Behavioural Types for Heterogeneous Systems (Position Paper)

Simon Fowler
University of Glasgow, UK

Philipp Haller
Digital Futures, KTH Royal Institute of Technology, SE

Roland Kuhn
Actyx AG, DE

Sam Lindley
The University of Edinburgh, UK

Alceste Scalas
Technical University of Denmark, DK

Vasco T. Vasconcelos
University of Lisbon, PT

Behavioural types provide a promising way to achieve lightweight, language-integrated verification for communication-centric software. However, a large barrier to the adoption of behavioural types is that the current state of the art expects software to be written using *the same* tools and typing discipline throughout a system, and has little support for components over which a developer has no control.

This position paper describes the outcomes of a working group discussion at Dagstuhl Seminar 24051 (Next-Generation Protocols for Heterogeneous Systems). We propose a methodology for integrating multiple behaviourally-typed components, written in different languages. Our proposed approach involves an *extensible protocol description language*, a *session IR* that can describe data transformations and boundary monitoring and which can be compiled into program-specific *session proxies*, and finally a *session middleware* to aid session establishment.

We hope that this position paper will stimulate discussion on one of the most pressing challenges facing the widespread adoption of behavioural typing.

1 Introduction

Behavioural types provide a powerful and lightweight mechanism for language-integrated verification of behavioural properties: whereas traditional data types rule out errors such as adding an integer to a string, behavioural types can rule out behavioural errors such as forgetting to close a file handle or sending an invalid message on a communication channel.

Session types [20, 21] are a behavioural typing discipline for checking adherence to communication protocols: if a process is typed according to its session type, then it is guaranteed to fulfil its role in the communication protocol at runtime. Although originally designed for two communicating participants, work on *multiparty* session types (MPSTs) [22] extends session typing to handle systems with multiple components. If all components are either derived from a well-formed global type, or all components have compatible local types, then the entire system should not encounter any communication errors at runtime. Many MPST disciplines include further guarantees such as global progress or liveness.

Extensive research has gone into behavioural types, in particular session types, over the years. A recent workshop celebrated 30 years of session types [2], and a recent book concentrates on how decades of research into behavioural types has given rise to a plethora of tools [18]. Many international programming languages conferences typically have sessions dedicated to behavioural type systems.

Nevertheless, and notwithstanding efforts to overcome barriers to practical adoption (e.g., work on failure handling [7, 17], graphical user interfaces [16, 32], and subsessions [13]), behavioural typing has not yet seen widespread industrial use. Arguably *the* key issue facing the behavioural types community[1] is

[1]And indeed, other neighbouring communities such as choreographic programming [33], although we concentrate on behavioural types in this paper.

the inability for behavioural types to work satisfactorily with *heterogeneous* systems, consisting of software components developed in different languages, using different tools, with different typing guarantees.

Heterogeneity may arise for practical reasons: for example, we may want to write some components in a given language due to better library support (e.g., using Python due to its rich support for data science), or because it is important to obtain stronger static guarantees for a particular participant or portion of a protocol. Heterogeneity may even arise in a single program, for example writing parts of an application in different programming languages (e.g., writing performance-critical code in a systems language such as C++ or Rust, and the remainder of the program in a managed language like Python). Furthermore, in any realistic setting, we would have to assume that some components are implemented using different languages or accessible only through an API (for example as is common with microservices). Similarly, we may have existing components that offer *similar* services that do not quite correspond to the expected types. We concentrate on the following scenario:

> **Scenario: Travel Booking System.** We want to design a travel booking system that includes both timing constraints and data refinements. A travel booking session consists of a client, travel agent, and flight provider; we are in control of the agent and client, but the provider is developed by an external company and accessible via an API. The client initiates a search, and receives a set of suitable flights. After receiving the results, the client has 6 minutes to select a flight before the results expire.
>
> When the client selects a flight, it receives a token, and should send the same token to the provider in order to continue the booking. Finally, the provider sends the client either a booking confirmation, or an error message.

Crucially, we will consider a heterogeneous version of this scenario in which the client, travel agent, and flight provider are implemented using different tools with quite different capabilities.

Paper structure. The paper proceeds as follows. Section 2 gives relevant background. Section 3 describes a motivating scenario of a travel agent application written in different tools with differing language features. Section 4 describes our proposed solution and speculates on several potential research challenges. Section 5 discusses related work, and Section 6 concludes.

2 Background

(Multiparty) Session Types. Session types are types for protocols: whereas a data type describes the shape of some data, ruling out errors such as adding an integer to a string, a session type describes both the type and direction of data to be communicated between participants [20, 21]. Session types were originally investigated in the *binary* setting between two participants, but later work on *multiparty* session types [22] describes communication between multiple communicating participants. We concentrate on multiparty session typing in the remainder of the paper.

The following example describes the classic *two-buyer* protocol where two participants collaborate in order to buy an expensive item (usually a book). `Buyer1` begins by sending the title to the `Seller`, who responds with a quote. `Buyer1` then sends the quote to `Buyer2`, who decides whether to accept the quote by sending their address to the `Seller` and subsequently receiving a delivery data, or declining the offer.

The *global type* describing all interactions in the system is described on the left. Global types can then be *projected* into *local types* for each participant; it is then possible to typecheck or monitor all participants against their local types. The local type for `Buyer2` is shown on the right.

$$\begin{array}{l}
\texttt{Buyer1} \to \texttt{Seller} : \mathsf{title}(\mathsf{String})\,.\\
\texttt{Seller} \to \texttt{Buyer1} : \mathsf{quote}(\mathsf{Int})\,.\\
\texttt{Buyer1} \to \texttt{Buyer2} : \mathsf{share}(\mathsf{Int})\,.\\
\texttt{Buyer2} \to \texttt{Seller} : \{\\
\quad \mathsf{address}(\mathsf{String})\,.\\
\qquad \texttt{Seller} \to \texttt{Buyer2} : \mathsf{date}(\mathsf{Date})\,.\,\mathsf{end},\\
\quad \mathsf{quit}(\mathsf{Unit})\,.\,\mathsf{end}\\
\}
\end{array}
\qquad
\begin{array}{l}
\mathsf{Buyer2} \triangleq \texttt{Buyer1}\,\&\,\mathsf{share}(\mathsf{Int})\,.\\
\qquad \texttt{Seller} \oplus \{\\
\qquad\quad \mathsf{address}(\mathsf{String})\,.\\
\qquad\qquad \texttt{Seller}\,\&\,\mathsf{date}(\mathsf{Date})\,.\,\mathsf{end},\\
\qquad\quad \mathsf{quit}(\mathsf{Unit})\,.\,\mathsf{end}\\
\qquad \}
\end{array}$$

A multitude of tools have been developed for checking against multiparty session types, for example in Java [28], Scala [40], Rust [12, 29], and F# [37]. However, each tool embeds the assumption that the entire system is written using that same tool; the possibility of combining heterogeneous components written using different tools and programming languages is not part of the tools' specification.

Runtime Monitoring against Session Types. Although the original work on session typing envisaged static checking, a correspondence between MPSTs and communicating automata [14] showed how it was possible to *monitor* processes against a session type, allowing a degree of runtime verification. The key idea is to translate a local type into a finite state machine where transition corresponds to a send or receive action; for each session endpoint, a monitor process observes incoming and outgoing messages — and upon receiving an invalid message, the monitor drops it and/or reports a violation. We describe runtime monitoring against session types in more depth in Section 5.

Multi-language Interoperability. There has been increasing attention given to semantic foundations for multi-language interoperability, for example through foreign function interfaces (FFIs). A major inspiration for our proposed solution is the approach of Patterson et al. [38] who introduce a methodology for interoperability by defining a common intermediate representation along with *convertibility relations* and *boundary conversions* and show how to verify semantic soundness using logical relations.

Our goal is to adopt an analogous methodology but for the world of message passing as opposed to shared memory. Rather than challenges such as linking and foreign function interfaces, our challenge is to describe the "glue" that can allow a program written in Go, for example, to safely interact with a program written in Java or Rust, while keeping as many of the guarantees that we would expect if a program was written in a single behaviourally-typed language.

3 Heterogeneous Multiparty Session Typing

We can start by writing an idealised global type (omitting some irrelevant timing constraints):

$$\begin{array}{l}
\texttt{Customer} \to \texttt{Agent} : \mathsf{search}(\mathit{origin} : \mathsf{AirportName}, \mathit{destination} : \mathsf{AirportName})\,.\\
\texttt{Agent} \to \texttt{Customer} : \mathsf{results}(\mathit{searchResults} : [(\mathsf{FlightNum}, \mathsf{Time}, \mathsf{Price}, \mathsf{Provider})])\,.\\
\texttt{Customer} \to \texttt{Agent} : \{\\
\quad \mathsf{select}\{t \leq 360\}(\mathit{flightNum} : \mathsf{FlightNum}) \mapsto\\
\qquad \texttt{Agent} \to \texttt{Customer} : \mathsf{providerRef}(\mathit{ref} : \mathsf{ProviderRef})\,.\\
\qquad \texttt{Customer} \to \texttt{Provider} : \mathsf{book}\{\mathit{token} == \mathit{ref}\}(\mathit{token} : \mathsf{ProviderRef}, \mathit{details} : \mathsf{PaymentDetails})\,.\\
\qquad \texttt{Provider} \to \texttt{Customer} : \{\\
\qquad\quad \mathsf{ok}() \mapsto \mathsf{end},\\
\qquad\quad \mathsf{error}() \mapsto \mathsf{end}\\
\qquad \},\\
\quad \mathsf{timeout}\{t > 360\}() \mapsto\\
\qquad \texttt{Customer} \to \texttt{Provider} : \mathsf{timeout}()\,.\,\mathsf{end}\\
\}
\end{array}$$

Note that the clock at the customer can only send a select message within 360 seconds, and otherwise must send a timeout message. Similarly, the data refinement on the book message ensures that the *same* reference is sent to the provider as is received from the agent. Say that our system consisted of:

- The `Agent` in Python using the time-aware framework proposed by Neykova et al. [36]
- The `Customer` in F⋆ using SESSION⋆ by Zhou et al. [46] to statically verify the data refinement
- The `Provider`, developed by a different company and accessible only through an API

In the above, the `Agent` has inbuilt verification of timing constraints, the `Customer` has static verification of data refinements, and the `Provider` does not even have verification of communication patterns. There are several research problems posed by this scenario:

Extensibility. At present there are different, incompatible, *dialects* of session types for each extension.

Precision mismatch. However, each tool available to implement a constraint (Python for timing constraints; SESSION⋆ for data refinement) only works in a single language. Thus, some checks must take place at runtime using boundary monitors.

Message rejection. Existing work on monitoring against multiparty session types takes a *suppression-based* approach to monitoring: a monitor will drop any non-conforming messages that it receives (either silently, or reporting a violation). Although suppression maintains safety (by stopping any non-conforming messages from being processed by the program logic), dropping a message may mean that the remaining actions in the protocol cannot be fulfilled—thus breaking liveness.

As well as research questions, there are also some more practical issues:

Session Initiation. The session needs to be *established*, which involves discovering each component, inviting it to the session, and setting up the communication infrastructure.

Wire format. Communication needs to occur over a *standard message layout*. Most session typing systems either do not include any wire communication, or communicate using non-standard message layouts that vary per tool.

4 Proposed Solution

Figure 1 gives an overview of our proposed approach. The overall idea is:

- A developer designs the protocol in an *extensible, language-agnostic protocol description language*
- The protocol is projected and compiled down into a program-specific *session IR*, whose purpose is to describe any necessary dynamic checks, message re-orderings, and data transformations
- The session IR is used to generate *session proxies* that act as adapters to each program

Extensible Protocol Description Language The first step is to write a protocol in a language-agnostic protocol description language. The *Scribble* protocol description language [44, 45] provides a good starting point, but the key point is that the language should be extensible in a modular way by supporting *plugins* supporting individual language features (e.g., value-dependency or timeouts).

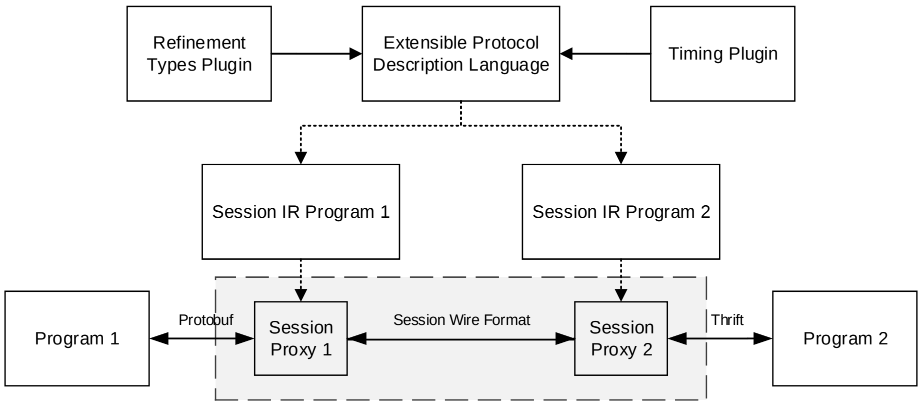

Figure 1: Proposed System

Session IR In traditional multiparty session programming, a global protocol description is projected as a *local type* for each participant. The local type serves as a language-agnostic *specification* for the communication actions that the participants should perform, and can be used for static type checking.

In addition to local types for static checking, we propose a *session intermediate representation* (Session IR). Whereas local types are meant to be implementation-agnostic, the Session IR is used to describe any monitoring or message transformations required in order to integrate a component with the system. In particular, we envisage the session IR being able to support at least the following:

- **Boundary Monitoring** To address the precision mismatches between the static checking supported by the tool, as well as maintaining timing properties, a core role of each session proxy is to perform boundary monitoring [8, 11] before an incoming message is delivered (and in some circumstances before an outgoing message is committed to the system). Monitor violations may result in *suppression* (i.e., dropping messages that are violated), but may also allow violations to be reported to the program in order to allow compensatory actions (e.g., raising an exception or requesting that the sender provides a revised message).
- **Message Insertion / Reordering** It may be that a component supports a compatible variation of its role in a protocol, but does not match the protocol exactly (for example, due to a version upgrade). In this case, the session IR can describe message reorderings (inspired by theoretical work on session type isomorphisms [5, 15]), or insertions of messages at a given point. This would enable migrating components to new versions with a different but compatible behaviour.
- **Wire Formatting** Each program will have its own expected communication mechanism (be that sockets, protocol buffers [19], or interface description languages like Thrift [3]). The final job of the session IR will be to describe transformations between the internal protocol representation and the wire protocol.

For trusted components that are statically checked to follow the protocol, the session proxy will only need to perform wire formatting and monitoring of incoming messages.

Session Proxies. Each session IR program can then be compiled into a *session proxy* to interact with each program. The session proxy acts as an adapter between the program and the other participants in the session. Interaction between session proxies happens using a standardised session wire format and communication medium.

There are several ways by which session proxies could be integrated with each application. We expect that for basic suppression-based monitors, it would suffice to leave the original application untouched and instead route all communication through the proxy (indeed, this would also support a level of session typing for components that are not programmed with session types). For more involved session proxies such as those that raise an application-level exception when a message violation occurs, it would likely be necessary to either adapt the current tooling to incorporate the session proxy or provide an API that a developer can code against.

Language Features. Another concern is the features that must be available to each language or tool in order for it to support any operations required by each session proxy. For suppression-based monitors, it is unlikely that any additional language features would be needed. If we would like to report any monitor violations to the program, however, then we would likely need some additional language support: for example, exception handling [17] in the case of needing to deal with linear resources, or additional failure handling callbacks in the case of a tool based around inversion-of-control [46].

Session Establishment. Figure 1 does not describe how sessions are established. We envisage a middleware application, similar to that described by Atzei et al. [6], is a potential solution: such a system would allow participants to register to take part in a session, discover other participants, and finally allow sessions to be established. Alternatively, session establishment could happen directly (as is done, for example, with explicit connection actions [24]).

4.1 Potential Challenges

Although we believe our proposed solution offers a promising framework for future research on heterogeneous session typing, we envisage several challenges, at least including the following:

Interactions between extensions. Our scenario considers two fairly orthogonal extensions: data refinements and timing. However, there could be other extensions (e.g., explicit connection actions [24] that require a more liberal syntax) that could pose challenges when combined with existing disciplines. How can we ensure that the extensible language is sufficiently general to both mediate between the different dialects of session typing, and how can we ensure that their combination does not lead to safety errors?

Formal guarantees. It is important to understand the desired guarantees to be given by the system. It seems reasonable to expect, at a minimum, that the system will ensure session fidelity (i.e., that every message that is exchanged will conform to the given protocol). Nevertheless, ensuring properties such as liveness is more challenging in a monitored setting. A further open challenge would be reasoning about the metatheory in a modular way.

Generality of the IR. The Session IR will at least need to be able to describe monitoring, message reordering, and message reformatting. However, there is a large design space and there are inherent trade-offs to ensuring the IR design remains sufficiently high-level while also sufficiently general to support the array of possible extensions.

Performance. Any runtime checking and message rewriting will inevitably incur a runtime overhead, so it will be necessary to ensure that any overhead is not prohibitive. This can be mitigated to an extent by only checking properties that are not guaranteed statically, and ensuring that the monitor is located on the same machine as the monitored process. In addition to runtime overheads, it is possible that monitors may need to record some message history, for example to enforce dependent type constraints [41]. It is therefore necessary to ensure that any generated monitor does not require unbounded space.

Location of error reporting. In addition to the language design challenges in allowing applications to handle any errors, it is possible that there are multiple places to report a violation. Some notion of blame [43] is likely to be important, but defining “more” or “less” typed is likely to be challenging in the presence of multiple extensions.

5 Related Work

Protocol description languages. MPSTs were designed without a particular implementation in mind. A good starting point for heterogeneity is the language-agnostic Scribble protocol description language [35, 44] for describing MPST specifications; implementations of Scribble (e.g., [45]) support well-formedness checking, projection, and monitor generation. A necessary step in the pursuit of heterogeneity would be to generalise a language like Scribble to allow modular and composable extensions with different language features, as opposed to the current status quo of *ad-hoc* extensions for each new feature.

Monitoring against MPSTs. Most closely relevant is work on runtime monitoring against local types [8, 11], based on the correspondence between multiparty session types and communicating automata [14]. The core idea is to translate each local type into an FSM and check each incoming and outgoing message against the monitor, rejecting the message if the message does not match any transition. In particular, Bocchi et al. [8] show *safety* (monitors ensure that ill-behaved participants do not send messages that violate their specification) and *transparency* (monitors do not affect the behaviour of well-behaved components). However, these monitors discard non-conforming messages without any feedback to the sender or receiver. In turn, this means that (especially for non-recursive protocols), non-conforming messages cause the protocol to silently stop. Later work by van den Heuvel et al. [42] addresses the black-box monitoring of multiparty sessions through monitoring processes that are directly generated from global types, and (unlike Bocchi et al. [8]) actively report violations by stopping execution. Burlò et al. [9] proposes a prototype implementation of a “hybrid” verification approach where session-typed components can interoperate with heterogeneous (possibly untyped) components through autogenerated monitors — which, in turn, are session-typed (only for two-party sessions), and can translate messages between different wire formats, and suppress and report protocol-violating messages; Burlò et al. [10] later studies the properties of such black-box monitors in terms of soundness and completeness of violation reports. In contrast to all the works on session monitoring listed above, we believe that it may be necessary to forego monitor transparency and instead allow compensatory behaviours upon a monitor violation (for example, raising an exception or requiring the sender to send a revised message).

Session types and heterogeneity. Only very little work has attempted to address heterogeneous session typing. Jongmans and Proença [27] describe the design of a system called ST4MP that aims to support multi-lingual programming through the established API generation approach [23]; the idea is to generate multiple compatible APIs for different languages from a given global type specification. In contrast to this position paper, ST4MP supports a base multiparty session typing discipline without any advanced language features (e.g., timing or refinement types), and does not make use of any form of runtime checking. In contrast we would expect our general session IR to be able to support even untyped components.

Language and system interoperability. Our proposal to use a common Session IR to provide safety properties even when composing heterogeneous components written in different languages is similar in spirit to recent work on sound language interoperability [38, 39]. While previous work only targets languages interoperating via shared memory, our proposal specifically aims to address interoperability for typed message-passing concurrency. Gradual session types [25] provide a framework for ensuring type and communication safety for programs integrating statically-typed sessions and dynamic types. It is assumed that programs share a common internal language with casts. Our proposal aims to decouple components even further by mediating communication via Session IR proxies which may, in addition to casts, insert or reorder messages and perform other forms of monitoring. Session IR and Session IR proxies are related to IDLs used for integrating distributed components [30] as well as systems for business-to-business interactions [31] which explicitly aim to address heterogeneity.

At a more abstract level, the ideas presented in this position paper can be related to another position paper by Albert *et al.* [4] that advocates a formal language for service-level agreements (SLAs) for (virtualised and heterogeneous) distributed services, to be enforced via e.g. static verification and/or runtime monitoring, possibly aided by code generation from executable models written in ABS [26]. Our approach is focused on behavioural types (which could be seen both as a form of formalised SLA) and as a form of executable specification (usable e.g. for monitor generation via the session IR).

Jolie [1, 34] is a service-oriented programming language. Programs can either be written in Jolie, or Jolie can serve as an interface to services written in a different programming language. Jolie programs can also serve as *orchestrators* to interact with multiple other services, potentially using different transport mechanisms. In contrast, our proposal takes a more protocol-centric approach: instead of specifying the services and an orchestration-based method of allowing them to interact, our proposal instead involves concentrates on boundary monitoring and manipulation of existing communication flows. An advantage is that we can (potentially statically) verify fine-grained data and timing constraints.

6 Conclusion

Although behavioural types offer a strong foundation for lightweight, language-integrated verification of behavioural properties, a large barrier to their adoption is that at present an *entire system* must be written in a single language. In this position paper we have described a potential line of work that, if completed successfully, could allow behavioural types in *heterogeneous* software systems where components can be written in different languages, using different tools, each of which support different static guarantees. Our approach relies on an *extensible* protocol description language that can support additional language features (e.g., timing or refinement types) as plugins, and a *session IR* that can describe transformations on data (e.g., wire formatting, message reordering, and boundary monitoring). Structured support for heterogeneity can greatly expand the reach of behavioural types in real-world systems, and we hope that these initial ideas serve as a starting point for addressing this challenging research topic.

Acknowledgements

We thank the organisers of Dagstuhl Seminar 24051 and Schloss Dagstuhl — Leibniz Center for Informatics for making this work possible. We are also grateful to the anonymous reviewers for their detailed and encouraging reviews. This work was supported by EPSRC grant EP/T014628/1 (STARDUST), Horizon Europe grant 101093006 (TaRDIS), Independent Research Fund Denmark RP-1 grant "Hyben", Digital Futures Research Pairs Consolidator Project "PORTALS", and UKRI Future Leaders Fellowship MR/T043830/1 (EHOP), FCT grant PTDC/CCI-COM/6453/2020 (SafeSessions), and the LASIGE Research Unit.

References

[1] Jolie programming language — official website. URL `https://www.jolie-lang.org/`. Accessed on 25/03/2024.

[2] ST30: 30 years of session types — workshop website. URL `https://2023.splashcon.org/home/st-anniversary-30`. Accessed on 11/01/2024.

[3] Apache Thrift - official website. URL `https://protobuf.dev/`. Accessed on 11/01/2024.

[4] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud '13, page 59–63. Association for Computing Machinery, 2013. doi: 10.1145/2513534.2513545.

[5] Assel Altayeva and Nobuko Yoshida. Service equivalence via multiparty session type isomorphisms. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 1–11, 2019. doi: 10.4204/eptcs.291.1.

[6] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Contract-oriented programming with timed session types. *Behavioural Types: from Theory to Tools*, page 27, 2017. doi: 10.1201/9781003337331-2.

[7] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised multiparty session types with crash-stop failures. In *CONCUR*, volume 243 of *LIPIcs*, pages 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.CONCUR.2022.35.

[8] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi: 10.1016/j.tcs.2017.02.009.

[9] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. Towards a hybrid verification methodology for communication protocols (short paper). In *FORTE*, volume 12136 of *Lecture Notes in Computer Science*, pages 227–235. Springer, 2020. doi: 10.1007/978-3-030-50086-3_13.

[10] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. On the monitorability of session types, in theory and practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 20:1–20:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPICS.ECOOP.2021.20.

[11] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011. doi: 10.1007/978-3-642-30065-3_2.

[12] Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In *PPoPP*, pages 246–261. ACM, 2022. doi: 10.1145/3503221.3508404.

[13] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi: 10.1007/978-3-642-32940-1_20.

[14] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi: 10.1007/978-3-642-28869-2_10.

[15] Mariangiola Dezani-Ciancaglini, Luca Padovani, and Jovanka Pantovic. Session type isomorphisms. In *PLACES*, volume 155 of *EPTCS*, pages 61–71, 2014. doi: 10.4204/eptcs.155.9.

[16] Simon Fowler. Model-view-update-communicate: Session types meet the Elm architecture. In *ECOOP*, volume 166 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.14.

[17] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi: 10.1145/3291617.

[18] Simon Gay and António Ravara. *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

[19] Google. Protocol buffers documentation. URL `https://protobuf.dev/`. Accessed on 11/01/2024.

[20] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35.

[21] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008. doi: 10.1145/1328438.1328472.

[23] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016. doi: 10.1007/978-3-662-49665-7_24.

[24] Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017. doi: 10.1007/978-3-662-54494-5_7.

[25] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi: 10.1017/S0956796819000169. URL `https://doi.org/10.1017/S0956796819000169`.

[26] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. doi: 10.1007/978-3-642-25271-6_8.

[27] Sung-Shik Jongmans and José Proença. ST4MP: A blueprint of multiparty session typing for multilingual programming. In *ISoLA (1)*, volume 13701 of *Lecture Notes in Computer Science*, pages 460–478. Springer, 2022. doi: 10.1007/978-3-031-19849-6_26.

[28] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for Java. *Sci. Comput. Program.*, 155:52–75, 2018. doi: 10.1016/j.scico.2017.10.006.

[29] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine Rust programming with multiparty session types. In *ECOOP*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ECOOP.2022.4.

[30] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Comput. Surv.*, 30(1):3–27, 1998. doi: 10.1145/274440.274441. URL `https://doi.org/10.1145/274440.274441`.

[31] Brahim Medjahed, Boualem Benatallah, Athman Bouguettaya, Anne H. H. Ngu, and Ahmed K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *VLDB J.*, 12(1):59–85, 2003. doi: 10.1007/S00778-003-0087-Z. URL `https://doi.org/10.1007/s00778-003-0087-z`.

[32] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC*, pages 94–106. ACM, 2021. doi: 10.1145/3446804.3446854.

[33] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. doi: 10.1017/9781108981491.

[34] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014. doi: 10.1007/978-1-4614-7518-7_4.

[35] Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In *Models, Languages, and Tools for Concurrent and Distributed Programming*, volume 11665 of *Lecture Notes in Computer Science*, pages 236–259. Springer, 2019. doi: 10.1007/978-3-030-21485-2_14.

[36] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017. doi: 10.1007/s00165-017-0420-8.

[37] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018. doi: 10.1145/3178372.3179495.

[38] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic soundness for language interoperability. In *PLDI*, pages 609–624. ACM, 2022. doi: 10.1145/3519939.3523703.

[39] Daniel Patterson, Andrew Wagner, and Amal Ahmed. Semantic encapsulation using linking types. In *TyDe@ICFP*, pages 14–28. ACM, 2023. doi: 10.1145/3609027.3609405.

[40] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPICS.ECOOP.2017.24. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2017.24`.

[41] Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.*, 90:61–83, 2017. doi: 10.1016/j.jlamp.2016.11.005.

[42] Bas van den Heuvel, Jorge A. Pérez, and Rares A. Dobre. Monitoring blackbox implementations of multiparty session protocols. In *RV*, volume 14245 of *Lecture Notes in Computer Science*, pages 66–85. Springer, 2023. doi: 10.1007/978-3-031-44267-4_4.

[43] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. doi: 10.1007/978-3-642-00590-9_1.

[44] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi: 10.1007/978-3-319-05119-2_3.

[45] Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *FCT*, volume 12867 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2021. doi: 10.1007/978-3-030-86593-1_2.

[46] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020. doi: 10.1145/3428216.