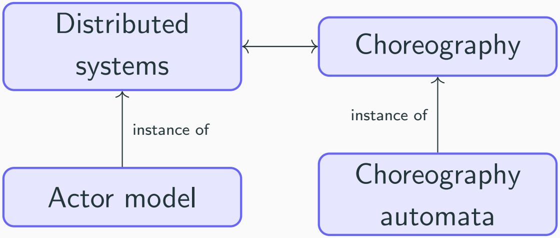
Choreographies for Program Understanding

Gabriele Genovese, Ivan Lanese, Cinzia Di Giusto,
Emilio Tuosto, and Germán Vidal

17 June 2025

Distributed Systems are Hard

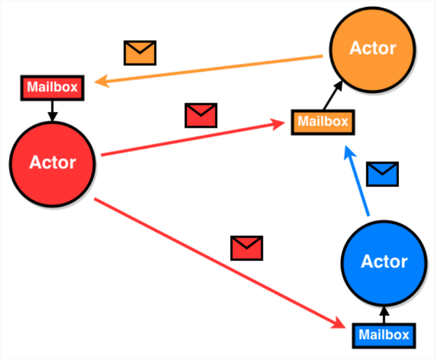
Abstractions in order to simplify:



The Actor Model

Main concepts

- Processes with mailbox
- Asynchronous messaging



The Actor Model

Ecosystem

- Erlang, Elixir, Scala
- Akka (Java), Actix (Rust)

Choreography Automata [1]

Informally

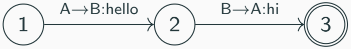
- Choreography: describes distributed protocols
- Paired with automata theory

$$\begin{tikzcd}
1 \arrow[r, "A\to B\text{:hello}"] & 2 \arrow[r, "B\to A\text{:hi}"] & 3
\end{tikzcd}$$

[1] Barbanera et al. "Choreography automata." COORDINATION, 2020.

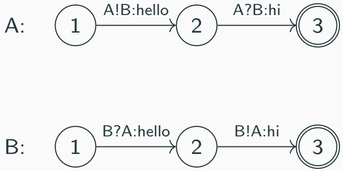
Global vs Local View

Global



The communication system is seen as a whole.

Local



A participant's individual perspective.

Usual practice: Top-Down Approach

$$\begin{tikzcd}
 & \boxed{\mathcal{G}} \arrow[dl, "\text{Projection}"'] \arrow[d] \arrow[dr] & \\
L_{\mathsf{A}} \arrow[d, leftrightarrow, "\text{Respect}"'] & L_{\mathsf{B}} \arrow[d, leftrightarrow] & L_{\mathsf{C}} \arrow[d, leftrightarrow] \\
\boxed{P_{\mathsf{A}}} & \boxed{P_{\mathsf{B}}} & \boxed{P_{\mathsf{C}}}
\end{tikzcd}$$

1. Global view

2. Local views

3. Processes

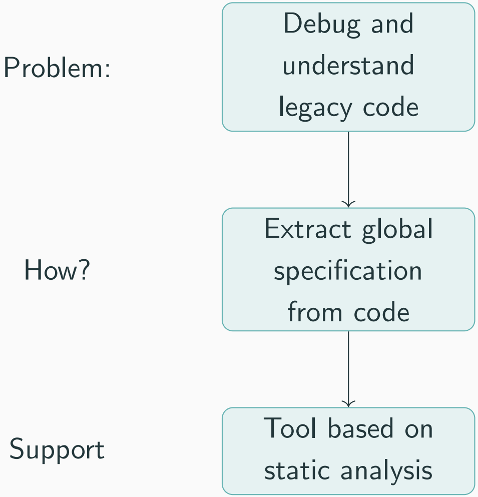
Usual practice: Top-Down Approach

Steps:

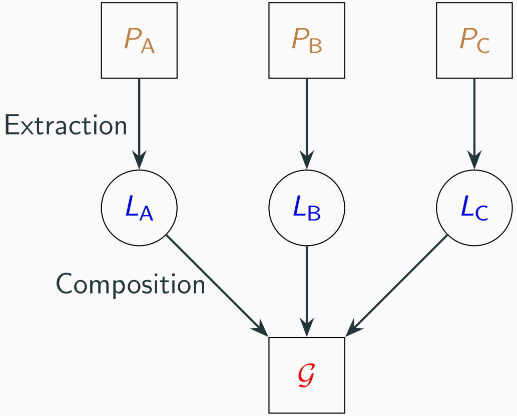
1. Write the global specification
2. Project to obtain the local specifications
3. Write local programs that respect the specifications

Problem: difficult to integrate existing code and architecture.

Aim


Problem:
Debug and understand legacy code
How?
Extract global specification from code
Support
Tool based on static analysis

Bottom-up Approach


$$\begin{tikzcd}
P_{\mathrm{A}} \arrow[d, "\text{Extraction}"'] & P_{\mathrm{B}} \arrow[d] & P_{\mathrm{C}} \arrow[d] \\
L_{\mathrm{A}} \arrow[dr, "\text{Composition}"'] & L_{\mathrm{B}} \arrow[d] & L_{\mathrm{C}} \arrow[dl] \\
 & \mathcal{G} &
\end{tikzcd}$$

1. Processes

2. Local view

3. Global view

Bottom-up approach

Extraction steps

1. Analyze an input source code
2. Extract the local views
3. Compose local views to create an approximated global view

Output: an abstraction that captures all the possible behaviors.

Why

Possible benefits

- Improves understanding of the code
    - Give all good behavior
- Highlight bugs
    - Like deadlock, race condition, etc...

Requirements

- Automatic extraction
- Target mainstream languages
- Support creation and removal of participants
- Capture good behaviors and highlight misbehaviors
- Use a simple notation as output

Extraction steps

1. Take the file input and perform simple analysis
2. Extract a local-view for each actor found
3. Compose a global-view using the local-views
4. Minimize and analyse the output graph

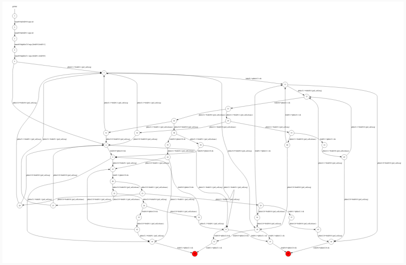
Problems - Undecidability

It would imply deciding termination. Choice of:

- Under-approximation: we can leave out some useful behavior
- **Over-approximation**: we include many behaviors (good or bad, present or not)

Problems - Huge descriptions

- Distributed system are naturally huge
- Needs of abstraction mechanisms: remove redundant information



Motivating Examples

Two examples:

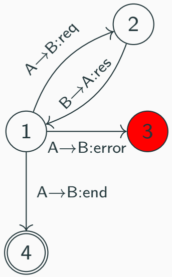
Dining Philosophers
(deadlock)

Bank Account
(race condition)

Deadlock

Final state: all the participants are in a *local* final state.

Deadlock: a non-final state without outgoing transitions.

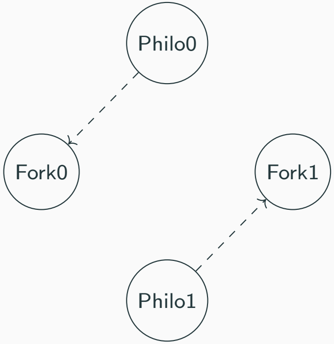


Dining Philosophers example

```
philosopher(Fork1, Fork2) →
  send req to Fork1,
  receive ack from Fork1,
  send req to Fork2,
  receive ack from Fork2,
  eat(),
  send release to Fork1,
  send release to Fork2,
  philosopher(Fork1, Fork2).

fork() →
  receive req from Phil,
  send ack to Phil,
  receive release from Phil,
  fork().
```

Participants:

- 2 philosophers
- 2 forks



Dining Philosophers example's local views

Note: merge of redundant communication!

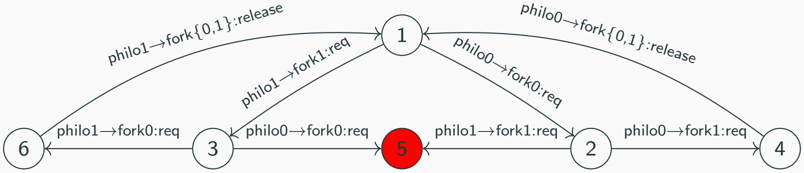
$$\begin{tikzcd}
1 \arrow[r, "\text{Philo!Fork\{0,1\}:req}"] & 2 \arrow[r, "\text{Philo?Fork\{0,1\}:ack}"] & 3 \arrow[ll, bend left, "\text{Philo!Fork\{0,1\}:release}"]
\end{tikzcd}$$

Philosopher's local view.

$$\begin{tikzcd}
1 \arrow[r, "\text{Fork?Philo:req}"] & 2 \arrow[r, "\text{Fork!Philo:ack}"] & 3 \arrow[ll, bend left, "\text{Fork?Philo:release}"]
\end{tikzcd}$$

Fork's local view.

Dining Philosophers global-view



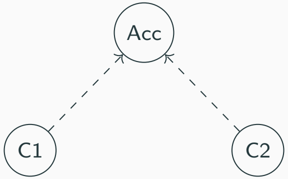
Global view of the Dining Philosophers example.

Bank Account example

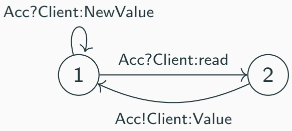
```
account(Value) →
  receive
    read from Client →
        send Value to Client,
        account(Value);
    NewValue from Client →
      account(NewValue).

client() →
  send read to Acc,
  receive Value from Acc,
  % operations on Value
  send NewValue to Acc.
```

Participants:

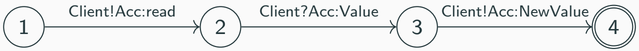
- 1 account
- 2 clients



Bank Account example's local views

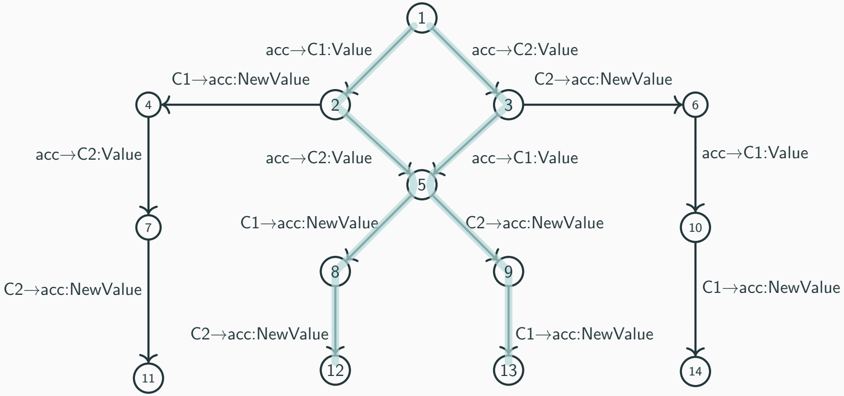


Account's local view.



Clients' local view.

Bank Account global-view



Global view of the Bank Account example.

Conclusion

Summary

- Problem: difficult to express legacy code
- Bridge the gap between programming and theoretical frameworks
- Use of bottom-up and over-approx techniques

Future work: create the tool as described (WIP on `github.com/gabrielegenovese/chorer`) and evaluate it on real-world examples.

Thanks for listening!