Introduction of model checking facilities in TASTE
ESA Final Days

Jérôme Hugues, ISAE/DISC

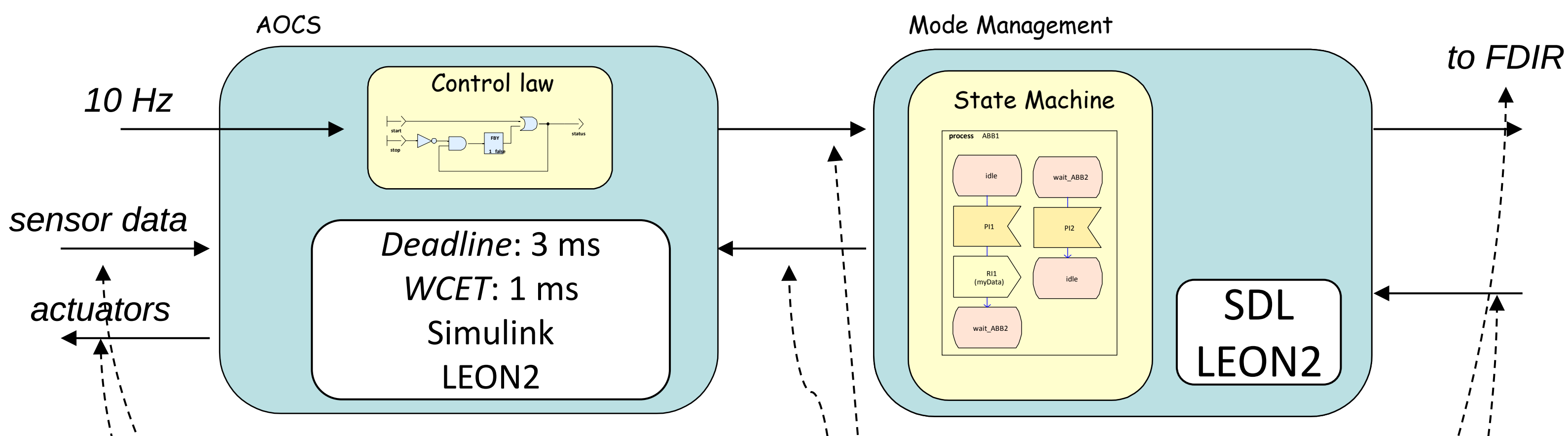
Agenda

1. TASTE process, code generation perspective
2. Introducing model checking @ runtime
3. Conclusion

TASTE COO3 objectives

- Goal: build state space of a TASTE-CV model (AADL) to support simulation and model checking (MC) objectives
- Rely on Ravenscar Computational Model + AADL semantics for port communication
  - Ravenscar = static set of tasks, ports, deterministic scheduling with worst case scenario
  - AADL semantics = precise timing for communication instants, and associated thread dispatch
- Combine these two information to build component state, and then system’s history from a set of external inputs

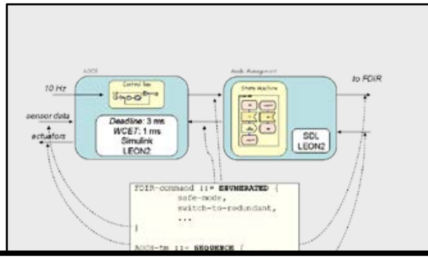
TASTE process in a nutshell



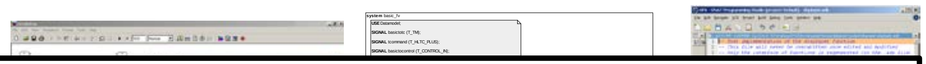

```
FDIR-command ::= ENUMERATED {
     safe-mode,
     switch-to-redundant,
     ...
}

AOCS-tm ::= SEQUENCE {
     attitude Attitude-ty,
     orbit Orbit-ty,
     ...
}
```

AADL and ASN.1 are combined to provide a formal, precise, and complete description of the system architecture and data.

TASTE process in a nutshell



① Generate "application skeletons" in Simulink, SDL, C, and Ada

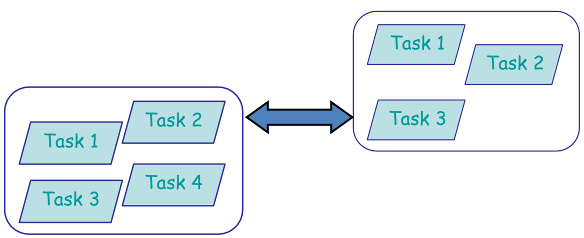


All these steps are **automated**, thanks

- Languages with good power of expression
  - AADL for architecture, ASN.1 for data typing,
  - SDL, Simulink, SCADE, C, Ada, etc. for behavior
- Tool to support this approach
  - TASTE toolchain (editors, code generators, orchestrator)

In the following, we focus in the Concurrency view level, leveraging AADL

real-time architecture

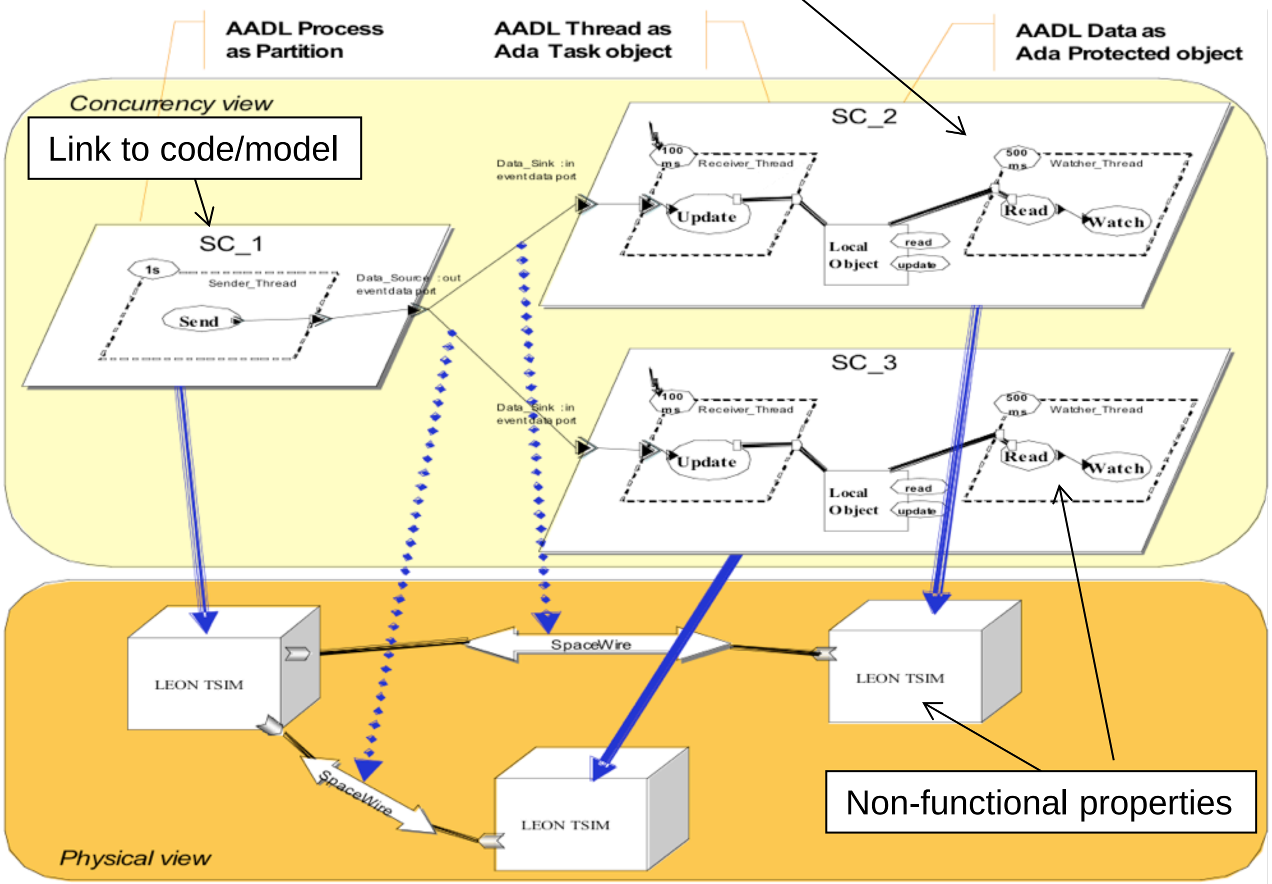


to put everything together on a real-time operating system

Research on AADL @ ISAE

Architecture helps you focusing on the actual system

Architectural patterns



AADL covers many parts of the V cycle: model checking, scheduling, safety and reliability and code generation

Lead on the Ocarina toolset

Development of AADL:

4 books, tutorials, 30+ papers

Code generation :

Ada, C (POSIX, ARINC653, RTEMS)

TRL 6-7 with ESA (ECSS E-40)

SPARK, ACSL **TRL 2-3**

Scheduling: Cheddar, MAST

External metrics: stack usage (gnatstack), WCET (Bound-T)

TRL 4-5 with ESA

Architectural Constraints/Requirements checks

TRL 6, being standardized

Model checking: Petri Nets, LNT

TRL 2 (PhD contributions)

System engineering: SysML, Capella **TRL 2-3 (with IRT-SE)**

Ocarina: an AADL code generator
http://www.openaadl.org

- Ocarina is a stand-alone tool for processing AADL models
  - Free Open Source Software (as in *Free* speech and *Free* beer)
  - Command-line, or integrated third-party tools
    - OSATE (CMU/SEI), TASTE (ESA), AADL Inspector (Ellidiss)
- Code generation facilities target PolyORB-HI runtimes
  - Ada HI integrity profiles, with Ada native and bare board runtimes
  - C POSIX or RTEMS, for RTOS & Embedded
  - C ARINC653 for avionics systems
- Generated code quality tested in various contexts
  - For WCET exploration, support for device drivers, …
- Written to meet most High-Integrity requirements
  - Follow Ravenscar model of computations, static configuration of all elements (memory, buffers, tasks, drivers, etc.)
- Contributions from PhD students, partners (SEI, ESA)

Ocarina runtimes: PolyORB-HI/Ada

> Target Ada Ravenscar and High-Integrity runtimes

> Based on the Ravenscar & HI Ada profiles

- » Meets stringent requirements for High-Integrity systems
- » Checked at compile-time by Ada compiler, GNAT
- » On-going work to support SPARK/Ada
  - Proof of absence of Run-Time Errors, contract-based programming

> Supports native, RTEMS, MaRTE OS, Ada bare-board

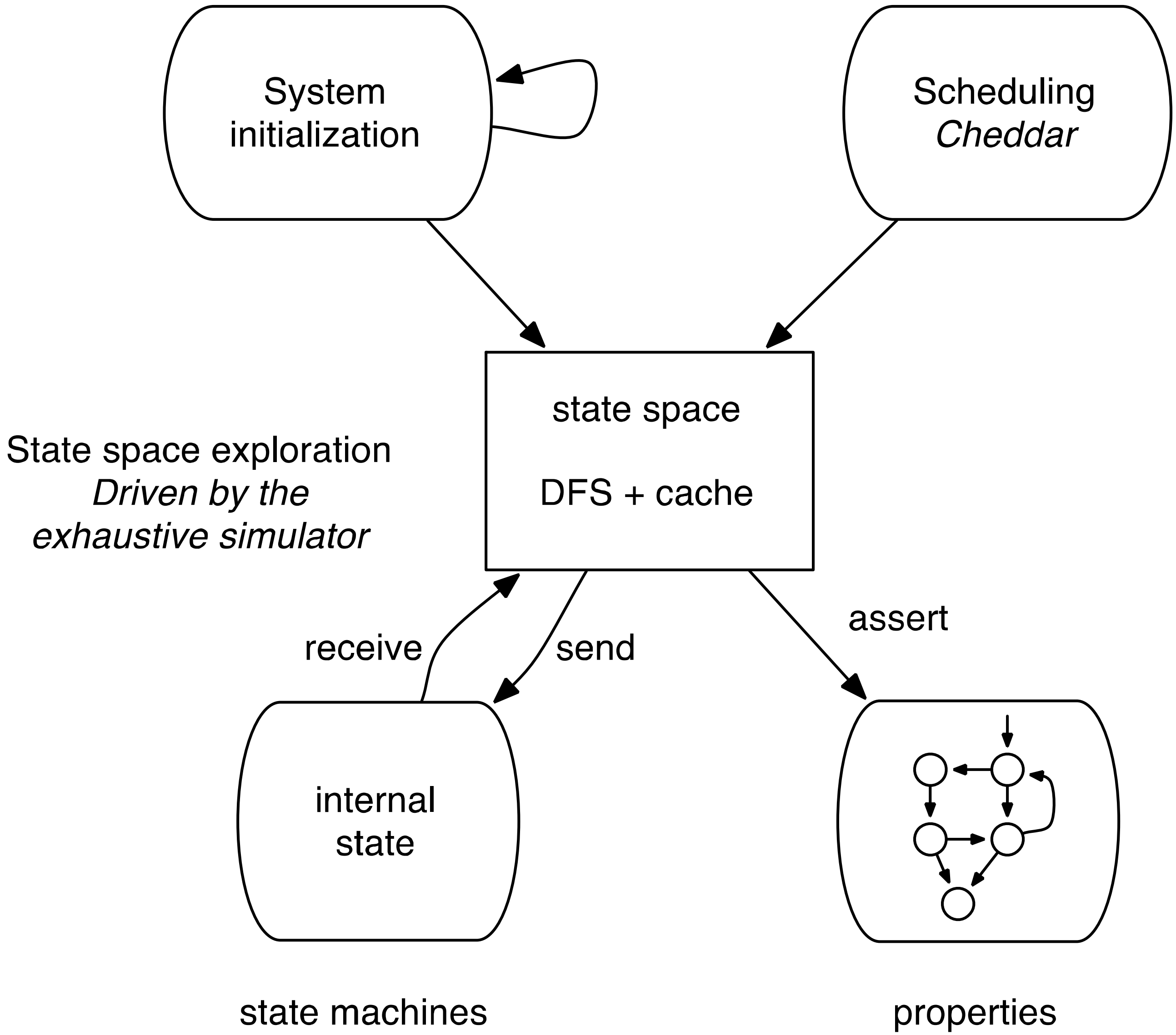
> Easy to retarget thanks to Ada portability

- » Reduced to configuration of the compilation chain
- » Any Ravenscar-capable runtime should work out-of-the box
- » GNAT support allows integration of 3rd-party API, e.g. ARINC653

Ocarina runtimes: PolyORB-HI/C

- Follow the same design principles from the Ada runtime
  - No memory allocation: static resources, threads, etc.
- Set of primitives to build all AADL entities (threads, ports)
- Set of macros to adapt runtime to target-dependent APIs
  - Supported: RT-POSIX, C/RTEMS, VxWorks classic API, Xenomai, Windows, FreeRTOS, ..
- Tested on different configurations:
  - Restricted libc: GNU/Linux on Nintendo DS and Nokia 770
  - POSIX RTOS: Linux, RTEMS, eLinOS (Linux)
  - RTEMS, VxWorks 6.2
- One mode to target directly ARINC653 APEX
  - Tested with DDC-I DeOS and WRS VxWorks 653

Generic approach for model checking

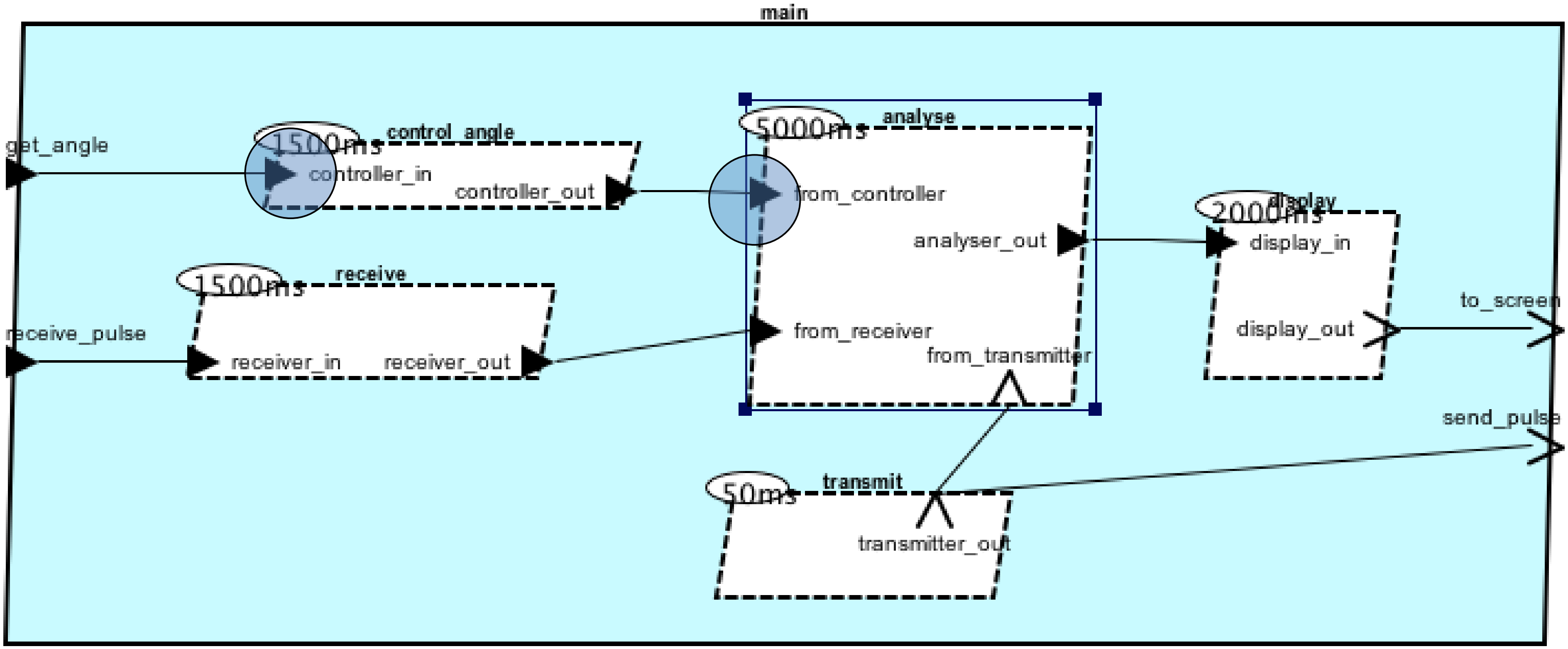


Point of interest for MC

- A TASTE CV model is made of
  - Interconnected components: interfaces, links, bindings to hardware platforms (buses, processors)
  - Implementation of components points either to
    - Other subcomponents (hierarchical model)
    - Leaf model (SDL, SCADE, etc.)
- Relevant properties
  - Observable set of states:
    - Monitored state variables of a component, from its interface
    - Content of messages exchanged
- Ravenscar MoC defines rule to update observable state
  - dispatch triggers, communication instant, computation states, …

Example

- Attach interceptors on ports
  - » State = request + meta data for building full state space



Formal definition of a state

> A event is a "step" in the execution of the model
> A state of a component is defined by

- » σ is the step of the event consumption at which the state is created. The event can either be the dispatch of a periodic thread or the consumption of a event in an event or an event data port;
- » ω is the occurrence of the hyperperiod;
- » τ is the identifier of the dispatched task or event consuming task;
- » ε is the port identifier of the consumed event (empty if it is a periodic dispatch);
- » $\upsilon_0 ... \upsilon_n$ is a tuple of values contained by the entry ports of the system, where *n* the global number of entry ports of the system.

> Parameters **ω**, **τ** will be used to rebuild the full history from a set of traces of the system

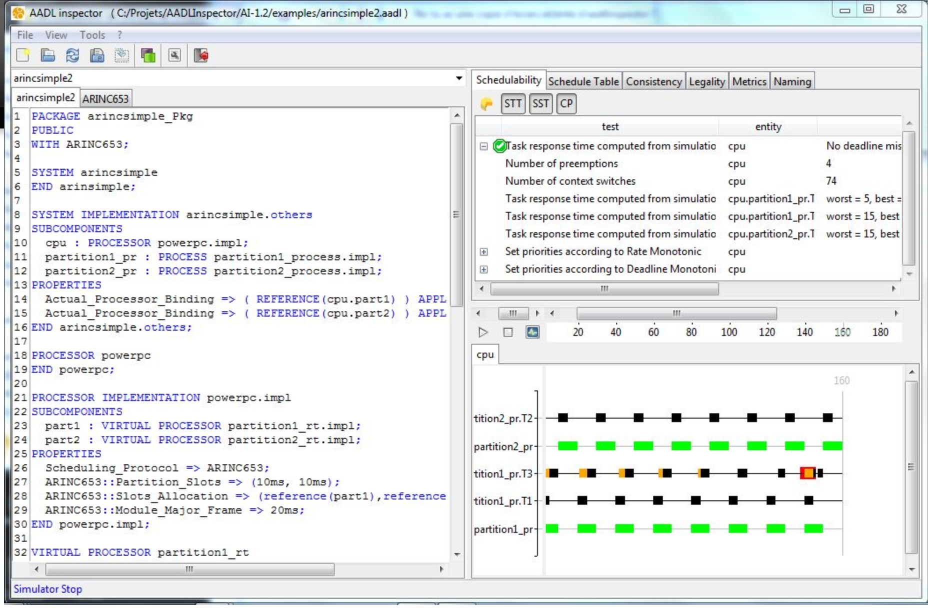
Implementation path

- Use a hybrid approach, combining MC and code generation
  - MC: interceptor on functional block viewed as black box
    - Only capture inputs/outputs/internal state + meta-data (timestamp, id)
    - Build a state space using an optimized hash function (model specific)
  - Code generation used to
    - Tune the hash function, build atomic state
    - Place interceptors on all or selected components
    - PolyORB-HI/C runtime (simulation + trace) or MC kernel (exhaustive)
  - Need halting condition
    - Driven by users (e.g. as part of observers, scenarios)
    - Or derived from scheduling (e.g. stop after one hyper period)
- Controlled by Python API, for future integration with TASTE ecosystem: TASTE TM/TC tool, automated testing, etc.

Implementation of the MC engine

- Main goal: reduce overhead on the generated code both in time and memory dimensions
- TASTE toolchain has detailed information to allocate all resources (buffers, marshallers, tasks, etc.)
  - Need to fine tune generation of state space, combination of data types + graph to store history of executions
- Solution: exploit meta-programming from C++ to instantiate at compile-time all required resources for monitoring
  - Allow for a clean separation between the monitoring engine and the existing run-time and code generation
  - Rely on Boost and C++11 meta-programming (introspection) facilities to allocated statically all types
    - no memory allocation required, can be embedded for logging

Managing time



- > Generic timed MC

  does not scale, need abstractions
- > Computing number of task dispatch per hyper-period
  - » Solution: use outputs from cheddarkernel (TASTE VM) + AADL to build dependency graph, triggering instants and worst case scenario for number of states per hyperperiod
- > Bounded by the number of worst-case number of context switches on a hyper-periodic of a system

- > Time abstracted thanks to Ravenscar MoC
- > Built from chain of events:
  - » E.g. T2 dispatched because of events in one of its predecessor
  - » No need to manage time explicitly

Benchmarks

- > Rely on efficient hashing to store states
  - » Decouple graphs connecting states (history) from repository of states (actual values)
- > Benchmarks from Ocarina tests

Example	Nb Tasks	Nb Ports	State size (bytes)	Hyperperiod size (steps)	Trace size (bytes)
ping	2	2	32	753	24192
producer-consumer	4	4	48	6	288
flight-mgmt	5	16	144	1000	144000
sunseeker	2	4	48	2	96
file-store	3	2	32	2	64
packet-store	3	2	32	2	64

- > Thanks to hashing, number of states reduced to true difference in values in ports, no impact on timing
  - » Graphs is generated once from worst-case scenario on hyperperiod, number of states depend on monitored data

Integration to Ocarina, take 1

- Monitoring is transparent to user
  - One additional configuration to Ocarina code generation to
    - Activate logging interceptors in communication API
    - Generate type for state from model elements
    - Evaluate number of state and allocate memory for storing the graph associated to the worst-case scenario
- Could be embedded in running application
  - Model checking is reduced to advanced non intrusive logging
  - Reduced penalty at runtime: storing events done as part of communication API, only read/write to hash tables

Interaction with user – step 1

- Default mode of operation is to use OS primitives for tasking
  - Running the system in operational scenario, for functional testing
  - Not adequate for model checking
- Need to give control to user to model-level debugger
  - Start/stop/step in model elements: tasks queues
  - Inject events, remove events, e.g. fault injection, introspection
  - Control of the clock to “pause” the model

- Introducing “user-mode” OS-like primitives
  - AADL runtime uses regular OS system calls
  - Emulate tasking and time management

About user-mode OS primitives

- Leveraging Linux ucontext.h API
  - Definition of "context of execution", aka thread control block
  - Used to emulate context switching, and scheduling policies
  - Time managed either using host clock, or emulated using "ticks"

- Defined a new UMthreads target configuration in runtime
  - Replace all calls to RTOS to user-mode OS
  - Emulate Ravenscar MoC: FIFO_Within_Priorities scheme, iCPP and absolute delay
  - Available as a regular target by user when building its concurrency view, yet restricted to Linux host

Interaction with user – step 2

> Need a way to interact with simulated models
  - Represented as an instrumented binary application

> Defined a Python API to interact with model@runtime
  - Uses SWIG to generate set/get methods to interact with models
    - Inject events, monitor queues, advance time, etc.
  - A few helper functions to start/stop model, configure logging, etc.

> Provide direct access to internals using the same API
  - Thin layer from SWIG, reduce uncertainty: you interact with the real code, not a simulator using a different code base

Example: a script to test the model

```
class TASTEModel(object):  # Handler to TASTE model
    def __init__(self):
        # Configuration (not shown)

    def run(self):
        taste_model.init ()

# Creating and starting thread running example
My_model = TASTEModel()

# Instanciating request factory
reqfac = RequestFactory()

# Calling a po_hi_gqueue function to set an in port value
po_hi_gqueue.__po_hi_gqueue_store_in
      (po_hi_gqueue.pc_consumer_k,                        # id of task
       po_hi_gqueue.consumer_local_data_sink,             # port
       reqfac.consumer_global_data_sink_request_init(40)) # value to be sent
```

Summary

- Integration of model checking facilities to TASTE in progress
  - Beyond regular model checking using formal methods
- Allow for model-checking@runtime
  1. Fine-tuned monitoring facilities
     - At runtime for assertion checking
     - Or used for model checking on specific scenarios or full exploration
     - Log could be dumped to user for off-line processing
  2. User-mode tasking API introduced
     - Use RTOS for running time-based scnearios
     - User-mode tasking for exploration, with time acceleration (no delay)
  3. Python API to control model execution
     - Inject events, monitor queues, etc.

Future work

- First step focused on enhancing infrastructure for supporting model-checking in multiple dimensions
  - » Logging, in-depth testing and model-checking (state exploration)
- Future directions include

1. Specification of properties and observers
   - Follow TASTE approach: property is a functional block (e.g. SDL) weaved to regular model through inspection point (observer)
2. Integration with testing GUI (TM/TC processing) to provide a uniform access to model internals at runtimerious dimensions
3. Scenario for testing, inline with project requirements
   - Observer for wanted/unwanted situation
   - Indication of relevant features to monitor (internal state, ports) to reduced memory overhead

That’s all folks