A correct-by-construction AADL runtime, proof of a safety-critical middleware using SPARK/Ada

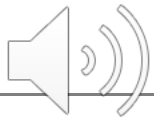
Jerome Hugues

SSD/ACPS/MBE

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. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM21-0463



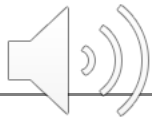
Model-Based System/Software Engineering Overarching objectives

MBSE complements typical software programming with **models** to

1. Organize stakeholders needs and elicit requirements
2. Capture system elements – design, reverse engineering or COTS
   - Inte
   - a sy
3. Apply
   - Syn
   - Qua
4. Synthesize portions of software from models

Research questions:
How to model and how to analyze models?
How to ”compile” models to code faithfully?

Context: safety-critical systems and AADL



This paper contributions

Main contribution:

A MBSE workflow from AADL to Ravenscar-compliant SPARK source code

How? *Expand previous research on code generation from AADL*

1. Define a subset of AADL compliant with the Ravenscar profile (extends [1])
   - As a set of validation rules exercised on model prior to code generation
   - Shared to foster interoperability with other tools
2. SPARK-compliant runtime for AADL, with updated code generator (completes [2,3])
   - Leverage the full power of SPARK2014 to assess both generated code and runtime

   [3] was limited to some functional blocks. This contribution covers the whole application

[1] O. Gilles and J. Hugues, “Expressing and enforcing user-defined constraints of AADL models,” in *Proceedings of the 5th UML& AADL Workshop*, University of Oxford, UK, 2010, pp. 337–342.
[2] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, “From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite,” *ACM TECS*, vol. 7, no. 4, pp. 1–25, Jul. 2008.
[3] J. Hugues and C. Garion, “Leveraging Ada 2012 and SPARK 2014 for assessing generated code from AADL models,” in *HILT 2014*, Portland, US, 2014, pp. 39–45.

AADL: Modeling in the small and the large

AADL modeling components: precise execution semantics

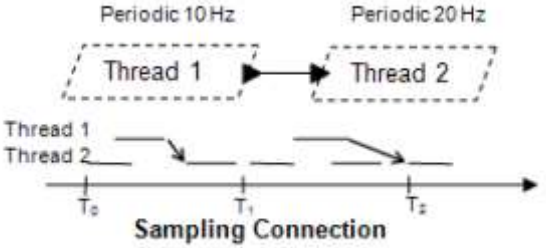
- Software: **thread (group), process, data, subprogram**
- Hardware: **processor, memory, bus, device, virtual processor, virtual bus**
- Composite: **system, abstract**

Supports system concepts of continuous control & event response processing

- Data and event **flows**, call/return, shared access
- End-to-End flow specifications

Modeling of large-scale systems

- Component variants (**refine**) layered system modeling, **packages**, **abstract**, **prototype**, parameterized templates, **arrays** of components, **connection** patterns, etc.



(from AADLv2 s[illegible]dard

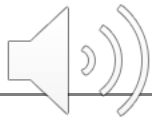
SAE International AADL Standard Suite (AS-5506 series)

Core AADL language standard [V1 2004, V2 2012, V2.2 2017]

- Focus on *embedded software system modeling, analysis, and generation*
- Strongly typed language with well-defined semantics for execution of threads, processes on partitions and processor, sampled/queued communication, modes, end to end flows
- Textual and graphical notation, XML/XMI interface to ease processing by 3rd party tool

Large set of analysis capabilities, but **disconnected**

- Integration with SysML, Simulink, SCADE
- Model checking:
- Timed/Stochastic/Colored Petri Nets
- Timed automata et al.: UPPAAL, TASM
- Scheduling: MAST, Cheddar, CARTS
- Performance evaluation: real-time and network calculu
- Fault analysis (FTA, FMEA): COMPASS, OSATE
- Simulation: ADeS, Marzhin
- Energy consumption of SoC: OpenPeople project
- Code generation: SystemC, C, Ada, RTSJ, Lustre
- WCET analysis: mapping to Bound-T



From AADL to Ravenscar
Architectural pattern enforcement

The Ravenscar pattern is well-known in the Ada Real-Time community

- Subset of Ada tasking for deterministic real-time systems
- Static task set, amenable to Worst-Case Response Time Analysis

Contribution #1: a set of constraints on an AADL model with **Resolute**

C1 Only one processor per node.

C2 The processor uses the POSIX 1003 scheduling policy, which is analogous to Ada FIFO per priority policy.

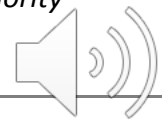
C3 All threads are matching the Ravenscar task model: all threads must be periodic or sporadic and define worst-case execution time (WCET), Priority, and Period.

[…]

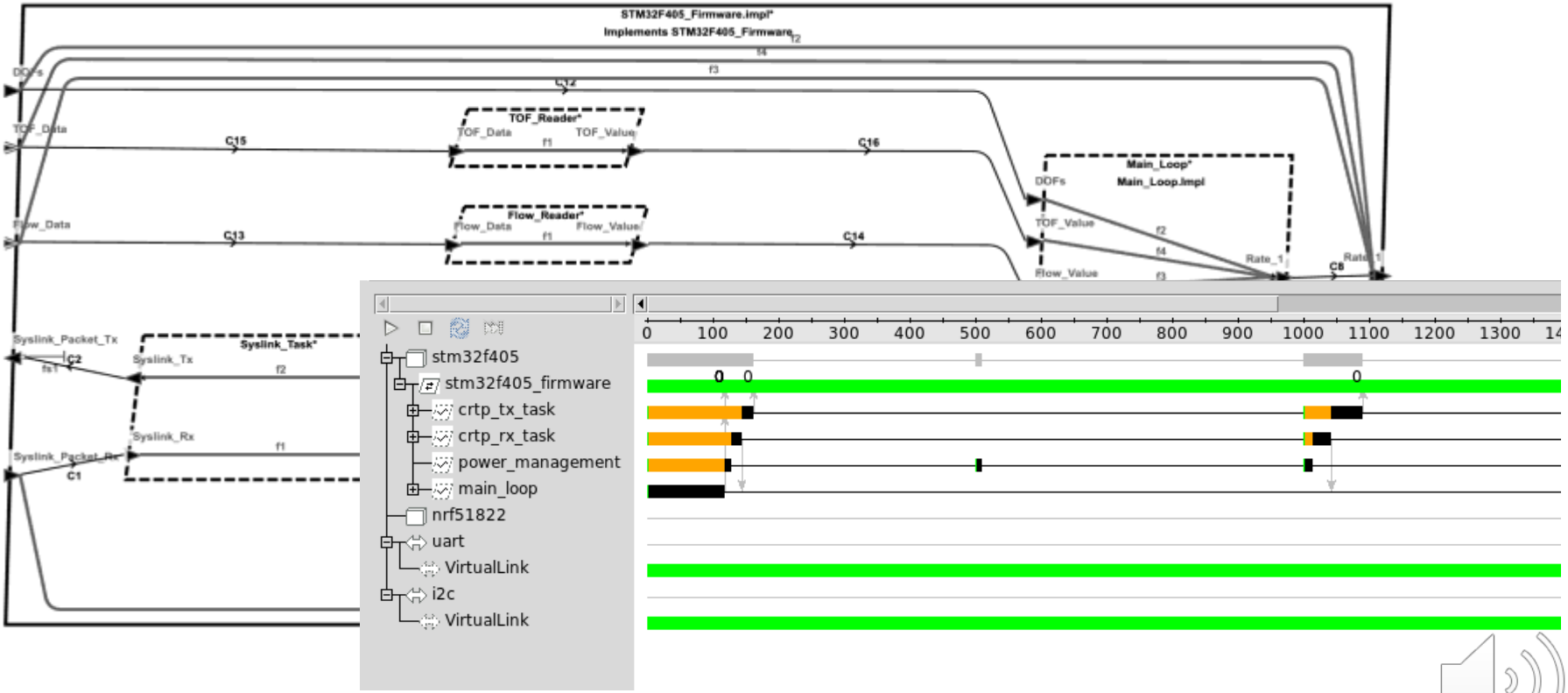
Extends [1] with accepted definition within the community, to support tool interoperability [4]

```
system ravenscar_sys
annex resolute {** prove ravenscar_rule_component(this)  **};
end ravenscar_sys;

is_Scheduling_Configured(c: component) <= -- C3
** "Thread " c " is correctly configured" **
has_property(c, Compute_Execution_Time) and -- Capacity
has_property(c, Period) and                 -- Period
has_property(c, Deadline) and               -- Deadline
has_property(c, Priority)                   -- Priority
```



Scheduling Analysis – AADLInspector / Cheddar



Code generation and middleware: Ocarina

Ocarina: AADL model “compiler”, FLOSS

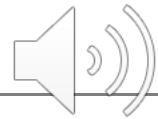
- Compiler architecture, AADL front-end, code generation back-ends

PolyORB-HI runtimes

- Ada High-Integrity profiles, with Ada native and bare board Ravenscar runtimes
- C POSIX or RTEMS, for RTOS & Embedded,
- Time and Space partitioning, e.g. ARINC653 C APEX, AIR, Xtratum

Generated code quality tested in various contexts and serve in the ESA TASTE

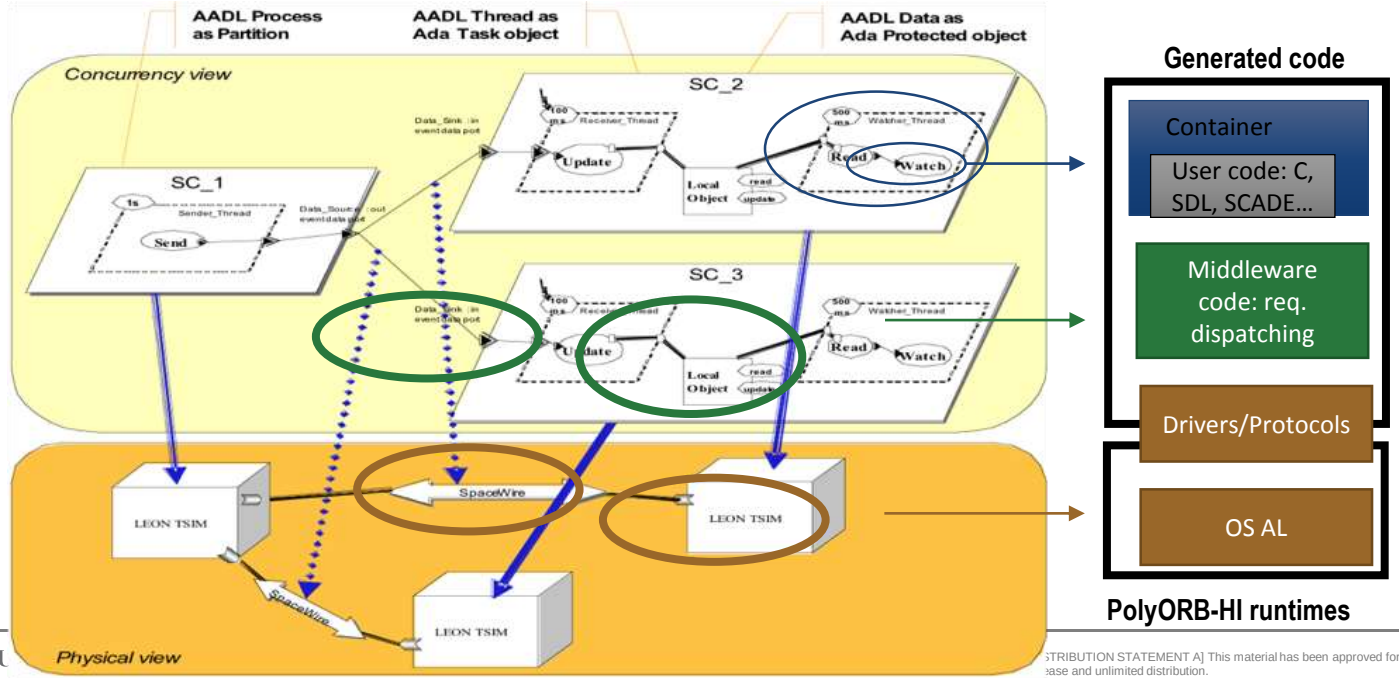
- WCET, quality, code coverage, etc.
- Designed to meet High-Integrity coding profiles
  - Ravenscar model of computations, static configuration of all elements (memory, buffers, tasks, drivers, etc.), no dynamicity



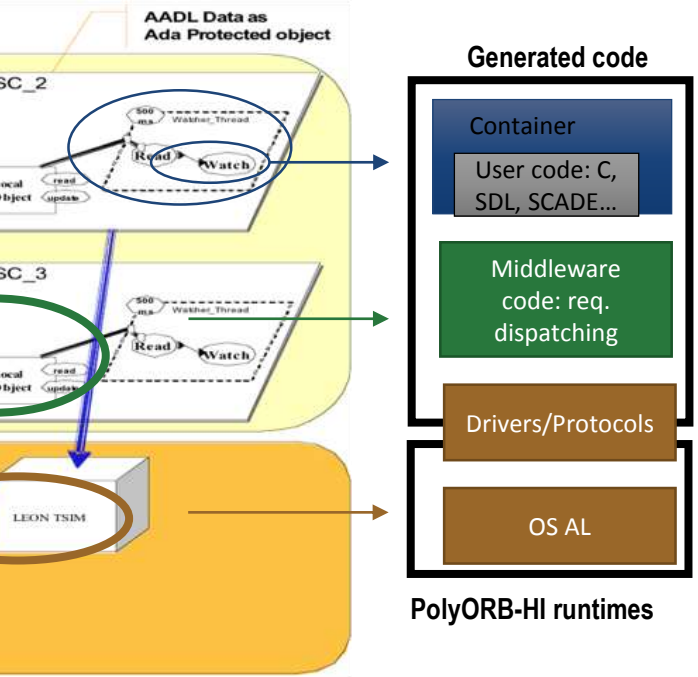
Code generation and middleware
Architecture-centric process

AADL captures tasks, queues, buffers, protocols

⇒ **Generate** middleware stack on top of minimal runtime/real-time OS



From model to generated code



User code ::= sequential code
Container ::= *instance of concurrency artefacts*, use middleware

Middleware ::= multiple static dispatch tables, marshallers, *instance of message queues*

Drivers/Protocols, OS Abstractions ::= implementation of Message queues, protocols, drivers as library elements

From generated code to SPARK2014 Gold Level

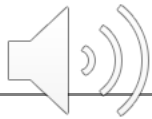
How to “trust” the code is correct?

- Code is compiled to be conformant to the Ravenscar profile, like its model
- Developing tests is challenging: no intermediate requirements exist

Contribution #2: use SPARK2014 to *prove* the code is free of runtime errors, and that its internals are formally specified and correctly used.

In [3], we tested GNATProve 2014, strong limitations that supported only sequential code.

But we need tasks, protected objects, volatile states (e.g. clocks), global state initialization, etc. All of this is now fully supported starting with GNATProve 2019



From PolyORB-HI/Ada to PolyORB-HI/SPARK

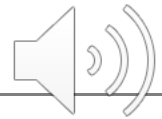
Original source code was in Ada 95, transitioning to Ada 2012/SPARK2014 is a huge gap

All the code (runtime and generated code) has been updated with annotations

- Pre/post conditions: assumptions on inputs, guarantees on outputs
- Type invariants: properties preserved by operations on a type, e.g. a buffer, a queue
- Global: state variables that are modified, e.g. global variables, file descriptors, etc.

```
procedure Put (Mode : in Verbosity := Normal; Text : in String)
with Global =>
(In_Out => (Epoch.Elaborated_Variables),
 --  Dependencies on the t0 value used to compute timestamp
 Input => (Elaborated_Variables, Ada.Real_Time.Clock_Time));
 --  Dependencies on the protected object state and clock
--  Display Text and a timestamp iff Mode is greater than
    Current_Mode.
```

Annotation process used pre-existing B. Zalila's thesis [5] for functional contracts

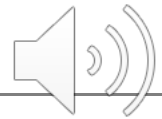


Verification results 1/2

- We defined minimal examples to exercise various interaction patterns:
  - Periodic/sporadic or periodic/periodic || event or data || local or distributed communication
  - 4 - 6.2 kSLOCs, proved using GNATProve 2020 CE, with caveats

SPARK Analysis results	Total	Flow	Provers	Justified	Unproved
Data Dependencies	35	32	.	3	.
Flow Dependencies	6	6	.	.	.
Initialization	58	55	.	3	.
Run-time Checks	281	.	281 (CVC4 93%, Trivial 7%)	.	.
Assertions	.	.	.	.	.
Functional Contracts	102	.	63 (CVC4 75%, Trivial 25%)	39	.
Concurrency	14	.	.	14	.
Total	496	93 (19%)	344 (69%)	59 (14%)	.

  - Number of VCs increase linearly with the number of threads *only* (1 skeleton + 1 queue)
  - Static typing and sizing allow GNAT front-ends to eliminate dead/inconsistent code



Verification results 2/2

Initialization and data dependencies: no issue, tools solve this statically

Functional contracts are typical software patterns: task skeletons, message queues

Concurrency: direct verifications of Ravenscar patterns, correctness of ICPP

SPARK Analysis results	Total	Flow	Provers	Justified	Unproved
Data Dependencies	35	32	.	3	.
Flow Dependencies	6	6	.	.	.
Initialization	58	55	.	3	.
Run-time Checks	281	.	281 (CVC4 93%, Trivial 7%)	.	.
Assertions	.	.	.	.	.
Functional Contracts	102	.	63 (CVC4 75%, Trivial 25%)	39	.
Concurrency	14	.	.	14	.
Total	496	93 (19%)	344 (69%)	59 (14%)	.

Justified elements

- Unmarshalling using Unchecked_Conversion
- Writing to data during thread initialization
- Potentially blocking operations for debug logs.

Conclusion

Main contribution:

A MBSE workflow from AADL to Ravenscar-compliant SPARK source code

Two facets

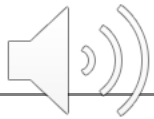
#1: Ravenscar-compatible subset of AADL, interoperability with other tools

#2: code generation from AADL to SPARK/Ada, with proof of correctness

Integrated in OSATE, Ocarina code generator and PolyORB-HI/Ada runtime

Lessons learnt: SPARK can scale to infrastructure code, note just algorithmic.

Obvious justification of some VCs required



References

[1] O. Gilles and J. Hugues, “Expressing and enforcing user-defined constraints of AADL models,” in Proceedings of the 5th UML& AADL Workshop (UML&AADL 2010), University of Oxford, UK, 2010, pp. 337–342.

[2] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, “From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite,” ACM TECS, vol. 7, no. 4, pp. 1–25, Jul. 2008.

[3] J. Hugues and C. Garion, “Leveraging Ada 2012 and SPARK 2014 for assessing generated code from AADL models,” in High Integrity Language Technology, HILT 2014, Portland, US, 2014, pp. 39–45.

[4] V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, and J. Legrand, “Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets,” in *IEEE International Symposium on Rapid System Prototyping*, Montreal, Canada, 2013.

[5] B. Zalila*. Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture*. PhD thesis, ENST, nov 2008.