Generating code from AADL models, code patterns used by Ocarina

Jérôme Hugues, ENST
ENST: Leading Engineering School in Information Technology

- 1330 students: engineer, Master, PhD degrees
  - C/S, Network, Signal, Electronics
- 160 full-time professors
- 10,000 ENST engineers all over the world
- 80 sustainable companies started since 1999

- Research in ENST is part of the LTCI laboratory
  - LTCI is part of French CNRS, for communication technologies
- ENST/INFRES is our C/S dept, focuses on
  - software & system engineering, network technologies, security, quantum networks, cryptology, artificial intelligence, expert systems, natural languages processing, databases, semantic Web.

- Industrial & Academic partnerships:
  - France Telecom, Alcatel, Thales, ESA, ONERA, SAGEM, AdaCore, EADS, EDF, Fraunhofer, Gemalto, …
  - UPMC / LIP6, INRIA, ENS, EPFL, Ecole des Mines, …
ENST Research topic: Methods for DRE

- Context: IST-ASSERT, industrial partnerships
  - AADL to help in automating system construction
    - From early requirements to refinements
    - Through V&V
    - And then code generation

- Use of AADL to build DRE
  - Address configuration, deployment issues
  - Suppress tedious/error-prone work
    - Setting up resources
    - Instantiating concurrent, distributed entities

- Take advantage of other AADL tools for V&V
Ocarina Tool Suite: [http://ocarina.enst.fr](http://ocarina.enst.fr)

- Library to manipulate AADL
  - Parsers and printers
  - Semantic checks
  - Model transformation, code generation
  - Run-time configuration
- Ocarina 1.0 (Jan. 2007)
  - Code generator
    - Ada/PolyORB
    - Ada/PolyORB-HI
  - V&V
    - Schedulability (Cheddar)
    - Petri Nets

Diagram:
- Core library
- Model Transformation
- Specific Analyzer
- Code Generator
- Run-Time
- Ada, C
- AADL
- Petri Nets
Two code generation strategies

- One common language: Ada
  - Strong typing, compiler checks

- Two middleware
  - PolyORB: QoS-based middleware
    - OO patterns for modularity, many QoS parameters
    - POSIX-like patterns as a wrapper for concurrent entities
      - No tasking, Ravenscar, pure POSIX, full Ada
  - PolyORB-HI: High Integrity middleware
    - No OO, everything is static
    - Ravenscar Ada concurrency

- Two situations for code generation
  - Different semantics, dimensioning, etc.
Common guidelines for code generation

- AADL to execute user code *and* to
  - Check and preserve properties
  - Reduce user intervention, to avoid adding its errors
- Chose to define a framework that calls user’s code
- Instantiate this framework with the architecture of the system
  - (almost) suppress manual API manipulation
  - Allows resource dimensioning by tools
- Differs from drafts of AADL language Annex we used
- Driven by experiments & IST-ASSERT project
PolyORB-QoS: POSIX-like patterns

- Mutexes, cond. var., threads built after POSIX
- Factories for configuration, dynamicity allowed
  - Memory allocation, OO (dynamic dispatching), …
- Easy points: dimensioning, configuration
  - Allocates, dispatches then runs
- Difficult point: interactions
  - Granularity of critical sections ?
  - Producer/consumer patterns ?
PolyORB-QoS: patterns (1/3)

- AADL processes -> partitions of the application
- Threads -> POSIX threads
  - One-to-one mapping of its properties, e.g.
    `Create_Periodic_Thread(My_Proc'Access);`
- Types -> Language types
  - No limits on types: arrays, strings, floats, etc.
    ```
    data message
    properties
    ARAO::data_type => integer;
    end message;
    ```
PolyORB-QoS: patterns (2/3)

- Protected type

```plaintext
data internal_message
properties
  ARAO::data_type => integer;
end internal_message;

data message
subcomponents
  Field : data internal_message;
features
  method : subprogram update;
properties
  ARAO::Access_Control_Protocol
  => Protected_Access;
end message;

type Message is record
  Data : Partition.Internal_Message;
  Mutex : Mutex_Access;
end record;

procedure Update
  (This : in out Message;
   Value : in Partition.Internal_Message) is
begin
  Enter (This.Mutex);
  Repository.Update (This, Value);
  Leave (This.Mutex);
end Update;
```
PolyORB-QoS: patterns (3/3)

- Inter-task communication through ports
  - Built after request-passing mechanism
  - Scale up to distribution

```plaintext
function Execute_Servant
(Obj : access th2_Object;
  Msg : PolyORB.Components.MessageClass)
return PolyORB.Components.MessageClass;

type th2_msg_in_buf_type is array (1 .. 1)
of Partition.message;

Ports are built after the worker lane pattern, equivalent to RT-CORBA lane
```
PolyORB-HI: HI patterns

- Requirements from the ASSERT project
  - Ravenscar profile for static concurrency
  - HI restrictions: No_Float, No_Allocator, No_Dispatching, ...
  - Checked at compile-time, ease validation

- Code generation more complex
  - Full allocation of resources (buffers, tasks, etc)
  - Finer configuration of mechanisms, unwind all dispatching calls
PolyORB-HI: patterns (1/3)

- AADL process -> executables
- Threads -> Ravenscar tasks
  - Canonical patterns
    - Periodic
    - Sporadic

```plaintext
task body My_Sporadic_Task is
  -- ...
  begin
    loop
      Monitoring_Event.Wait (..);
      -- Do_Useful_Work
      Next_Activation := Activation_Time
      + My_Min_Interarrival_Time;
      delay until Next_Activation;
      end loop;
    end My_Sporadic_Task;
```

- Types -> Language types
  - Only types of bounded size, no floats
PolyORB-HI: patterns (2/3)

- Protected data are mapped onto Ada PO

```ada
data POS features
    Update: subprogram Update;
    Read: subprogram Read;
end POS;

data implementation POS_Impl subcomponents
    Field: data POS_Internal_Type;
end POS_Impl;

type POS_Internal_Type is
    new Standard.Integer;

protected type POS_Impl is
    procedure Update;
    procedure Read;
private
    Field: POS_Internal_Type;
end POS_Impl;
```
PolyORB-HI: patterns (3/3)

- Inter-task communication through ports
  - Ravenscar pattern: protected object
  - Scale up to distribution, PO hides internals
    • Caller drops a message, callee is notified

```adam
thread implementation Q.Impl
  calls {
    Q_Spg : subprogram Ping_Spg;
  };
  connections
    parameter Data_Sink -> Q_Spg.Data_Sink;
  properties
    Dispatch_Protocol => Sporadic;
    Period => 10 Ms;
    Compute_Execution_time => 1ms .. 3ms;
  end Q.Impl;

PingMeHandler : Protocols.HandlerTaskType
(From => Deployment.Node_A_K,
  Dispatcher => Process_Request'Access,
  Priority => System.Default_Priority);
```

---

AADL to code patterns
2007/01/22
Assessments

- Code generation tested on “real-world” examples
- PolyORB-QoS
  - Ring, multicast, BBS canonical examples
- PolyORB-HI
  - Ping, Some_Types, ASSERT tests
- Demonstrate it can be applied to complete systems
- Need to concentrate on models to express both code generation patterns and V&V
  - Working in collaboration with Cheddar
Conclusion and Ongoing Work

- AADL as a basis for code generation
- Ocarina targets two complementary middleware
  - QoS: POSIX-like API, dynamic
  - HI: Ravenscar, HI restrictions, no dynamicity
- Ongoing work in the context of IST-FP6 ASSERT (ESA, …)
  - Addition of more properties for Ada2005 APIs
    - Group budgets, execution timers
  - Consensus pattern
- Papers (more on Ocarina’s web pages)
  - ISORC’07: “Combining Model processing and Middleware Configuration for Building Distributed High-Integrity Systems”
  - AdaEurope’07: “Generating Distributed High Integrity Applications from their Architectural Description”
  - RTSS-WiP’06: “Middleware and Tool suite for High Integrity Systems”