Configuration and Deployment of Distributed Real-time Embedded Applications Using an Architecture Description Language

Bechir ZALILA (bechir.zalila@enst.fr)

Definitions

- Deployment : "*Placement of the distributed application components on their corresponding physical locations and preparing them to be run*"
  - Requires the placement of additional middleware components
    - Send messages through the network from sender nodes (*stubs*)
    - Receive messages from the network on receiver nodes (*skeletons*)
    - Addressing tables to allow nodes to "reach" each others
- Configuration: "*The opportunity to parameterize the components selected and placed during the deployment phase*"
  - Communication protocol parameters
  - Number of the communication channels to be opened on each node
  - Data marshalling/unmarshalling parameters

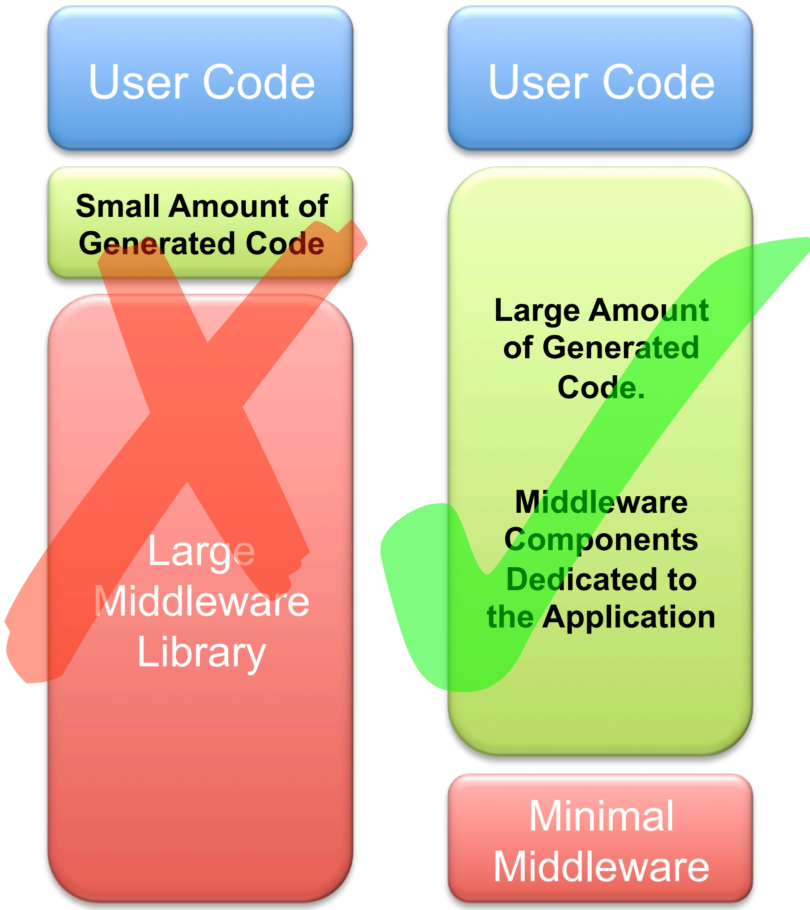
Deployment and Configuration: Origins

- Middleware architectures that help D&C
  - ⌘ Configurable middleware (TAO…)
    - Components selected and parameterized depending on the application properties
    - Uses design patterns
  - ⌘ Schizophrenic middleware (PolyORB…)
    - Cohabitation and interoperability between heterogeneous distribution paradigms
    - Choice of the concurrence profile
- Standards that make D&C easier
  - ⌘ OMG's Deployment and Configuration Specification
- Tools that automate D&C
  - ⌘ COSMIC: Based on CCM and TAO
  - ⌘ AUTOSAR: Flexibility and scalability for automotive systems

- Critical systems additional requirements
  - ⌘ Ravenscar profile (Ada)
    - Guarantee the static analyzability of high integrity systems
  - ⌘ SPARK (Ada)
    - Add annotations to Ada code to allow performing proofs on the applications

Research Issues

- Building a production process that includes:
  - Analysis of the distributed application
    - Semantics, schedulability, verification…
    - Analyses often **not** correlated and must be done by different tools
    - Compatibility with well-known analyzes (RMA…)
  - Deployment of the distributed application
  - Automatic configuration of the middleware according to the application properties
  - Automatic Integration along with the user components



Objectives

- Model and analyze DRE applications
  - ADLs, especially SAE AADL (*Architecture Analysis & Design Language*)
- Deploy and configure an *ad hoc* middleware
  - Execution platform for the AADL
  - Schizophrenic middleware architecture
- Rely on a massive code generation
  - Encapsulate the user code (*glue code*)
  - Produce a large part of the middleware
  - Generate for several languages (Ada, C…)
  - One code generator per language
    - Easily extensible production process

Approach: Modeling

- Specify a subset of AADL that must be used by the user
  - Additional semantic analyses to ensure model coherence
- Rules to interface user code with generated glue code
  - Data type mapping rules
  - Subprogram mapping rules
- Rules to interface user code with applicative components
  - Access to thread interfaces
  - Shared data
- New AADL properties to control deployment
  - Programming language
  - Execution platform

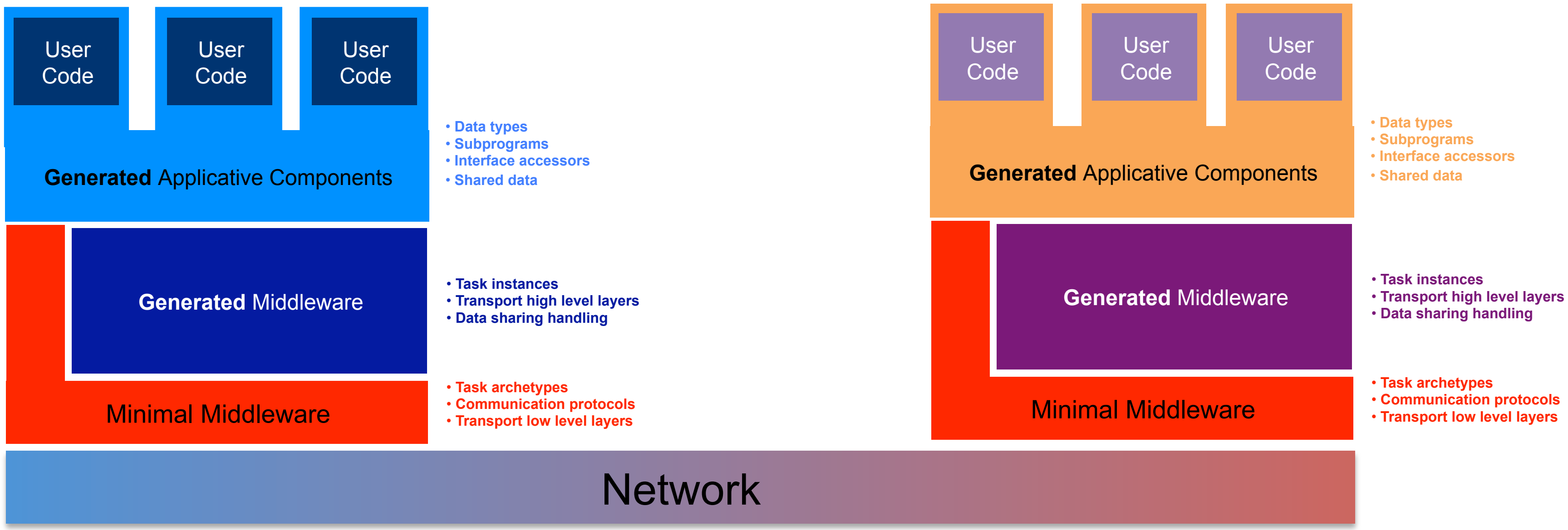
Approach: Middleware

- Design and build a minimal middleware which contains the components that are common to all applications
  - ⌘ One minimal middleware per programming language
    - Same provided services
  - ⌘ Guarantee an efficient and high-performance middleware core layer
  - ⌘ The biggest part of the middleware is automatically generated

- Components:
  - ⌘ Task archetypes
    - Periodic tasks
    - Sporadic tasks
    - Timed tasks
  - ⌘ Communication protocols
  - ⌘ Low level transport layers

Approach: Code Generation

- Automatically generate code depending on the AADL component properties
  - ⌘ One code generator per programming language
  - ⌘ Generate applicative components and middleware components



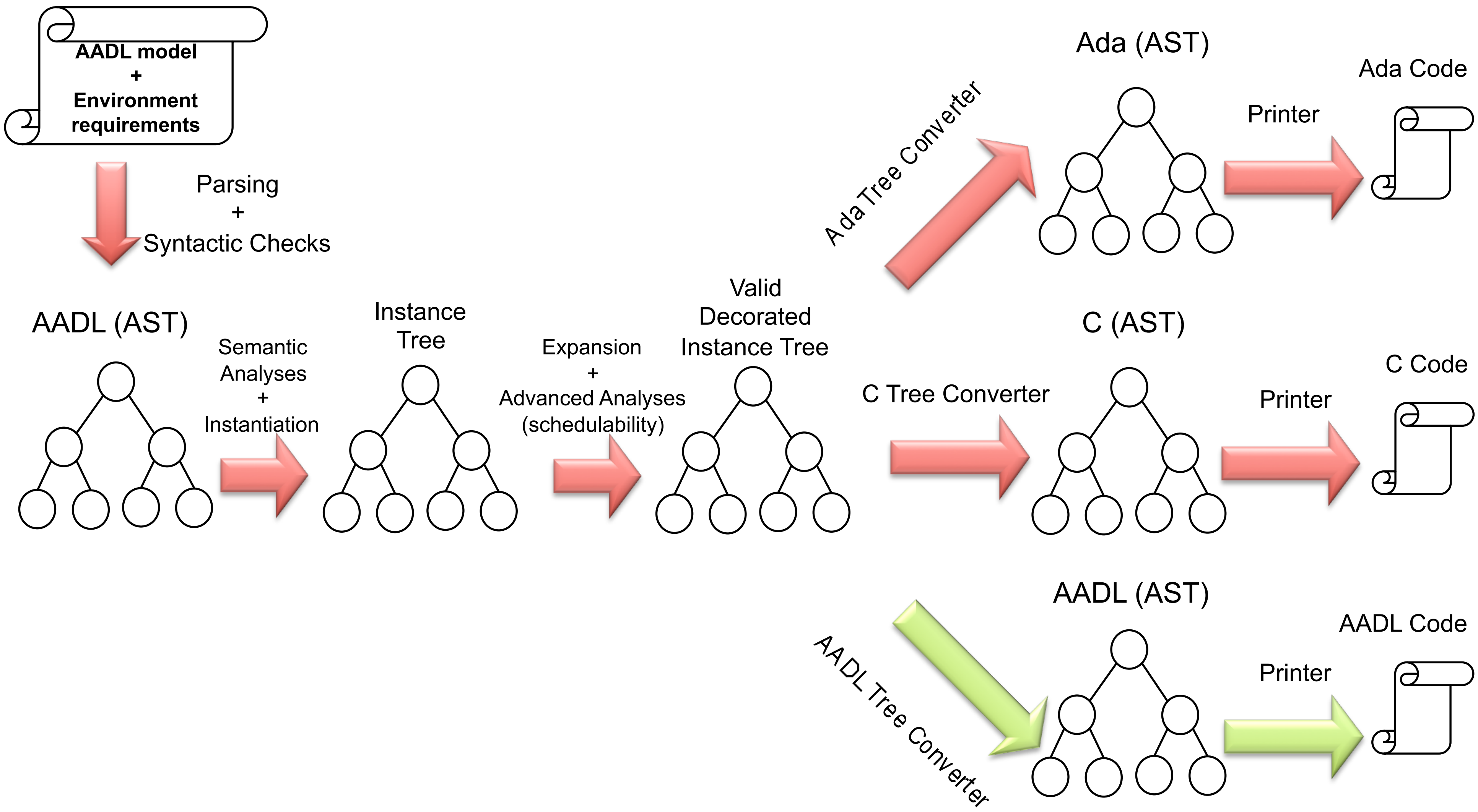
Code generation (1/2)

- Classical approach
  - ⌘ Model processing and model transformation framework (meta-modeling, Eclipse)
  - ⌘ AADL syntactic tree traversal and “on the fly” code generation

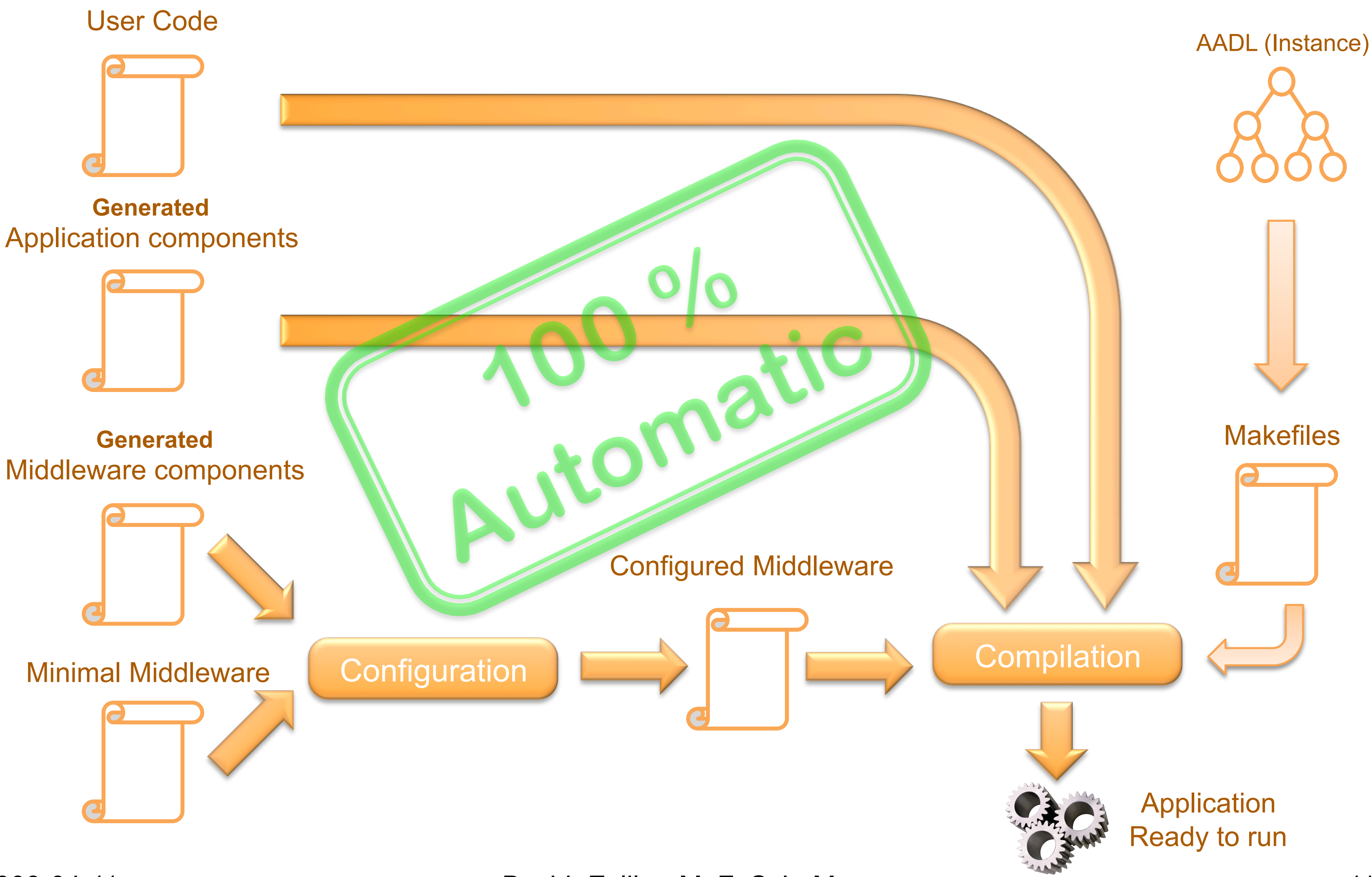
- Motivation for our approach
  - ⌘ An experience and a set of tools to manipulate abstract syntax tree and generate code from description languages (Conception of an improved IDL compiler)
  - ⌘ Total control of the production process while still having a maintainable product
  - ⌘ Better handling for the dependency against a well known API

- Adopted approach
  - ⌘ Generator structure very similar to a compiler:
    - Frontend
    - **Instantiation**
    - Expansion
    - Backend
  - ⌘ Build an abstract syntax tree (AST) for the target language by applying transformation rules on the AADL tree
  - ⌘ Code generation from the target AST

Code generation (2/2)



Results: 1 - Production Process



Results: 2 - Analyzable Code Generation

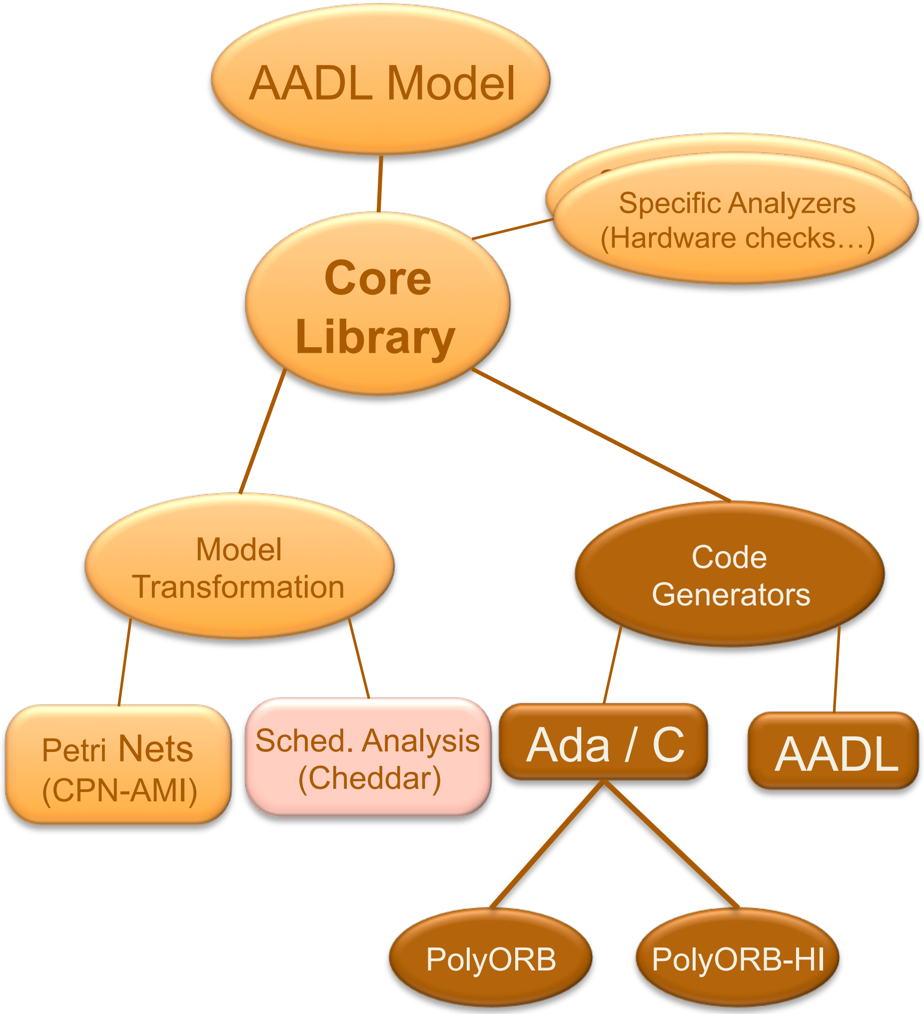
- Generate only the code the application needs
- All resources and requirements computed at code generation time
  - ⌘ No runtime configuration (communication protocol…)
    - No dynamic allocation
  - ⌘ No complex circuitry to select a service at runtime
    - No object oriented programming
- Hard real time constraints specific to High Integrity (HI) systems
  - ⌘ Analyzable concurrency model:
    - Ravenscar profile, Ada
    - Equivalent concurrency profile for the C language (Work in progress…)
  - ⌘ Restrictions of the programming languages to the high integrity systems
    - Even more restrictive than the Ravenscar profile
  - ⌘ Interface with GNAT (**`gnatcheck`**, **`gnatstack`**, **`gnatmetric`**)
    - Advanced memory verification (Ada)

Results: 3 - Middleware

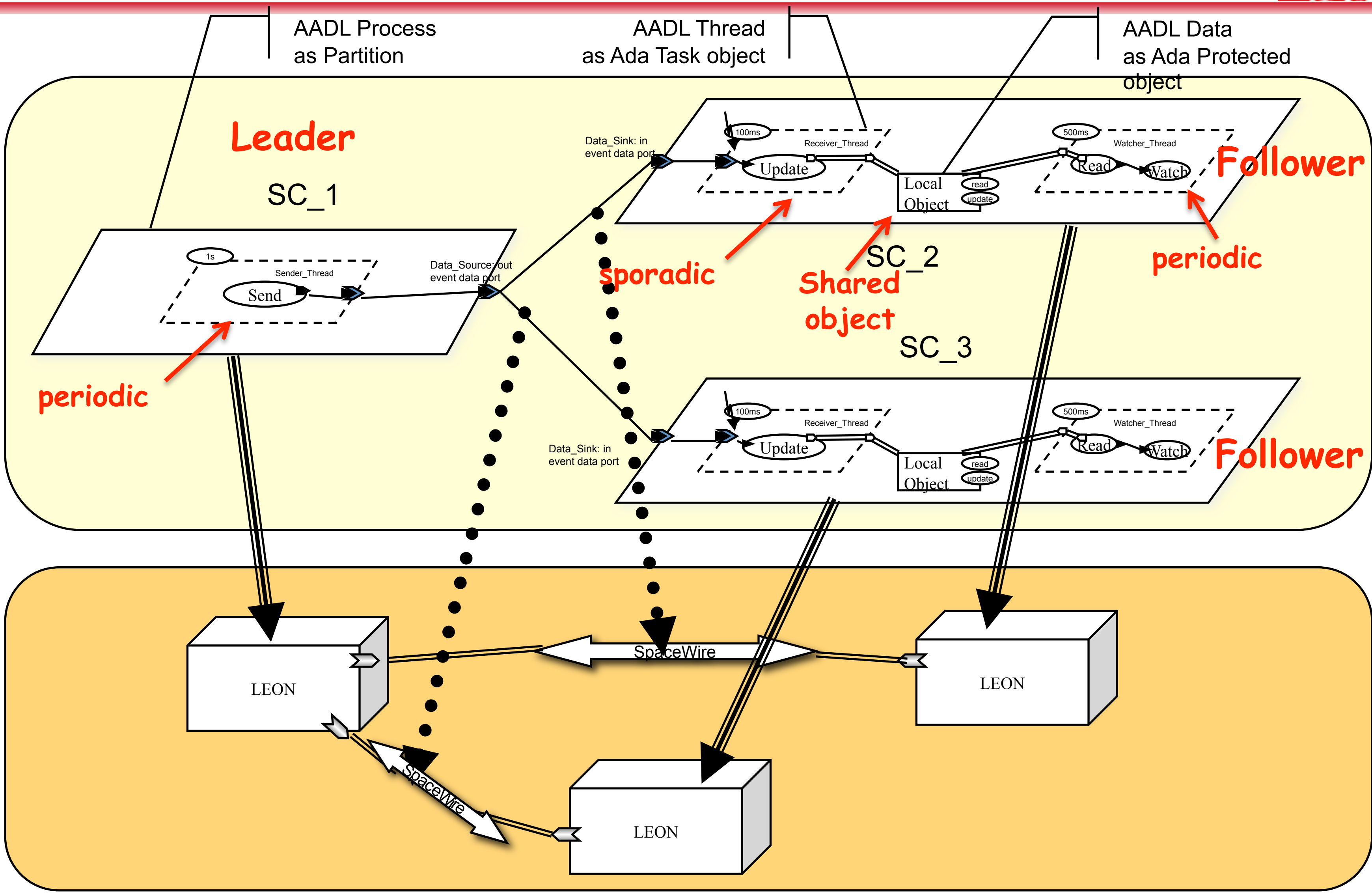
- PolyORB-HI: an AADL runtime
  - Supports AADL constructs
    - Periodic and sporadic threads, data sharing, etc.
  - Automatically configured from the AADL model
    - Resources computed and allocated statically
    - No intervention required from the user
  - Small memory footprint
    - The larger part of the application is produced during code generation
  - Conformant to the Ravenscar Profile and the HI system annex
  - Contributed to the thematics of “middleware factories”
    - For each DRE application, generate a dedicated middleware
- Ported to several embedded platforms
  - Native
  - ERC32
  - LEON2

Results: 4 – Tool chain

- Continuation of the research work of T. Vergnaud
- Ocarina: libraries and tools to manipulate AADL
  - AADL parsers and printers
  - Semantics verification
- Specific operations
  - Model transformation
  - Proposition on configuration and code generation
- Code generation
  - Ada/PolyORB
  - (Ada, C)/PolyORB-HI



Case study: MPC (Multi-Platform Cooperation)



Case study: Metrics

- Most of the interaction patterns
  - Periodic and sporadic threads
  - Shared data
  - Distribution performed transparently
  - Configuration and deployment of the nodes
- Verification of the AADL model
  - Links between the hardware and the software resources
  - Data types
  - Connection coherence (data flow)
  - Schedulability (Cheddar)
- ASSERT project
  - Final demonstration
- Generated code
  - All the code is generated by Ocarina from the AADL model (except the behavioral part)
  - Conform to all HI system restriction
- Executables
  - Memory footprint: 1.1MB
    - Generated MW : **54,3 KB**
    - Minimal MW : **47,7 KB**
    - User code : **8,4 KB**
    - Task stacks : 512 KB
      - Allocated statically in the executable
    - OS Libs : 249,7 KB
    - Drivers : 28,3 KB
    - Kernel : 238 KB
  - Demonstration
    - LEON2 + SpaceWire Bus
    - Simulated using *tsim Pro*

Conclusions and Perspectives

- Fulfilled objectives
  - ⌘ Proposition of a production process for the DRE systems
  - ⌘ Instantiation of this process using Ocarina and PolyORB-HI
    - Ocarina: Modeling, analysis and code generation
    - PolyORB-HI: Runtime for the AADL
  - ⌘ Possible interface with generated user code
    - LUSTRE
    - SDL
  - ⌘ Positive experimentation and feedback for the examples
    - ASSERT project partners
    - SAE partners
  - ⌘ Successful final demonstration of the ASSERT

- What remains to be done
  - ⌘ Support of AADLv2
  - ⌘ Writing of the code generation annex for AADLv2 standard
  - ⌘ A performance comparison with other tools