POK – Code Generation for Partitioned Architectures

Julien Delange <julien@gunnm.org>
Forewords

• The POK project
  • Design and implement safe and secure system
  • Complete development process with model-based engineering

• Now, focus on code generation
  • How you can generate code from AADL models
  • Code generation benefits
  • Code generation patterns
Outline

• Code generation overview

• Generation patterns

• Benefits

• Conclusion
Outline

• Code generation overview

• Generation patterns

• Benefits

• Conclusion
Code Generation Overview

- Generate the whole architecture
  - Configuration for module an partitions
  - Deployment

- Existing work
  - Ada/C code generation
  - Rely on general-purpose operating systems
Code Generation - toolset

• Ocarina
  • C code generation functionalities
  • POK-specific code generator
  • ARINC653 flavor to generate ARINC653-compliant code

• POK toolchain
  • Automatize code generation
  • Also verify architecture correctness before generation
Ocarina - functionalities

- ARINC653 configuration
  - AADL to XML code generator
- Partitioned architecture code generator
  - POK flavor: use the POK API
  - ARINC653 flavor
  - Auto-generation of assertions, improve error detection
- CARTS generator (scheduling simulation)
  - Interfacing with other analysis tools
- REAL: model validation
POK toolchain

• Model validation with Ocarina/REAL
  • Architecture consistency
  • Health Monitoring policy impacts
  • Security analysis

• Generation of ARINC653 configuration
  • XML file generation with Ocarina
  • Ease system integration & portability

• Auto-generation of the whole application
  • Code generation with Ocarina
  • Compilation & integration with the POK operating system
Generation process

Code for each partition

Module conf.

Task 2  Task 1
Part 1  Part 2

ARINC653 Module

Julien Delange <julien@gunnm.org> - http://pok.gunnm.org
Outline

- Code generation overview
- Generation patterns
- Benefits
- Conclusion
Generation requirements

- Partition configuration
  - Resources dimensioning
  - Intra-partition ports routing
- Partition initialization
  - Resources creation
  - Tasks initialization
- Tasks execution
  - Interface with application
  - Send/receive data
- Module configuration
  - Partitions scheduling
  - Memory segments
  - Resources dimensioning
- Communications
  - Inter-partitions ports routing
Generated files

• Module
  • deployment.[h|c]: module configuration

• Partition
  • deployment.[h|c]: partition configuration
  • subprograms.[h|c]: interface with application code
  • activity.[c|h]: tasks activity
  • main.[c|h]: partition initialization
  • types.h : types

Julien Delange <julien@gunnm.org> - http://pok.gunnm.org
Module configuration
(deployment.h)

- Partition scheduling (time isolation)
  - Module scheduling policy: time frame allocation
- Memory segments (space isolation)
  - Segments allocation across partitions
- Resources dimensioning
  - Amount of partitions, ports, etc.
- Inter-partitions ports
  - Ports kind, routing policy
Module configuration - sample

ARINC653 Module
Partition configuration
(deployment.h)

• Process scheduling
  • Partition scheduling policy (algorithm)

• Resources dimensioning
  • Amount of threads, ports, allocatable memory, ...

• Determine included functionalities
  • Cipher algorithms, libc, libm, ...

• Intra-partition ports
  • Routing policy
Partition configuration

Conf part 1
Task 2
Task 1
Conf part 2

Task 1

Julien Delange <julien@gunnm.org> - http://pok.gunnm.org
Partition initialization (main.c)

- AADL threads
  - Create threads according to their type

- AADL ports
  - Initialize inter and intra-partition channels
  - Consistency with configuration

- Misc initialization
  - Cipher algorithms
Partition configuration
Partition behavior

- **AADL threads** *(activity.c)*
  - Thread type (period vs. sporadic)
  - Call sequence
  - Ports (intra and inter-partition) : receive/send data

- **Subprograms** *(subprograms.c)*
  - Traditional subprograms (C/Ada vs. Simulink/Esterel)

- **Types** *(types.h)*
  - Interface AADL with C/Ada types
Partition behavior

Task 2
Part 1

Task 1
Part 2
Outline

- Code generation overview
- Generation patterns
- Benefits
- Conclusion
Improve reliability

- Avoid traditional errors
  - Ease certification

- Enforce specification
  - Code generation according to designer's requirements

- Consistency with configuration
  - Avoid use of unallocated resource
Better analysis

- Predictable code
  - Code created from patterns
  - Predictable overhead

- Ravenscar compliance
  - Useful for High Integrity systems
  - Past experiments with PolyORB-HI-C
Outline

- Code generation overview
- Generation patterns
- Benefits
- Conclusion
Conclusion

• Improve system creation
  • Predictable code, better analysis
  • Enforce specification requirements
  • Avoid all errors introduced by developers

• Support for various operating systems
  • Generation of ARINC653 XML files
  • ARINC653 compliant code generation
Questions ?