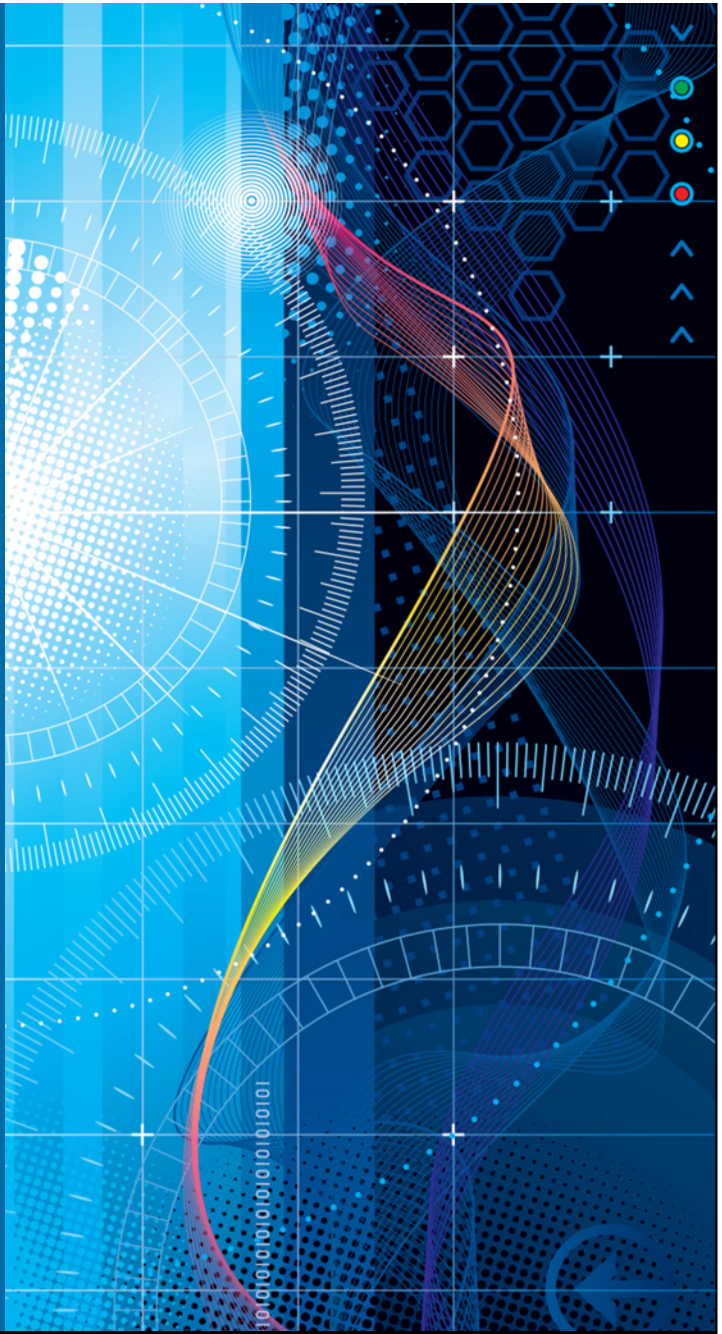
Reducing Architecture Complexity with AADL

Julien Delange <jdelange@sei.cmu.edu>
Jerome Hugues <jerome.hugues@isae.fr>

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

DM-0002644

Objectives

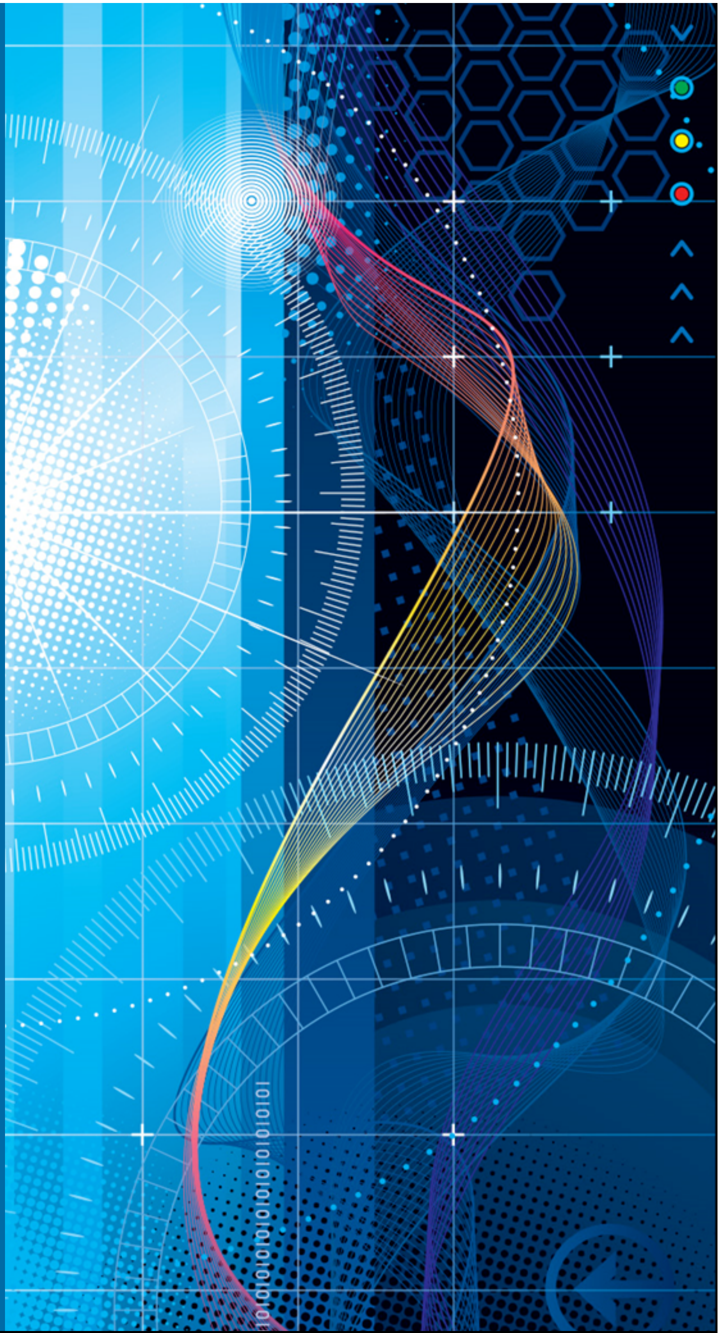
Explain Complexity in models

Detect Complexity in Models

Mitigate Complexity

Cost of Complexity

Complexity in Architecture Models



What is complexity?

Software hard to read & understand

Several factors

- Components coupling (fan-in/fan-out)
- Number of entities (components, connectors)
- Nesting levels

Supporting Metrics

- **McCabe**: connections between component
- **Halstead**: # operands and # operators
- **Zage**: fan-in/fan-out

Why complexity matters?

Increase development costs

Testing: increase # tests and ultimately, development costs

Maintenance: more time to update/upgrade

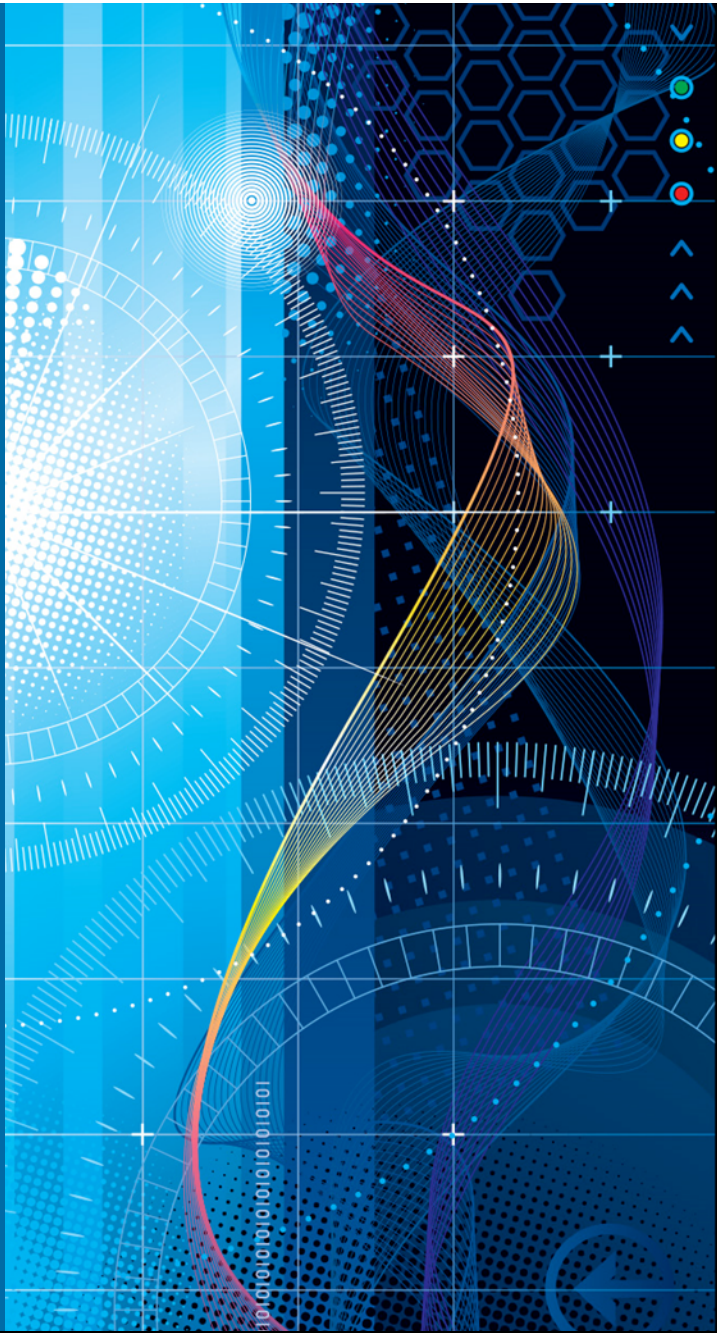
Reduce component reuse

Interfaces design issues

Software is getting a major costs for safety-critical

Reducing complexity = reduce costs, deliver quicker

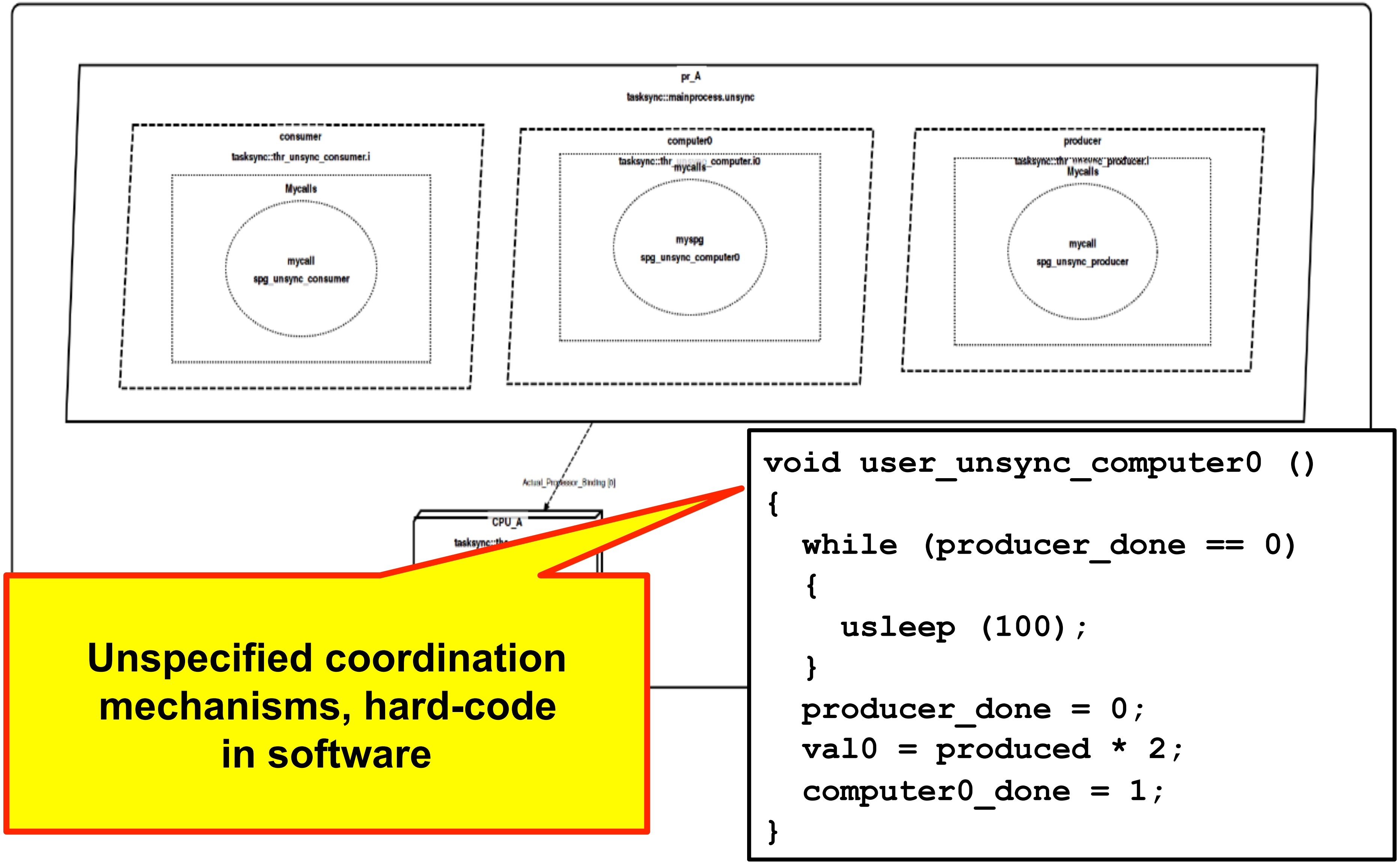
Identifying and Mitigating Complexity



Software Engineering Institute | Carnegie Mellon University

Distribution Statement A: Approved for Public Release; Distribution is Unlimited

Synchronization methods – usual design issue


pr_A
tasksync::mainprocess.unsync
consumer
tasksync::thr_unsync_consumer.i
Mycalls
mycall
spg_unsync_consumer
computer0
tasksync::thr_unsync_computer.i0
mycalls
myspg
spg_unsync_computer0
producer
tasksync::thr_unsync_producer.i
Mycalls
mycall
spg_unsync_producer
Actual_Processor_Binding [b]
CPU_A
tasksync::th

```
void user_unsync_computer0 ()
{
  while (producer_done == 0)
  {
    usleep (100);
  }
  producer_done = 0;
  val0 = produced * 2;
  computer0_done = 1;
}
```

Unspecified coordination mechanisms, hard-code in software

Synchronization methods - impacts

Architecture decision embedded in the code

- Bad separation of concerns
- Architecture decision tightly coupled with code

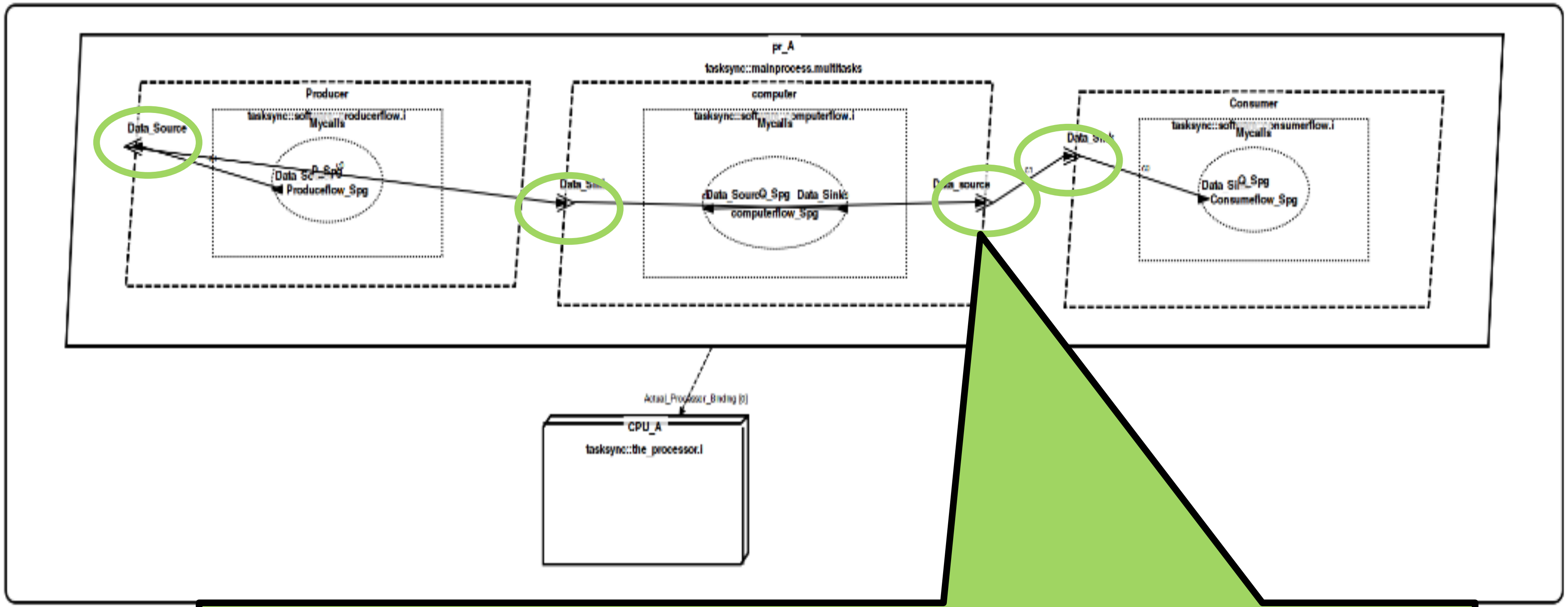
Potential for Analysis

- Scheduling Analysis
- Data flow (latency) analysis

Performance issues

- Performance issues (*busy wait*)

Synchronization methods



Synchronization through event data port, show the synchronization method at the architecture level.

Synchronization methods - benefits

Highlight architecture decisions in models

- Disconnect the non-functional and functional aspects
- Improve code and components reusability

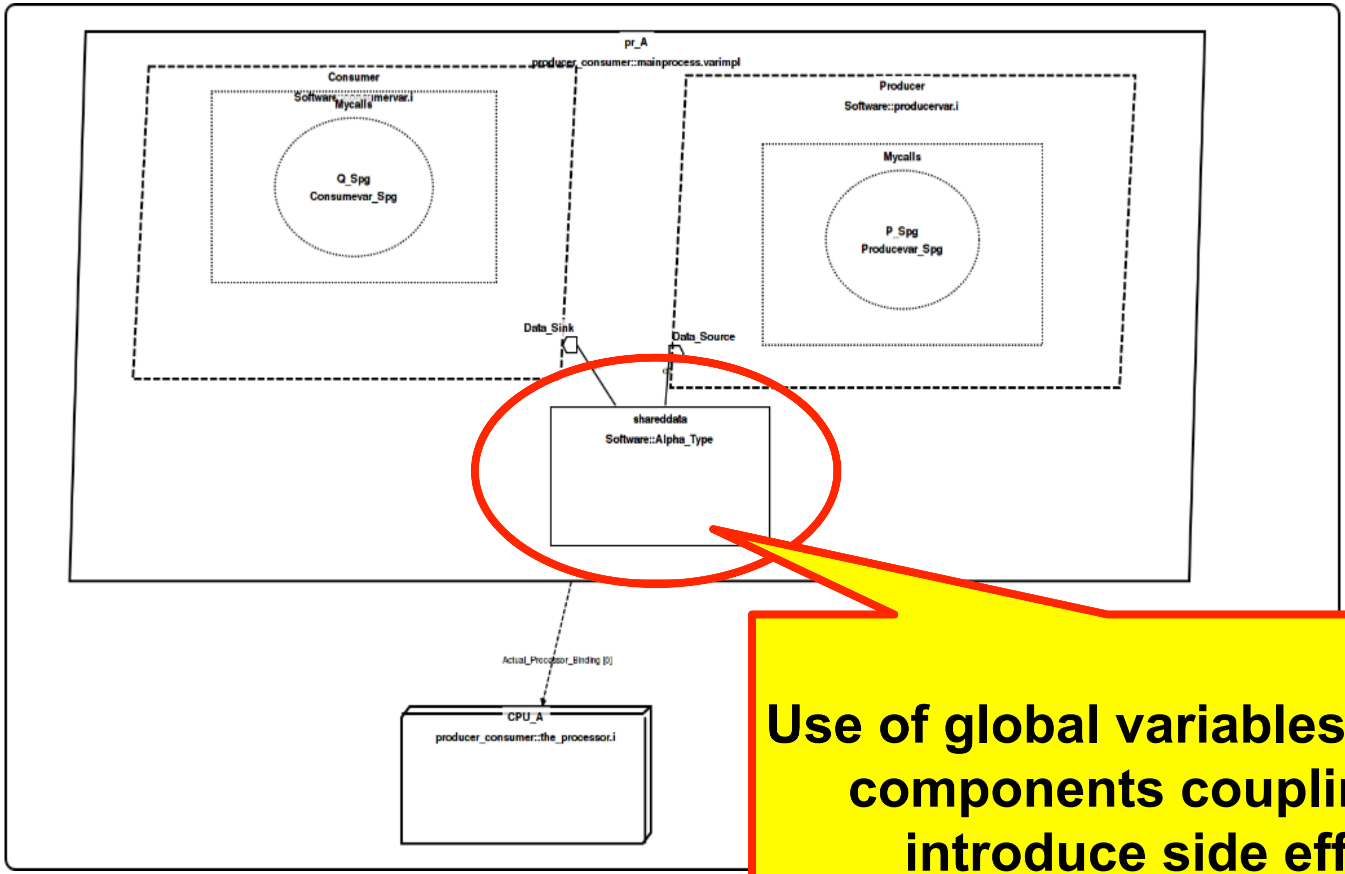
Enable Architecture Analysis

- Scheduling Analysis (tasks scheduling)
- Data flow (latency) analysis

Performance issues

- Avoid inappropriate implementation decisions
- Rely on code generator and potential optimizations

Data Sharing – usual design issues



Use of global variables, increase components coupling and introduce side effects

Data Sharing - impacts

Loss of data flow analysis

- Hard to keep track of who is reading/writing

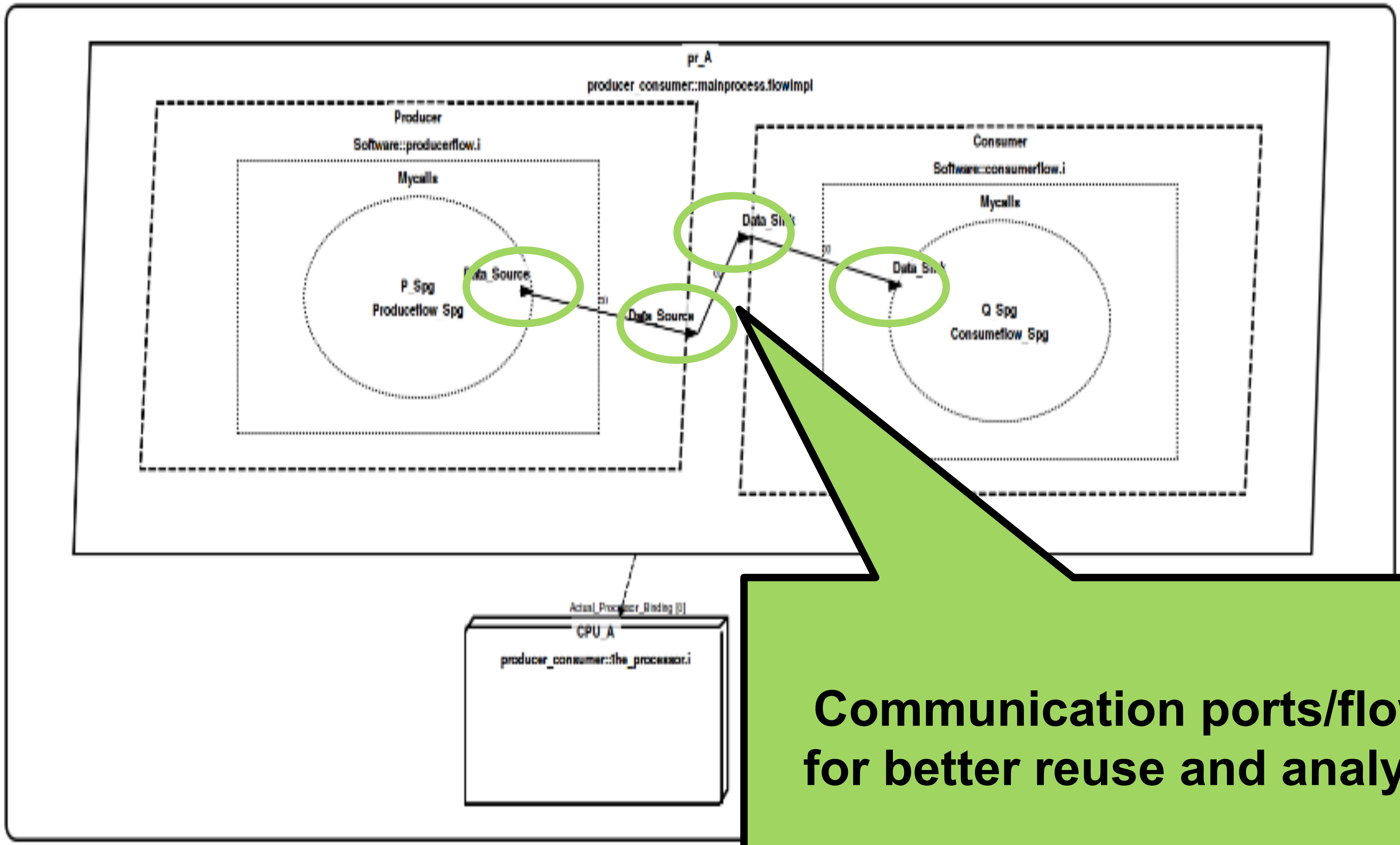
Lack of modularity

- Reuse a component must bring the variable

Potential side-effects

- Unexpected data change, must use synchronizations
- Hard to detect, analyze, might lead to catastrophic issue

Data Sharing – architecture solution



Communication ports/flows for better reuse and analysis

Data Sharing – benefits

Enable data flow analysis

Support for end to end latency analysis

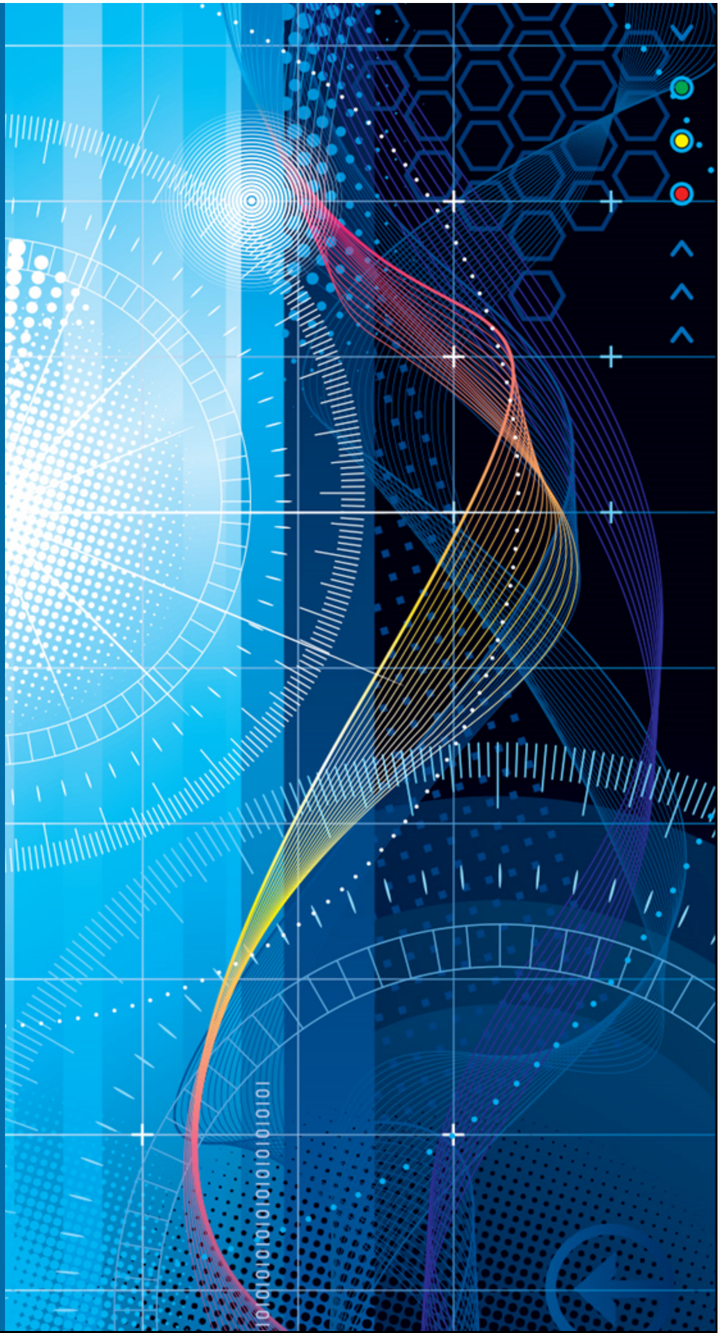
Modular design, simple component reuse

Re-import the component and connect its interface

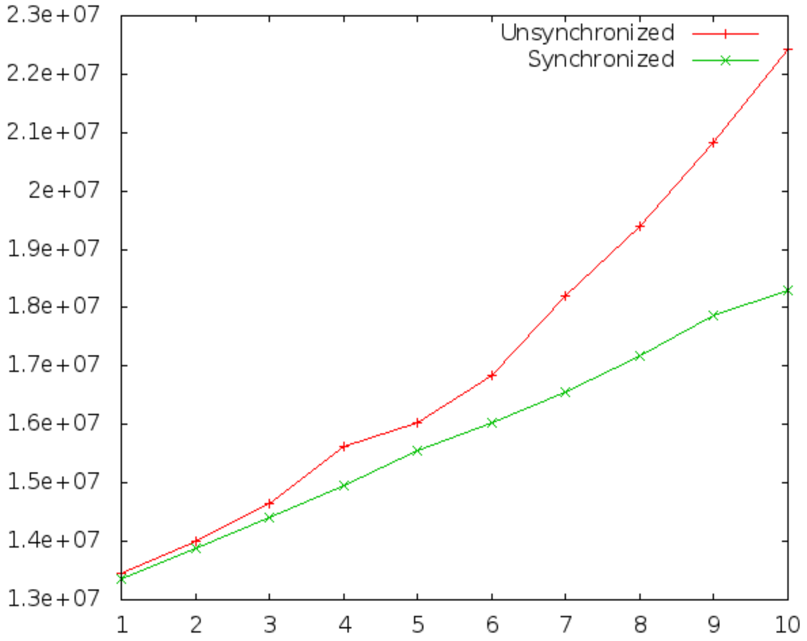
Limited side-effects

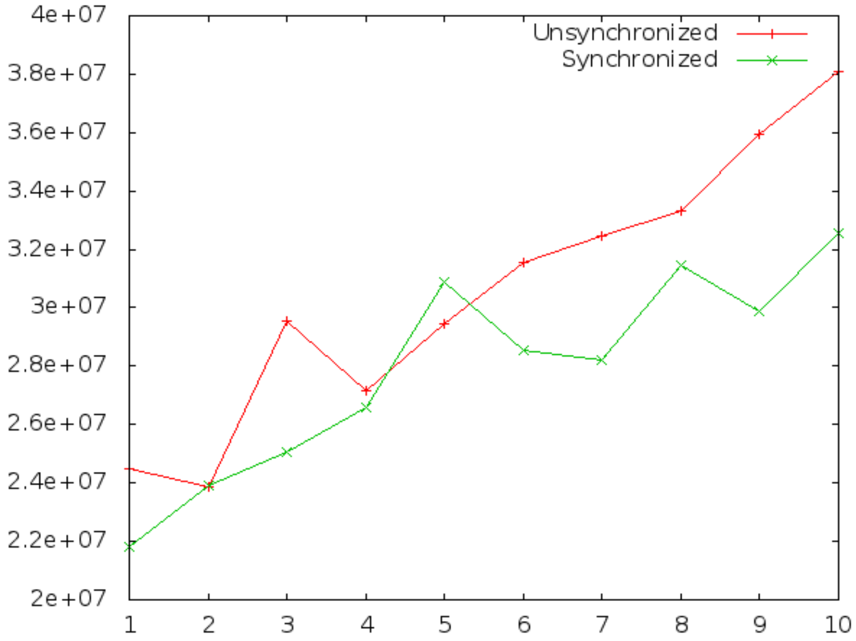
New data impact only the component

Cost of Good Designs



Synchronization - Performance

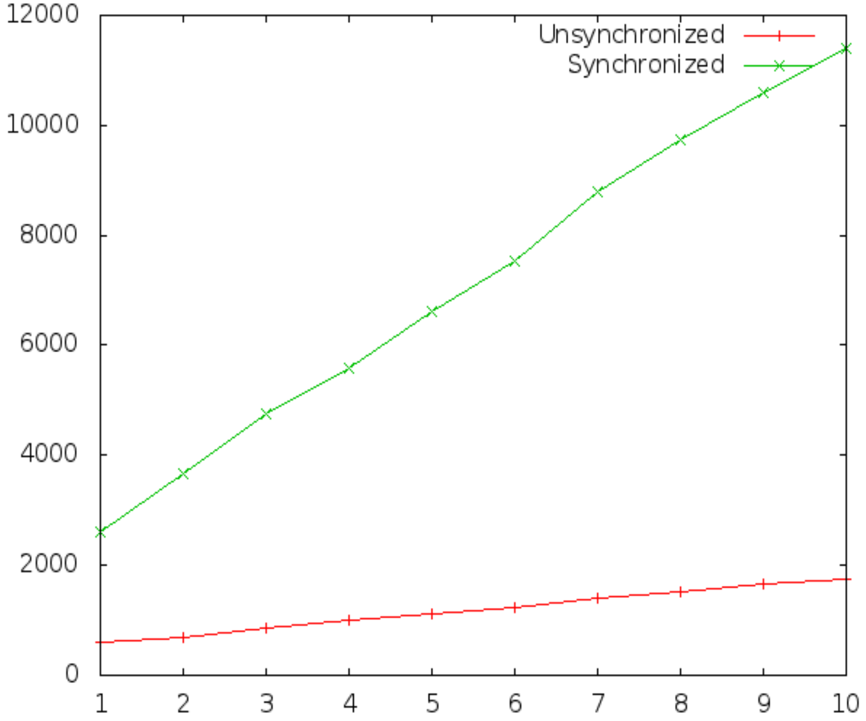




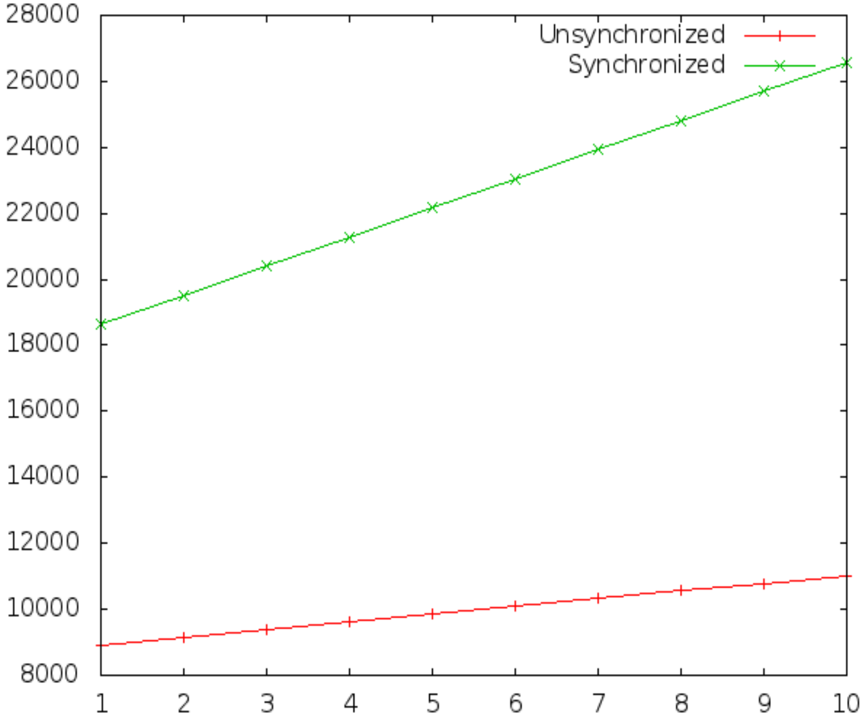
Number of Instructions

Number of CPU cycles

Synchronization - Memory

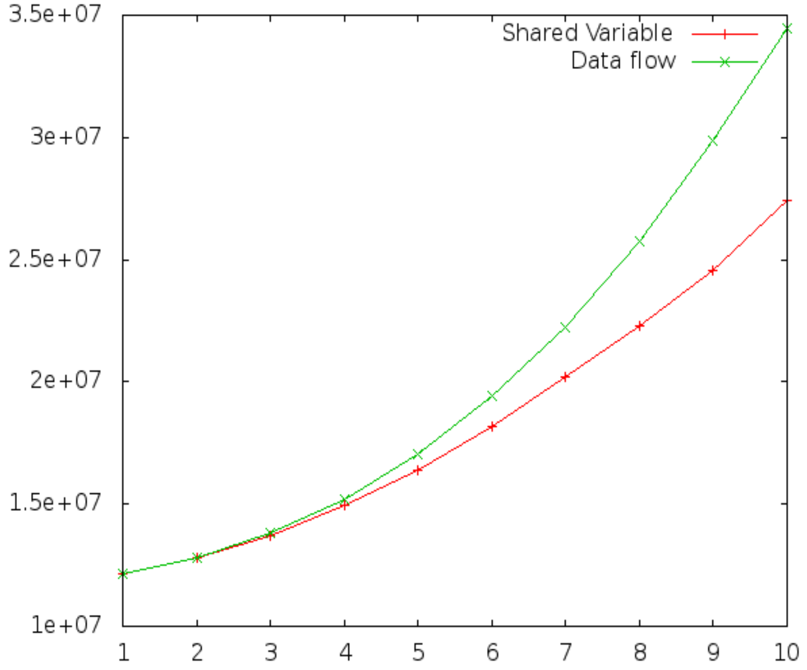


BSS size (global variables)

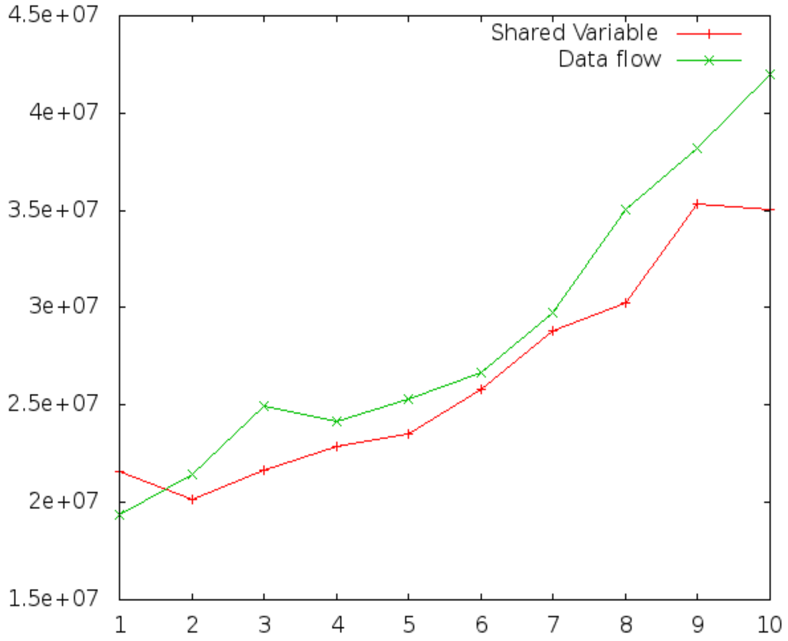


Text Size (code)

Global Variable vs. Data Flow - Performance

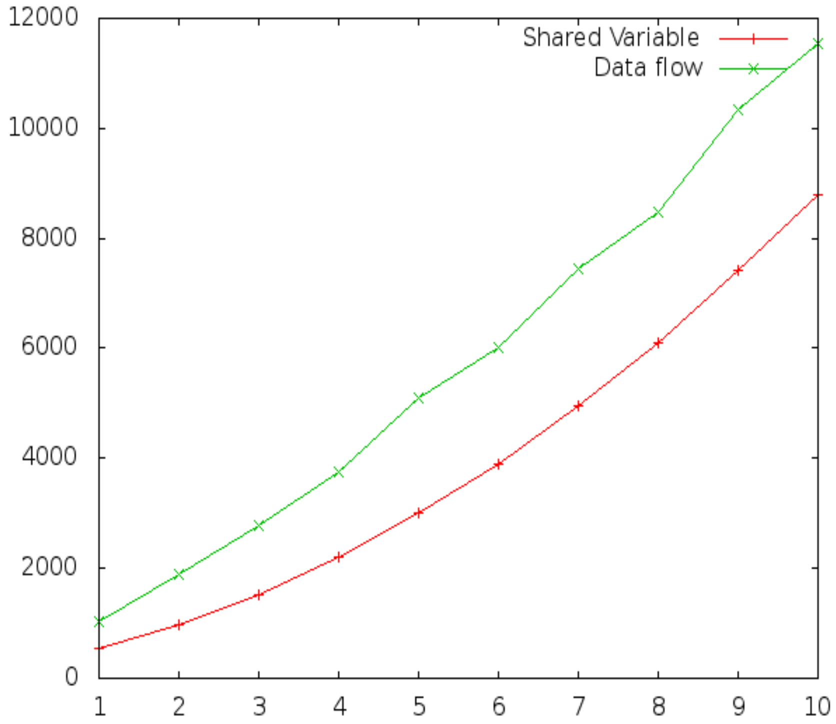


Number of Instructions

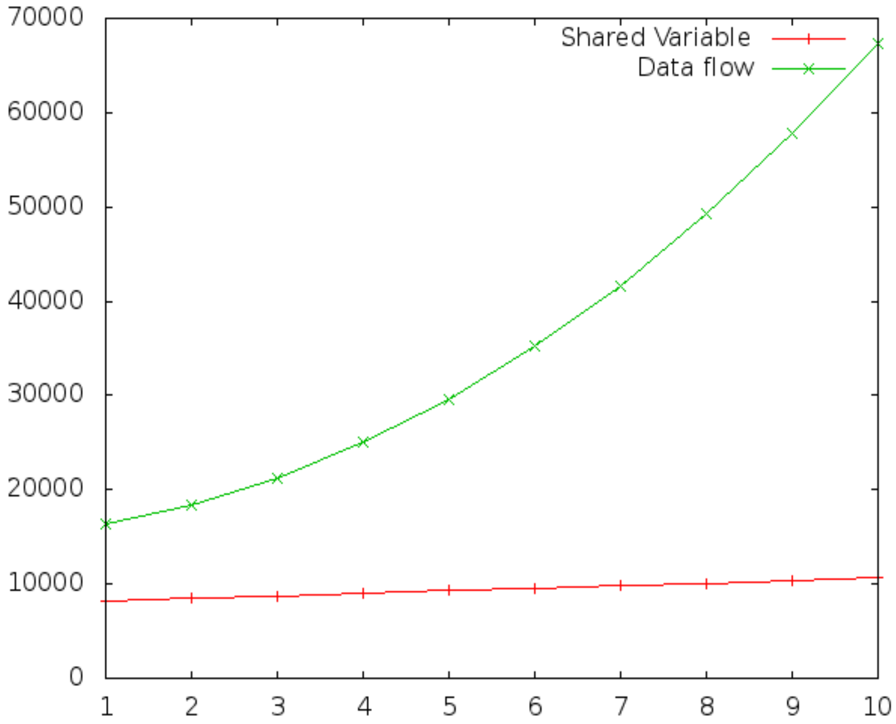


Number of CPU cycles

Global Variable vs. Data Flow - Memory

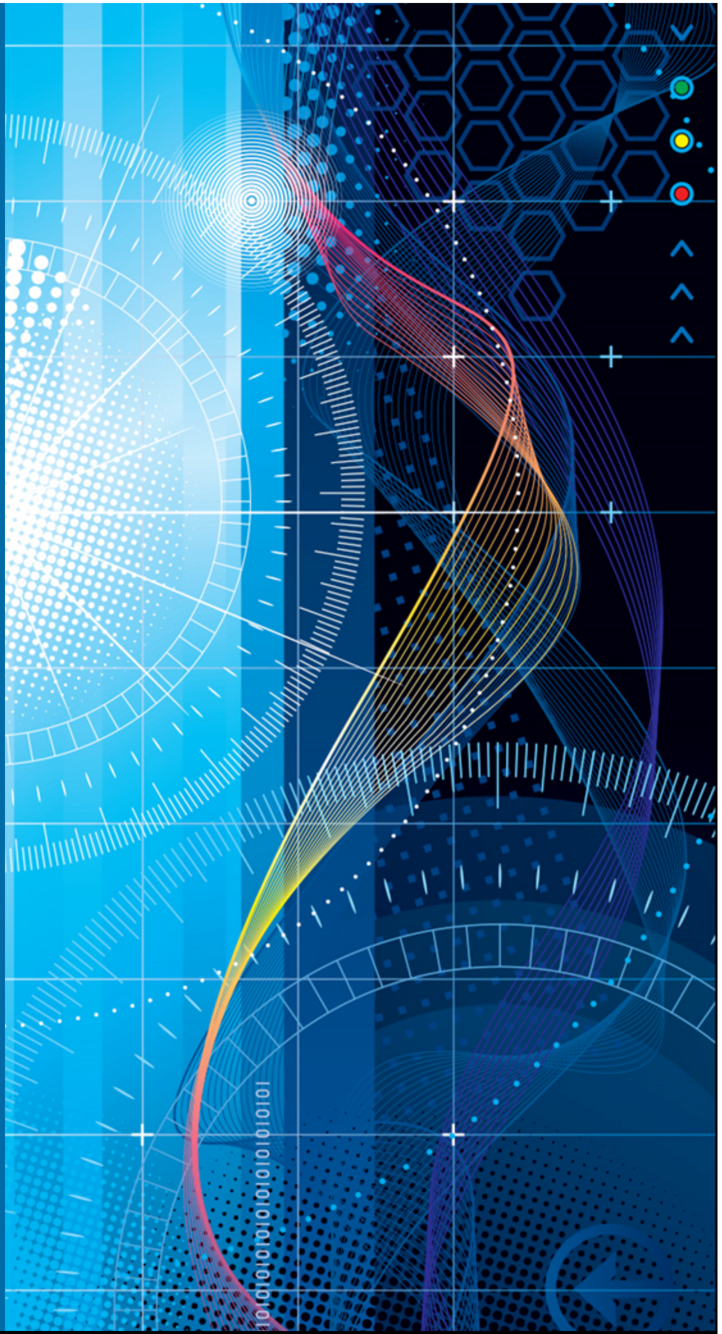


BSS size (global variables)



Text Size (code)

Conclusion



Conclusion

AADL for mitigating architecture-related complexity

Detecting complexity through architecture review

Architecture Analysis for discovering other complexity area

Reducing Development Costs

Testing efforts (e.g. DO178C, ISO26262)

Maintenance & Software reuse

Rely on Code Generator and Associated Optimizations

Avoid manual optimizations

Rely on code generator for creating efficient code