A Decentralized Analysis of Multiparty Protocols[*]

Bas van den Heuvel and Jorge A. Pérez

Bernoulli Institute for Math, CS, and AI
University of Groningen, The Netherlands

NWPT 2021

[*]Research partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.
- *Multiparty* protocols have *two or more* participants.

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.
- *Multiparty* protocols have *two or more* participants.
- Our analysis:
    - *Message-passing processes* (π-calculus) implement roles of protocol participants, network of processes implements protocol.

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.
- *Multiparty* protocols have *two or more* participants.
- Our analysis:
  - *Message-passing processes* (π-calculus) implement roles of protocol participants, network of processes implements protocol.
  - Verify *correctness, safety, and deadlock-freedom* of implementations, by means of a *binary session type system*.

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.
- *Multiparty* protocols have *two or more* participants.
- Our analysis:
  - *Message-passing processes* (π-calculus) implement roles of protocol participants, network of processes implements protocol.
  - Verify *correctness, safety, and deadlock-freedom* of implementations, by means of a *binary session type system*.
  - Support expressive classes of protocols (delegation, interleaving).

Introduction

- Protocol: describes how communicating components should interact, to avoid errors.
- Here: sequence of directed exchanges (messages), specified as *session types*.
- *Multiparty* protocols have *two or more* participants.
- Our analysis:
  - *Message-passing processes* (π-calculus) implement roles of protocol participants, network of processes implements protocol.
  - Verify *correctness, safety, and deadlock-freedom* of implementations, by means of a *binary session type system*.
  - Support expressive classes of protocols (delegation, interleaving).
  - New approach: process networks should be *decentralized*.

Multiparty Session Types

- Multiparty Session Types use *global and local types* to verify protocol implementations.

Multiparty Session Types

- Multiparty Session Types use *global and local types* to verify protocol implementations.

- An example protocol: global type G_{auth}, server authentication with three participants *server* (s), *client* (c), and *authorization service* (a).

$$\mu X \,.\, s \twoheadrightarrow c \left\{\begin{array}{l} \mathsf{login} \,.\, c \twoheadrightarrow a : \mathsf{passwd}\langle \mathsf{str} \rangle \,.\, a \twoheadrightarrow s : \mathsf{result}\langle \mathsf{bool} \rangle \,.\, X, \\ \mathsf{quit} \,.\, c \twoheadrightarrow a : \mathsf{quit} \,.\, \mathsf{end} \end{array}\right\}$$

Global Type Type Checking Workflow

G_{auth} global type

Global Type Type Checking Workflow

$$\begin{tikzcd}
 & G_{\mathsf{auth}} \arrow[dl] \arrow[d] \arrow[dr] & \\
L_s & L_c & L_a
\end{tikzcd}$$

global type

local projection

local types

Global Type Type Checking Workflow

$$\begin{tikzcd}
& G_{\text{auth}} \arrow[ddl] \arrow[dd] \arrow[ddr] & & \text{global type} \\
& & & \text{local projection} \\
L_s & L_c & L_a & \text{local types} \\
& & & \\
P_s & P_c & P_a & \text{implementations}
\end{tikzcd}$$

Global Type Type Checking Workflow

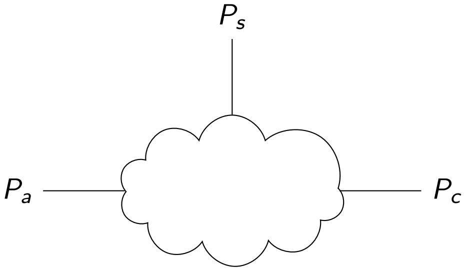
$$\begin{tikzcd}
 & G_{\mathsf{auth}} \arrow[ddl] \arrow[dd] \arrow[ddr] & & \text{global type} \\
 & & & \text{local projection} \\
L_s & L_c & L_a & \text{local types} \\
\vdash & \vdash & \vdash & \text{type checking} \\
P_s & P_c & P_a & \text{implementations}
\end{tikzcd}$$

Type Checking Implementations

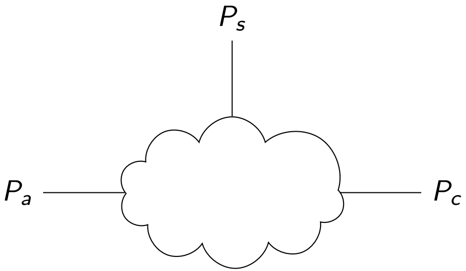
P_s

P_a P_c

Type Checking Implementations

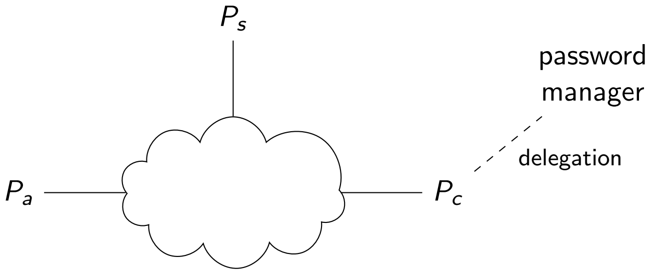


Type Checking Implementations



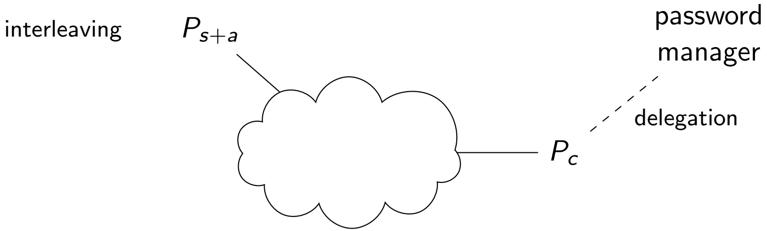
- Open problem: guarantee deadlock-freedom while supporting *delegation* and *interleaving*.

Type Checking Implementations



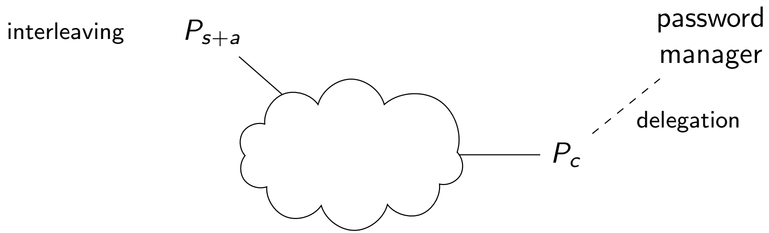
- Open problem: guarantee deadlock-freedom while supporting *delegation* and *interleaving*.

Type Checking Implementations



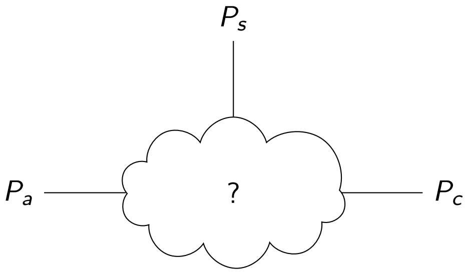
- Open problem: guarantee deadlock-freedom while supporting *delegation* and *interleaving*.

Type Checking Implementations



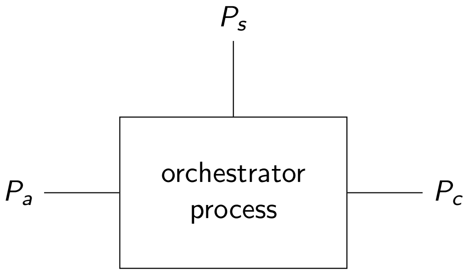
- Open problem: guarantee deadlock-freedom while supporting *delegation* and *interleaving*.
- Our approach: reduce the problem to *binary* session types, where deadlock-freedom follows from typing and delegation and interleaving are naturally supported.

Routers



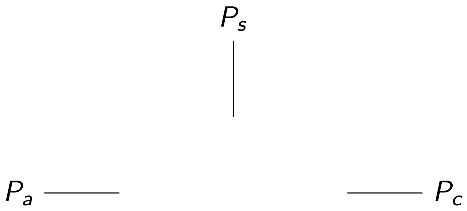
- How to connect processes?

Routers



- How to connect processes?
- *Centralized* solution is natural, but undesirable: makes assumptions of the network and scales poorly.

Routers



- How to connect processes?
- *Centralized* solution is natural, but undesirable: makes assumptions of the network and scales poorly.
- We want a *decentralized* solution.

Routers

$$\begin{tikzcd}
 & P_s \arrow[d, dash] & \\
 & \mathcal{R}_s & \\
P_a \arrow[r, dash] & \mathcal{R}_a \quad\quad\quad\quad \mathcal{R}_c \arrow[r, dash] & P_c
\end{tikzcd}$$

- How to connect processes?
- *Centralized* solution is natural, but undesirable:
  makes assumptions of the network and scales poorly.
- We want a *decentralized* solution.
- Our solution: a *router* per participant, derived from global type.

Routers

$$\begin{tikzcd}
 & & P_s \arrow[d, dash] & & \\
 & & \mathcal{R}_s \arrow[dl, dash] \arrow[dr, dash] & & \\
P_a \arrow[r, dash] & \mathcal{R}_a \arrow[rr, dash] & & \mathcal{R}_c \arrow[r, dash] & P_c
\end{tikzcd}$$

- How to connect processes?
- *Centralized* solution is natural, but undesirable:
  makes assumptions of the network and scales poorly.
- We want a *decentralized* solution.
- Our solution: a *router* per participant, derived from global type.

Routers

$$\begin{tikzcd}
 & & P_s \arrow[d, dash] & & \\
 & & \mathcal{R}_s \arrow[dl, dash] \arrow[dr, dash] & & \\
P_a \arrow[r, dash] & \mathcal{R}_a \arrow[rr, dash] & & \mathcal{R}_c \arrow[r, dash] & P_c
\end{tikzcd}$$

- How to connect processes?
- *Centralized* solution is natural, but undesirable:
  makes assumptions of the network and scales poorly.
- We want a *decentralized* solution.
- Our solution: a *router* per participant, derived from global type.
- Routers are local to their participant implementation,
  but decentralized from each other.

Routers

$$\begin{tikzcd}
 & & P_s \arrow[d, no head] & & \\
 & & \mathcal{R}_s \arrow[dl, no head] \arrow[dr, no head] & & \\
P_a \arrow[r, no head] & \mathcal{R}_a \arrow[rr, no head] & & \mathcal{R}_c \arrow[r, no head] & P_c
\end{tikzcd}$$

- How to connect processes?
- *Centralized* solution is natural, but undesirable:
  makes assumptions of the network and scales poorly.
- We want a *decentralized* solution.
- Our solution: a *router* per participant, derived from global type.
- Routers are local to their participant implementation,
  but decentralized from each other.
- Result: decentralized network of routed implementations.

Type Checking Networks of Routed Implementations

$$\begin{tikzcd}
 & & P_s \arrow[d, dash] & & \\
 & & \mathcal{R}_s \arrow[dl, dash] \arrow[dr, dash] & & \\
P_a \arrow[r, dash] & \mathcal{R}_a \arrow[rr, dash] & & \mathcal{R}_c \arrow[r, dash] & P_c
\end{tikzcd}$$

- How to type check networks of routed implementations?

Type Checking Networks of Routed Implementations

$$\begin{tikzcd}
 & & P_s & & \\
 & & \mathcal{R}_s \arrow[u, dash] \arrow[dl, dash] \arrow[dr, dash] & & \\
P_a \arrow[r, dash] & \mathcal{R}_a \arrow[rr, dash] & & \mathcal{R}_c \arrow[r, dash] & P_c
\end{tikzcd}$$

- How to type check networks of routed implementations?
- Type system *APCP*, with deadlock-freedom guarantee for cyclic process networks *without cyclic dependencies*.

Type Checking Networks of Routed Implementations

$$\begin{tikzcd}
 & & P_s \arrow[d, dash, "L_s"] & & \\
 & & \mathcal{R}_s \arrow[dl, dash] \arrow[dr, dash] & & \\
P_a \arrow[r, dash, "L_a"] & \mathcal{R}_a \arrow[rr, dash] & & \mathcal{R}_c \arrow[r, dash, "L_c"] & P_c
\end{tikzcd}$$

- How to type check networks of routed implementations?
- Type system *APCP*, with deadlock-freedom guarantee for cyclic process networks *without cyclic dependencies*.
- Channels between implementations and routers: use local projections.

Type Checking Networks of Routed Implementations

$$\begin{tikzcd}
 & & P_s \arrow[d, dash, "L_s"] & & \\
 & & \mathcal{R}_s \arrow[dl, dash, "R_{s,a}"'] \arrow[dr, dash, "R_{c,s}"] & & \\
P_a \arrow[r, dash, "L_a"] & \mathcal{R}_a \arrow[rr, dash, "R_{a,c}"'] & & \mathcal{R}_c \arrow[r, dash, "L_c"] & P_c
\end{tikzcd}$$

- How to type check networks of routed implementations?
- Type system *APCP*, with deadlock-freedom guarantee for cyclic process networks *without cyclic dependencies*.
- Channels between implementations and routers: use local projections.
- Channels between routers: we introduce *relative projection*.

$$R_{s,a} = G_{\mathsf{auth}} \downharpoonright (s, a) \quad R_{a,c} = G_{\mathsf{auth}} \downharpoonright (a, c) \quad R_{c,s} = G_{\mathsf{auth}} \downharpoonright (c, s)$$

Relative Projection

- Relative projection reduces a global type to a binary protocol, relative to a *pair of participants*.

Relative Projection

- Relative projection reduces a global type to a binary protocol, relative to a *pair of participants*.
- Recall G_{auth}:

$$\mu X \,.\, s \twoheadrightarrow c \left\{ \begin{matrix} \text{login} \,.\, c \twoheadrightarrow a : \text{passwd}\langle \text{str} \rangle \,.\, a \twoheadrightarrow s : \text{result}\langle \text{bool} \rangle \,.\, X, \\ \text{quit} \,.\, c \twoheadrightarrow a : \text{quit} \,.\, \text{end} \end{matrix} \right\}$$

Relative Projection

- Relative projection reduces a global type to a binary protocol, relative to a *pair of participants*.
- Recall G_{auth}:

$$\mu X \mathrel{.} s \twoheadrightarrow c \begin{Bmatrix} \text{login} \mathrel{.} c \twoheadrightarrow a : \text{passwd}\langle \text{str} \rangle \mathrel{.} a \twoheadrightarrow s : \text{result}\langle \text{bool} \rangle \mathrel{.} X, \\ \text{quit} \mathrel{.} c \twoheadrightarrow a : \text{quit} \mathrel{.} \text{end} \end{Bmatrix}$$

- Relative projection onto (c, s) is straightforward:

$$G_{\text{auth}} \downharpoonleft (c, s) = \mu X \mathrel{.} s\{\text{login} \mathrel{.} X, \quad \text{quit} \mathrel{.} \text{end}\}$$

Relative Projection

- Relative projection reduces a global type to a binary protocol, relative to a *pair of participants*.
- Recall G_{auth}:

$$\mu X \,.\, s \twoheadrightarrow c \begin{Bmatrix} \mathsf{login} \,.\, c \twoheadrightarrow a : \mathsf{passwd}\langle \mathsf{str} \rangle \,.\, a \twoheadrightarrow s : \mathsf{result}\langle \mathsf{bool} \rangle \,.\, X, \\ \mathsf{quit} \,.\, c \twoheadrightarrow a : \mathsf{quit} \,.\, \mathsf{end} \end{Bmatrix}$$

- Relative projection onto (c, s) is straightforward:

$$G_{\mathsf{auth}} \downharpoonright (c, s) = \mu X \,.\, s\{\mathsf{login} \,.\, X, \quad \mathsf{quit} \,.\, \mathsf{end}\}$$

- Relative projection onto (c, a) is more complicated: non-local choices.

Relative Projection

- Relative projection reduces a global type to a binary protocol, relative to a *pair of participants*.
- Recall G_{auth}:

$$\mu X \mathrel{.} s \twoheadrightarrow c \begin{Bmatrix} \text{login} \mathrel{.} c \twoheadrightarrow a : \text{passwd}\langle \text{str} \rangle \mathrel{.} a \twoheadrightarrow s : \text{result}\langle \text{bool} \rangle \mathrel{.} X, \\ \text{quit} \mathrel{.} c \twoheadrightarrow a : \text{quit} \mathrel{.} \text{end} \end{Bmatrix}$$

- Relative projection onto (c, s) is straightforward:

$$G_{\text{auth}} \downharpoonleft (c, s) = \mu X \mathrel{.} s\{\text{login} \mathrel{.} X, \quad \text{quit} \mathrel{.} \text{end}\}$$

- Relative projection onto (c, a) is more complicated: non-local choices.
- Solution: non-local choices as explicit *dependency* messages.

$$G_{\text{auth}} \downharpoonleft (c, a) = \mu X \mathrel{.} c?s \begin{Bmatrix} \text{login} \mathrel{.} c : \text{passwd}\langle \text{str} \rangle \mathrel{.} X, \\ \text{quit} \mathrel{.} c : \text{quit} \mathrel{.} \text{end} \end{Bmatrix}$$

Well-formedness

- Not all global types derived from the syntax can be analyzed.

Well-formedness

- Not all global types derived from the syntax can be analyzed.
- As usual, we only consider *well-formed* global types, where projections onto all pairs of participants are defined.

Well-formedness

- Not all global types derived from the syntax can be analyzed.
- As usual, we only consider *well-formed* global types, where projections onto all pairs of participants are defined.
- For example, the following global type is not well-formed:

$$G = b \twoheadrightarrow a \begin{Bmatrix} \text{ok} \,.\, b \twoheadrightarrow s : \text{ok} \,.\, s \twoheadrightarrow m : \text{deliver}\langle \text{str} \rangle \,.\, \text{end}, \\ \text{quit} \,.\, b \twoheadrightarrow s : \text{quit} \,.\, s \twoheadrightarrow m : \text{quit} \,.\, \text{end} \end{Bmatrix}$$

Relative projection onto (s, m) is undefined.

Well-formedness

- Not all global types derived from the syntax can be analyzed.
- As usual, we only consider *well-formed* global types, where projections onto all pairs of participants are defined.
- For example, the following global type is not well-formed:

$$G = b \twoheadrightarrow a \begin{Bmatrix} \mathsf{ok} \,.\, b \twoheadrightarrow s : \mathsf{ok} \,.\, s \twoheadrightarrow m : \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \mathsf{end}, \\ \mathsf{quit} \,.\, b \twoheadrightarrow s : \mathsf{quit} \,.\, s \twoheadrightarrow m : \mathsf{quit} \,.\, \mathsf{end} \end{Bmatrix}$$

  Relative projection onto (s, m) is undefined.
- We can derive from G a well-formed global type:

$$G' = b \twoheadrightarrow a \begin{Bmatrix} \mathsf{ok} \,.\, b \twoheadrightarrow s : \mathsf{ok} \,.\, s \twoheadrightarrow m \{ \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \mathsf{end},\ \mathsf{quit} \,.\, \mathsf{end} \}, \\ \mathsf{quit} \,.\, b \twoheadrightarrow s : \mathsf{quit} \,.\, s \twoheadrightarrow m \{ \mathsf{deliver}\langle \mathsf{str} \rangle \,.\, \mathsf{end},\ \mathsf{quit} \,.\, \mathsf{end} \} \end{Bmatrix}$$

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.
- Supports decentralized process networks of routed participant implementation.

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.
- Supports decentralized process networks of routed participant implementation.
- Guarantees deadlock freedom by typing,
  with support for delegation and interleaving.

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.
- Supports decentralized process networks of routed participant implementation.
- Guarantees deadlock freedom by typing,
  with support for delegation and interleaving.
- Key innovations: routers and relative projection.

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.
- Supports decentralized process networks of routed participant implementation.
- Guarantees deadlock freedom by typing, with support for delegation and interleaving.
- Key innovations: routers and relative projection.
- New class of well-formed global types.

Conclusion

- New approach to analyzing correctness and deadlock-freedom of implementations of multiparty protocols.
- Supports decentralized process networks of routed participant implementation.
- Guarantees deadlock freedom by typing,
  with support for delegation and interleaving.
- Key innovations: routers and relative projection.
- New class of well-formed global types.
- Work currently under submission, draft available at `https://basvdheuvel.github.io`.