A Logical Basis for the Verification of Message-Passing Programs

Jorge A. Pérez

Fundamental Computing Group
University of Groningen, The Netherlands
https://www.rug.nl/fse/fc

Dutch Formal Methods Day
April 16, 2024

UNIFYING
C•RRECTNESS FOR
C•MMUNICATING
S•FTWARE

My Group's Research: Keywords (and Slogans)

Concurrency Theory, Message-Passing, Programming Languages, Verification

My Group's Research: Keywords (and Slogans)

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**
  Slogan: Well-typed programs can't go wrong (Milner)

My Group's Research: Keywords (and Slogans)

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**
  Slogan: Well-typed programs can't go wrong (Milner)
- **Session types** for communication correctness
  Slogan: **What** and **when** should be sent through a channel

My Group's Research: Keywords (and Slogans)

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**
  Slogan: Well-typed programs can't go wrong (Milner)
- **Session types** for communication correctness
  Slogan: **What** and **when** should be sent through a channel
- **Process calculi**
  Slogan: The π-calculus treats **processes** like the λ-calculus treats **functions**
- **Propositions as sessions**

My Group's Research: Keywords (and Slogans)

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**
  Slogan: Well-typed programs can't go wrong (Milner)
- **Session types** for communication correctness
  Slogan: **What** and **when** should be sent through a channel
- **Process calculi**
  Slogan: The π-calculus treats **processes** like the λ-calculus treats **functions**
- **Propositions as sessions**

Today An overview in two parts

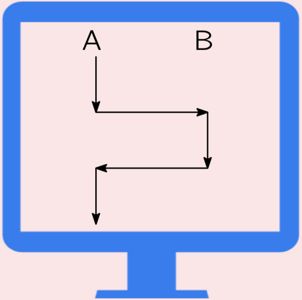
- A gentle introduction to session types
- Runtime verification based on session types (presented in RV'23)

Part I

Session Types for Message-Passing Concurrency

When is a Program Correct?

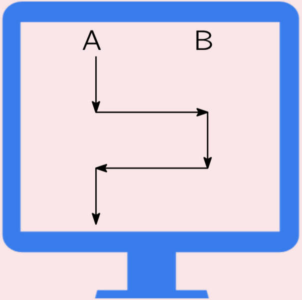
Sequential Programs



"Programs produce outputs that are consistent with their input"

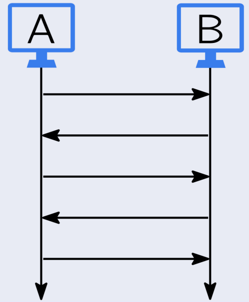
When is a Program Correct?

Sequential Programs



"Programs produce outputs that are consistent with their input"

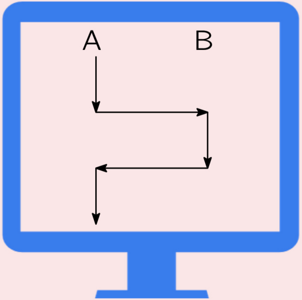
Concurrent Programs



"Programs always respect their intended protocols"

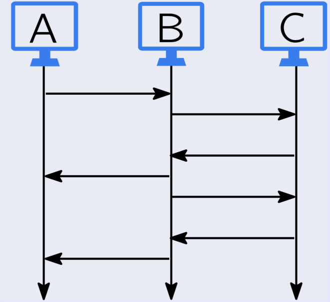
When is a Program Correct?

Sequential Programs



“Programs produce outputs that are consistent with their input”

Concurrent Programs

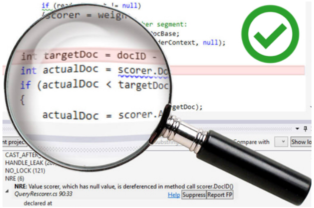


“Programs always respect their intended protocols”

Type Systems: From Data to Behaviors

- Can detect bugs before programs are run
- Present in many programming languages
- A sound notion of correctness

  A program is either correct or incorrect



```
if (rea        != null)
{scorer = weig
               ber segment:
               ocBase;
               derContext, null);
int targetDoc = docID -
int actualDoc = scorer.Dc
if (actualDoc < targetDoc
{
    actualDoc = scorer.A         getDoc);
```

CAST_AFTER_

HANDLE_LEAK (2

NO_LOCK (121)

NRE (6)

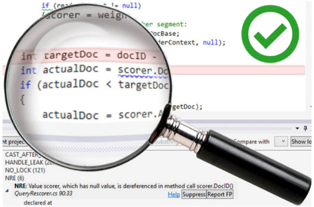
NRE: Value scorer, which has null value, is dereferenced in method call scorer.DocID()

QueryRescorer.cs 90:33

declared at

Type Systems: From Data to Behaviors

- Can detect bugs before programs are run
- Present in many programming languages
- A sound notion of correctness
  A program is either correct or incorrect



```
if (rea          != null)
{scorer = weigh
                   her segment:
                   ocBase;
                   derContext, null);
int targetDoc = docID -
int actualDoc = scorer.Dc
if (actualDoc < targetDoc
{
    actualDoc = scorer.A       getDoc);
```

CAST_AFTER
HANDLE_LEAK (2
NO_LOCK (121)
NRE (6)
NRE: Value scorer, which has null value, is dereferenced in method call scorer.DocID()
QueryRescorer.cs 90:33
declared at

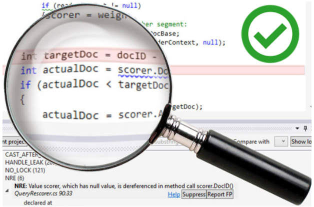
Sequential Languages

- **Data type systems** classify values in a program
- Examples: Integers, strings of characters

Type Systems: From Data to Behaviors

- Can detect bugs before programs are run
- Present in many programming languages
- A sound notion of correctness

  A program is either correct or incorrect



```
if (rea      t != null)
{scorer = weigh
                  her segment:
                  ocBase;
                  derContext, null);
int targetDoc = docID -
int actualDoc = scorer.Do
if (actualDoc < targetDoc
{
                  rgetDoc);
    actualDoc = scorer.A
```

CAST_AFTER_
HANDLE_LEAK (2
NO_LOCK (121)
NRE (6)

NRE: Value scorer, which has null value, is dereferenced in method call scorer.DocID()
QueryRescorer.cs 90:33
declared at

Sequential Languages

- **Data type systems** classify values in a program
- Examples: Integers, strings of characters

Concurrent Languages

- **Behavioral type systems** classify protocols in a program
- Example: “first send username, then receive true/false, finally close”
- A typical bug: sending messages in the wrong order

Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (input and output)
- labeled choices (offers and selections)
- sequential composition
- recursion

Session protocols are attached to **interaction devices**:

- channel endpoints
- channels in languages like Go
- π-calculus names
- $\cdots$

Sequentiality in types goes **hand-in-hand** with sequentiality in processes

Example: A Two-Buyer Protocol

Alice and Bob cooperate in buying a book from Seller:

bruna

Example: A Two-Buyer Protocol

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.

Example: A Two-Buyer Protocol

bruna

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.

Example: A Two-Buyer Protocol

bruna

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
    a) completing the payment and arranging delivery details
    b) canceling the transaction

Example: A Two-Buyer Protocol

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
    a) completing the payment and arranging delivery details
    b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

Example: A Two-Buyer Protocol

bruna

Alice and Bob cooperate in buying a book from Seller:

1. Alice sends a book title to Seller, who sends a quote back.
2. Alice checks whether Bob can contribute in buying the book.
3. Alice uses the answer from Bob to interact with Seller, either:
   a) completing the payment and arranging delivery details
   b) canceling the transaction
4. In case 3(a) Alice contacts Bob to get his address, and forwards it to Seller.

4’. In case 3(b) Alice is in charge of gracefully concluding the conversation.

The Syntax of Session Types

$S ::=$	$!U;S$	**output** value of type U, continue as S
$\mid$	$?U;S$	**input** value of type U, continue as S
$\mid$	$\&\{l_i : S_i\}_{i \in I}$	**offer** a selection between $S_1, \ldots, S_n$
$\mid$	$\oplus\{l_i : S_i\}_{i \in I}$	**select** between $S_1, \ldots, S_n$
$\mid$	$\mu t.S \mid t$	**recursion**
$\mid$	$\mathtt{end}$	**terminated protocol**

(Labels $l_1, \ldots, l_n$ are pairwise different.)

Example: A Two-Buyer Protocol

Two separate protocols, with Alice "leading" the interactions:

- A session type for Seller (in its interaction with Alice):

$$S_{\mathsf{SA}} = \ ?\mathtt{book};\, !\mathtt{quote};\, \& \begin{cases} \mathsf{buy}: & ?\mathtt{paym};\, ?\mathtt{address};\, !\mathtt{ok};\, \mathsf{end} \\ \mathsf{cancel}: & ?\mathtt{thanks};\, !\mathtt{bye};\, \mathsf{end} \end{cases}$$

- A session type for Alice (in its interaction with Bob):

$$S_{\mathsf{AB}} = \ !\mathtt{cost};\, \& \begin{cases} \mathsf{share}: & ?\mathtt{address};\, !\mathtt{ok};\, \mathsf{end} \\ \mathsf{close}: & !\mathtt{bye};\, \mathsf{end} \end{cases}$$

Example: A Two-Buyer Protocol

Two separate protocols, with Alice "leading" the interactions:

- A session type for Seller (in its interaction with Alice):

$$S_{\mathsf{SA}} = \ ?\mathtt{book};\, !\mathtt{quote};\, \& \begin{cases} \mathsf{buy}: & ?\mathtt{paym};\, ?\mathtt{address};\, !\mathtt{ok};\, \mathsf{end} \\ \mathsf{cancel}: & ?\mathtt{thanks};\, !\mathtt{bye};\, \mathsf{end} \end{cases}$$

- A session type for Alice (in its interaction with Bob):

$$S_{\mathsf{AB}} = \ !\mathtt{cost};\, \& \begin{cases} \mathsf{share}: & ?\mathtt{address};\, !\mathtt{ok};\, \mathsf{end} \\ \mathsf{close}: & !\mathtt{bye};\, \mathsf{end} \end{cases}$$

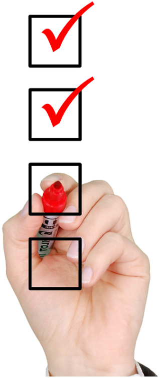
Note:

- The above protocols are specified in the **binary** setting
- Session types have been developed also in the more general **multiparty** setting

Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

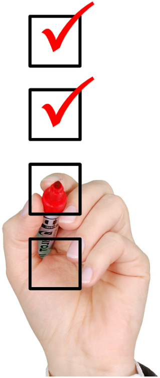
- **Fidelity** – they **follow the intended protocol**.
  - Alice doesn't continue the transaction if Bob can't contribute
  - Alice chooses among the options provided by Seller



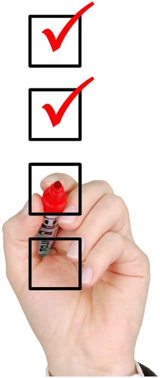
Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
  - Seller always returns an integer when Alice requests a quote



Example: A Two-Buyer Protocol



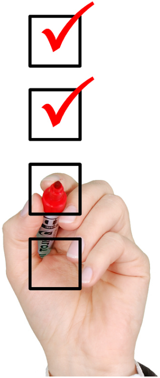
Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
  - Alice eventually receives an answer from Bob on his contribution.

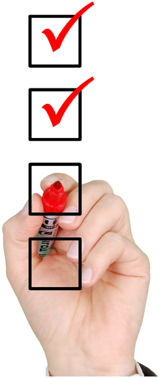
Example: A Two-Buyer Protocol

Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)



Example: A Two-Buyer Protocol



Desiderata for the implementations of Alice, Bob, and Seller:

- **Fidelity** – they **follow the intended protocol**.
- **Safety** – they don't feature **communication errors**.
- **Deadlock-Freedom** – they do not **"get stuck"** while running the protocol.
- **Termination** – they do not engage in **infinite behavior** (that may prevent them from completing the protocol)

Correctness follows from the interplay of these properties.
Hard to enforce, especially when actions are "scattered around" in source programs.

Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.
  - the dual of input is output (and vice versa)
  - branching is the dual of selection (and vice versa)

Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.
  - the dual of input is output (and vice versa)
  - branching is the dual of selection (and vice versa)

- Recall that S_{AB} describes Alice's viewpoint in her interaction with Bob:

$$S_{\mathsf{AB}} = \; !\mathtt{cost}; \& \begin{cases} \text{share}: & ?\mathtt{address}; !\mathtt{ok}; \text{end} \\ \text{close}: & !\mathtt{bye}; \text{end} \end{cases}$$

- Given this, Bob's implementation should conform to $\overline{S_{\mathsf{AB}}}$, the dual of S_{AB}:

$$\overline{S_{\mathsf{AB}}} = \; ?\mathtt{cost}; \oplus \begin{cases} \text{share}: & !\mathtt{address}; ?\mathtt{ok}; \text{end} \\ \text{close}: & ?\mathtt{bye}; \text{end} \end{cases}$$

Example: A Two-Buyer Protocol

Implementations for Alice, Bob, Seller should be **compatible**.

- **Duality** relates session types with opposite behaviors.
  - the dual of input is output (and vice versa)
  - branching is the dual of selection (and vice versa)

- Recall that S_{AB} describes Alice's viewpoint in her interaction with Bob:

$$S_{\mathsf{AB}} = \;!\mathtt{cost}; \& \begin{cases} \mathsf{share}: & ?\mathtt{address}; !\mathtt{ok}; \mathsf{end} \\ \mathsf{close}: & !\mathtt{bye}; \mathsf{end} \end{cases}$$

- Given this, Bob's implementation should conform to $\overline{S_{\mathsf{AB}}}$, the dual of S_{AB}:

$$\overline{S_{\mathsf{AB}}} = \;?\mathtt{cost}; \oplus \begin{cases} \mathsf{share}: & !\mathtt{address}; ?\mathtt{ok}; \mathsf{end} \\ \mathsf{close}: & ?\mathtt{bye}; \mathsf{end} \end{cases}$$

- Also, Alice's implementation should conform to both $\overline{S_{\mathsf{SA}}}$ and S_{AB}.

Propositions as Sessions

Concurrency		**Logic**
session types	↔	linear logic propositions
π-calculus processes	↔	proofs
process communication	↔	cut elimination

- All four correctness properties hold "for free"
- Firm justification for seminal work on session types
- Reference framework for expressiveness
- Canonical platform for extensions (e.g., sharing)

Part II

Runtime Verification Based on Session Types

In A Nutshell

- A verification methodology based on **routers**, protocol descriptions synthesized from multiparty protocols.
- Combining and improving existing techniques, leveraging on propositions-as-sessions. Validated with a practical implementation.
- **Key idea**: Routers enrich local descriptions by capturing intra-participant dependencies.
- Routers be can used for **static verification** (type systems, SCP'22) and also in a **dynamic verification** setup (RV'23).

Multiparty Session Types

- A **global type** provides the entire protocol's specification for multiple participants. Participant implementations communicate with each other, without a coordinator.
- A simple authorization protocol:

$$G_{\mathsf{auth}} = \mu X.s!c\{\mathsf{login}.c!a(\mathsf{passwd}).a!s(\mathsf{succ}).X \ ,\ \mathsf{quit}.\mathsf{end}\}$$

Three participants: client (c), server (s), authorization server (a)

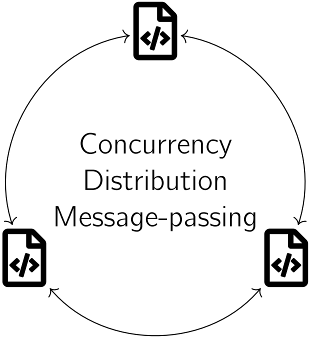
Multiparty Session Types

- A **global type** provides the entire protocol's specification for multiple participants. Participant implementations communicate with each other, without a coordinator.
- A simple authorization protocol:

  $$G_{\text{auth}} = \mu X.s!c\{\text{login}.c!a(\text{passwd}).a!s(\text{succ}).X\ ,\ \text{quit}.\text{end}\}$$

  Three participants: client (c), server (s), authorization server (a)
- The global type is projected onto **local types**, one per participant, which provide a basis for static or dynamic verification.
- Note: not all conceivable global types are projectable onto local types.

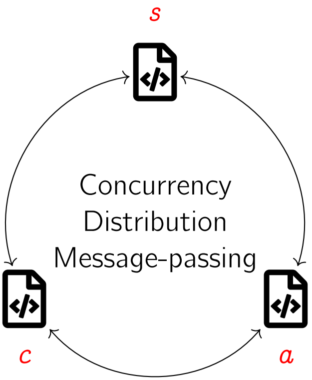
Dynamic Approach



Concurrency
Distribution
Message-passing

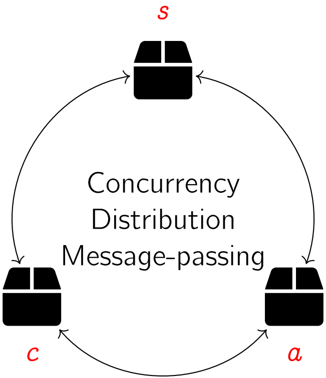
- Multiparty session types (MPSTs): **protocols** for distributed message-passing.
- MPSTs enable useful runtime verification techniques. They rely on usual notions of well-formedness, which **limits their applicability**.
- Many practical protocols **not supported** by existing RV techniques: e.g., our running example, **s**erver requests **c**lient to login through **a**uthorization service.
- Existing techniques require **too much information** about components.

Dynamic Approach



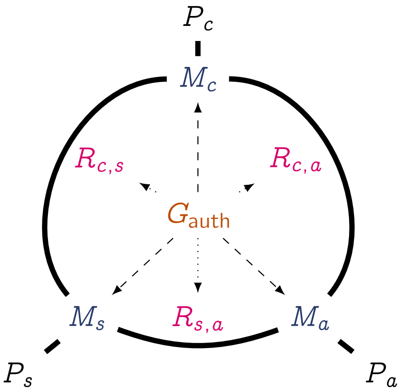
- Multiparty session types (MPSTs): **protocols** for distributed message-passing.
- MPSTs enable useful runtime verification techniques. They rely on usual notions of well-formedness, which **limits their applicability**.
- Many practical protocols **not supported** by existing RV techniques: e.g., our running example, **s**erver requests **c**lient to login through **a**uthorization service.
- Existing techniques require **too much information** about components.

Dynamic Approach: Overview



- New approach to **runtime** verification of distributed components using MPSTs as **monitors** to verify **protocol conformance**.
- Support **expressive** class of protocols.
- Components with **unknown** specification but **observable** message-passing behavior.
- LTSs with **minimal assumptions**: “blackboxes”.
- Contributions:
  - Verification framework.
  - Compositional verification.
  - Protocol conformance and **transparency**.
  - Prototype implementation.

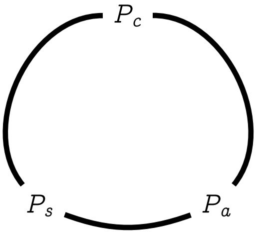
Dynamic Verification: Setup



We use the global type (multiparty protocol) in different ways:

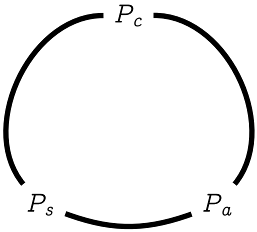
- Obtain local views for verifying protocol conformance
- Synthesize monitors for each participant
- Detect additional coordination messages

Our Setup: Blackboxes



- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

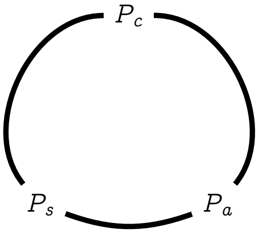
Our Setup: Blackboxes



- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

P_c

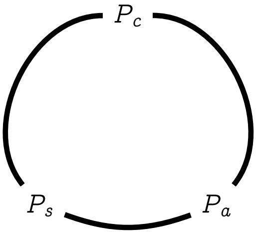
Our Setup: Blackboxes



- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

$$P_c^{|} \xleftarrow{c?s} P_c$$

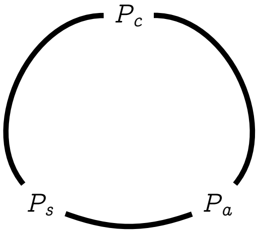
Our Setup: Blackboxes



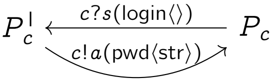
- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m}\rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

$$P_c^{|} \xleftarrow{c?s(\mathsf{login}\langle\rangle)} P_c$$

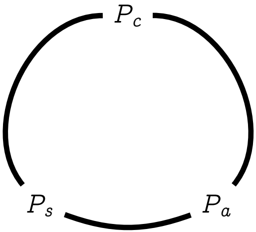
Our Setup: Blackboxes



- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.



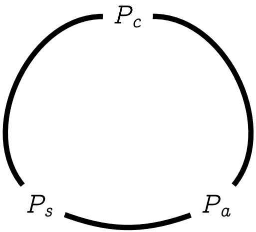
Our Setup: Blackboxes



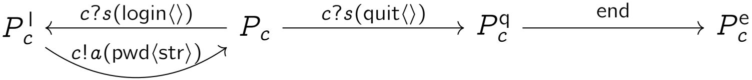
- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

$$\begin{tikzcd} P_c^{\mathsf{l}} \arrow[r, bend right, "c!a(\mathsf{pwd}\langle \mathsf{str} \rangle)"'] & P_c \arrow[l, "c?s(\mathsf{login}\langle\rangle)"'] \arrow[r, "c?s(\mathsf{quit}\langle\rangle)"] & P_c^{\mathsf{q}} \end{tikzcd}$$

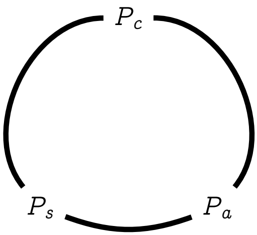
Our Setup: Blackboxes



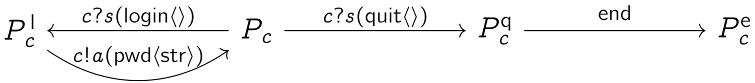
- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.



Our Setup: Blackboxes

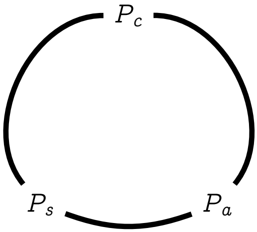


- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.



$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle) \rangle$

Our Setup: Blackboxes

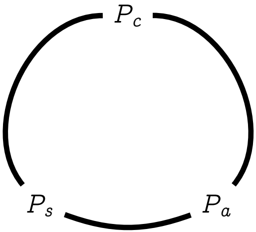


- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

$$\begin{tikzcd}
P_c^{\mathsf{!}} \arrow[r, bend right, "c!a(\mathsf{pwd}\langle\mathsf{str}\rangle)"'] & P_c \arrow[l, "c?s(\mathsf{login}\langle\rangle)"'] \arrow[r, "c?s(\mathsf{quit}\langle\rangle)"] & P_c^{\mathsf{q}} \arrow[r, "\mathsf{end}"] & P_c^{\mathsf{e}}
\end{tikzcd}$$

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle) \rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon \rangle$$

Our Setup: Blackboxes

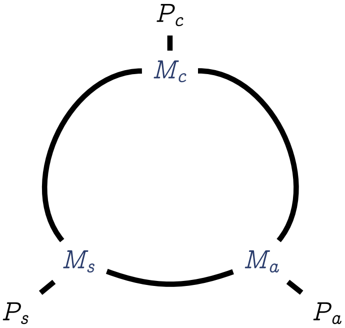


- Each blackbox assumed to belong to a protocol **participant**: e.g., P_c.
- Blackboxes exchange messages **asynchronously** through buffers: e.g., $\langle c : P_c : \vec{m} \rangle$.
- Messages carry **data** or **choices** to resolve **branching**.

$$\begin{tikzcd}
P_c^{\mathsf{!}} \arrow[r, bend right, "c!a(\mathsf{pwd}\langle\mathsf{str}\rangle)"'] & P_c \arrow[l, "c?s(\mathsf{login}\langle\rangle)"'] \arrow[r, "c?s(\mathsf{quit}\langle\rangle)"] & P_c^{\mathsf{q}} \arrow[r, "\mathsf{end}"] & P_c^{\mathsf{e}}
\end{tikzcd}$$

$$\begin{tikzcd}
\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \arrow[r, "c?s(\mathsf{quit}\langle\rangle)"] & \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \arrow[r, "\mathsf{end}"] & \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle
\end{tikzcd}$$

Our Setup: Monitors



- **Monitoring** to shield the system against **unexpected behavior**.
- Monitors: FSMs describing sequences of **expected** incoming/outgoing messages.
- **Forwards** expected messages between its blackbox and other monitored blackboxes.
- Unexpected messages result in **error state**.
- **Monitored blackboxes**: e.g., $[\langle c : P_c : \vec{m} \rangle : M_c : \vec{n}]$.
- Broadcast messages to **coordinate** blackboxes to support **more expressive** protocols.

Networks of monitored blackboxes (1/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

Networks of monitored blackboxes (1/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$M_c = c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} \qquad M_s = s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\}$$

Networks of monitored blackboxes (1/2)

$$\frac{\begin{array}{c} \langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle \\ \langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle \end{array}}{[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \quad \mid \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]}$$

Networks of monitored blackboxes (1/2)

$$
\begin{array}{c}
\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle \\
\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle \\
\hline
\end{array}
$$

$$
\begin{array}{l}
[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \quad \mid \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \\
\qquad\qquad \downarrow\tau \\
[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad \mid \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]
\end{array}
$$

Networks of monitored blackboxes (1/2)

$$
\begin{array}{c}
\langle c : P_c : s!c(\text{quit}\langle\rangle)\rangle \xrightarrow{c?s(\text{quit}\langle\rangle)} \langle c : P_c^{\text{q}} : \varepsilon\rangle \xrightarrow{\quad\text{end}\quad} \langle c : P_c^{\text{e}} : \varepsilon\rangle \\
\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\text{quit}\langle\rangle)} \langle s : P_s^{\text{q}} : \varepsilon\rangle \xrightarrow{\quad\text{end}\quad} \langle s : P_s^{\text{e}} : \varepsilon\rangle \\
\hline
\end{array}
$$

$$
\begin{array}{l}
[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\text{quit}\langle\rangle.\text{end}\}\!\} : \varepsilon] \quad | \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\text{quit}\langle\rangle.\text{end}\}\!\} : \varepsilon] \\
\qquad\qquad \downarrow_{\tau} \\
[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\text{quit}\langle\rangle.\text{end}\}\!\} : s!c(\text{quit}\langle\rangle)] \quad | \quad [\langle s : P_s^{\text{q}} : \varepsilon\rangle : \text{end} : \varepsilon] \\
\qquad\qquad \downarrow_{\tau} \\
[\langle c : P_c : s!c(\text{quit}\langle\rangle)\rangle : \text{end} : \varepsilon] \quad | \quad [\langle s : P_s^{\text{q}} : \varepsilon\rangle : \text{end} : \varepsilon]
\end{array}
$$

Networks of monitored blackboxes (1/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \quad | \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]$$

$$\downarrow\tau$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow\tau$$

$$[\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle : \mathsf{end} : \varepsilon] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow\tau$$

$$[\langle c : P_c^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

Networks of monitored blackboxes (1/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \quad \mid \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]$$

$$\downarrow_{\tau}$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad \mid \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow_{\tau}$$

$$[\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle : \mathsf{end} : \varepsilon] \quad \mid \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow_{\tau}$$

$$[\langle c : P_c^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon] \quad \mid \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow_{\mathsf{end}}$$

$$\downarrow_{\mathsf{end}}$$

$$[\langle c : P_c^{\mathsf{e}} : \varepsilon\rangle : \checkmark : \varepsilon] \quad \mid \quad [\langle s : P_s^{\mathsf{e}} : \varepsilon\rangle : \checkmark : \varepsilon]$$

Networks of monitored blackboxes (2/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\qquad\mathsf{end}\qquad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\qquad\mathsf{end}\qquad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$M_c = c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} \qquad\qquad M_s = s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\}$$

Networks of monitored blackboxes (2/2)

$$\frac{\begin{array}{c}\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle \\ \langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\mathsf{end}} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle\end{array}}{[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} : \varepsilon] \quad \mid \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]}$$

Networks of monitored blackboxes (2/2)

$$\frac{\begin{array}{c} \langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle \\ \langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle \end{array}}{\begin{array}{l} [\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} : \varepsilon] \quad | \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon] \\ \qquad\qquad\quad \downarrow\tau \\ [\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon] \end{array}}$$

Networks of monitored blackboxes (2/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\qquad\mathsf{end}\qquad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\qquad\mathsf{end}\qquad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} : \varepsilon] \quad | \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]$$

$$\downarrow\tau$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle \ldots\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow\tau$$

$$\mathsf{error}_c \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

Networks of monitored blackboxes (2/2)

$$\langle c : P_c : s!c(\mathsf{quit}\langle\rangle)\rangle \xrightarrow{c?s(\mathsf{quit}\langle\rangle)} \langle c : P_c^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle c : P_c^{\mathsf{e}} : \varepsilon\rangle$$

$$\langle s : P_s : \varepsilon\rangle \xrightarrow{s!c(\mathsf{quit}\langle\rangle)} \langle s : P_s^{\mathsf{q}} : \varepsilon\rangle \xrightarrow{\quad\mathsf{end}\quad} \langle s : P_s^{\mathsf{e}} : \varepsilon\rangle$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle\ \ldots\}\!\} : \varepsilon] \quad | \quad [\langle s : P_s : \varepsilon\rangle : s!c\{\!\{\mathsf{quit}\langle\rangle.\mathsf{end}\}\!\} : \varepsilon]$$

$$\downarrow\tau$$

$$[\langle c : P_c : \varepsilon\rangle : c?s\{\!\{\mathsf{login}\langle\rangle\ \ldots\}\!\} : s!c(\mathsf{quit}\langle\rangle)] \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow\tau$$

$$\mathsf{error}_c \quad | \quad [\langle s : P_s^{\mathsf{q}} : \varepsilon\rangle : \mathsf{end} : \varepsilon]$$

$$\downarrow\tau$$

$$\mathsf{error}_{c,s}$$

Results

Theorem (Soundness)

If all the monitored blackboxes in a network N *satisfy a protocol* G*, then* N *behaves as specified by* G*.*

Theorem (Transparency)

If a monitored blackbox satisfies a protocol, then it is behaviorally equivalent to its contained blackbox (modulo coordination messages).

Conclusion

- Types for message-passing concurrency
- Session types: A class of behavioral types for communication correctness
- A new framework for runtime verification based on multiparty session types
- More details in our papers; see `www.jperez.nl`

Current and future work:

- Coalgebraic and coinductive approaches to session types
- Session types beyond linear logic / propositions as sessions
- Session-based concurrency implemented on Maude (rewriting logic)

A Logical Basis for the Verification of Message-Passing Programs

Jorge A. Pérez

Fundamental Computing Group
University of Groningen, The Netherlands
https://www.rug.nl/fse/fc

Dutch Formal Methods Day
April 16, 2024

UNIFYING
C•RRECTNESS FOR
C•MMUNICATING
S•FTWARE