

2 Session Types

The section introduces session types through a a discussion about the need of session types in software engineering. It also presents the Scribble [13] protocol

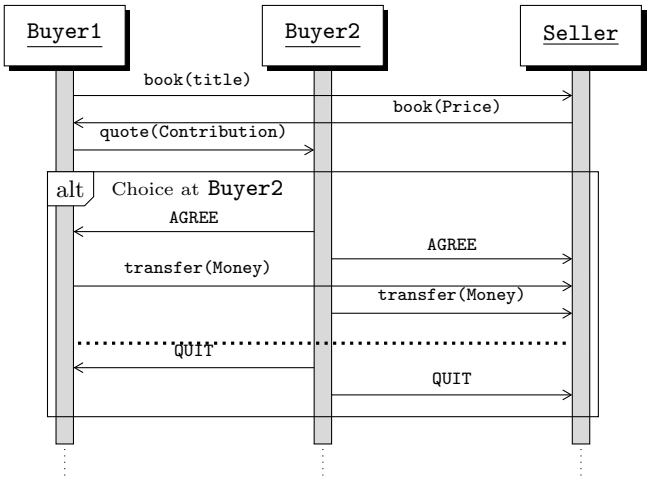

Fig. 1. The Book-store protocol: Sequence Diagram

description language which is a formal language for describing protocols using session type principles. The overall discussion in this section identifies and sets a clear and distinctive terminology needed to for transferring to a wider audience the basic ideas 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 charts (see Figure 1). Sequence charts are used to depict communication flow as ordered sequence of communication *message passing* among communication participants that represent particular *roles* in communication protocol.

In Figure 1, the case of the Bookstore protocol, which is a usecase in the ABCD online repository [1], is presented. 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 expenses. In the sequence chart, each role is represented with vertical line and communication messages are represented with horizontal directed arrows indicating the direction of communication flow. Intuitively, the communication progress is proceeds downwards. Thus, visually the protocol behaviour is represented within the chart. However, sequence charts are not considered as formal tools because such described protocols can not be verified as being complete or unambiguous.

For example, in industrial practice, the communication protocols for distributed software, are usually described in natural language and discussed with help of the sequence charts. With the help of sequence charts, each software module that participates in the protocol communication implements its *local* software role within the protocol. Usually the process of implementing the lo-

cal implementation impose numerous possible solutions that are often hidden and omitted from sequence chart. Another limitation of sequence charts, is the inability to deal with complex communication protocols where complex communication interactions that involve multiple parties become hard to follow and reason.

what is hidden? - overall structure of the protocol, logic on choices

A solution to manage communication complexity is the introduction of *structure* within the communication protocol. Session types describes communication in a structured and rigorous fashion, and is aiming to define concepts for structured communication that should enable better mastering of communication complexity.

Typically, a type is broadly understood as a meta-information concept, that is used describe a class of a data. Structures of 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 a communication behaviour in terms of a series of send and receive interactions between a set of communicating entities and it is used to characterise data structures that are used to implement communication.*

2.2 Sessions as Protocols: The Scribble Protocol Description Language

The Scribble tool-chain [13] is a tool that indents to serve as a core element in the process of integrating the practical aspects of session types in the software design and development process. The basic module of the Scribble tool-chain is the Scribble protocol description language, which is a syntax that expresses session type specifications. The design of the Scribble language draws directly from the principles of session types. For the sake of example consider part of the session type protocol for the Book-store usecase as in Figure 2. This section defines the main terminology for session types and at the same time clarifies the notions through the presentation of the Scribble language.

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 the 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 type value, i.e. a structure that implements communication, is passed as a message. Choice requires for a role to use a value called *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 com-

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

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

Fig. 2. Scribble: Session Type Protocol for the Bookstore Usecase

munication and among all possible cases that may happen, in contrast to the limiting structure of a sequence diagram.

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

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

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

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

In line 5 of the protocol their the usage of a choice interaction, where role `Buyer2` makes an internal decision and selects out a two possible outcomes, 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.

describe the recursion construct?

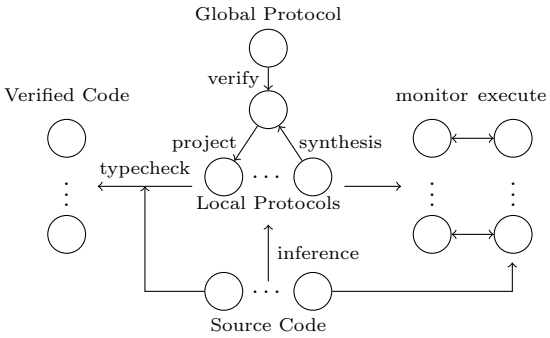

Fig. 3. Use of Session Types in Software Development

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

2.3 Integrating Session Types in Software Development: The Scribble Tool-chain

Apart from a protocol description language, Scribble offers a tool-chain that integrates session types in the software development process.

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

In Figure 3 circles represent source code for either the software under development or for the definition of the session type protocols. 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 is developed using a programming language that makes use of a communication interface that can be automatically identified.

In a typical top-down approach, a developer will first develop a global protocol and express it syntactic form. 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 correctness . After the global protocol verification, a projection procedure takes place where a local protocol for each global protocol role is derived. Verification and projection are supported by the Scribble tool-chain.

what does it verify?

One way to use the set of local protocols is to ensure conformance between the source code of the software under development and the local protocols, by

using a type-checker procedure. The type-checker takes as an input a software module, identifies its communication operations, and derives the communication behaviour of the module. The type-checker also takes as an input the corresponding local protocol and checks whether the software module implements the corresponding role communication. Upon success, the result is verified software that enjoys the session types *safety* and *progress* properties.

Another way of using the set of local protocol is for monitoring the execution of a concurrent set of communicating software modules. The local protocols are used to feed a set of local monitors. Essentially, the local monitors run a state machine that is derived out of the local protocol syntax. The monitor platform also requires for the concurrent software execution. The assumption to be made here is that monitor platform implements the communication interface on top of the monitoring interface. Scribble does support the procedure of deriving state machines out of local protocol syntax.

In Figure 3 there is the bottom-up approach as well, were the communicating modules of a concurrent software are developed independently. Then an inference procedure identifies the communication behaviour of each module that results in a set of local protocols. Then the synthesis procedure takes place where the derived local protocols are synthesised to create a global protocol. Upon success, the developed software is guaranteed to ensure the desirable session type properties.

The case where the local protocols of all the roles in a global protocol are implemented by the corresponding communicating modules of a system is called session fidelity. When session fidelity is ensured then a number of communication properties can be guaranteed for the concurrent software:

- Every execution state of the system has *safe* communication behaviour:
  - *Communication operator matching*: Every send (resp., select) operation has a corresponding receive (resp., branch) operation.
  - *Message type matching*: The type of every message being send matches the type of the message expected to be receive.
  - *Deadlock-freedom*: Consequently, every message send will be eventually received.
- Every state of the protocol will *progress* to a safe state.

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

2.4 Usecase Classification on Technologies used

Just presentation and references

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

1. Session Java [7] is the first tool developed as a tool with a library that support binary session types. The tool requires compiler support to perform session type-check.
2. The Eventful Session Java [6] is an extension of session types to support asynchronous, event-driven programming.
3. Multiparty Session C [11] is the first tool that implemented a multiparty session type library for MPI. The tool requires compiler support to perform session type-check.
4. Session fidelity can also be ensured using runtime monitoring. Modules for monitoring were implemented in python and are presented in [9].
5. Session types are embedded in the Actors paradigm in the following technologies:
   - An implementation that uses the python monitoring module for session types to monitor python threads that simulate actors [8].
   - An similar approach on monitoring session types is also used for the Erlang programming language, which is a programming language for the Actors model.
6. Pabble [10] is an extension to Scribble used to express communication structures that have a parametrised number of roles. Pabble protocols are implemented and type-checked for the MPI framework.
7. Mungo [3] is a tool that integrates session types in the object-oriented programming paradigm through the typestate structure; communication interactions can be seen as an object interface with a behaviour. Mungo uses a protocol description language for capturing typestate. The tool Scribble to Mungo (StMungo) is used to transform Scribble protocols into Mungo protocols.
8. GV: A functional programming implementation of binary session types.
9. LINKS: Functional programming based on the linear logic interpretation of session types.
10. SILL: Functional programming based on the linear dual intuitionistic interpretation of session types.
11. A state pattern implementation as a library. Uses Scribble to automatically create an API for state pattern programming and uses runtime checks to cope with linearity requirements.

2.5 Fine/coarse grain representation

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

this should go as future work

3 Usecases

This section will give a summary of the usecases in the ABCD online repository [1]. At the same time it will give a more detail presentation on how session types are used in the implementation of the network application usecase for a

book-store, which is a standard example for introducing session types. It will also focus on the implementation of a session type Simple Mail Transfer Protocol, which is a standard network protocol.

A table summary for the usecases in the repository can be found in Figure 4.

3.1 Domain Classification of Usecases

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

1. *Network Application/Business Logic.* Session types can be used to develop protocols for applications that run inside a network. A protocol given in a session type structure, apart from the specification of the communication of the application, will reveal a kind of business logic for the application.
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.
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.
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.
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.
6. *Hardware.* Hardware mechanisms complete the stratification of domains. The communication of hardware modules may also be expressed using session types.

Security

7. . Session types can also find applications in the security domain, which is a domain that supports all other domains in the above list.

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/.

	Usecase	Domain	Technologies/Tools	Description
1.	Book Store	Business Application	Mungo, Session Java	Interaction Logic
2.	Chat Server	Network Application	Erlang	Application Logic
3.	Hyper-Text Transfer Protocol	Network Protocol		Request / Response protocol
4.	Simple Mail Transfer Protocol	Network Protocol	Mungo/StMungo, Session Java, LINKS	Stateful protocol
5.	Domain Name System	Network Protocol	Erlang	
6.	Concurrency • Dinning Philosophers • Sleeping Barber • Cigarette Smokers	Operating Systems	Python/Actors	Race conditions
7.	Lock	Operating Systems	Eventful Session Java	Race conditions
8.	Collection	Data Structures	Mungo	A stack client
9.	File Access	Data Structures	Mungo	File Access client
10.	Concurrent Fibonacci	Concurrent Algorithms	Mungo	
11.	Network Topologies • Ring • Butterfly • All to All • Stencil • Farm (Master-Worker) • Map Reduce	Parallel Algorithms	Pabble and MPI	Parametric algorithms
12.	Concurrent Algorithms • Peano Numbers • Add Server	Concurrent Algorithms	Links, GV	
13.	Memory Coherence	Systems, Hardware	Mungo	

Fig. 4. Usecases in the ABCD online repository

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. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. Scribble Project homepage. www.scribble.org.