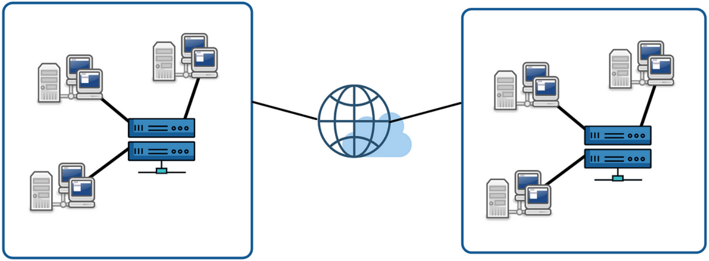
Extracting Choreography Automata for Program Understanding

Gabriele Genovese
Supervisor: Prof. Cinzia Di Giusto

March 24, 2025

Introduction

- Aim: explore formal model for development of distributed application for protocol specification



Context

- Distributed system: set of **participants** working for a common goal
- **Asynchronous** communication channel
- Programming with the *actor model* (Erlang)
- *Send* and *receive* main operations
- Protocols usually designed with *top-down approach*

Brief introduction to Choreography Automata

- Choreography: a formal model for distributed systems used for protocol specification
- **Global view**: communication system seen from above
- **Local view**: point of view of a single participant
- Choreography Automata: a *graphical* way of expressing Choreography

Motivations

- *Debugging*, *verification* of concurrent properties (*deadlock*, *liveness*, etc...), *program understanding*
- Bottom-up approach like the one used in programming
- Chorer: a prototype for a static analyzer that extract Choreography Automata

Case study: Dining philosopher

- ! : send
- ? : receive

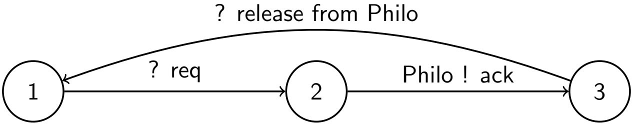


Figure: Local view of a fork.

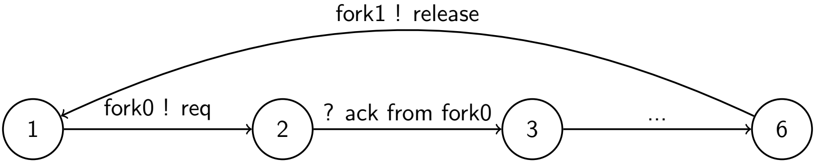


Figure: Local view of a philosopher.

Global view example

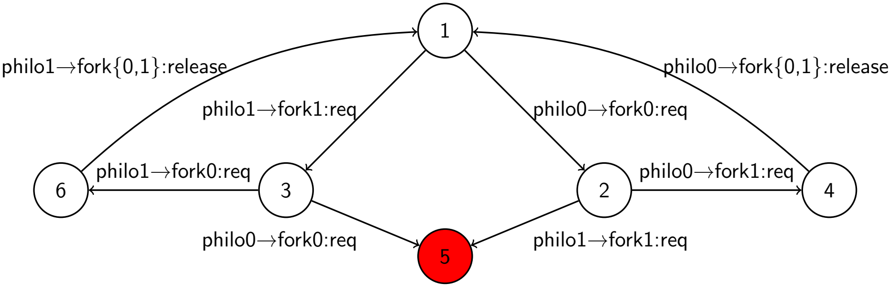


Figure: Global view of the dining philosophers example.

Characteristics of the tool

- Automatic **bottom-up** extraction
- **Over-approximated** approach when computing the globalview
- Always capture the good behaviors, and highlight possible misbehavior
- Applicable to mainstream languages

Chorer

Proof of Concept for a static analyzer developed as part of my Bachelor's thesis at the University of Bologna. Old state:

- Few working examples and no case study
- Some important feature missing (basic use of functions)
- Very difficult to use and understand

Contributions

- Formalization journey began (attributed grammar)
- Lots of improvements on the codebase (feature, bug, misc)
- Benchmarks suite improved and case studies created

Attributed grammar

- Aim: generalize some aspect of the tool
- Formal grammar enriched with attributes assigned to symbols that define how these attributes are computed.
- Used in compilers and parsers design
- Begin a formalization and refactoring process for localviews.

Attributes

Bottom-up technique. Attributes found:

- Nodes and Edges with Labels: used to show the automata
- First and Last Node of the automata: used to link some production
- Context and Returned Variable: used to manage process identifiers and data in general

Improvements

Feature:

- Value passing in functions (localview)
- ANY data overapproximation added (globalview)
- Improvements on CLI argument parsing, error and warning report
- Benchmark suite and testing enhanced
- Some important bugs fixed

Benchmarks - Empirical data

- Examples made by myself
- Show some empirical data about the output produced
- Algorithm $O(n!)$ but in practice very efficient

Example	Tot LV	GV Nodes	GV Edges	Warns	Errors	Time
async	3	7	6	0	0	0.194s
dining	3	45	72	0	2	0.232s
account	3	28	39	0	2	0.211s
if-cases	4	148	210	**185**	30	0.525s
foo6	5	9	9	15	2	0.190s
foo7	3	149	229	0	6	0.513s
foo8	5	**561**	**560**	0	**191**	**3.590s**

Table: Global view empirical data

Benchmarks - Correctness

Set ground for a precision benchmark (a correct version of the global view must be present):

- True: the global view given in output is the same of its correct version
- False: there are some difference

Example	Check
unknown	False
async	True
ticktackloop	True
ticktackstop	False
customer	False

Table: Global view correctness data

Conclusion

Key contributions:

- Study of the requirements and of the challenges
- Several improvements to the codebase and test suite
- Set groundwork for a formalization process and refactoring

Future works

- Continue the formalization and refactoring process
- Extend correctness benchmarks
- Continue feature (new keyword or BIF) and bug fix process

Thank you for your attention!