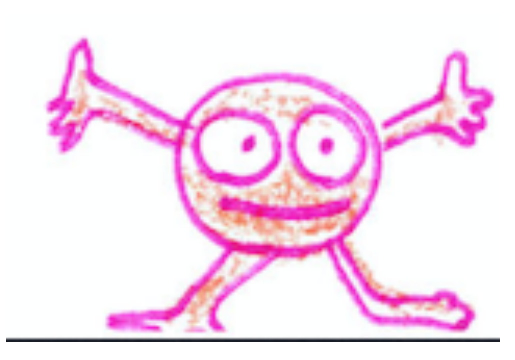
mCRL2 and Multiparty Session Types



Nobuko Yoshida (Oxford)

Multiparty Session Types

Specification-Guided Concurrent and Distributed Programming

1916-1975

- **Christopher Strachey** (sequential computation)
  - *Types* = **abstract** and **digest computation (data types, polymorphism)**
  - **Structured programming = High-level programming**
- **Session types** (concurrency & communication)
  - Structured programming = ***protocols***

mCRL2 and Multiparty Session Types

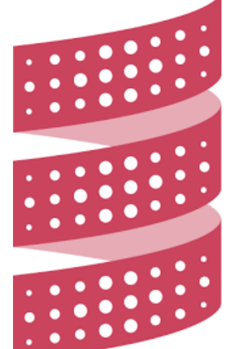
- How can we integrate local verifications (by mCRL2) and global protocol conformance?

- What are possible extensions of local verifications of session types?
  1. probabilities
  2. counter-example guided verification
  3. bisimulation checking
  4. asynchrony
  5. mCRL2 graphics tool

Communications are Ubiquitous

- Increasingly, **communications** are the way to organise software and systems.
- Industry trend – programming languages with **explicit message-passing primitives.**

microservices



Problems: Ambiguity

- Protocol descriptions are **ambiguous**
- **SMTP: simple mail transfer protocol**
    - They are written in English, often very long

RFC 821 August 1982
Simple Mail Transfer Protocol

TABLE OF CONTENTS

1. INTRODUCTION 1
2. THE SMTP MODEL 2
3. THE SMTP PROCEDURE 4
3.1. Mail 4
3.2. Forwarding 7
3.3. Verifying and Expanding 8
3.4. Sending and Mailing 11
3.5. Opening and Closing 13
3.6. Relaying 14
3.7. Domains 17
3.8. Changing Roles 18
4. THE SMTP SPECIFICATIONS 19
4.1. SMTP Commands 19
4.1.1. Command Semantics 19
4.1.2. Command Syntax 27
4.2. SMTP Replies 34
4.2.1. Reply Codes by Function Group 35
4.2.2. Reply Codes in Numeric Order 36
4.3. Sequencing of Commands and Replies 37
4.4. State Diagrams 39
4.5. Details 41
4.5.1. Minimum Implementation 41
4.5.2. Transparency 41
4.5.3. Sizes 42

Problems: Ambiguity

- Protocol descriptions are **ambiguous**
- **SMTP: simple mail transfer protocol**
  - They are written in English, often very long

```
3.1.  MAIL

   There are three steps to SMTP mail transactions.  The transaction
   is started with a MAIL command which gives the sender
   identification.  A series of one or more RCPT commands follows
   giving the receiver information.  Then a DATA command gives the
   mail data.  And finally, the end of mail data indicator confirms
   the transaction.

      The first step in the procedure is the MAIL command.  The
      <reverse-path> contains the source mailbox.

         MAIL <SP> FROM:<reverse-path> <CRLF>

      This command tells the SMTP-receiver that a new mail
      transaction is starting and to reset all its state tables and
      buffers, including any recipients or mail data.  It gives the
      reverse-path which can be used to report errors.  If accepted,
      the receiver-SMTP returns a 250 OK reply.

      The <reverse-path> can contain more than just a mailbox.  The
      <reverse-path> is a reverse source routing list of hosts and
      source mailbox.  The first host in the <reverse-path> should be
      the host sending this command.

      The second step in the procedure is the RCPT command.

         RCPT <SP> TO:<forward-path> <CRLF>

      This command gives a forward-path identifying one recipient.
      If accepted, the receiver-SMTP returns a 250 OK reply, and
      stores the forward-path.  If the recipient is unknown the
      receiver-SMTP returns a 550 Failure reply.  This second step of
      the procedure can be repeated any number of times.
```

Problems: Concurrency Bugs

- Communications increase **concurrency bugs**
  - Survey of 4k users [golang.org]
  - Analysis of 6 large software systems [ASPLOS 19]

The Go Gopher

Do not communicate by sharing memory; share memory by communicating

– Go Philosophy

Problems: Concurrency Bugs

- Communications increase **concurrency bugs**
  - Survey of 4K users **[golang.org]**
  - Analysis of 6 large software systems **[ASPLOS 19]**

    Uber code base [PLDI 2022]

More than a half of concurrency bugs in Go are caused by communications.

channel errors

The Go Gopher

Problems: Concurrency Bugs

- Communications increase **concurrency bugs**
  - Survey of 4k users **[golang.org]**
  - Analysis of 6 large software systems **[ASPLOS 19]**

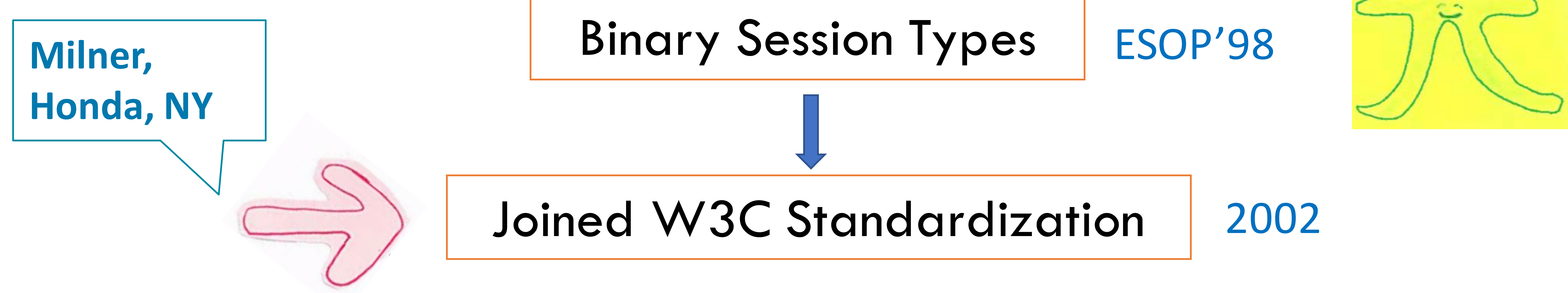
More than a half of concurrency bugs in Go are caused by communications.

Session Types

- Prevent concurrency bugs.
- Can abstract, implement and manage communications as **Protocols.**
- **Clean, Cheap** and **Retrofittable.**

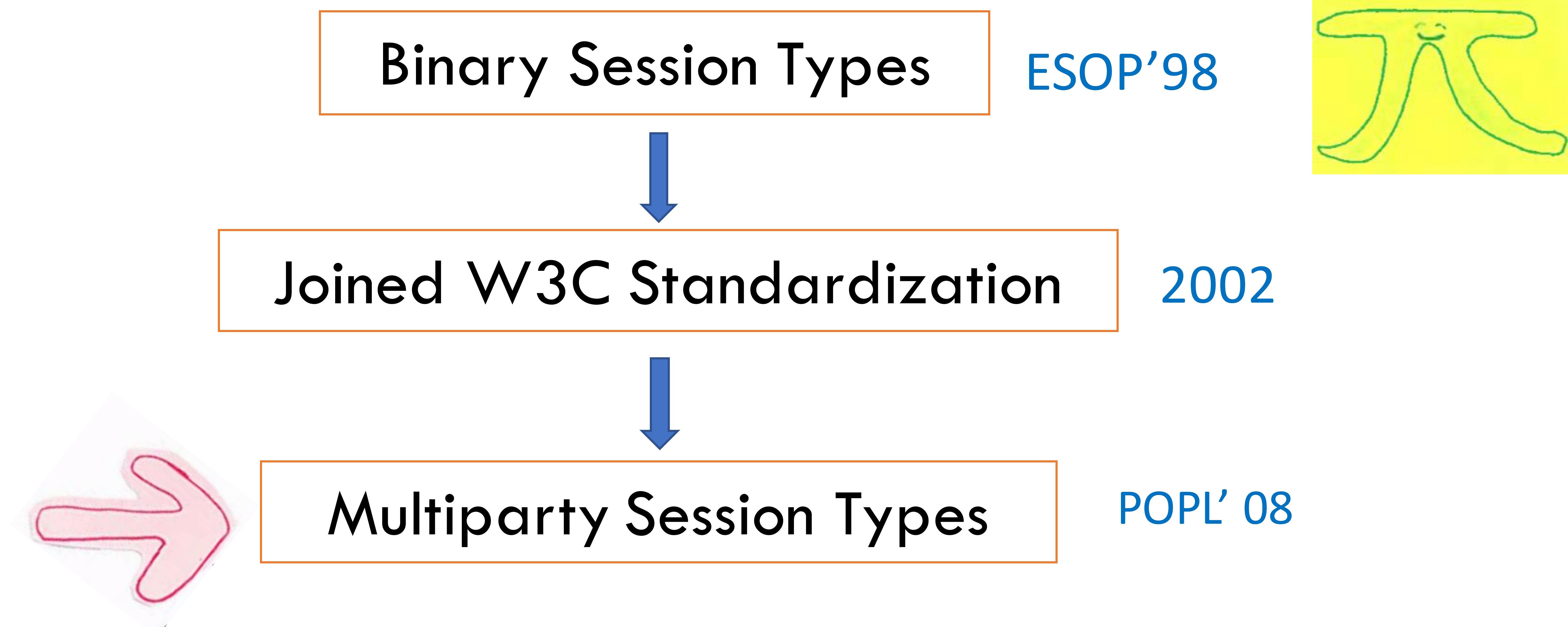
Why Session Types, Why Now?

Significant academic and industry interests via fundamental breakthroughs



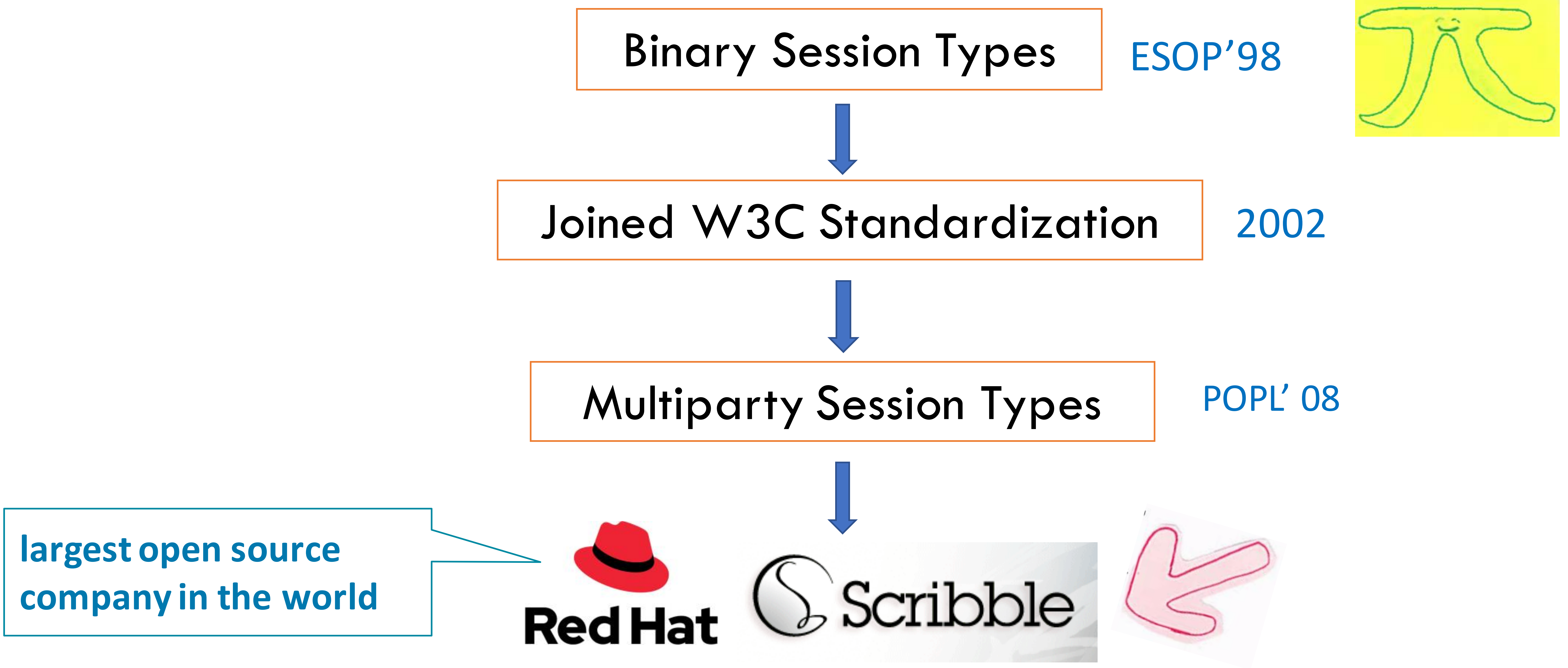
Why Session Types, Why Now?

Significant academic and industry interests via fundamental breakthroughs



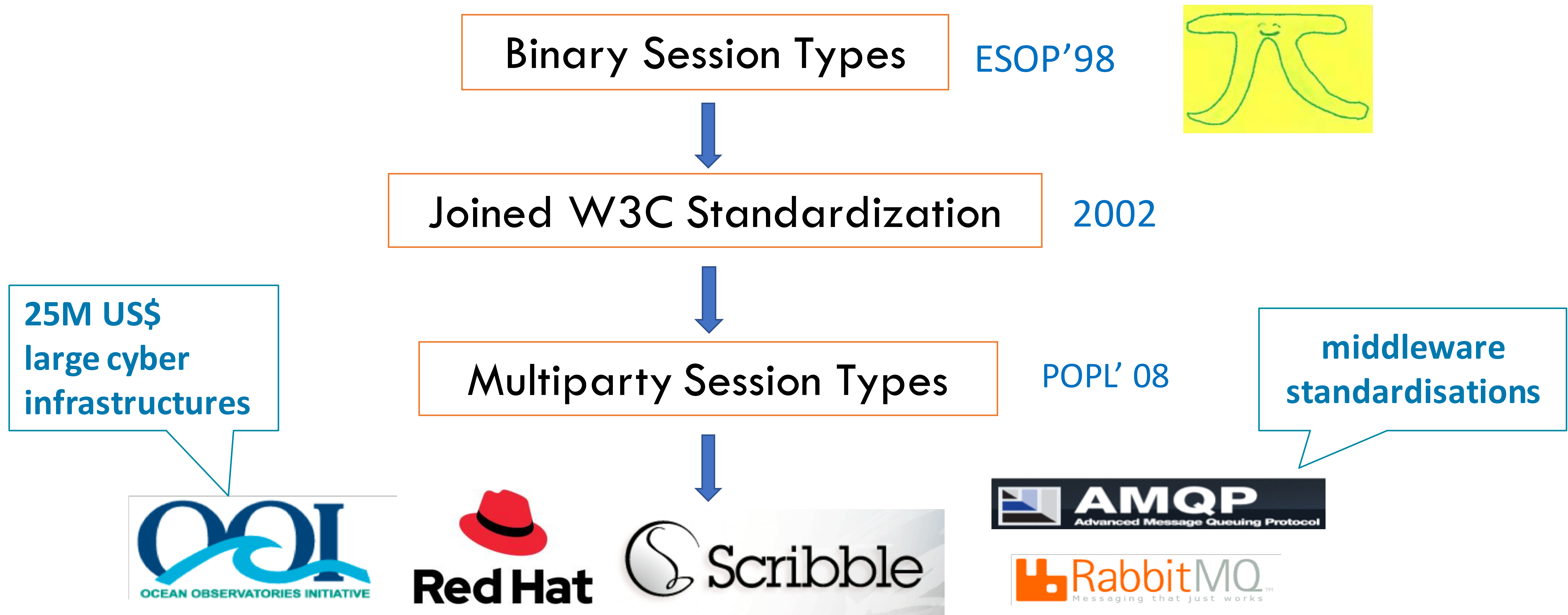
Why Session Types, Why Now?

Significant academic and industry interests via fundamental breakthroughs



Why Session Types, Why Now?

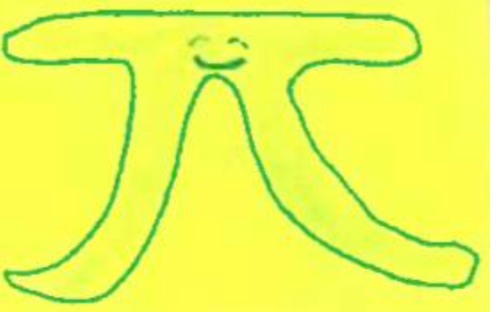
Significant academic and industry interests via fundamental breakthroughs



Why Session Types, Why Now?

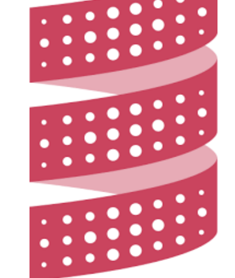
Significant academic and industry interests via fundamental breakthroughs

Binary Session Types — ESOP'98



Joined W3C Standardization — 2002

Multiparty Session Types — POPL' 08



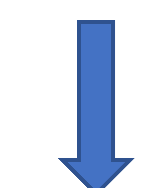
Why Session Types, Why Now?

Significant academic and industry interests via fundamental breakthroughs

ETAPS Test Time Award 2019

Binary Session Types

ESOP'98



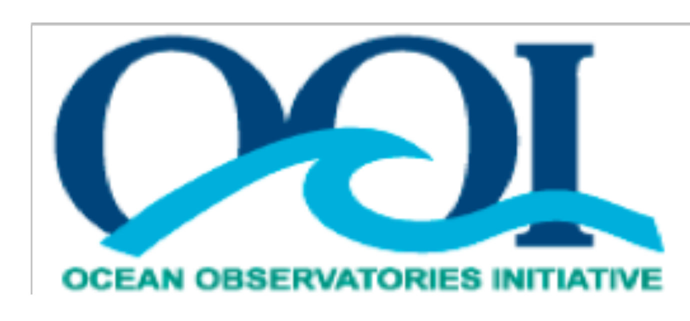
Joined W3C Standardization

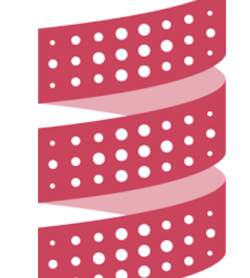
2002

POPL Influential Paper Award 2018

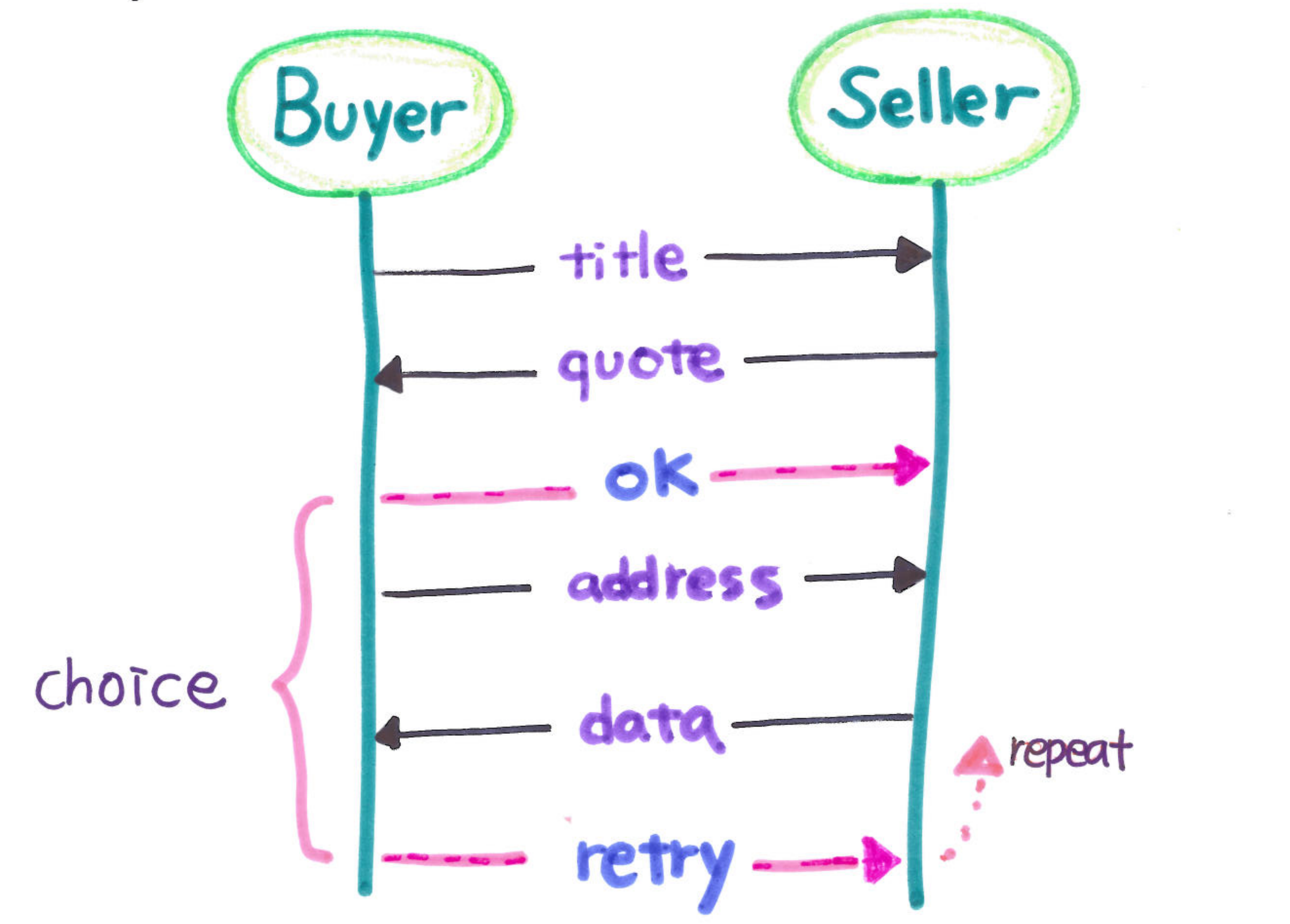
Multiparty Session Types

POPL' 08

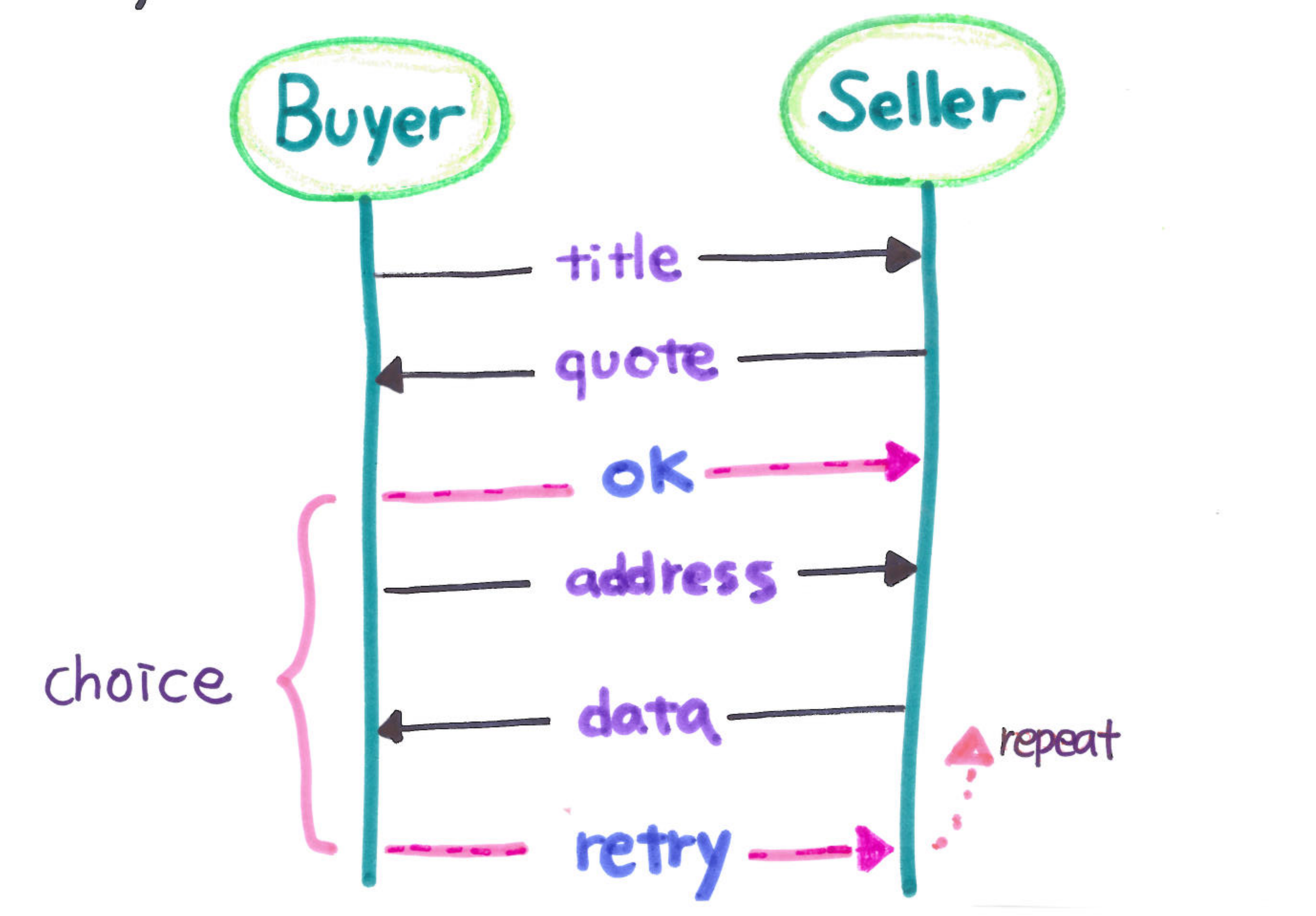




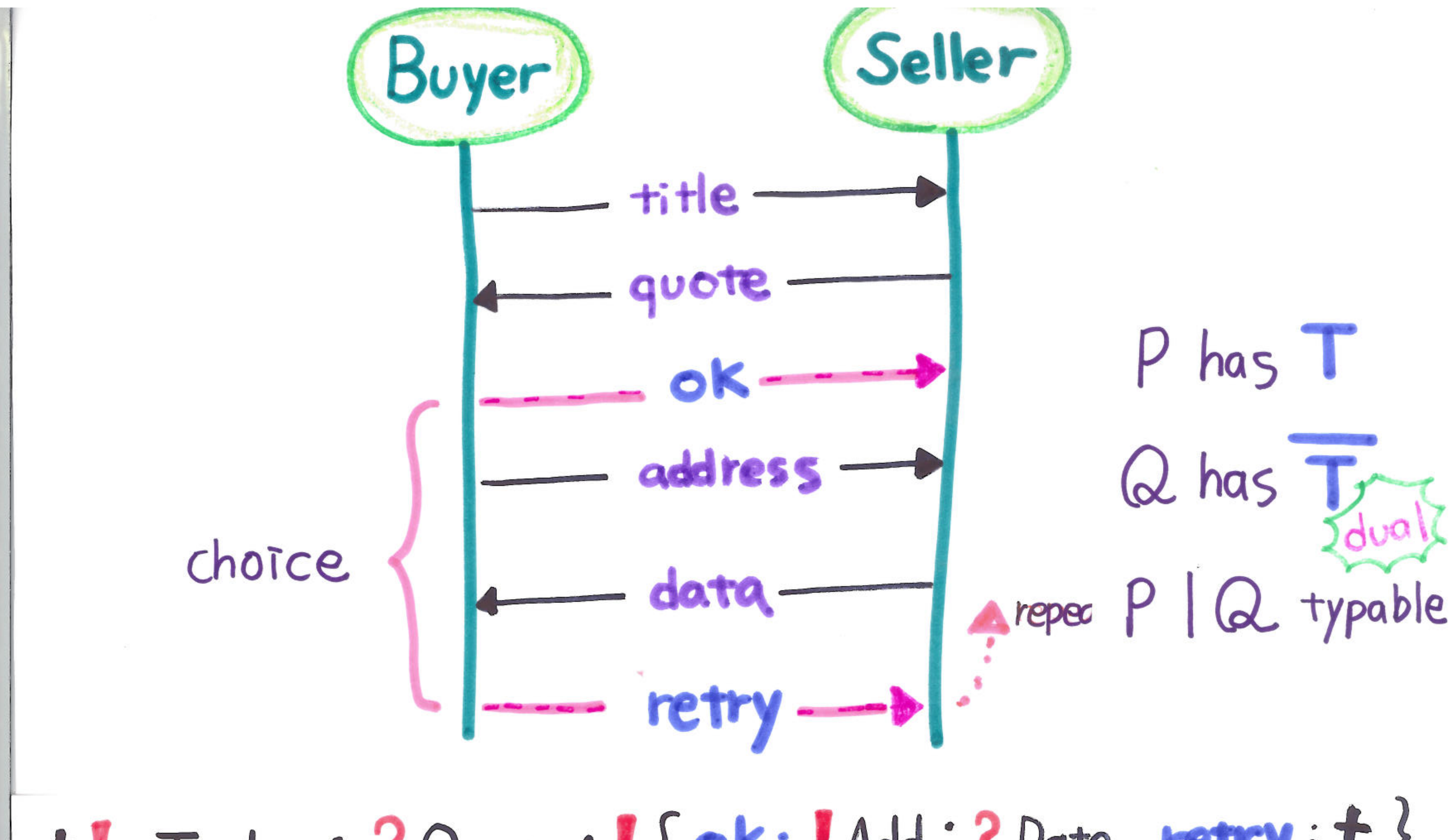
Binary Session Types: Buyer-Seller Protocol



Binary Session Types: Buyer-Seller Protocol



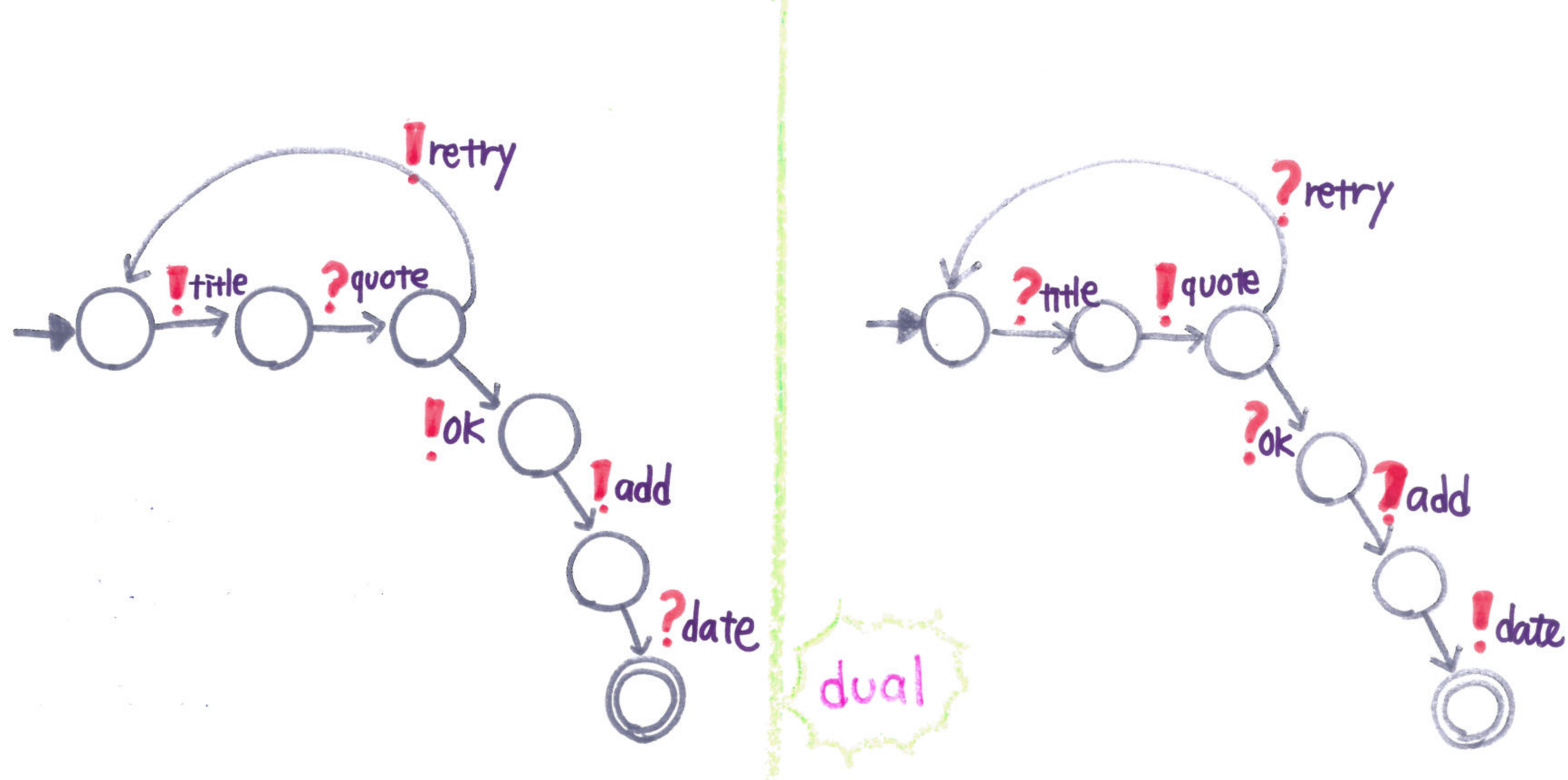
$\mu t.\,!\,\mathrm{Title}\,;\,?\,\mathrm{Quote}\,;\,!\,\{\mathrm{ok} : !\,\mathrm{Add}\,;\,?\,\mathrm{Date},\ \mathrm{retry} : t\,\}$

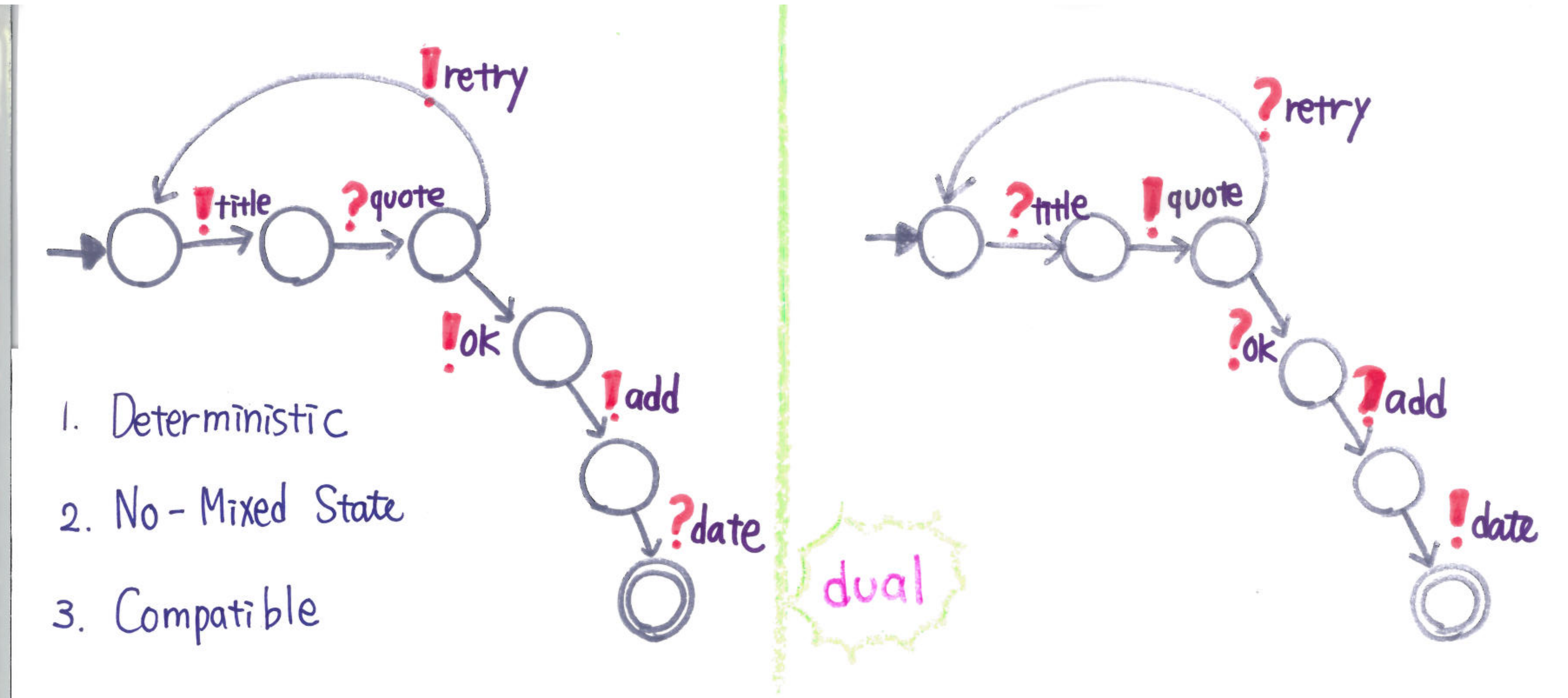


$\mu t.\,!\,\mathrm{Title}\,;\,?\,\mathrm{Quote}\,;\,!\,\{\mathbf{ok}:\,!\,\mathrm{Add}\,;\,?\,\mathrm{Date},\ \mathbf{retry}:\mathbf{t}\,\}$

$\mu t\,?\,\mathrm{Title}\,;\,!\,\mathrm{Quote}\,;\,?\,\{\mathbf{ok}:\,?\,\mathrm{Add}\,;\,!\,\mathrm{Date},\ \mathbf{retry}:\mathbf{t}\,\}$

Communicating Automata [1980s]





[Gouda et al 1986] Two compatible machines without mixed states which are deterministic satisfy deadlock-freedom.

Multiparty Session Types

Buyer 1

Seller

Buyer 2

title

quote

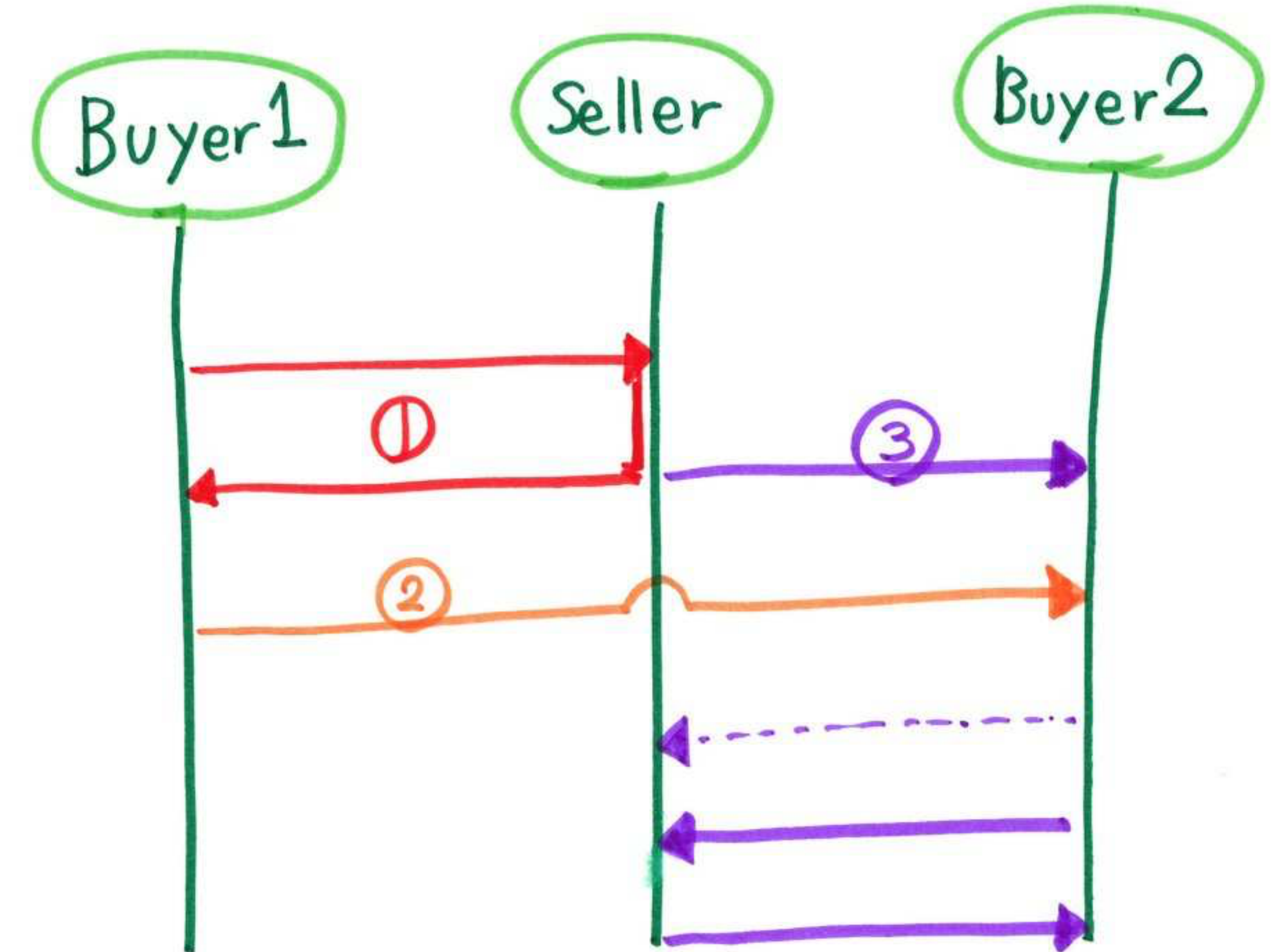
quote

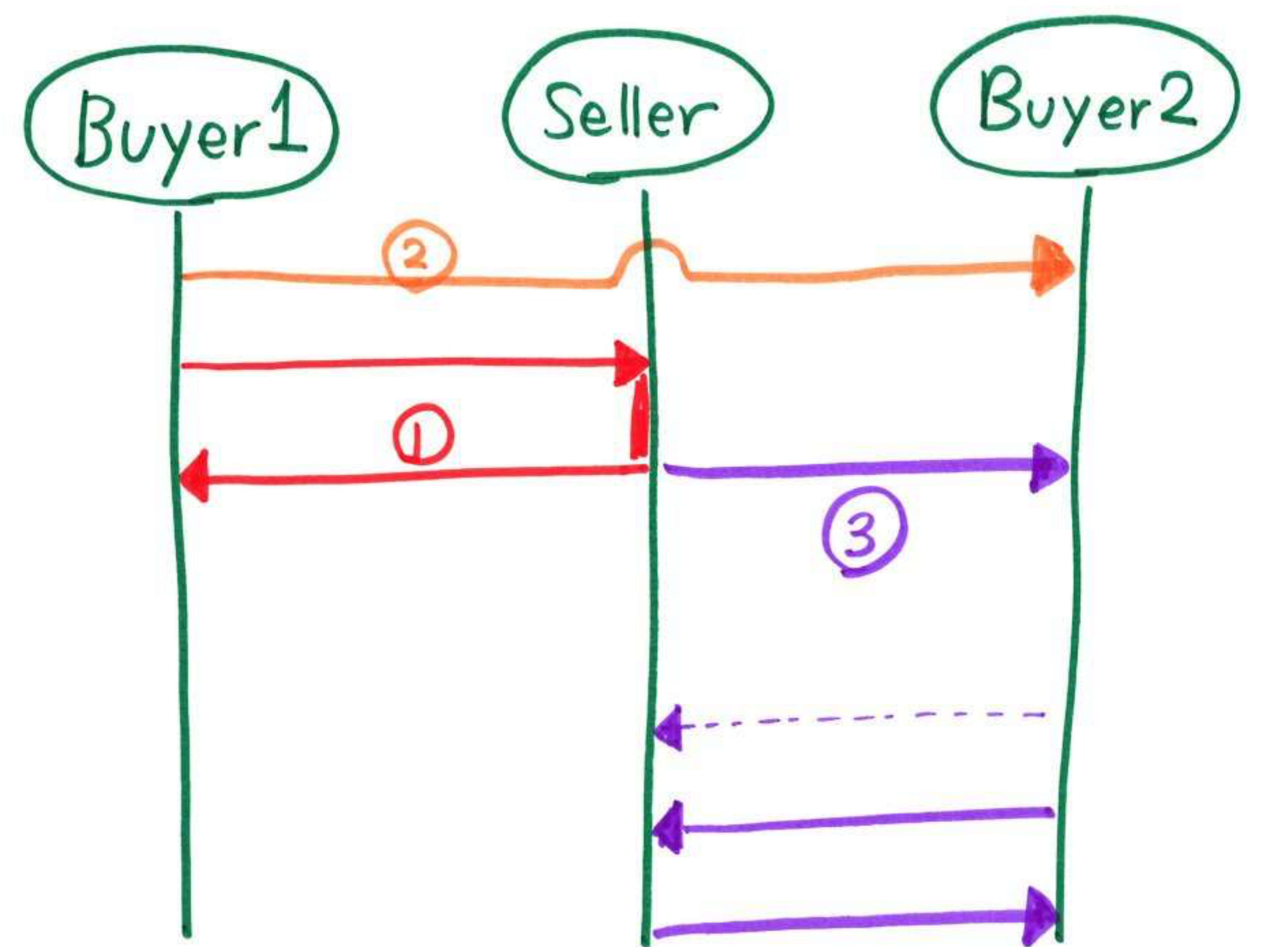
quote ÷ 2

ok

address

date


Buyer1
Seller
Buyer2
①
③
②


Buyer1
Seller
Buyer2
2
1
3

Multiparty Session Types

Buyer 1

Seller

Buyer 2

title

quote

quote

quote ÷ 2

ok

address

date

Multiparty Session Types [Honda, Yoshida, Carbone 2008]

B1 S B2

G

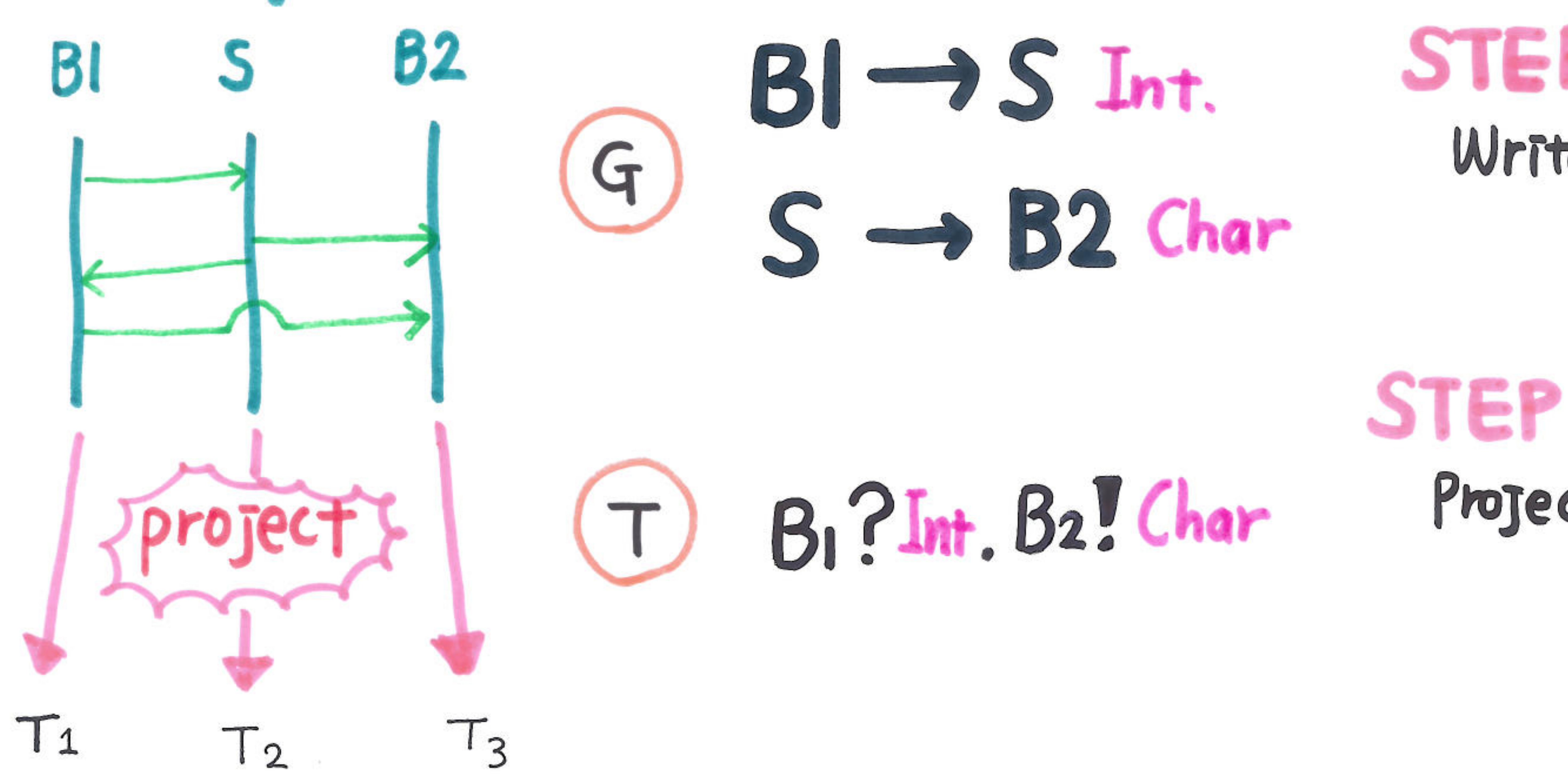
B1 → S Int.

S → B2 Char

STEP 1

Write Global Type

Multiparty Session Types [Honda, Yoshida, Carbone 2008]



STEP 1

Write Global Type

STEP 2

Project to Local Types

Multiparty Session Types [Honda, Yoshida, Carbone 2008]

B1 S B2

project

T_1 T_2 T_3

P_1 P_2 P_3

(G) B1 → S Int.
S → B2 Char

(T) B_1?Int. B_2!Char

(P) B_1?(x). B_2!⟨"apple"⟩

STEP 1

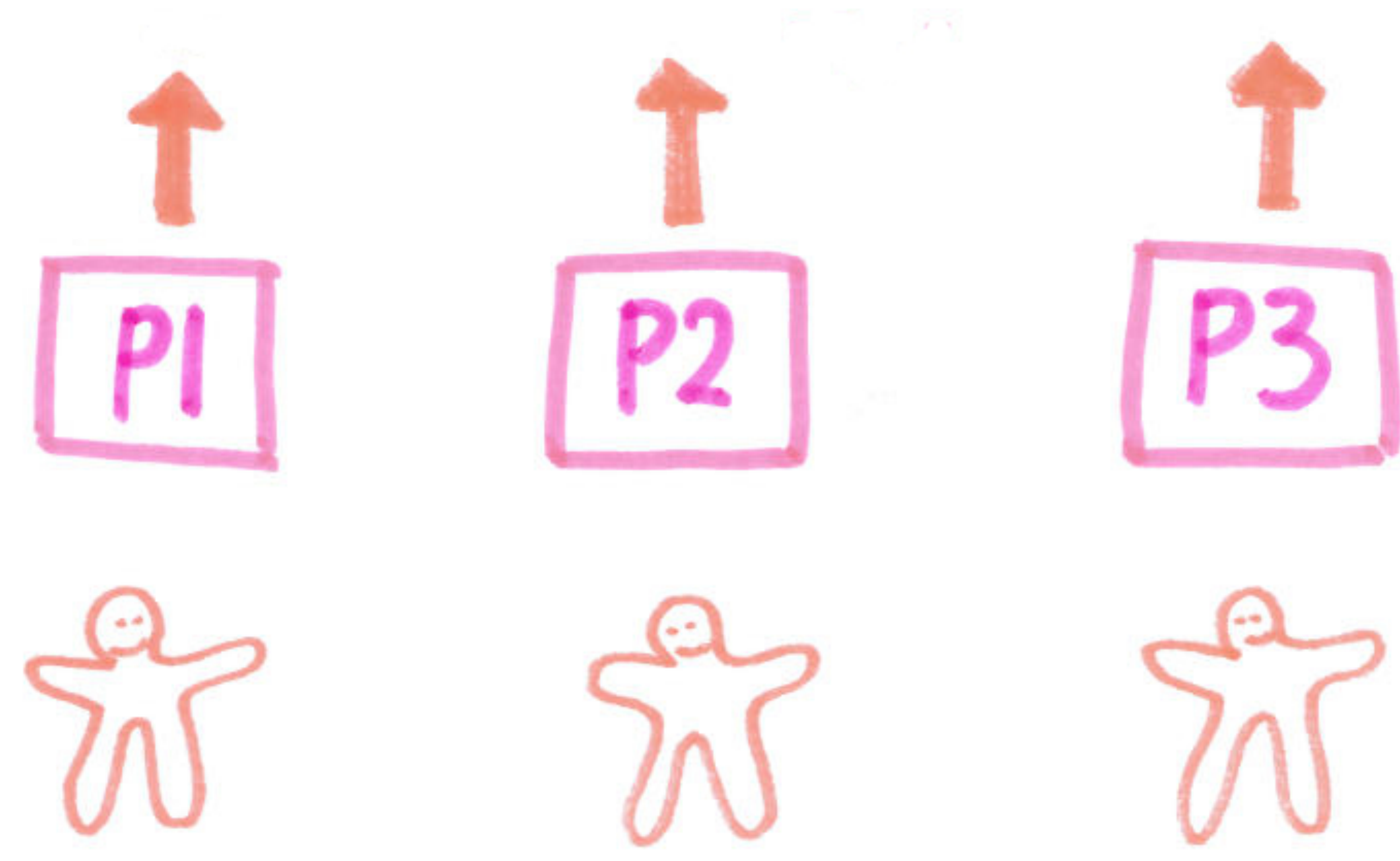
Write Global Type

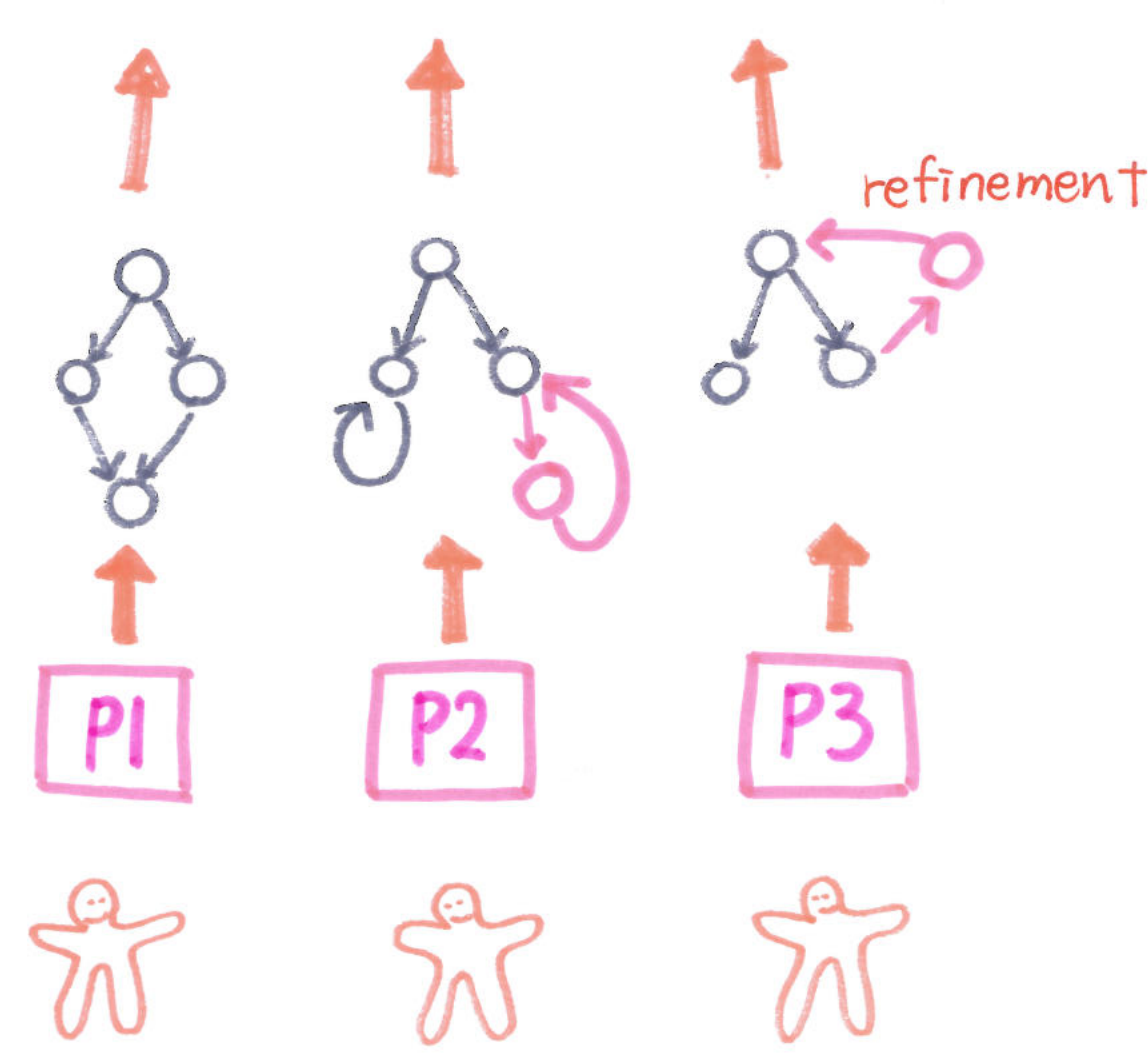
STEP 2

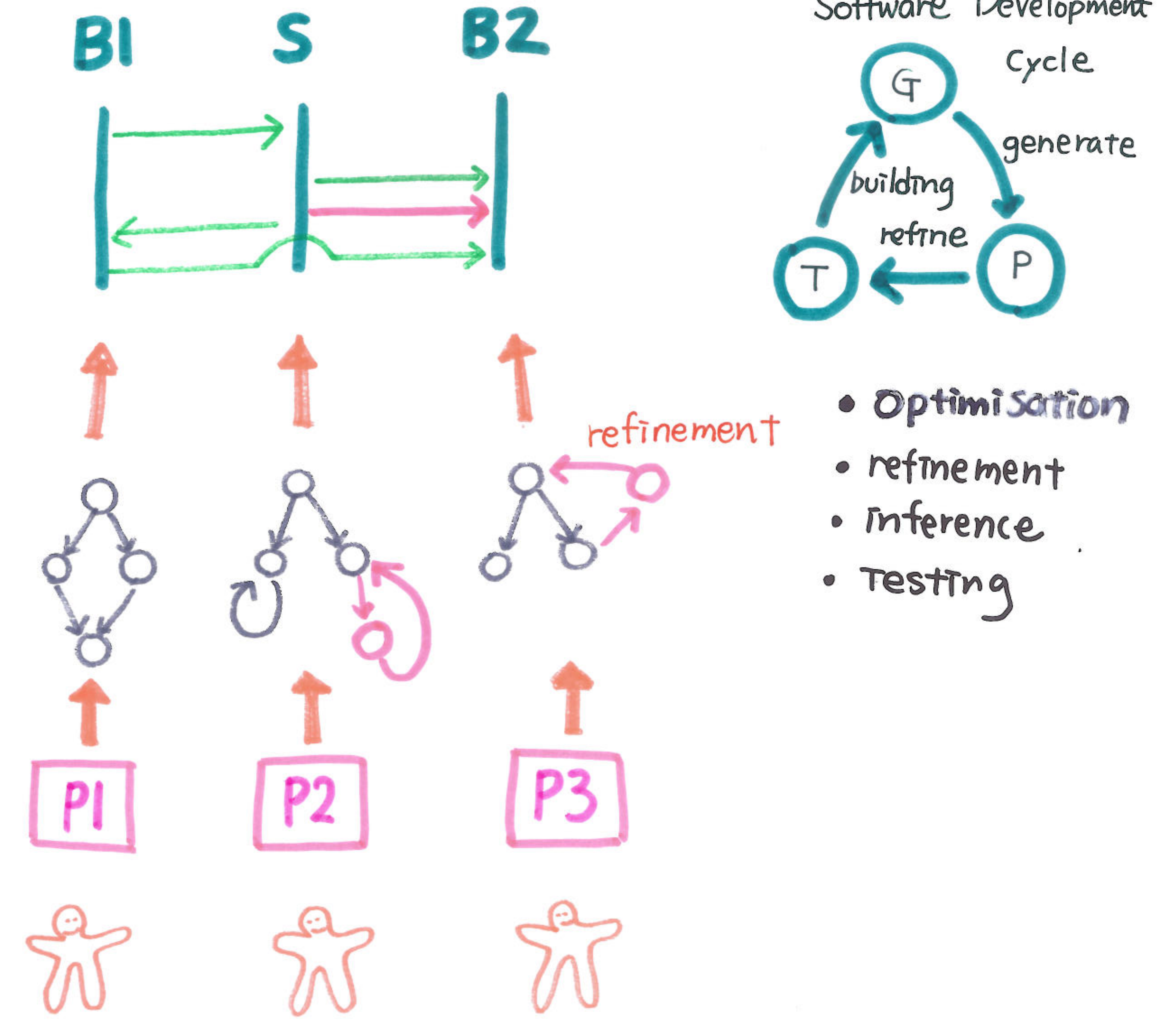
Project to Local Type

STEP 3

- Static Check
- Generate Code
- Run-time check

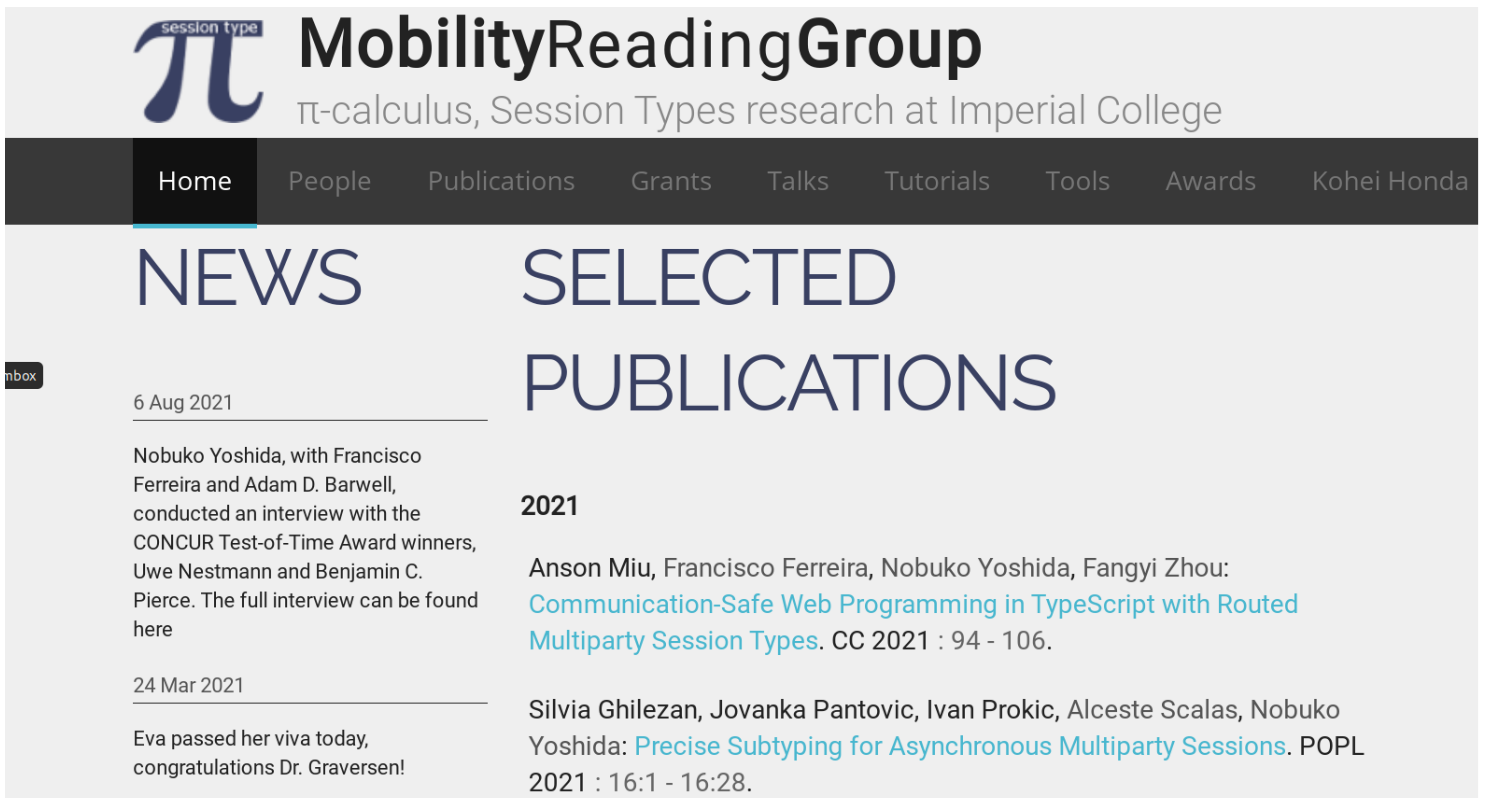

P1
P2
P3


refinement
P1
P2
P3


B1
S
B2
Software Development Cycle
G
generate
building
refine
T
P
refinement
• Optimisation
• refinement
• inference
• Testing
P1
P2
P3

Mobility Reading Group

http://mrg.doc.ic.ac.uk/



History of Session Types with mCRL2

- [TACAS'17] Using mCRL2 to benchmark **session subtyping** checking
- [POPL'17] Analysing properties (e.g., deadlock-freedom) of **Go** programs
- [ICSE'18] Analysing properties of **message-passing Go** programs
- [POPL'19] Analysing properties of **synchronous multiparty session pi-calculus**
- [PLDI'19] Analysing properties of **distributed Scala programs**
- [ECOOP'20] Analysing properties of **shared memory Go** programs
- [CONCUR'22] Extension of [POPL'19] with **Stop-Failure**

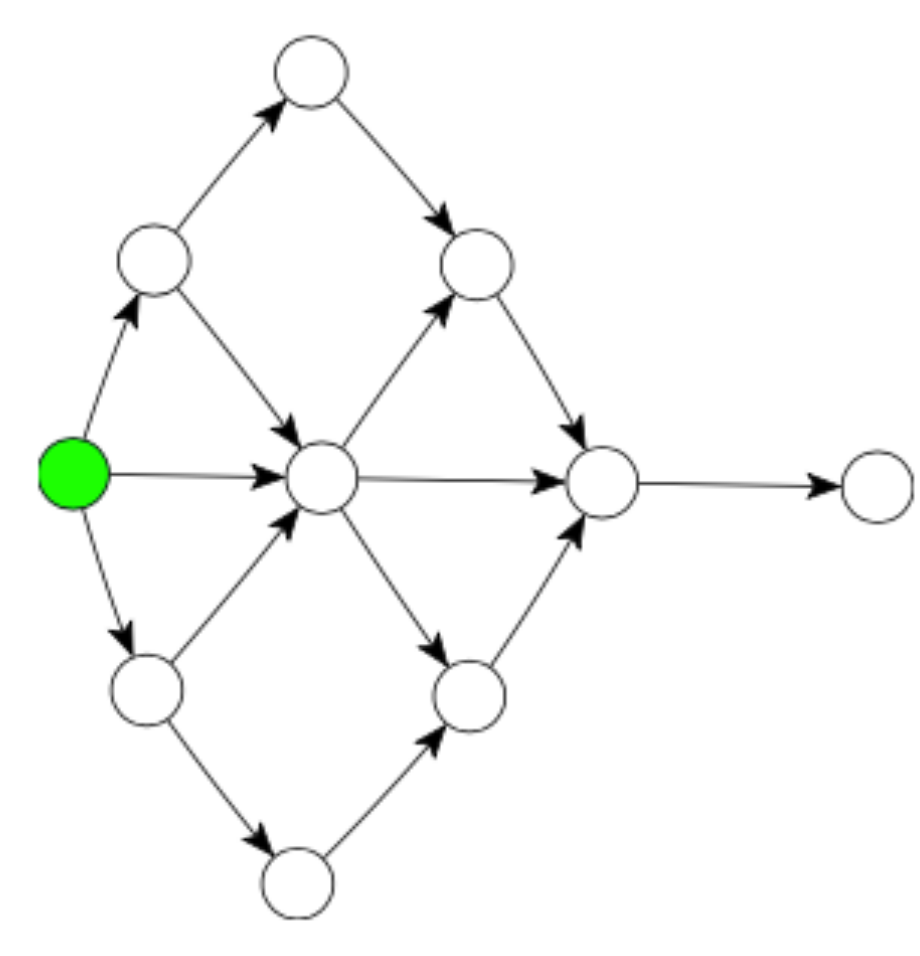
History of Session Types with mCRL2

- [TACAS'17] Using mCRL2 to benchmark **session subtyping** checking
- [POPL'17] Analysing properties (e.g., deadlock-freedom) of **Go** programs
- [ICSE'18] Analysing properties of **message-passing Go** programs
- [POPL'19] Analysing properties of **synchronous multiparty session pi-calculus**
- [PLDI'19] Analysing properties of **distributed Scala programs**
- [ECOOP'20] Analysing properties of **shared memory Go** programs
- [CONCUR'22] Extension of [POPL'19] with **Stop-Failure**

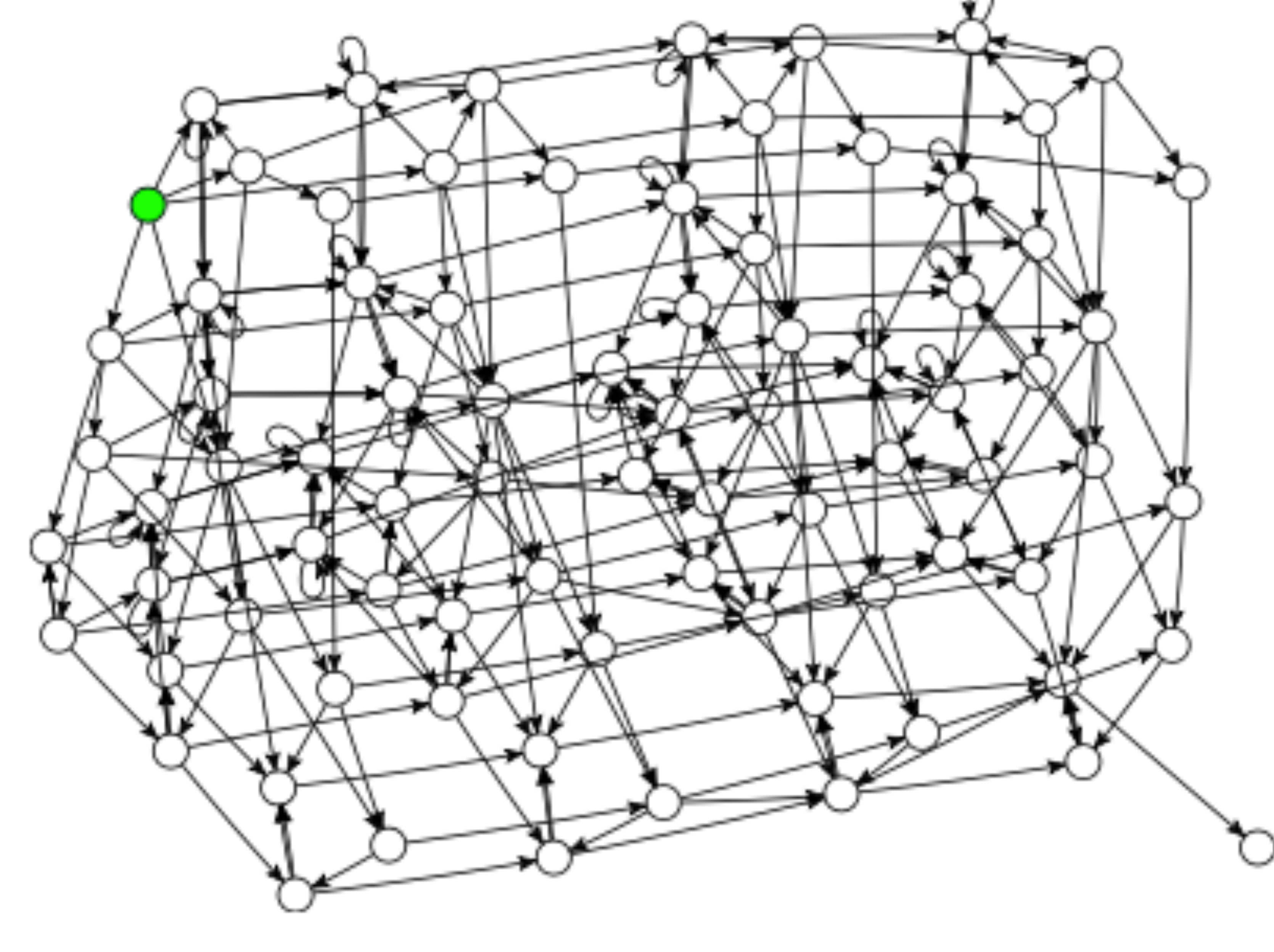
No gap between type and process properties

Why mCRL2?

- Can analyse properties than the top-down approach
- Can scale

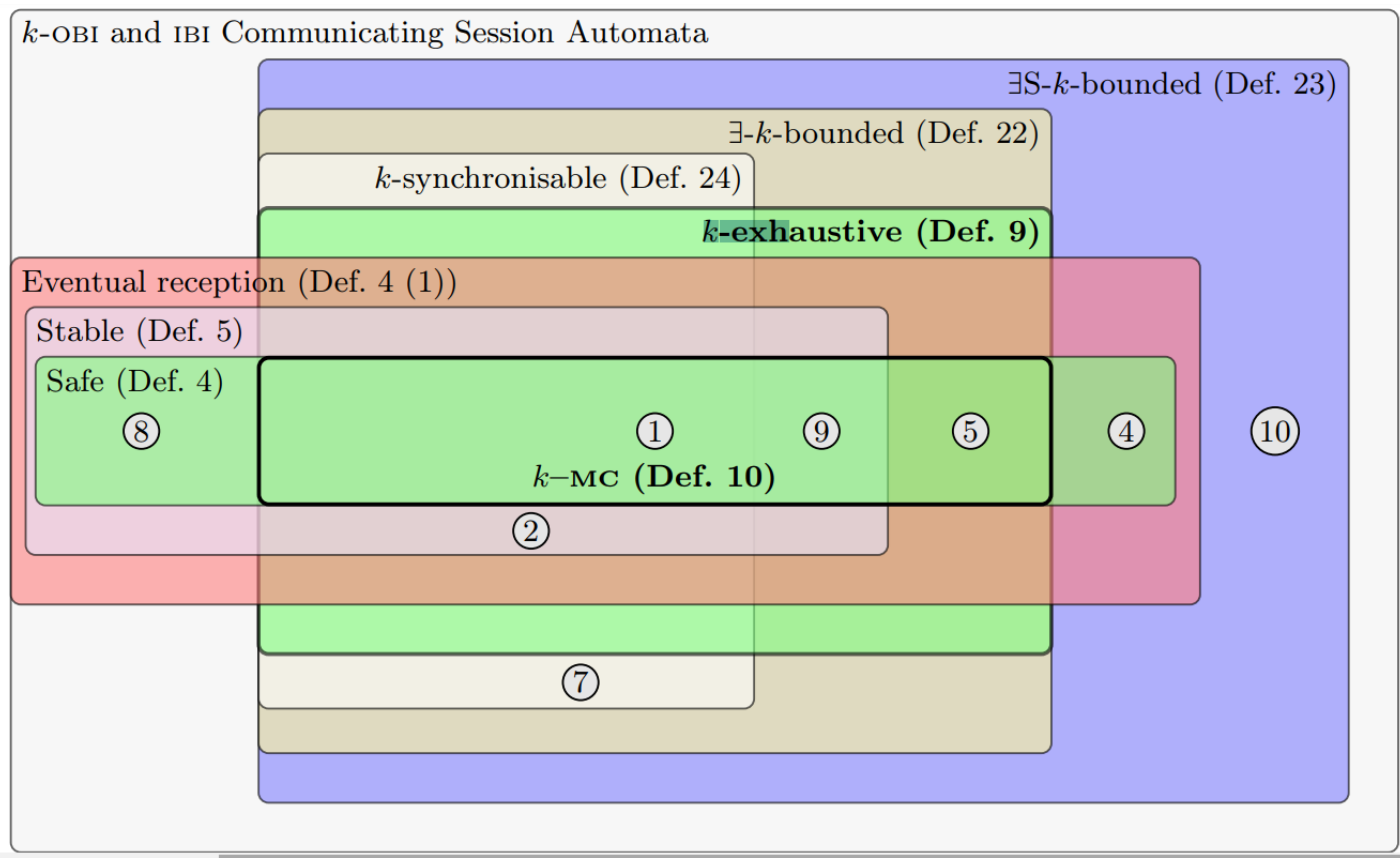


No Failure



with Failure

k-Multiparty Compatibility [CAV'19]

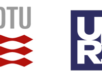


Generalised Multiparty Session Types with Crash-Stop Failures

Adam D. Barwell[1] Alceste Scalas[2] Nobuko Yoshida[1] Fangyi Zhou[1]

[1]Imperial College London
[2]DTU Compute – Technical University of Denmark

33rd International Conference on Concurrency Theory
16 September 2022



The Case For Session Types

- » Formally describe **communications behaviour** between two or more systems
- » Communications behaviour is **enforced statically**
- » **Guarantee** desirable behavioural properties
  - Deadlock-Freedom
  - Liveness
  - Termination
  - Never-Termination
- » Libraries permit use with **real-world languages**
  - Rust, Haskell, Scala, Erlang, OCaml, Java, Go, Typescript, &c.

The Case Against Session Types

- **Limited practicality** in the real-world
- Basic assumptions **restrict** what we can express and guarantee
- One notable example: **fault-tolerant protocols**

Fault-Tolerance or Lack Thereof

- Most MPST systems assume a **perfect world**
  - No process failures
  - No message loss, duplication, or corruption

Fault-Tolerance or Lack Thereof

- Most MPST systems assume a **perfect world**
  - No process failures
  - No message loss, duplication, or corruption
- **Reality** tends to disagree
  - Things break, unfortunately

Fault-Tolerance or Lack Thereof

- Most MPST systems assume a **perfect world**
  - No process failures
  - No message loss, duplication, or corruption
- **Reality** tends to disagree
  - Things break, unfortunately

Aim

Represent fault-tolerant protocols.

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

» **Crash-stop failures**: our failure model; provides failures to tolerate

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

- » **Crash-stop failures**: our failure model; provides failures to tolerate
- » **Generalised MPST**: our type system is parametric on a safety invariant
  - – Safety can be a behavioural property (e.g. deadlock-freedom)
  - – Enables integration with model checking

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

- **Crash-stop failures**: our failure model; provides failures to tolerate
- **Generalised MPST**: our type system is parametric on a **safety invariant**
  - Safety can be a behavioural property (e.g. deadlock-freedom)
  - Enables integration with model checking

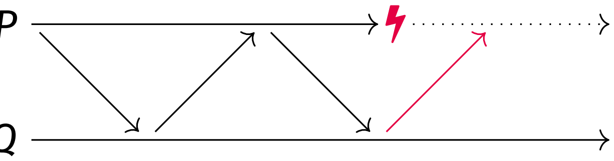
Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

1. Session π-calculus with crash-stop failures
2. Multiparty session types with crashes
3. Type-level model checking
4. Optional reliability assumptions

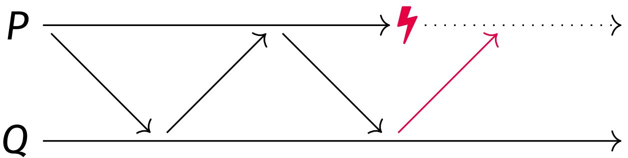
Crash-Stop Failures

- Processes can **crash arbitrarily**
- Crashed processes **make no progress**
- Crashed processes **do not recover**
- Communications channels deliver messages in order and without losses



Crash-Stop Failures

- Processes can **crash arbitrarily**
- Crashed processes **make no progress**
- Crashed processes **do not recover**
- Communications channels deliver messages in order and without losses



- How do we represent crash-stop failures in our π-calculus and types?

A Session π-Calculus with Crash-Stop Failures

'act as participant $\mathtt{p}$ in session s'

$c ::= x \mid s[\mathtt{p}]$		Variables and session endpoints
$d ::= v \mid c$		Basic value, variable, or session endpoint
$P, Q ::= \mathbf{0} \mid (\nu s)\, P \mid P \mid Q$		Inaction, restriction, and parallel composition
$\mid c[\mathtt{q}] \oplus \mathtt{m}\langle d \rangle . P$	(where $\mathtt{m} \neq$ **crash**)	Send to participant $\mathtt{q}$ as c
$\mid c[\mathtt{q}] \& \{\mathtt{m}_i(x_i) . P_i\}_{i \in I}$		Receive from participant $\mathtt{q}$ as c
$\mid s[\mathtt{p}]\lightning$		Crashed session endpoint

- $s[\mathtt{p}]\lightning$ represents that participant $\mathtt{p}$ has crashed in session s
- **crash** is a reserved label that cannot be sent

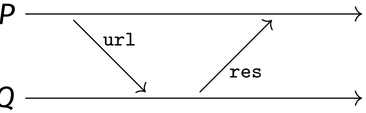
Some standard constructs have been omitted for clarity of presentation.

A Simple DNS Protocol

A client P queries DNS server Q for an IP address.

$$P \quad = \quad s[\mathrm{p}][\mathrm{q}] \oplus \mathtt{url}.s[\mathrm{p}][\mathrm{q}] \& \mathtt{res}.\mathbf{0}$$

$$Q \quad = \quad s[\mathrm{q}][\mathrm{p}] \& \mathtt{url}.s[\mathrm{q}][\mathrm{p}] \oplus \mathtt{res}.\mathbf{0}$$



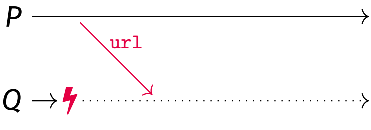
A Simple DNS Protocol

A client P queries DNS server Q for an IP address.

$$P = s[p][q] \oplus \texttt{url}.s[p][q] \& \texttt{res}.\mathbf{0}$$

$$Q = s[q] ↯$$

How does P handle Q's crash?



Handling Crashes

- Receive operations may contain a **crash-handling branch**

$$s[\mathrm{p}][\mathrm{q}]\&\{\cdots,\ \mathbf{crash}.k\}$$

- Each process has a **failure detector**
  - Strong completeness & accuracy: processes detected to have crashed must have crashed
  - Exact method is abstract; a variety to choose from: timeouts, heartbeat, &c.
  - Relaxing this restriction is future work
- A crash-handling branch is taken when the sender is detected to have crashed
  - Crashes are handled locally
  - Failures are not implicitly broadcast to peers

Adding Crash-Handling Behaviour to the DNS Protocol

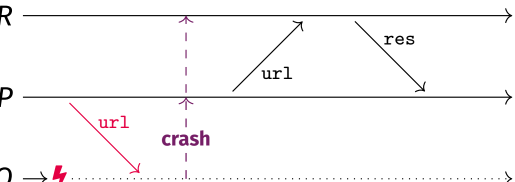
Fail-over server

Activates only when q is detected to have crashed

$$R = s[r][q] \& \mathbf{crash}.s[r][p] \& \texttt{url}.s[r][p] \oplus \texttt{res}.\mathbf{0}$$

$$P = s[p][q] \oplus \texttt{url}.s[p][q] \& \left\{ \begin{array}{l} \texttt{res}.\mathbf{0} \\ \mathbf{crash}.s[p][r] \oplus \texttt{url}.s[p][r] \& \texttt{res}.\mathbf{0} \end{array} \right\}$$

$$Q = s[q] \lightning$$



Multiparty Session Types with Crash-Stop Failures

T	$::=$	$\mathtt{p}\oplus\{\mathtt{m_i}(S_i).T_i\}_{i\in I}$	Send to participant $\mathtt{p}$
	$\mid$	$\mathtt{p}\&\{\mathtt{m_i}(S_i).T_i\}_{i\in I}$	Receive from participant $\mathtt{p}$
	$\mid$	$\mu\mathbf{t}.T \quad\mid\quad \mathbf{t} \quad\mid\quad \mathsf{end}$	Recursion, type variable, and termination
U	$::=$	T	Pure session type
	$\mid$	stop	Crash type

- Session types, U, are assigned to session endpoints
- stop represents a crashed endpoint; a **runtime type**
- **crash** is a reserved label, denotes crash-handling branches

---

Basic and payload types have been omitted for clarity of presentation.

Adding Session Types to the DNS Protocol

$$R \quad = \quad s[\mathtt{r}][\mathtt{q}]\&\textbf{crash}.s[\mathtt{r}][\mathtt{p}]\&\mathtt{url}.s[\mathtt{r}][\mathtt{p}]\oplus\mathtt{res}.\mathbf{0}$$

$$P \quad = \quad s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{q}]\&\left\{\begin{array}{l}\mathtt{res}.\mathbf{0}\\ \textbf{crash}.s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{r}]\&\mathtt{res}.\mathbf{0}\end{array}\right\}$$

$$Q \quad = \quad s[\mathtt{q}]\lightning$$

Adding Session Types to the DNS Protocol

$$s[\mathtt{r}] \quad : \quad \mathtt{q}\&\mathbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

$$R \quad = \quad s[\mathtt{r}][\mathtt{q}]\&\mathbf{crash}.s[\mathtt{r}][\mathtt{p}]\&\mathtt{url}.s[\mathtt{r}][\mathtt{p}]\oplus\mathtt{res}.\mathbf{0}$$

$$P \quad = \quad s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{q}]\&\left\{\begin{array}{l}\mathtt{res}.\mathbf{0} \\ \mathbf{crash}.s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{r}]\&\mathtt{res}.\mathbf{0}\end{array}\right\}$$

$$Q \quad = \quad s[\mathtt{q}]\lightning$$

Adding Session Types to the DNS Protocol

$$s[\mathtt{r}] \; : \; \mathtt{q}\&\mathbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

$$R = s[\mathtt{r}][\mathtt{q}]\&\mathbf{crash}.s[\mathtt{r}][\mathtt{p}]\&\mathtt{url}.s[\mathtt{r}][\mathtt{p}]\oplus\mathtt{res}.\mathbf{0}$$

$$s[\mathtt{p}] \; : \; \mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end}\\ \mathbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$P = s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{q}]\&\left\{\begin{array}{l}\mathtt{res}.\mathbf{0}\\ \mathbf{crash}.s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{r}]\&\mathtt{res}.\mathbf{0}\end{array}\right\}$$

$$Q = s[\mathtt{q}]\lightning$$

Adding Session Types to the DNS Protocol

$$s[\mathtt{r}] \quad : \quad \mathtt{q}\&\textbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

$$R \quad = \quad s[\mathtt{r}][\mathtt{q}]\&\textbf{crash}.s[\mathtt{r}][\mathtt{p}]\&\mathtt{url}.s[\mathtt{r}][\mathtt{p}]\oplus\mathtt{res}.\mathbf{0}$$

$$s[\mathtt{p}] \quad : \quad \mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end}\\ \textbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$P \quad = \quad s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{q}]\&\left\{\begin{array}{l}\mathtt{res}.\mathbf{0}\\ \textbf{crash}.s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{url}.s[\mathtt{p}][\mathtt{r}]\&\mathtt{res}.\mathbf{0}\end{array}\right\}$$

$$s[\mathtt{q}] \quad : \quad \mathsf{stop}$$

$$Q \quad = \quad s[\mathtt{q}]\lightning$$

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

- ✓ Session π-calculus with crash-stop failures
- ✓ Multiparty session types with crashes
- $\Rightarrow$ Type-level model checking for behavioural properties

3. Optional reliability assumptions

Desirable Behavioural Properties

Deadlock-Freedom

Lack of progress implies termination.

Liveness

All communications eventually occur.

Terminating

The protocol will eventually cease.

Never-Terminating

The protocol will continue for all eternity.

Desirable Behavioural Properties

Deadlock-Freedom

Lack of progress implies termination.

Liveness

All communications eventually occur.

Terminating

The protocol will eventually cease.

Never-Terminating

The protocol will continue for all eternity.

» All examples of **safety properties**

Desirable Behavioural Properties

Deadlock-Freedom

Lack of progress implies termination.

Liveness

All communications eventually occur.

Terminating

The protocol will eventually cease.

Never-Terminating

The protocol will continue for all eternity.

- All examples of **safety properties**
- Abstract properties to one general **safety invariant**, φ

Desirable Behavioural Properties

Deadlock-Freedom

Lack of progress implies termination.

Liveness

All communications eventually occur.

Terminating

The protocol will eventually cease.

Never-Terminating

The protocol will continue for all eternity.

- All examples of **safety properties**
- Abstract properties to one general **safety invariant**, φ
- Typing rules are **parameterised** by φ
  - Enables checking different properties

Desirable Behavioural Properties

Deadlock-Freedom

Lack of progress implies termination.

Liveness

All communications eventually occur.

Terminating

The protocol will eventually cease.

Never-Terminating

The protocol will continue for all eternity.

- All examples of **safety properties**
- Abstract properties to one general **safety invariant**, φ
- Typing rules are **parameterised** by φ
  - Enables checking different properties
- We check that safety holds inspecting **typing contexts**, Γ

Guaranteeing Safety

Ignore $\mathcal{R}$ for now, we'll come back to it later.

We say that φ is an $(s; \mathcal{R})$-*safety property* of typing contexts iff, whenever $\varphi(\Gamma)$, we have:

Communication is possible if $\mathtt{p}$ and $\mathtt{q}$ try to communicate.

$$[\text{S-}\oplus\&] \quad \Gamma \xrightarrow{s[\mathtt{p}]:\mathtt{q}\oplus\mathtt{m}(S)} \text{ and } \Gamma \xrightarrow{s[\mathtt{q}]:\mathtt{p}\&\mathtt{m}'(S')} \text{ implies } \Gamma \xrightarrow{s[\mathtt{p}][\mathtt{q}]\mathtt{m}};$$

Guaranteeing Safety

Ignore $\mathcal{R}$ for now, we'll come back to it later.

We say that φ is an $(s; \mathcal{R})$-*safety property* of typing contexts iff, whenever $\varphi(\Gamma)$, we have:

Communication is possible if p and q try to communicate.

[S-⊕&] $\Gamma \xrightarrow{s[p]:q\oplus m(S)}$ and $\Gamma \xrightarrow{s[q]:p\& m'(S')}$ implies $\Gamma \xrightarrow{s[p][q]m}$;

q has a crash handling branch if it receives from an unreliable p.

[S-↯&] $\Gamma \xrightarrow{s[p]\text{stop}}$ and $\Gamma \xrightarrow{s[q]:p\& m(S)}$ implies $\Gamma \xrightarrow{s[q]\odot p}$;

Guaranteeing Safety

Ignore $\mathcal{R}$ for now, we'll come back to it later.

We say that φ is an $(s;\mathcal{R})$-*safety property* of typing contexts iff, whenever $\varphi(\Gamma)$, we have:

Communication is possible if $\mathtt{p}$ and $\mathtt{q}$ try to communicate.

[S-$\oplus$&] $\Gamma \xrightarrow{s[\mathtt{p}]:\mathtt{q}\oplus\mathtt{m}(S)}$ and $\Gamma \xrightarrow{s[\mathtt{q}]:\mathtt{p}\&\mathtt{m}'(S')}$ implies $\Gamma \xrightarrow{s[\mathtt{p}][\mathtt{q}]\mathtt{m}}$;

$\mathtt{q}$ has a crash handling branch if it receives from an unreliable $\mathtt{p}$.

[S-$\lightning$&] $\Gamma \xrightarrow{s[\mathtt{p}]\mathrm{stop}}$ and $\Gamma \xrightarrow{s[\mathtt{q}]:\mathtt{p}\&\mathtt{m}(S)}$ implies $\Gamma \xrightarrow{s[\mathtt{q}]\odot\mathtt{p}}$;

A safe Γ cannot beget an unsafe Γ.

[S-$\rightarrow_{\lightning}$] $\Gamma \rightarrow_{\lightning\setminus s;\mathcal{R}} \Gamma'$ implies $\varphi(\Gamma')$.

Safety Checking the DNS Protocol

$$\Gamma = \Gamma_{\mathtt{p}}, \Gamma_{\mathtt{q}}, \Gamma_{\mathtt{r}}$$

$$\Gamma_{\mathtt{p}} = s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end} \\ \mathbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$\Gamma_{\mathtt{q}} = s[\mathtt{q}]{:}\mathsf{stop}$$

$$\Gamma_{\mathtt{r}} = s[\mathtt{r}]{:}\mathtt{q}\&\mathbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

By inspection, we know that Γ is safe

Safety Checking the DNS Protocol

$$\Gamma = \Gamma_{\mathtt{p}}, \Gamma_{\mathtt{q}}, \Gamma_{\mathtt{r}}$$

$$\Gamma_{\mathtt{p}} = s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end} \\ \textbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$\Gamma_{\mathtt{q}} = s[\mathtt{q}]{:}\mathsf{stop}$$

$$\Gamma_{\mathtt{r}} = s[\mathtt{r}]{:}\mathtt{q}\&\textbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

By inspection, we know that Γ is safe

$$[\text{S-}\oplus\&] \quad \begin{array}{l} s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{url}\cdots \\ s[\mathtt{q}]{:}\mathtt{p}\&\mathtt{url}\cdots \end{array} \qquad \begin{array}{l} s[\mathtt{p}]{:}\mathtt{r}\&\mathtt{res}.\mathsf{end} \\ s[\mathtt{p}]{:}\mathtt{p}\oplus\mathtt{res}.\mathsf{end} \end{array}$$

Safety Checking the DNS Protocol

$$\Gamma = \Gamma_{p}, \Gamma_{q}, \Gamma_{r}$$

$$\Gamma_{p} = s[p]{:}q{\oplus}\texttt{url}.q\&\left\{\begin{array}{l}\texttt{res}.\mathsf{end} \\ \textbf{crash}.r{\oplus}\texttt{url}.r\&\texttt{res}.\mathsf{end}\end{array}\right\}$$

$$\Gamma_{q} = s[q]{:}\mathsf{stop}$$

$$\Gamma_{r} = s[r]{:}q\&\textbf{crash}.p\&\texttt{url}.p{\oplus}\texttt{res}.\mathsf{end}$$

By inspection, we know that Γ is safe

$$[\text{S-}\lightning\&] \quad \begin{array}{l} s[p]{:}q\&\left\{\begin{array}{l}\cdots \\ \textbf{crash}\cdots\end{array}\right\} \\ s[q]{:}\mathsf{stop}\end{array}$$

Safety Checking the DNS Protocol

$$\Gamma = \Gamma_{p}, \Gamma_{q}, \Gamma_{r}$$

$$\Gamma_{p} = s[p]{:}q\oplus \mathtt{url}.q\&\left\{ \begin{array}{l} \mathtt{res}.\mathsf{end} \\ \mathbf{crash}.r\oplus \mathtt{url}.r\&\mathtt{res}.\mathsf{end} \end{array} \right\}$$

$$\Gamma_{q} = s[q]{:}\mathsf{stop}$$

$$\Gamma_{r} = s[r]{:}q\&\mathbf{crash}.p\&\mathtt{url}.p\oplus \mathtt{res}.\mathsf{end}$$

By inspection, we know that Γ is safe

» By further inspection, Γ deadlock-free, live, and terminating.

Safety Checking the DNS Protocol

$$\Gamma = \Gamma_{\mathtt{p}}, \Gamma_{\mathtt{q}}, \Gamma_{\mathtt{r}}$$

$$\Gamma_{\mathtt{p}} = s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end} \\ \mathbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$\Gamma_{\mathtt{q}} = s[\mathtt{q}]{:}\mathsf{stop}$$

$$\Gamma_{\mathtt{r}} = s[\mathtt{r}]{:}\mathtt{q}\&\mathbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

By inspection, we know that Γ is safe

- By further inspection, Γ deadlock-free, live, and terminating.
- How do we determine this **automatically?**

Type-Level Model Checking

Desirable behavioural properties can be checked *via* **model checking**.

- Off-the-shelf model checker: **mCRL2**
- Typing contexts are expressed as **models**
- Behavioural properties are expressed as **modal μ-calculus formulæ**

Type-Level Model Checking

Desirable behavioural properties can be checked *via* **model checking**.

- Off-the-shelf model checker: **mCRL2**
- Typing contexts are expressed as **models**
- Behavioural properties are expressed as **modal μ-calculus formulæ**

$$
\begin{array}{ll}
[\text{S-}\oplus\&] & \Gamma \xrightarrow{s[\mathtt{p}]:\mathtt{q}\oplus\mathtt{m}(S)} \text{ and } \Gamma \xrightarrow{s[\mathtt{q}]:\mathtt{p}\&\mathtt{m}'(S')} \text{ implies } \Gamma \xrightarrow{s[\mathtt{p}][\mathtt{q}]\mathtt{m}}; \\
[\text{S-}\lightning\&] & \Gamma \xrightarrow{s[\mathtt{p}]\mathsf{stop}} \text{ and } \Gamma \xrightarrow{s[\mathtt{q}]:\mathtt{p}\&\mathtt{m}(S)} \text{ implies } \Gamma \xrightarrow{s[\mathtt{q}]\odot\mathtt{p}}; \\
[\text{S-}\rightarrow_{\lightning}] & \Gamma \rightarrow_{\lightning \setminus s;\mathcal{R}} \Gamma' \text{ implies } \varphi(\Gamma').
\end{array}
$$

Type-Level Model Checking

Desirable behavioural properties can be checked *via* **model checking**.

- Off-the-shelf model checker: **mCRL2**
- Typing contexts are expressed as **models**
- Behavioural properties are expressed as **modal μ-calculus formulæ**

[S-↯&]

$$\Gamma \models \nu Z.\left(\begin{array}{l}\forall s, p, q, m, m', S, S'.\ (\langle s[p]\mathsf{stop}\rangle\top \land \langle s[q]{:}p\&m'(S')\rangle\top \Rightarrow \langle s[q]\odot p\rangle\top) \\ \land\ (\langle s[p]{:}q\oplus m(S)\rangle\top \land (\langle s[q]{:}p\&m'(S')\rangle\top \lor \langle s[q]\mathsf{stop}\rangle\top) \Rightarrow \langle s[p][q]m\rangle\top) \land \phi_{\rightarrow}(Z)\end{array}\right)$$

[S-⊕&]

Type-Level Model Checking is Practical

example	states	transitions	safe	df	live	nterm	term
DNS	101	427	$12.28 \pm 1\%$	$17.14 \pm 1\%$	$11.24 \pm 1\%$	$15.47 \pm 0\%$	$12.33 \pm 0\%$
Adder	37	159	$12.43 \pm 0\%$	$15.74 \pm 0\%$	$12.24 \pm 1\%$	$14.46 \pm 0\%$	$12.06 \pm 1\%$
TwoBuyers	1409	10248	$45.6 \pm 0\%$	$88.26 \pm 0\%$	$31.33 \pm 0\%$	$77.2 \pm 0\%$	$45.65 \pm 0\%$
Negotiate	1089	8106	$34.61 \pm 0\%$	$55.07 \pm 0\%$	$25.69 \pm 0\%$	$47.46 \pm 0\%$	$26.04 \pm 0\%$
Broadcast	161	925	$17.99 \pm 1\%$	$28.13 \pm 0\%$	$14.08 \pm 0\%$	$25.72 \pm 1\%$	$17.74 \pm 0\%$

- Property checking takes at $<$ 100ms on our example suite
- Our prototype tool, MPSTK, is available on GitHub at `https://github.com/alcestes/mpstk-crash-stop`

Type-Level Model Checking is Practical

example	states	transitions	safe	df	live	nterm	term
DNS	101	427	$12.28 \pm 1\%$	$17.14 \pm 1\%$	$11.24 \pm 1\%$	$15.47 \pm 0\%$	$12.33 \pm 0\%$
Adder	37	159	$12.43 \pm 0\%$	$15.74 \pm 0\%$	$12.24 \pm 1\%$	$14.46 \pm 0\%$	$12.06 \pm 1\%$
TwoBuyers	1409	10248	$45.6 \pm 0\%$	$88.26 \pm 0\%$	$31.33 \pm 0\%$	$77.2 \pm 0\%$	$45.65 \pm 0\%$
Negotiate	1089	8106	$34.61 \pm 0\%$	$55.07 \pm 0\%$	$25.69 \pm 0\%$	$47.46 \pm 0\%$	$26.04 \pm 0\%$
Broadcast	161	925	$17.99 \pm 1\%$	$28.13 \pm 0\%$	$14.08 \pm 0\%$	$25.72 \pm 1\%$	$17.74 \pm 0\%$

- Property checking takes at $<$ 100ms on our example suite
- Our prototype tool, MPSTK, is available on GitHub at `https://github.com/alcestes/mpstk-crash-stop`

Type-Level Model Checking is Practical

example	states	transitions	safe	df	live	nterm	term
DNS	101	427	$12.28 \pm 1\%$	$17.14 \pm 1\%$	$11.24 \pm 1\%$	$15.47 \pm 0\%$	$12.33 \pm 0\%$
Adder	37	159	$12.43 \pm 0\%$	$15.74 \pm 0\%$	$12.24 \pm 1\%$	$14.46 \pm 0\%$	$12.06 \pm 1\%$
TwoBuyers	1409	10248	$45.6 \pm 0\%$	$88.26 \pm 0\%$	$31.33 \pm 0\%$	$77.2 \pm 0\%$	$45.65 \pm 0\%$
Negotiate	1089	8106	$34.61 \pm 0\%$	$55.07 \pm 0\%$	$25.69 \pm 0\%$	$47.46 \pm 0\%$	$26.04 \pm 0\%$
Broadcast	161	925	$17.99 \pm 1\%$	$28.13 \pm 0\%$	$14.08 \pm 0\%$	$25.72 \pm 1\%$	$17.74 \pm 0\%$

- Property checking takes at $<$ 100ms on our example suite
- Our prototype tool, MPSTK, is available on GitHub at `https://github.com/alcestes/mpstk-crash-stop`

mCRL2 improvement [POPL'19]

Table 2. Average time (in seconds ± std. dev.) for the verification of the protocols in Fig. 4. Protocols (3) and (4) are instantiated with $n = 3$. The outcome of the verification is shown in Table 1. *(Benchmarking specs: Intel Core i7-4790 CPU, 3.60GHz, 16 GB RAM, mCRL2 201808.0 invoked 30 times (by* `mpstk`*) with:* `pbes2bool --strategy=2`*)*

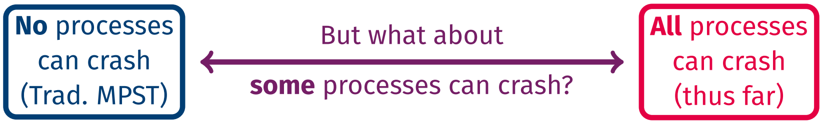
	states	safe	deadlock-free	live	live^{+}	live^{++}	never-terminat.	terminat.
(1) OAuth2 fragment	37	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	0.98 ± 9%
(2) Rec. two-buyers	85	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	0.99 ± 3%
(3) Rec. map/reduce	2561	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	1.00 ± 0%	0.99 ± 3%
(4) MP workers	442369	1.01 ± 4%	0.98 ± 8%	0.98 ± 9%	1.03 ± 14%	1.02 ± 7%	0.99 ± 6%	1.00 ± 1%

Fault-Tolerant Session Types

We present a **generalised multiparty session type theory** with **crash-stop failures**.

- ✓ Session π-calculus with crash-stop failures
- ✓ Multiparty session types with crashes
- ✓ Type-level model checking for behavioural properties
- ⇒ Optional reliability assumptions

One More Step Towards Realistic Procotols



- Some participants may represent services that can be **assumed reliable**
  - À la Amazon AWS, Microsoft Azure, Google Cloud Platform, &c.
- Can we **safely** avoid writing crash handling branches for **reliable** participants?

Optional Reliability Assumptions

- Specific participants can be marked as **reliable**
- Reliable participants **will not crash**
- The set of reliable participants is denoted $\mathcal{R}$
  - As seen in the safety definition
- Reduction rules, model checking, &c. conform to these assumptions
  - Behavioural properties can be satisfied for partially reliable protocols

Optionally Assuming Reliability in the DNS Protocol

$$
\begin{aligned}
\Gamma &= \Gamma_{p}, \Gamma_{q}, \Gamma_{r} \\
\Gamma_{p} &= s[p]{:}\, q \oplus \mathtt{url}.q \,\&\, \left\{ \begin{array}{l} \mathtt{res}.\mathsf{end} \\ \mathbf{crash}.r \oplus \mathtt{url}.r \,\&\, \mathtt{res}.\mathsf{end} \end{array} \right\} \\
\Gamma_{q} &= s[q]{:}\, \mathsf{stop} \\
\Gamma_{r} &= s[r]{:}\, q \,\&\, \mathbf{crash}.p \,\&\, \mathtt{url}.p \oplus \mathtt{res}.\mathsf{end}
\end{aligned}
$$

If we assume that **all participants are unreliable**

- Both p and r can crash arbitrarily
- Currently missing crash handling branches...
- The safety property **does not hold**

Optionally Assuming Reliability in the DNS Protocol

$$\Gamma = \Gamma_{\mathtt{p}}, \Gamma_{\mathtt{q}}, \Gamma_{\mathtt{r}}$$

$$\Gamma_{\mathtt{p}} = s[\mathtt{p}]:\mathtt{q}\oplus\mathtt{url}.\mathtt{q}\&\left\{\begin{array}{l}\mathtt{res}.\mathsf{end}\\ \mathbf{crash}.\mathtt{r}\oplus\mathtt{url}.\mathtt{r}\&\mathtt{res}.\mathsf{end}\end{array}\right\}$$

$$\Gamma_{\mathtt{q}} = s[\mathtt{q}]:\mathsf{stop}$$

$$\Gamma_{\mathtt{r}} = s[\mathtt{r}]:\mathtt{q}\&\mathbf{crash}.\mathtt{p}\&\mathtt{url}.\mathtt{p}\oplus\mathtt{res}.\mathsf{end}$$

If we assume that both $\mathtt{p}$ **and** $\mathtt{r}$ **are reliable**

- Both $\mathtt{p}$ and $\mathtt{r}$ **cannot** crash
- No further adjustments necessary
- Safety, deadlock-freedom, liveness, and terminating properties hold

Summary and Future Work

- » Generalised MPST theory with crash-stop failures
- » Model checking support to check for behavioural properties
  - `https://github.com/alcestes/mpstk-crash-stop`
- » Optional Reliability Assumptions
  - Support for fully-reliable to fully-unreliable protocols

In the paper

- » Type system: typing rules and typing context transitions
- » How optional reliability is respected in considering process reductions
- » How properties are formulated as modal μ-calculus formulæ
- » Benchmarks and examples
- » Full version: `https://arxiv.org/abs/2207.02015`

Summary and Future Work

- Generalised MPST theory with crash-stop failures
- Model checking support to check for behavioural properties
  - `https://github.com/alcestes/mpstk-crash-stop`
- Optional Reliability Assumptions
  - Support for fully-reliable to fully-unreliable protocols

Future Work

- Support more failure models
- Support a top-down approach to MPST with global types and projection

Summary and Future Work

- Generalised MPST theory with crash-stop failures
- Model checking support to check for behavioural properties
  - `https://github.com/alcestes/mpstk-crash-stop`
- Optional Reliability Assumptions
  - Support for fully-reliable to fully-unreliable protocols

In the paper

- Type system: typing rules and typing context transitions
- How optional reliability is respected in considering process reductions
- How properties are formulated as modal μ-calculus formulæ
- Benchmarks and examples
- Full version: `https://arxiv.org/abs/2207.02015`

Aides Memoire: Affine Approaches

- *Affine Sessions* by Mostrous & Vasconcelos, 2018
  - Binary sessions
- *Exceptional Asynchronous Session Types* by Fowler et al. 2019
  - λ-calculus
  - Binary sessions
  - Exception handling
- More exceptions: Cappechi et al. 2016 & Carbone et al. 2008
- This approach models failures at the application level via throw/catch constructs.
  1. We model arbitrary failures (e.g. hardware failures);
  2. We specify what to do when a failure is detected at the type level;
  3. We support multiparty sessions;
  4. We support a participant's crash whilst handling another's (the do-catch constructs cannot be nested).

Aides Memoire: Coordinator Model Approaches

- *Session Types for Link Failures* by Adameit et al., 2017
  - Optional blocks and default values
- *Robust Failure Handling in Distributed Systems* by Chen et al., 2016
  - Synchronisation points to detect and handle failures
- *Statically Verified Crash Failure Handling* by Viering et al., 2018
  - Try-handle construct, a reliable coordinator, and broadcast failures
- We do not assume reliable processes, failure broadcasts, or coordination/synchronisation points.

Aides Memoire: Other Approaches

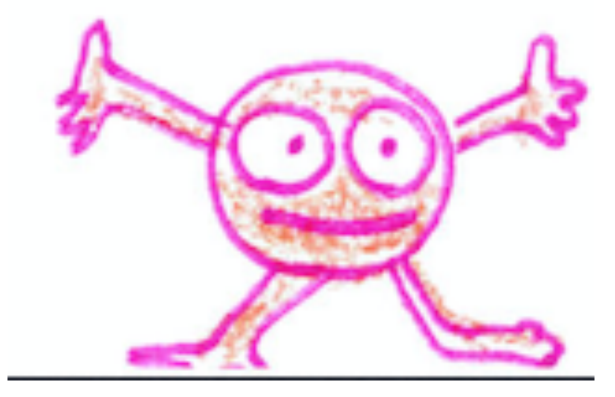
- *Fault-Tolerant Multiparty Session Types* by Peters et al. 2022
  - Their failure model is different from ours; and they handle failures by continuing the protocol via default branches and values.
  - **crash** is not synonymous to a default branch
- *Let it Recover* by Neykova & Yoshida, 2017
  - Supervision trees approach to MPST à la Erlang
- *Fault-Tolerant Event-Driven Distributed Programming* by Viering et al. 2018
  - Processes are monitored and restarted if they fail; needs reliable participants; but they do tolerate false suspicions
- *Linearity, Control Effects, and Behavioural Types* by Caires et al. 2017
  - Curry-Howard interpretation; binary session types; failures are propagated to relevant peers
- *Localities* approaches
  - Model distributed systems with failures; no types

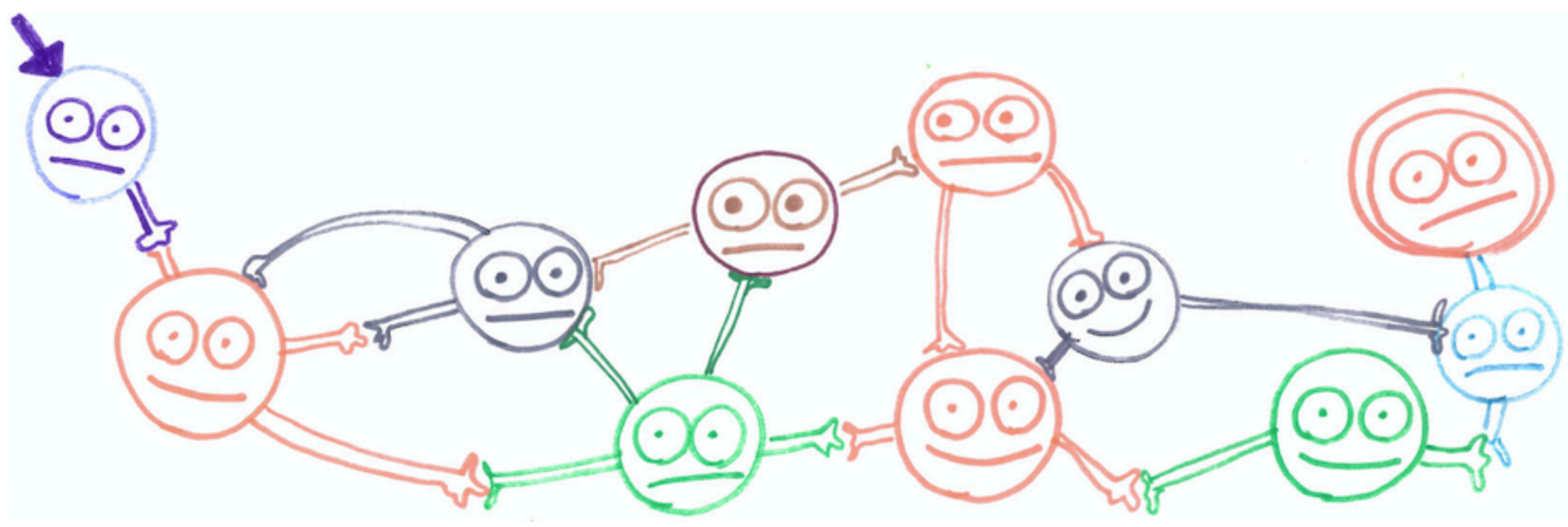
mCRL2 and Multiparty Session Types

- How can we integrate local verifications (by mCRL2) and global protocol conformance?

- What are possible extensions of local verifications of session types?
  1. probabilities
  2. counter-example guided verification
  3. bisimulation checking
  4. asynchrony
  5. mCLR2 graphics tool

Thank you! Questions?





CFSMs [1980-] ITU notation SDL · MSCS ···

Def A CFSM $M = (Q, C, q0, \Sigma, \delta)$

Q a finite set of states

$C = \{ pq \in Participant^2 \mid p \neq q \}$

$q0$ initial state

Σ a finite alphabet of messages

$\delta \subseteq Q \times (C \times \{!, ?\} \times \Sigma) \times Q$ a finite set of transitions

Def CS $S = (M_p)_{p \in Participant}$

Multiparty Compatibility

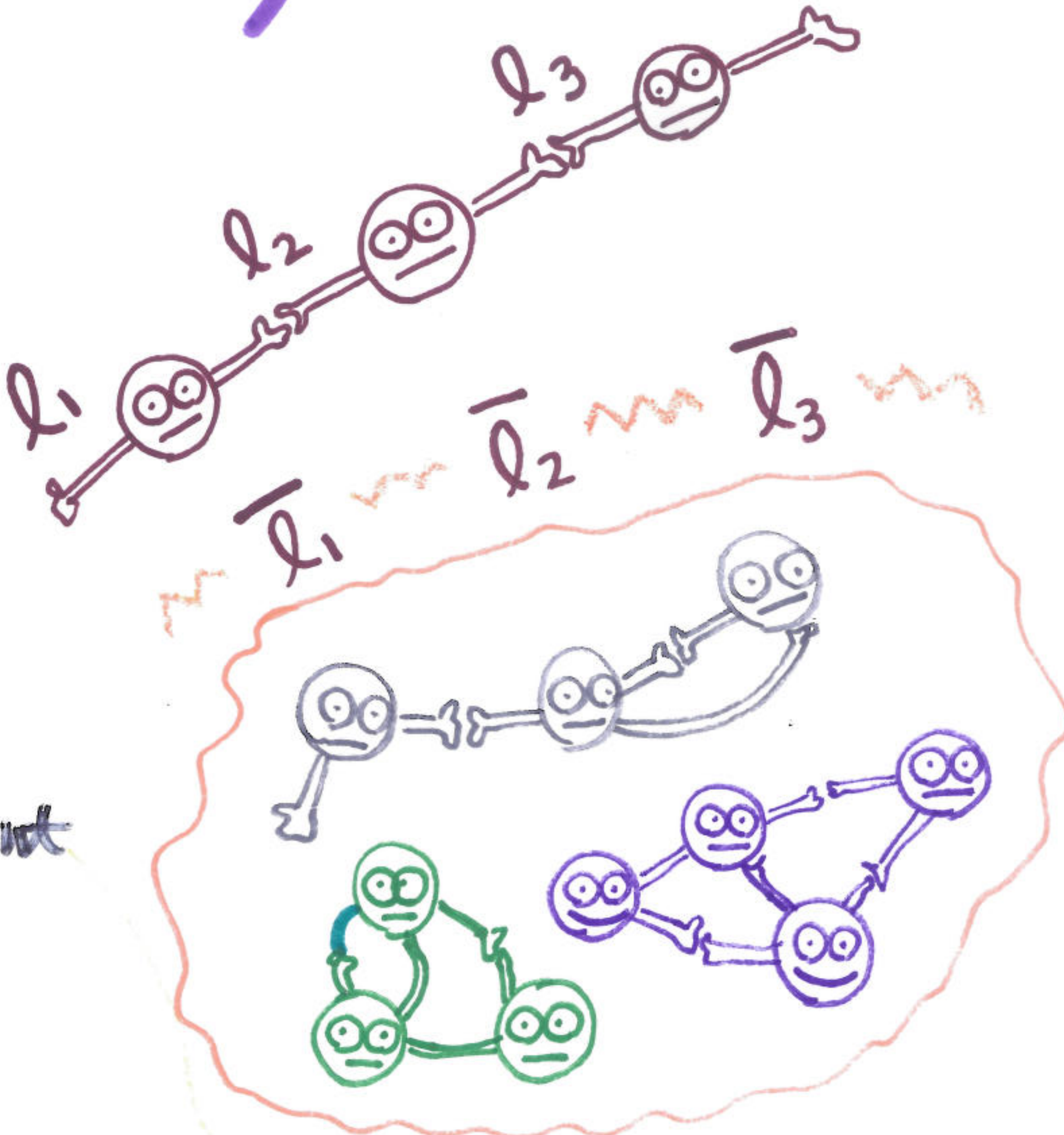
A B C

l_1 l_2 l_3

$\overline{l_1}$ $\overline{l_2}$ $\overline{l_3}$

Multiparty Compatibility

A B C



Def $S = (M_p)_{p \in \text{Participant}}$

$\forall s .\ s_0 \rightsquigarrow s$ (1-buffer execution)

if M_i does action ℓ

then $(M_j)_{j \in P \setminus i}$ do action $\bar{\ell}$

after some $\rightsquigarrow$

Multiparty Compatibility

Definition System $S = (M_p)_{p \in \mathcal{P}}$ is **MC** if for any 1-bound reachable state $s \in RS_1(S)$, and any output action $pq!a$ from s in M_p, there exists an alternation $\varphi \cdot t$ from s in a system where $act(t) = pq?a$ and $p \notin act(\varphi)$

(Dual for input)

$s \xrightarrow{t} s'$ configuration $s = (\vec{q} \,;\, \vec{w})$

states q_p queues w_{pq}

Send

$$(\cdots q_p \cdots ; \cdots w_{pq} \cdots) \xrightarrow{pq!\ell} (\cdots q_p' \cdots ; \cdots w_{pq} \cdot \ell \cdots)$$

Receive

$$(\cdots q_q \cdots ; \cdots \ell \cdot w_{pq} \cdots) \xrightarrow{pq?\ell} (\cdots q_q' \cdots ; \cdots w_{pq} \cdots)$$

Deteministic CFSM

$$s \xrightarrow{t} s', \quad s \xrightarrow{t} s'' \quad \Rightarrow \quad s' = s''$$

Basic CFSMs

A CFSM is **Basic** if **deterministic directed**, has **no mixed states**