Event-Driven Multiparty Session Actors

Simon Fowler
simon.fowler@glasgow.ac.uk
University of Glasgow
United Kingdom

Raymond Hu
r.hu@qmul.ac.uk
Queen Mary University of London
United Kingdom

Abstract

Actor languages such as Erlang and Elixir have emerged as popular tools for designing reliable, fault-tolerant distributed applications, but communication patterns used by actors are often informally specified. Multiparty session types (MPSTs) are a type discipline for communication protocols: if a program typechecks according to its session type, then it is guaranteed to fulfil its role in a communication protocol, but the unidirectional communication mechanism used by actors makes it difficult to apply session types to actor languages directly. By combining a flow-sensitive effect system with an event-driven programming model, we show the first statically-typed session type system for actors that can participate in multiple sessions.

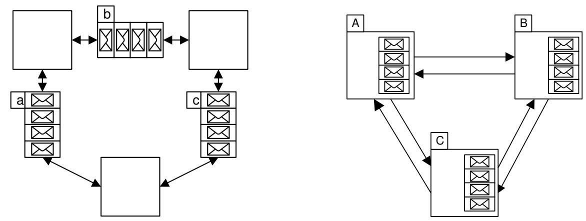
1 Introduction

Actor languages and frameworks are a mainstay of reliable distributed software development: an actor is an addressable process that can spawn and send messages to other actors, and react to incoming messages. Actor languages support powerful idioms such as *supervision hierarchies* [1], which have helped companies such as Ericsson develop systems with "nine nines" reliability; Erlang is also used as the backbone of WhatsApp, with billions of users worldwide.

Alas, message passing introduces the threats of deadlocks and communication mismatches. *Multiparty session types* [9] encode communication patterns as types; successful typechecking ensures communication safety. To date, session typing for actors has either been checked *dynamically* [14], or has restricted each actor to a single session [7]. Restricting an actor to a single session is disadvantageous as it is often useful for an actor to have some common state (*e.g.*, the stock of a warehouse) common to multiple sessions.

> **Problem statement.** Multiparty session types can rule out communication errors, and actor languages support robust distributed programming. However, due to the mailbox-oriented communication model supported by actors, applying session types to actors is challenging.
> How can we support *static* checking of multiparty session types in actors which can take part in *multiple sessions*?

In this extended abstract, we detail ongoing work showing how combining a flow-sensitive effect system (*i.e.*, an effect system with pre- and post-conditions) with event-driven programming supports flexible actor programming with *statically-checked* MPSTs. Although we do not introduce any novel effects machinery, the relevance of this work to HOPE is how effect typing and event-driven programming can address a pressing open problem in the session types community.



(a) Channels **(b)** Actors

Figure 1. Channels and actors (taken from [4])

2 Preliminaries

We begin by introducing some preliminary concepts.

(Multiparty) session types. Session types [8] are a type discipline that enforce conformance to communication protocols: if a program typechecks against its session type, then it is guaranteed to implement the protocol. Originally, session types described communication between two participants. *Multiparty* session types [9] allow communication patterns to be described between more than two participants: a *global type* describes the interactions between all participants, and can be projected to a *local type* for each participant.

Actor- and channel-based communication. A *channel* is a shared name or buffer that allows two or more processes to communicate. An *actor* is a process which can react to incoming messages; typically actor languages associate processes with a *mailbox*. Figure 1 (taken from [4], which provides a detailed formal comparison) shows the difference between the two models.

Whereas it is straightforward to give types to session channel endpoints, the unidirectional nature of mailboxes makes it difficult to apply session types to mailboxes directly. Therefore, session types are typically used to govern the *actions that an actor performs*, rather than the mailbox itself.

Event-driven programming. In the *event-driven programming* programming model, computation is triggered by an *event* (for example, a mouse click, or an incoming message). Each event is handled by an *event handler*; after the handler has completed, the thread reverts to being idle.

Flow-sensitive effect systems. Standard Gifford-style type-and-effect systems [12] record the effects performed by an expression (e.g., writing to a reference cell or opening a file handle). These type-and-effect systems are typically set-based and do not reason about ordering of effects. In contrast, flow-sensitive [13] (also known as *sequential* [6]) effect systems reason about the *ordering* of effects, and their typing judgements typically express effects as pre- and post-conditions. Our functional formulation is reminiscent of Atkey's parameterised monads [2].

Putting them together. Since it is difficult to attach session types to mailboxes, previous work has used session types to govern the communication actions performed by an actor, either using runtime monitoring [3, 14], or static typechecking for a single session using session types as pre- and post-conditions in a flow-sensitive effect system [7].

In short, our programming model combines the benefits of actor style programming (e.g., data locality) with statically-checked multiparty session types. Session typing is enforced by a type-and-effect system, and event-driven programming allows an actor to take part in more than one session.

3 Programming model by example

We introduce our model using the two-buyer protocol [9], intended as an abstraction of financial protocols. Two buyers (Buyer1 and Buyer2) interact with a Seller to buy a book: Buyer1 sends a title to Seller, who responds with a price. At this point, Buyer1 sends Buyer2 their share of the price. Buyer2 can then either accept the offer, sending their address to Seller and receiving a delivery date, or reject the offer. The *global type* for the protocol is described on the left; by *projecting* the global type, we obtain a *local type* for each role. The local type for Buyer2, B2, is shown on the right; & denotes branching and ⊕ denotes making a choice.

```
Buyer1 → Seller : title(String) .            B2 ≜
Seller → Buyer1 : quote(Int) .                 Buyer1 & share(Int) .
Buyer1 → Buyer2 : share(Int) .                 Seller ⊕{
Buyer2 → Seller : {                              address(String) .
  address(String) .                                Seller & date(Date)
    Seller → Buyer2 : date(Date) . end,             . end,
  quit(Unit) . end                               quit(Unit) . end
}                                              }
```

Although our core calculus is in the style of fine-grain call-by-value [11], we allow ourselves nested expressions in examples. We wish to allow a single seller to interact with an arbitrary number of requests. We include the main process used to set up the sessions, and the implementation of the Seller role below; the implementations and local types for the other roles can be found in Appendix A.

The program creates a new *access point* [5]—a name with which actors can register in order to take part in a session—and spawns a seller actor, and two sets of buyers.

```
main ≜
  let ap ⇐ newAP(Seller:S,Buyer1:B1,Buyer2:B2) in
  spawn (seller(ap) ());
  spawnBuyers(ap, "Types and Programming Languages");
  spawnBuyers(ap, "Compiling with Continuations")

spawnBuyers(ap, title) ≜
  spawn buyer1(ap, title); spawn buyer2(ap)
```

The seller is defined as a recursive function that registers with *ap* to play role Seller. The **register** construct is given a callback to be evaluated once a session is established: here it will re-register to take part in another instance of the session before awaiting a title message from Buyer1. To await a message, the actor invokes **suspend** with a (first-class) message handler titleHandler; this installs the given message handler and reverts the actor to being idle until a message arrives and an installed handler can be invoked.

Note: Our use of the term "handler" refers to a *message* handler; since our proposal does not include algebraic effects, our message handlers are *not* effect handlers.

```
seller(ap) ≜
  register ap Seller (( rec install(_).
                          register ap Seller (install ());
                          suspend titleHandler        ) ())
titleHandler ≜
  handler Buyer1 {
    title(x) ↦
      Buyer1 ! quote(lookupPrice(x)); suspend decisionHandler }
```

Once the titleHandler has been invoked with the title, it sends the price to Buyer1, and awaits a decision from Buyer2:

```
decisionHandler ≜
  handler Buyer2 {
    address(addr) ↦ Buyer2 ! date(shippingDate(addr)),
    quit(_) ↦ return () }
```

If Buyer2 agrees and sends their address, then Seller will send the date; otherwise, the Seller instance will finish.

Although an actor can be involved in multiple sessions simultaneously, it will only evaluate a term in the context of one session at a time; after an actor suspends, it may invoke a handler from a different session. In our case, this allows the same seller actor to participate in *both* sessions.

4 Calculus

Syntax. Our core language is a standard fine-grain call-by-value λ-calculus extended with constructs for event-based concurrency and session communication; we will describe the typing rules for each shortly.

Types include base types C, function types $A \xrightarrow{S,T} B$ (a function from A to B with session precondition S and post-condition T), and access point types $\mathsf{AP}((\mathsf{p} : S_i)_i)$.

Local session types S, T consist of input session types $\mathsf{p}\,\&\{\ell_i(A_i)\,.\,S_i\}_{i \in I}$ (ranged over by $S^?$) and output session types $\mathsf{p} \oplus \{\ell_i(A_i) . S_i\}$, recursive session types $\mu X.S$ and recursion variables X, and the completed session type end. We omit the braces for unary branches and selections.

Typing. The value typing judgement $\Gamma \vdash V : A$ is standard; following [7] the computation typing judgement $\Gamma \mid S \triangleright M : A \triangleleft T$ can be read "under type environment Γ, with session type S, term M has type A and updates the session type to T". We concentrate on the session- and event-based rules here; full typing rules can be found in Appendix B.

The $\textbf{newAP}_{(\mathsf{p}_i:S_i)_i}$ construct creates a new access point for the specified roles and session types; following Scalas and Yoshida [15], we require that the session types satisfy a base *safety property* φ that precludes communication mismatches.

$$\frac{\varphi \text{ is a safety property} \qquad \varphi((\mathsf{p}_i : T_i)_{i\in I})}{\Gamma \mid S \triangleright \textbf{newAP}_{(\mathsf{p}_i:T_i)_{i\in I}} : \mathsf{AP}((\mathsf{p}_i : T_i)_{i\in I}) \triangleleft S}$$

The **register** $V\ \mathsf{p}_j\ M$ term is well typed if V is an access point supporting p_j of type T_j, and the continuation is typable with session type T_j. Note that evaluating **register** does not affect the current evaluation context, so does not modify the session type and has return type **1**.

$$\frac{j \in I \qquad \Gamma \vdash V : \mathsf{AP}((\mathsf{p}_i : T_i)_{i\in I}) \qquad \Gamma \mid T_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \textbf{register}\ V\ \mathsf{p}_j\ M : \mathbf{1} \triangleleft S}$$

A message handler $\textbf{handler}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}$ has type $\mathsf{Handler}(\mathsf{p}\,\&\{\ell_i(A_i).S_i\}_i)$ if, given an environment extended with the parameter name x_i and payload type A_i, each continuation M_i has type **1** and final session type end, denoting that the session type has completed.

$$\frac{(\Gamma, x : A_i \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end})_i}{\Gamma \vdash \textbf{handler}\ \mathsf{p}\ \{\ell_i(x_i) \mapsto M_i\}_i : \mathsf{Handler}(\mathsf{p}\,\&\{\ell_i(A_i).S_i\}_i)}$$

An actor can only send a message if it is permitted by the current session type; the rule requires that the payload type matches the one described in the session type, and updates the session type to the relevant continuation.

$$\frac{j \in I \qquad \Gamma \vdash V : A_j}{\Gamma \mid \mathsf{p} \oplus \{\ell_i(A_i).S_i\}_{i\in I} \triangleright \mathsf{p}\,!\,\ell_j(V) : \mathbf{1} \triangleleft S_j}$$

Finally, we can type **suspend** V if the current session type matches the session type of the message handler V. Note that **suspend** has an arbitrary return type and postcondition, since it will abort the current evaluation context.

$$\frac{\Gamma \vdash V : \mathsf{Handler}(S^?)}{\Gamma \mid S^? \triangleright \textbf{suspend}\ V : A \triangleleft T}$$

Semantics. For space, we give an informal overview of the semantics: we model an actor as a triple $\langle \mathcal{T}, \sigma, \rho \rangle$ where $\mathcal{T}$ is either **idle**, or a term $(M)^{s[\mathsf{p}]}$ denoting evaluating M in the context of role p in session s, or a term M evaluating outwith the context of a session (*i.e.* when an actor has just been created). Environment σ maps session-role pairs $s[\mathsf{p}]$ to message handlers; and ρ stores initialisation callbacks.

Access points establish a session if there are idle registered actors for each role. Suspending aborts the current evaluation context and adds the given handler to σ. We model asynchronous communication through the use of a session-level queue (which, due to message reordering rules, is isomorphic to individual role-level queues). Sending appends a message to the queue. If an actor is idle and has a stored event handler $s[\mathsf{p}] \mapsto \textbf{handler}\ \mathsf{q}\ \{\ell_i(x_i) \mapsto M_j\}_{i\in I}$ and the head of the queue contains a message $(\mathsf{q}, \mathsf{p}, \ell_j(V_j))$ (where $j \in I$), then the thread state will become $(M_j\{V_j/x_j\})^{s[\mathsf{p}]}$.

Metatheory. We have proved a type preservation theorem using the generalised approach introduced by Scalas and Yoshida [15]. We conjecture that the event-driven nature of the system will also permit a global progress result like that of Viering et al. [16], although this is a work-in-progress.

5 Conclusion

We have shown how the combination of a flow-sensitive effect system and event-driven programming allows static typing of actors that participate in multiple sessions. We have also implemented a typechecker and small-step interpreter for the calculus; we further plan to build on the code generation approach introduced by Hu and Yoshida [10] to allow the programming model to be used in mainstream programming languages such as Scala. Our next steps are to prove a global progress theorem and to investigate how to switch between sessions while maintaining progress guarantees.

Acknowledgments

Thanks to Matthew Alan Le Brun and Alceste Scalas for discussion of the theory of asynchronous generalised multiparty session types. Thanks also to the HOPE'23 reviewers for their valuable comments; hopefully the abstract is more accessible as a result. Fowler was funded by EPSRC grant EP/T014628/1 (STARDUST).

References

[1] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[2] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376.

[3] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE (EPTCS, Vol. 223)*. 36–50.

[4] Simon Fowler, Sam Lindley, and Philip Wadler. 2017. Mixing Metaphors: Actors as Channels and Channels as Actors. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:28.

[5] Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50.

[6] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79.

[7] Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor

Language. In *ECOOP (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:30.
[8] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (Lecture Notes in Computer Science, Vol. 715)*. Springer, 509–523.
[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.
[10] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *FASE (Lecture Notes in Computer Science, Vol. 9633)*. Springer, 401–418.
[11] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210.
[12] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57.
[13] Daniel Marino and Todd D. Millstein. 2009. A generic type-and-effect system. In *TLDI*. ACM, 39–50.
[14] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Log. Methods Comput. Sci.* 13, 1 (2017).
[15] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29.
[16] Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. 2021. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30.

A Full two-buyer example

Local types.

$$
\begin{array}{lcl}
\mathsf{B1} & \triangleq & \mathsf{Seller} \oplus \mathtt{title}(\mathtt{String}) \,.\, \mathsf{Seller} \,\&\, \mathtt{quote}(\mathtt{String}) \,.\\
& & \mathsf{Buyer2} \oplus \mathtt{share}(\mathtt{Int}) \,.\, \mathtt{end} \\
\mathsf{B2} & \triangleq & \mathsf{Buyer1} \,\&\, \mathtt{share}(\mathtt{Int}) \,.\, \mathsf{Seller} \oplus \{ \\
& & \quad \mathtt{address}(\mathtt{String}) \,.\, \mathsf{Seller} \,\&\, \mathtt{date}(\mathtt{Date}) \,.\, \mathtt{end} \\
& & \quad \mathtt{quit}(\mathtt{Unit}) \,.\, \mathtt{end} \, \} \\
\mathsf{S} & \triangleq & \mathsf{Buyer1} \,\&\, \mathtt{title}(\mathtt{String}) \,.\, \mathsf{Buyer1} \oplus \mathtt{quote}(\mathtt{Int}) \,.\\
& & \mathsf{Buyer2} \,\&\{ \\
& & \quad \mathtt{address}(\mathtt{String}) \,.\, \mathsf{Buyer2} \oplus \mathtt{date}(\mathtt{Date}) \,.\, \mathtt{end} \\
& & \quad \mathtt{quit}(\mathtt{Unit}) \,.\, \mathtt{end} \, \}
\end{array}
$$

Main function.

```
main ≜
  let ap ⇐ newAP(Seller:S,Buyer1:B1,Buyer2:B2) in
  spawn (seller(ap) ());
  spawnBuyers(ap, "Types and Programming Languages");
  spawnBuyers(ap, "Compiling with Continuations")

spawnBuyers(ap, title) ≜
  spawn buyer1(ap, title); spawn buyer2(ap)
```

Seller.

```
seller(ap) ≜
  rec install(_).
    register ap Seller ( install ();
                         suspend titleHandler )
titleHandler ≜
  handler Buyer1 {
    title(x) ↦
      Buyer1 ! quote(lookupPrice(x));
      suspend decisionHandler
  }

decisionHandler ≜
  handler Buyer2 {
    address(addr) ↦
      Buyer2 ! date(shippingDate(addr))
    quit(_) ↦ return ()
  }
```

Here, shippingDate is left abstract and calculates a shipping date given an address.

Buyer 1.

```
buyer1(ap, title) ≜
  register ap Buyer1
    ( Seller ! title(title);
      suspend quoteHandler )

quoteHandler ≜
  handler Seller {
    quote(amount) ↦
      Buyer2 ! share(amount/2)
  }
```

Buyer 2.

```
buyer2(ap) ≜
  register ap Buyer2 (suspend shareHandler)

shareHandler ≜
  handler Buyer {
    share(amount) ↦
      if (amount > 100) then
        Seller ! quit(())
      else
        Seller ! address("18 Lilybank Gardens");
        suspend dateHandler
  }

dateHandler ≜
  handler Seller {
    date(date) ↦ log(date)
  }
```

Here, log is left abstract, and logs the received date.

B Full typing rules

Value typing $\boxed{\Gamma \vdash V : A}$

$$\text{TV-VAR}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{TV-LAM}\ \frac{\Gamma, x : A \mid S \triangleright M : B \triangleleft T}{\Gamma \vdash \lambda x.M : A \xrightarrow{S,T} B} \qquad \text{TV-REC}\ \frac{\Gamma, x : A, f : A \xrightarrow{S,T} B \mid S \triangleright M : A \xrightarrow{S,T} B \triangleleft T}{\Gamma \vdash \mathbf{rec}\ f(x).M : A \xrightarrow{S,T} B} \qquad \text{TV-CONST}\ \frac{c \text{ has base type } C}{\Gamma \vdash c : C}$$

$$\text{TV-HANDLER}\ \frac{(\Gamma, x : A_i \mid S_i \triangleright M_i : \mathbf{1} \triangleleft \mathsf{end})_i}{\Gamma \vdash \mathbf{handler}\ \mathtt{p}\ \{\ell_i(x_i) \mapsto M_i\}_i : \mathsf{Handler}(\mathtt{p}\,\&\{\ell_i(A_i).S_i\}_i)}$$

Computation typing $\boxed{\Gamma \mid S \triangleright M : A \triangleleft T}$

$$\text{T-RETURN}\ \frac{\Gamma \vdash V : A}{\Gamma \mid S \triangleright \mathbf{return}\ V : A \triangleleft S} \qquad \text{T-LET}\ \frac{\Gamma \mid S_1 \triangleright M : A \triangleleft S_2 \quad \Gamma, x : A \mid S_2 \triangleright N : B \triangleleft S_3}{\Gamma \mid S_1 \triangleright \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N : B \triangleleft S_3} \qquad \text{T-APP}\ \frac{\Gamma \vdash V : A \xrightarrow{S,T} B \qquad \Gamma \vdash W : A}{\Gamma \mid S \triangleright V\ W : B \triangleleft T}$$

$$\text{T-IF}\ \frac{\Gamma \vdash V : \mathsf{Bool} \qquad \Gamma \mid S_1 \triangleright M : A \triangleleft S_2 \qquad \Gamma \mid S_1 \triangleright N : A \triangleleft S_2}{\Gamma \mid S_1 \triangleright \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N : A \triangleleft S_2} \qquad \text{T-SPAWN}\ \frac{\Gamma \mid \mathsf{end} \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{spawn}\ M : \mathbf{1} \triangleleft S} \qquad \text{T-SEND}\ \frac{j \in I \qquad \Gamma \vdash V : A_j}{\Gamma \mid \mathtt{p} \oplus \{\ell_i(A_i).S_i\}_{i \in I} \triangleright \mathtt{p}\,!\,\ell_j(V) : \mathbf{1} \triangleleft S_j}$$

$$\text{T-SUSPEND}\ \frac{\Gamma \vdash V : \mathsf{Handler}(S^?)}{\Gamma \mid S^? \triangleright \mathbf{suspend}\ V : A \triangleleft T} \qquad \text{T-NEWAP}\ \frac{\varphi \text{ is a safety property} \qquad \varphi((\mathtt{p}_i : T_i)_{i \in I})}{\Gamma \mid S \triangleright \mathbf{newAP}_{(\mathtt{p}_i : T_i)_{i \in I}} : \mathsf{AP}((\mathtt{p}_i : T_i)_{i \in I}) \triangleleft S} \qquad \text{T-REGISTER}\ \frac{j \in I \qquad \Gamma \vdash V : \mathsf{AP}((\mathtt{p}_i : T_i)_{i \in I}) \qquad \Gamma \mid T_j \triangleright M : \mathbf{1} \triangleleft \mathsf{end}}{\Gamma \mid S \triangleright \mathbf{register}\ V\ \mathtt{p}_j\ M : \mathbf{1} \triangleleft S}$$