Multi-Processing for Distributed Summary Synthesis

Matthias Kettl, Thomas Lemberger, and Akshay Warrier

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SoSy-Lab Software Systems

Multi-Processing for Distributed Summary Synthesis

The next evolution of Distributed Summary Synthesis (DSS), for multi-processing.
Based on microservice technology.
Backend uses original DSS implementation in CPAchecker.

- ➔ Mitigates resource limits
- ➔ Simpler and more flexible in its setup
- ➔ Enables huge set of existing tooling

Issues with DSS

- On average, DSS runs 100 concurrent worker threads (cf. supplementary webpage)
- DSS requires more memory than traditional predicate abstraction (cf. paper, RQ 4)

- But DSS is a single, multi-threaded Java process
  - Limit on CPU cores
  - Limit on available memory

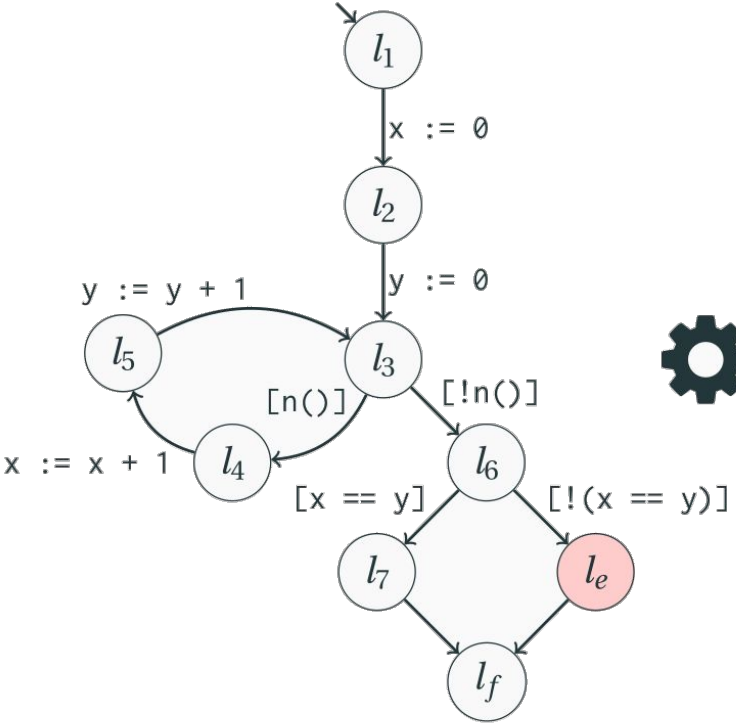
Advantages of microservice architecture

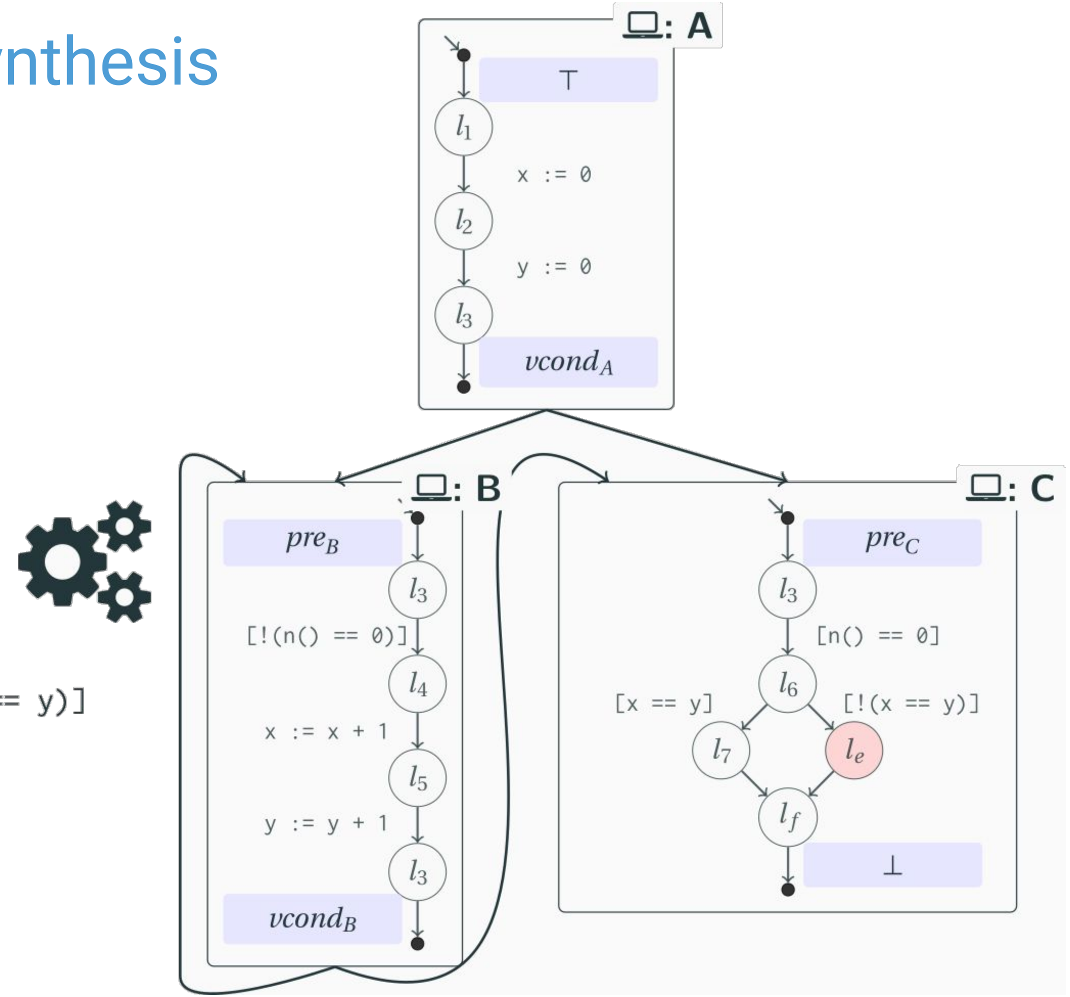
- Java microservices are popular ➔ lots of tooling and support
- Frameworks provide lots of functionality out of the box

Examples:

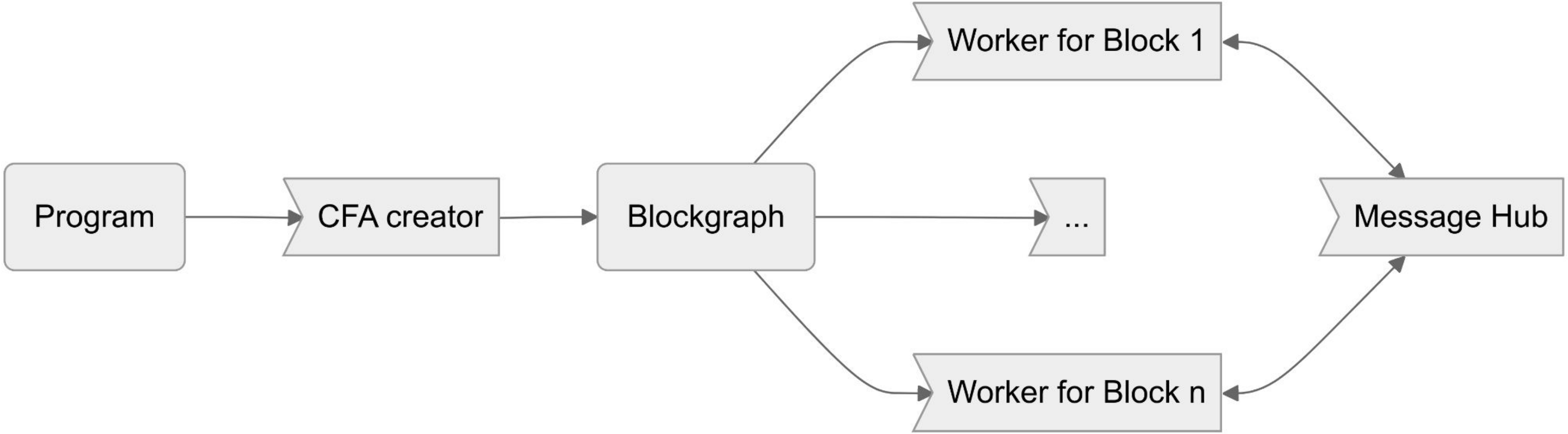
- Quarkus automatically generates network communication and glue code from interface definition
- Minikube provides quick infrastructure setup based on small yaml file
- Linkerd injects load balancing and basic profiling with a single command-line
- Prometheus collects, persists and visualizes profiling data without configuration

Distributed Summary Synthesis





Distributed Summary Synthesis

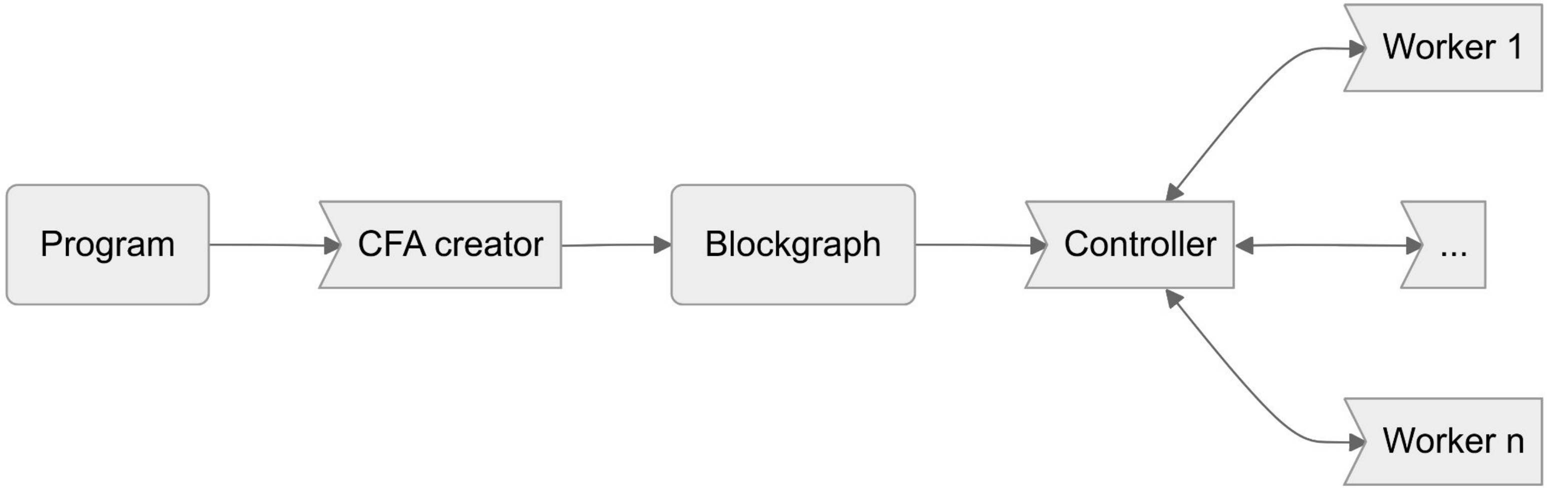


- Workers maintain precondition and violation condition for their block
- Workers decide when to re-analyze

➔ Behavior is distributed among workers

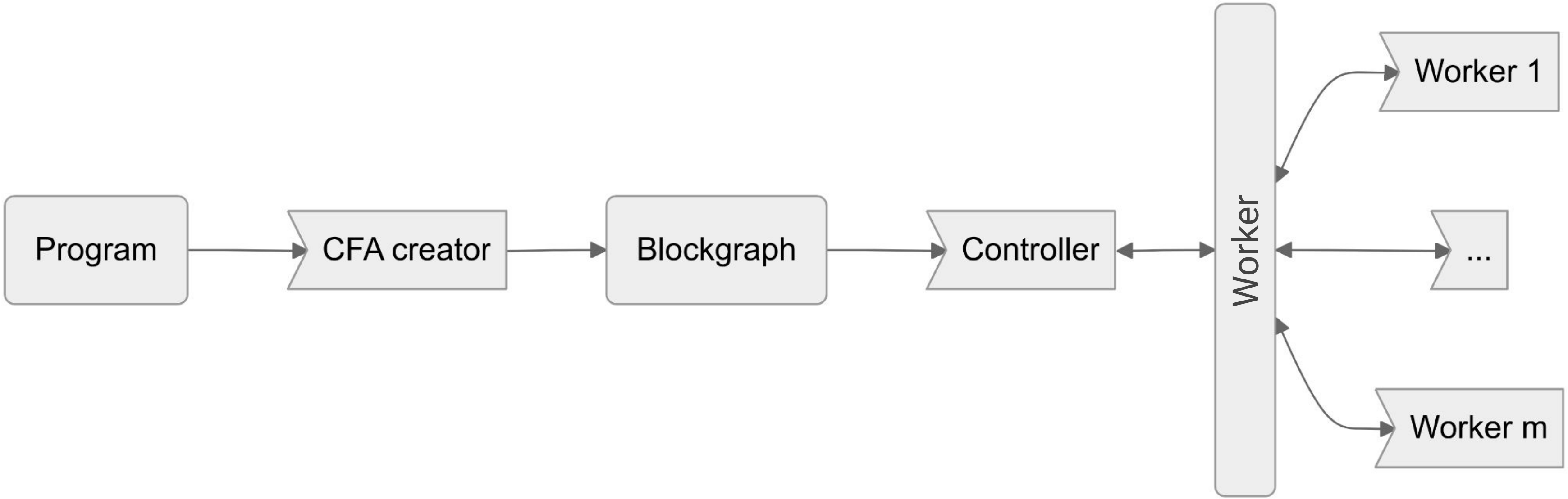
□ Inflexible dependence between workers and blocks

Multi-Processing for Distributed Summary Synthesis



- Worker is fully stateless, not related to any block
- Controller manages conditions for all blocks
- Controller schedules the verification analyses for all blocks

Multi-Processing for Distributed Summary Synthesis



- Worker is fully stateless, not related to any block
- Controller manages conditions for all blocks
- Controller schedules the verification analyses for all blocks

Controller

- Asynchronous communication
- New block verifications are triggered based on incoming messages and block graph
- Current schedule: FIFO

```
StartRequest {
  string programCode
  string specification
  bytes blockGraph
}
```

Controller

Conditions

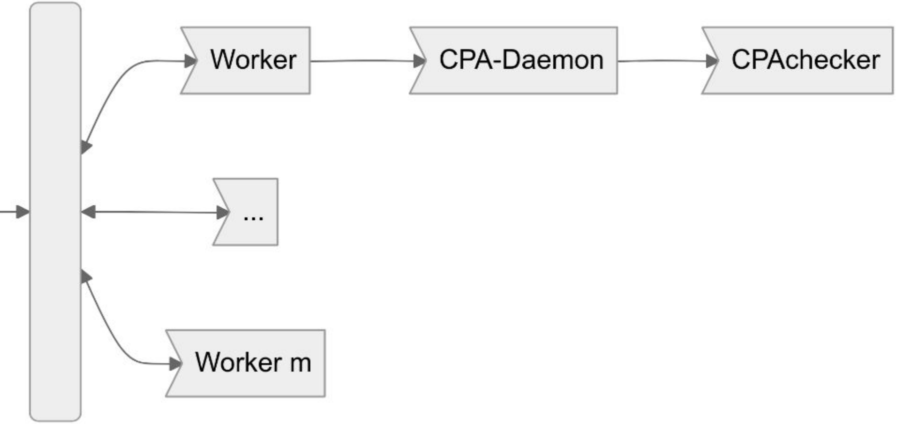
```
VerificationRequest {
  int64 timepoint
  string programCode
  string specification
  bytes blockGraph
  string blockId
  Condition[] oldMessages
  Condition[] newMessages
}
```

```
VerificationResponse {
  int64 timepoint
  Condition[] messages
}
```

Worker

Worker

- Uses CPA-Daemon in backend
- Translates block verification requests into CPA-Daemon calls/CPAchecker options
- Other translation schemes/verification engines possible



Worker: CPAchecker Backend

- We extended CPAchecker to run a single-block analysis for DSS

➔ JSON exports for condition messages and block graph

```
blocks.json:
{
    "L1": {
        "predecessors": [ "L0", "L2" ],
        "successors": [ "L2", "L3" ],
        "startNode": 5,
        "endNode": 9,
        "edges": [ [ 5, 8 ], [ 8, 9 ] ]
    },
    // … snip …
}
```

Worker: CPAchecker Backend

- We extended CPAchecker to run a single-block analysis for DSS

➔ JSON exports for condition messages and block graph

➔ Configuration options to trigger single-block run

```
$ bin/cpachecker \
    --predicateAnalysis-block \
    --option distributedSummaries.importDecomposition=blocks.json \
    --option distributedSummaries.spawnWorkerForId=L1 \
    --option distributedSummaries.knownConditions=L2-vCond.json \
    --option distributedSummaries.newConditions=L0-postCond.json \
    program.i
```

Technology

- Scheduler and Worker implemented as Kotlin Quarkus services
- Communication through gRPC (procedure-based communication)
- Load-balancing with linkerd
- Orchestration with kubernetes

Current State

- Experiments are work in progress
- Example: ldv-linux-3.4-simple/...leds--leds-regulator... with 91 blocks
- Message size is huge because we always send everything
  - Verification request + response: 150–2000 kB each
  - Total: 355MB
  - The message size is not a bottleneck so far
- Redundant work because of poor analysis order: 1037 runs over 91 blocks
- Redundant work in CFA creation and parsing
  - Make CPAchecker directly import CFA, send that instead of program code
  - Use CPA-Daemon backend that reuses CFA

- Next steps: Visualization of work process, smarter controller

Conclusion

- Next Evolution of Distributed Summary Synthesis (DSS)
- CPAchecker as verification engine in the backend
- Enables multi-processing
- Simplifies DSS architecture
- Base for future exploration and development