The SMART Project: Multi-Agent Scheduling Simulation of Real-time Architectures

P. Dissaux[1], O. Marc[2], S. Rubini[3] , C. Fotsing[3], V. Gaudel[3], F. Singhoff[3], A. Plantec[3],
Vương Nguyễn-Hồng[3], Hải Nam Trần[3]

1: Ellidiss Technologies, 24 quai de la douane, 29200 Brest, France
2: Virtualys, 41 rue Yves Collet, 29200 Brest, France
3: Lab-STICC, CNRS UMR 6285, Université de Bretagne Occidentale, 20, av Le Gorgeu, 29200 Brest, France

Abstract

The ongoing SMART collaborative project addresses modeling and analysis techniques for software intensive real-time systems. The AADL modeling language has been selected to describe multi-thread, multi-partition, multi-processor and multi-core architectures.

This paper focuses on the use of the Marzhin simulator that is based on a Multi-Agent technology for providing scheduling analysis results of real-time systems. This simulator is integrated in the AADL Inspector product and can also be used to animate realistic 3D animations.

Introduction

The SMART project (*Simulation Multi-Agent d'ARchitectures Temps-réel)* is a collaborative project focused on real-time analysis and simulation of real-time hardware and software architectures. It aims at emulating the behavior of the control-command logic of a system and animating a 3D graphical representation of this system.

The first section of this article briefly summarizes the capabilities of the AADL language for specifying real-time architectures. The second section introduces Marzhin, an original approach for performing real-time simulations at a model level. The third section explains how the Cheddar scheduling analysis tool has been used to verify the results generated by Marzhin. Finally, the fourth section shows one of the main benefits of the approach which consists in using the real-time simulation output to animate a 3D graphical representation of the system.

1. Modeling Real-Time Architectures

The AADL Language (Architecture Analysis and Design Language) is an international standard of the SAE (AS-5506B) [1].

The standard defines a default execution model that specifies the way the various components interact at run-time. This enables precise timing analysis and simulation of AADL models.

A typical AADL model is composed of one or several execution units (*Processor*) that can communicate via *Buses*. The software application is composed of one or several memory address spaces (*Process*) that contain concurrent *Threads* and shared *Data*. Various inter-threads communication paradigms are supported by the standard.

1.1. Processors

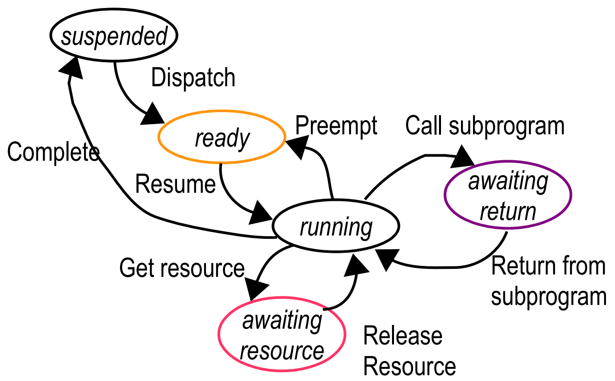
In AADL, the *Processor* represents the hardware computation unit associated with a scheduler. It must declare a *Scheduling_Protocol* property whose value corresponds to one of those that are actually supported by the analysis, simulation or code generator.

Typically supported *Scheduling_Protocols* are:

- Rate Monotonic Protocol, based on the period of the Threads.
- Deadline Monotonic Protocol, based on the deadline of the Threads.
- POSIX 1003, based on the priority of the threads.
- ...

1.2. Threads

The default behavior of AADL *Threads* is specified in the standard by a state-transition automaton.



A *Thread* must also have a *Dispatch_Protocol* property that defines when it is ready to execute. Supported protocols are:

- Periodic
- Aperiodic
- Sporadic
- Timed
- Hybrid
- Background

Thread interfaces contain *Features* that are used to implement communication channels. They can be:

- *Data Ports*
- *Event Ports* or *Event Data Ports*
- *Access* to shared *Data*
- *Access* to remote *Subprograms*

1.3. Shared Data

One particular way to exchange information between *Threads* is to let them have access to the same shared data. Shared data are represented in AADL by *Data* subcomponents to which *Threads* can have access through *Data Access Connections*.

It is possible to specify critical sections thanks to the AADL *Behavior Annex*. In order to ensure mutual exclusion of all the threads accessing a given shared data component, a *Concurrency_Control_Protocol* property can be set.

1.4. AADL Behavior Annex

The core definition of AADL restricts its scope to the architectural description of the system. It specifies which components are instantiated and how they are connected together. The functional activity of threads is summarized by a *Compute_Execution_Time* property that must be given with its Min and Max values. The Max value of this property thus corresponds to the usual WCET (Worst Case Execution Time).

However, in order to perform precise timing analysis or simulation, it is necessary to provide a more detailed description of the functional behavior of a *Thread*. The AADL *Behavior Annex* is an action language that can be used to provide a simplified representation of the code logic (pseudo-code).

Examples of actions that can be described with the AADL Behavior *Annex* are:

- `p!` : sending an event on port p (*Put_Value* and *Send_Output*)
- `d!<` : entering a critical section on shared data access d (*Get_Resource*)
- `d!>` : leaving a critical section on shared data access d (*Release_Resource*)
- `computation(a..b)` : use of the processor for a duration between the minimum duration a and the maximum duration b.

2. The Marzhin Multi-Agent Simulator

Usually, the implementation of real-time simulators is based on an emulation of an actual Real-Time Operating System (RTOS). These emulators must thus provide the same Application Programming Interface (API) and have the same dynamic behavior as the real software. Due to these strong requirements, the development and validation of such simulators is a complex and costly task.

An alternate and original approach has been followed to build the Marzhin real-time simulator. After presenting the general notions of Multi-Agent systems, we describe how Marzhin has been implemented.

2.1 Notions of Multi-Agent Systems

According to [8], an agent is a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives. This definition means that an agent must be able to react to events in its environment, to be proactive, social, in order to maintain its goal.

When multiple agents work within the same system, we call such a system a Multi-Agent system. In this case, they need a framework to communicate, coordinate and negotiate in order to meet their goals.

Several models for agent communication have been developed [9]: agent to agent, where each agent knows the name of any other agents with which it might need to communicate, agent broker and agent matchmaker, where globally, a special agent is tasked with finding agents to fulfill services required by requesting agents. And, in mostly case, the communication among agents is facilitated by an agent communication language [10]: The Knowledge Query Manipulation Language (KQML).

Finally, many agent architectures have been developed to support Multi-Agent systems. We can distinguish: DECAF [11] (Distributed, Environment - Centered Agent Framework) Agent Framework, which is a Java-based Multi-Agent system, and provides a matchmaker agent that accepts KQML “performatives” to allow for agent communication; FIPA [12] (Foundation for Intelligent Physical Agents), which have developed a standard for Multi-Agent systems, that include specification for agent architecture, agent management, agent communication and interoperability; and COBALT [13] which is an agent framework based on KQML and CORBA, and provides a complete communication layer that can support cooperation in Multi-Agent systems.

Marzhin is based on the reuse of an existing Multi-Agents simulation kernel called VAgent. As opposed to traditional simulators that implement deterministic scheduling algorithms, this kernel randomly stimulates a set of autonomous interconnected elementary entities (the Agents) in order to exhibit a resulting macroscopic behavior.

However, VAgent only provide generic Agents. The precise behavior of each category of Agent must be refined in order to implement a simulator. In our case, this behavior is specified by the semantics of the simulated AADL component categories.

2.2 Marzhin Implementation Details

The realization of such a simulator thus consists in specifying and implementing the awaited behavior for each individual Agent, which is a much easier task than doing so globally for a complete RTOS emulator.

Practically, the realization of Marzhin mostly consists in implementing the AADL real-time specifications for the *Processors*, *Threads* and shared *Data* components as they are described individually in the previous section in order to enrich the basic behavior of the VAgent kernel.

The design of the simulation kernel is thus very modular and the implementation of each specialized Agent can be performed by a small piece of source code that can be easily traced against its requirements.

However, due to the intrinsic low determinism of the Multi-Agent simulation approach, there is a need to provide some kind of evidence that the resulting global behavior of Marzhin is realistic and can be used with a certain level of confidence for critical real-time system development. A specific verification task is thus mandatory.

3. Verifying Simulation Results with Cheddar

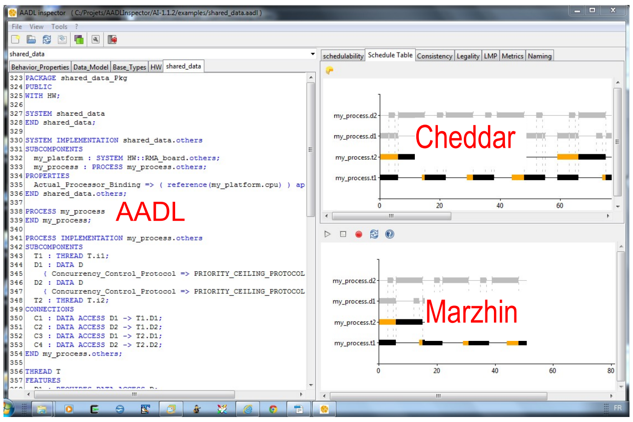
Cheddar [3] is a scheduling analysis tool that performs real-time analysis complying with the “scheduling theory” [2]. It is mostly composed of a set of analytical feasibility tests providing mathematically proven results on analyzed models.

Unfortunately, it is not possible to analyze any kind of systems by this mean, and some theoretical results are often known as being too pessimistic. That's why additional techniques such as dynamic simulation can help to increase the “confidence” the designer has on his design.

However, it is quite difficult to demonstrate that non exhaustive and non deterministic simulation results are realistic enough to be used in a real-time analysis process.

The approach that has been followed in the SMART project is to identify a set of “validation points” each of them being described by an AADL architectural model. The goal is then to compare the results given by Cheddar and Marzhin for these well identified models.

This comparison task is facilitated by the fact that Cheddar and Marzhin are embedded in the AADL Inspector product [17]. It is thus easy to run both verification tools on the same AADL models.


AADL inspector (C:/Projets/AADLInspector/AI-1.1.2/examples/shared_data.aadl)
File View Tools ?
shared_data
Behavior_Properties Data_Model Base_Types HW shared_data
schedulability Schedule Table Consistency Legality LMP Metrics Naming
AADL
Cheddar
Marzhin
my_process.d2
my_process.d1
my_process.t2
my_process.t1
0
20
40
60
80

```
PACKAGE shared_data_Pkg
PUBLIC
WITH HW;

SYSTEM shared_data
END shared_data;

SYSTEM IMPLEMENTATION shared_data.others
SUBCOMPONENTS
  my_platform : SYSTEM HW::RMA_board.others;
  my_process : PROCESS my_process.others;
PROPERTIES
  Actual_Processor_Binding => ( reference(my_platform.cpu) ) ap
END shared_data.others;

PROCESS my_process
END my_process;

PROCESS IMPLEMENTATION my_process.others
SUBCOMPONENTS
  T1 : THREAD T.i1;
  D1 : DATA D
    { Concurrency_Control_Protocol => PRIORITY_CEILING_PROTOCOL
  D2 : DATA D
    { Concurrency_Control_Protocol => PRIORITY_CEILING_PROTOCOL
  T2 : THREAD T.i2;
CONNECTIONS
  C1 : DATA ACCESS D1 -> T1.D1;
  C2 : DATA ACCESS D2 -> T1.D2;
  C3 : DATA ACCESS D1 -> T2.D1;
  C4 : DATA ACCESS D2 -> T2.D2;
END my_process.others;

THREAD T
FEATURES
```

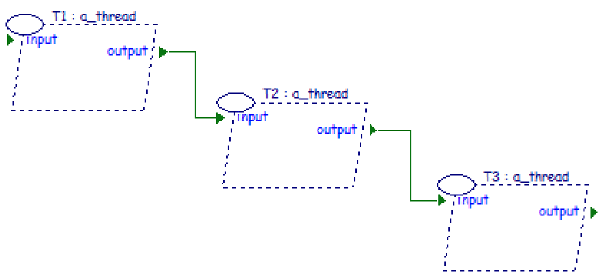
This verification approach is of course limited by the fact that Cheddar cannot provide a significant result for any kind of AADL architecture. However, it must be recalled that the role of Marzhin is specifically to complement Cheddar with cases that cannot be processed in a deterministic manner.

3.1. AADL Design Patterns

A first set of validation points have been defined on the basis of predefined AADL Design Patterns that highlight various inter-thread communication paradigms [4]. The advantage of reusing some of these patterns is that they have already been analyzed in details with Cheddar. The simulation results are thus well known and the comparison with the outputs of Marzhin consists in checking that the time lines for each thread are similar during a bounded period of time.

3.1.1. Data flow communication

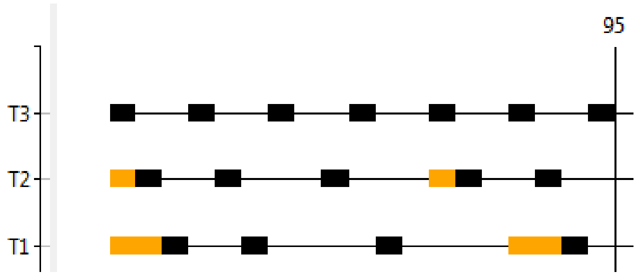
The simplest communication pattern consists in a set of periodic *Threads* communicating by data ports connected in a sampling mode. This implies that communication does not interfere with the thread dispatch policy.



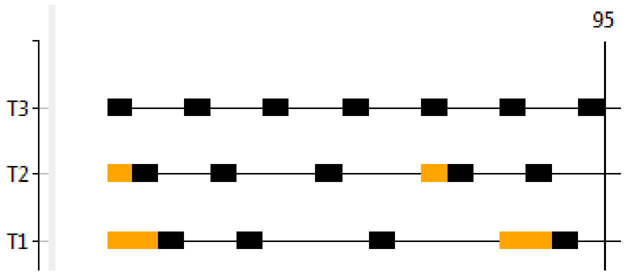
The real-time characteristics of each thread instance is given by the appropriate AADL properties, as shown by the following AADL source code fragment:

```
PROCESS IMPLEMENTATION my_process.others
SUBCOMPONENTS
  T1  : THREAD a_thread
    { Dispatch_Protocol => Periodic;
      Compute_Execution_Time => 5ms..5ms;
      Period => 25ms;
      Deadline => 25ms; };
  T2  : THREAD a_thread
    { Dispatch_Protocol => Periodic;
      Compute_Execution_Time => 5ms..5ms;
      Period => 20ms;
      Deadline => 20ms; };
  T3  : THREAD a_thread
    { Dispatch_Protocol => Periodic;
      Compute_Execution_Time => 5ms..5ms;
      Period => 15 ms;
      Deadline => 15 ms; };
CONNECTIONS
  C1  : PORT T1.output -> T2.input;
  C2  : PORT T2.output -> T3.input;
END my_process.others;
```

If such a real-time architecture is bound to a processor and scheduled for instance with a Rate Monotonic protocol, it will be possible to use Cheddar to apply feasibility tests and run the static simulation. The first part of this simulation trace is given below (the black rectangles represent the *Threads* in a running state, and the orange ones the *Threads* in a ready state, awaiting for the *Processor* resource):

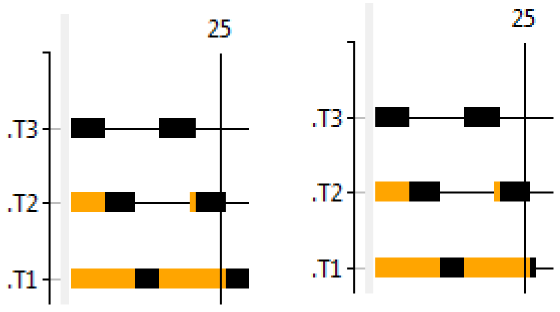


The same AADL specification can also feed the Marzhin dynamic simulator, in order to get another simulation trace, as shown below:



Simple direct visual comparison of the two simulation traces during the complete simulation period (the time required for the simulation trace to repeat itself) shows that the two simulators give a similar result in that case. Another analysis method consists in automatically perform the comparison of the output data that are produced by each tool.

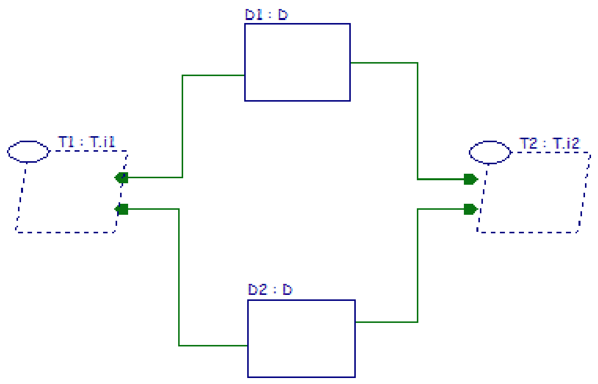
A light change in the AADL specification can show a difference in the way the two simulators behave in case the system becomes non schedulable. If the capacity of *Thread* T3 is increased by 1ms, the simulation traces of Cheddar and Marzhin become as shown below:



At tick 25 indeed, *Thread* T1 should start a new period, but it cannot happen as it is still preempted by T2 at that time and has not completed its previous job. With the current implementation of the two tools, Cheddar (on the left) starts the new job of T2 even if late, whereas Marzhin (on the right) skips the missed job. Further investigation would be required to determine which of these two behaviors is the more realistic according to actual RTOS implementations.

3.1.2. Shared data communication

The second communication pattern that has been selected for our study makes use of *Access Connections* to shared *Data* component instead of direct point-to-point communication between the *Threads*.



A fragment of the corresponding AADL specification is as follows. Note that we initially applied a *Priority Ceiling Protocol* (PCP) to access the shared *Data* components.

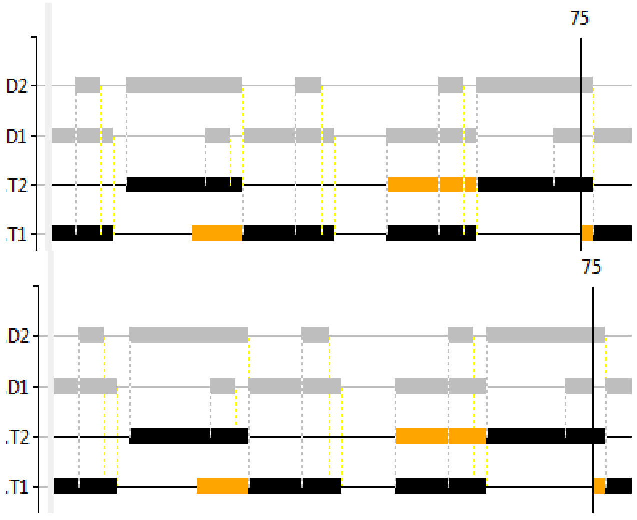
```
PROCESS IMPLEMENTATION my_process.others
SUBCOMPONENTS
  T1 : THREAD T.i1;
  T2 : THREAD T.i2;
  D1 : DATA D.others
    { Concurrency_Control_Protocol =>
      PRIORITY_CEILING_PROTOCOL; };
  D2 : DATA D.others
    { Concurrency_Control_Protocol =>
      PRIORITY_CEILING_PROTOCOL; };
CONNECTIONS
  cnx_0 : DATA ACCESS D1 -> T1.D1;
  cnx_1 : DATA ACCESS D2 -> T1.D2;
  cnx_2 : DATA ACCESS D1 -> T2.D1;
  cnx_3 : DATA ACCESS D2 -> T2.D2;
END my_process.others;
```

The precise timing of the interaction between each *Thread* and shared *Data* component has now a direct impact on the global scheduling. The critical sections can be expressed thanks to the AADL *Behavior Annex* subclause that is attached to each *Thread*.

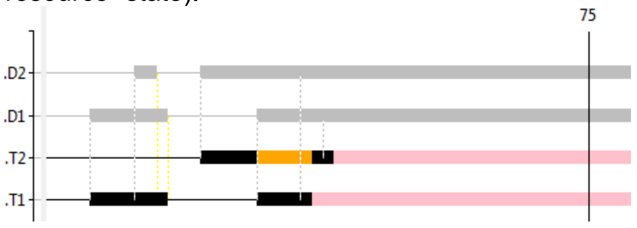
```
THREAD IMPLEMENTATION T.i1
PROPERTIES
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 5ms..5ms;
  Period => 15 ms;
ANNEX Behavior_Specification {**
  states
    s : initial complete final state;
  transitions
    t : s -[on dispatch]-> s {
      D1 !<;
      computation(3 ms);
      D2 !<;
      D2 !>;
      D1 !>
    };
**};
END T.i1;
```

```
THREAD IMPLEMENTATION T.i2
PROPERTIES
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 5ms..5ms;
  Period => 20 ms;
ANNEX Behavior_Specification {**
  states
    s : initial complete final state;
  transitions
    t : s -[on dispatch]-> s {
      D2 !<;
      computation(5 ms);
      D1 !<;
      D1 !>;
      D2 !>
    };
**};
END T.i2;
```

With such an AADL specification, running the two analysis tools gives again a similar execution trace (the gray rectangles indicate when Threads are accessing the Data).



However, if the concurrency control protocol property is removed, the execution trace produced by Marzhin quickly shows that a deadlock situation has occurred (the pink rectangles indicate a "wait for resource" state).



3.2. Composite Examples

We have shown in the previous section how the results obtained by Cheddar and Marzhin could be compared when applied to popular real-time modeling patterns. We will now address three case studies that are a little closer to industrial considerations.

3.2.1. ARINC 653

It is possible to describe partitioned architectures with AADL and its ARINC 653 Annex.

```
PROCESSOR IMPLEMENTATION powerpc.impl
SUBCOMPONENTS
  part1 : VIRTUAL PROCESSOR p1.impl;
  part2 : VIRTUAL PROCESSOR p2.impl;
PROPERTIES
  Scheduling_Protocol => ARINC653;
  ARINC653::Partition_Slots =>
    (10ms, 10ms);
  ARINC653::Slots_Allocation =>
   ERTS2014.odt
(reference(part1),reference(part2));
  ARINC653::Module_Major_Frame => 20ms;
END powerpc.impl;
```

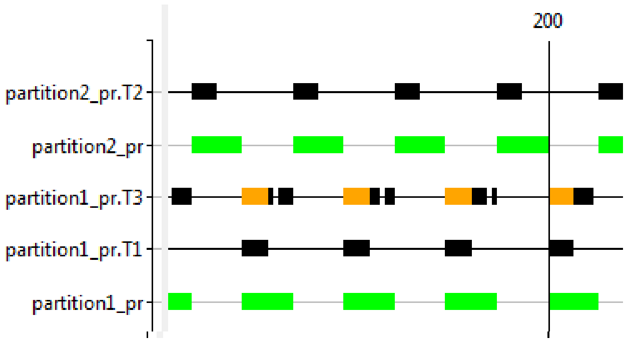
An ARINC 653 partition is declared in AADL by the association of a *Virtual Processor* handling its own scheduler and a *Process* containing a set of *Threads*. At the higher level, the partitions that are co-located on the same *Processor* are scheduled according to a time sharing protocol. A set of properties specifies how the time slots are allocated to the partitions.

```
PROCESS IMPLEMENTATION part1_process.impl
SUBCOMPONENTS
  T1 : THREAD T.i;
  T3 : THREAD T.i {Period => 21ms;};
END part_process.impl;

PROCESS IMPLEMENTATION part2_process.impl
SUBCOMPONENTS
  T2 : THREAD T.i;
END part2_process.impl;

THREAD IMPLEMENTATION T.i
PROPERTIES
  Dispatch_Protocol => Periodic;
  Compute_Execution_Time => 5 ms..5 ms;
  Deadline => 20 ms;
  Period => 20 ms;
END T.i;
```

In our simplified example, the first partition encompasses two *Threads* whereas the second partition has only one *Thread*.

The schedule table that is generated by Cheddar for this example is as shown below (the green rectangles show which partition is active):

And once again the simulation trace of Marzhin provide a similar result:

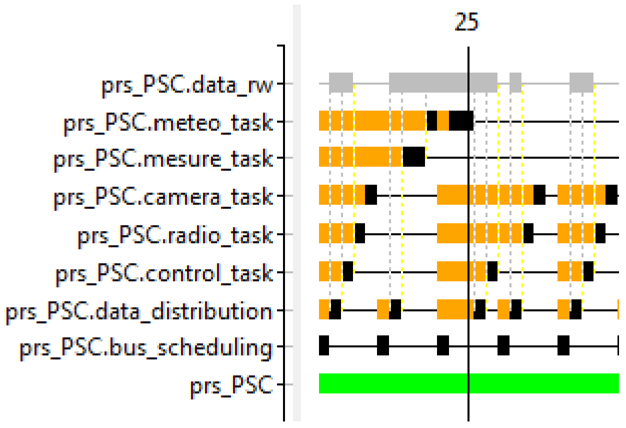


It must be noted that at the time of the redaction of this paper, inter-partition communications have not been implemented in Marzhin yet.

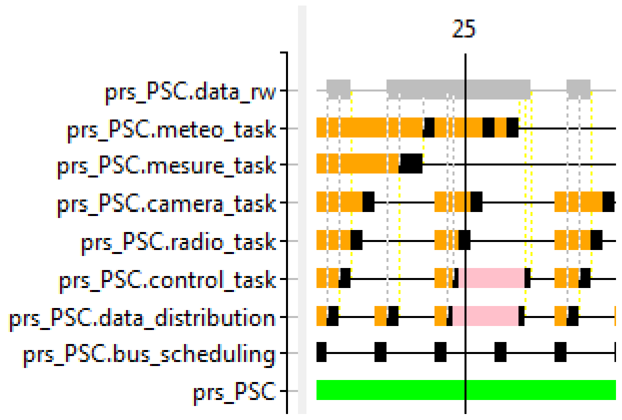
3.2.2. Priority Inversion

The following example is used to compare the results produced by Cheddar and Marzhin on a simplified representation of the Mars Pathfinder probe which failed because of an error known as "priority inversion". The AADL code used for this example is a lightly modified copy of the one that can be found at http://www.openaadl.org/aadlib.html.

Several *Threads* have access to the same shared *Data* component and are scheduled with a Highest Priority First protocol. When PCP is applied to the shared resource, the simulation traces that is generated by both Cheddar and Marzhin look as follows:



Note that the *Threads* are shown with an increasing level of priority (i.e. the bus scheduling *Thread* has the highest priority). If the PCP concurrency protocol is removed from the shared *Data* component, the "priority inversion" problem is displayed by Marzhin as follows:



We can observe that some high priority *Threads* are postponed by low priority *Threads* that are blocking the access to the shared *Data*. The priority inversion phenomena is highlighted by the fact that the high priority *Threads* are in a "wait for resource" state

3.2.3. Large Multi-processor System

It is also important to check that the use of this simulator is scalable. The test case that has been used for this verification is an open source AADL model developed by Rockwell Collins to describe the architecture of an avionic display system.

This model contains more than 12000 lines of AADL code, and defines 5 *Processors*, 22 *Processes* and 123 *Threads*. With the current state of this model, only a single Process can run on a Processor which restricts the complexity to 5 *Processors*, 5 *Processes* and 33 *Threads*. A separate set of time lines is produced by Marzhin for each *Processor*.

The next step in this scalability check will be to modify the case study into a set of multi-partition system, so that the entire set of *Processes* and *Threads* would have to be actually managed by the Multi-Agent simulator.

It must be noted that the version of Marzhin that is available at the time this paper is written does not support inter-*Processors* communications that would have to be modeled by *Buses*.

4. 3D Graphical Animation

We saw that a Multi-Agent kernel can be used to contribute to Real-Time system analysis activities. However an original usage of this technology is to realize 3D graphical animations. One of the tasks of the SMART project consists in studying the feasibility of using the same Multi-Agent application to animate a 3D graphical representation of a system and to simulate the corresponding control/command logic of this system.

The Real-Time behavior of 3D applications entities are usually described independently of graphical rendering issues, by the mean of state-transition diagrams or behavior trees. The implementation of the control logic for the interaction between the various entities must be done in the 3D application under the responsibility of its designer.

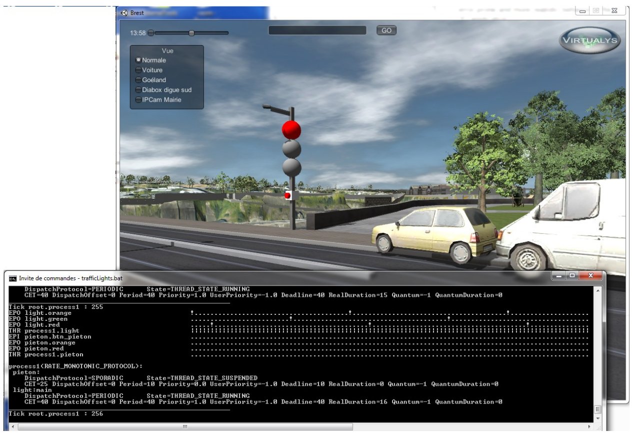
There is thus a great interest in reusing the implementation of the control logic that is derived from the corresponding AADL specification when such a model has already been developed for Real-Time analysis purposes. The benefit of this approach is a less error-prone and more realistic behavior of the 3D application.

The Marzhin simulator acts as a server that implements a bi-directional communication interface in order to send commands to the simulator and receive dynamic information about the state of the simulation. A client application can thus collect the state of the various entities (*Threads*, *Ports*, …) at each simulation cycle. This data collection can be organized by specifying measurement probes that observe the dynamic behavior of the application.

The graphical time-lines editor that is embedded in AADL inspector is one of the client applications of the Marzhin simulator. Similarly, the 3D environment must also implement the client side of the communication interface with the simulator so that it can be used to animate a 3D representation of the system and send user-controlled events to Marzhin.

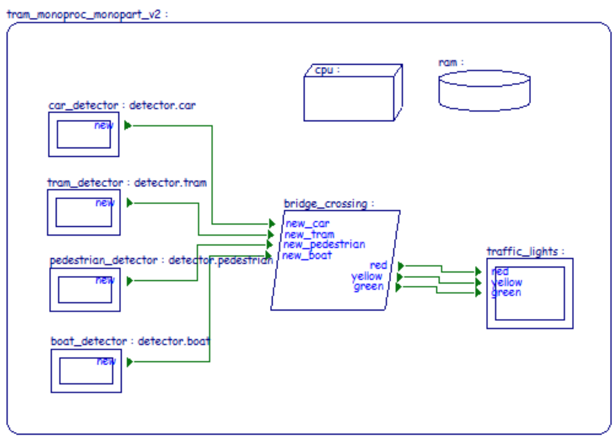
At this stage of the exploratory part of the SMART project for the animation of a 3D environment, the first experiments show that the communication between the 3D virtual world and Marzhin simulation is done through an interface integrating AADL features. These are *Event Ports*, *Data Ports* and and *Event Data Ports* in both input and output.

In output of the Marzhin simulator, the 3D animation application retrieves the state changes of the various AADL output *Ports* and the subscribing 3D entities are dynamically warned of the changes of the status of these *Ports*. The AADL input *Ports* of the simulator Marzhin are similarly connected to events occurring in the 3D environment.



```
    DispatchProtocol=PERIODIC      State=THREAD_STATE_RUNNING
    CET=40 DispatchOffset=0 Period=40 Priority=1.0 UserPriority=-1.0 Deadline=40 RealDuration=15 Quantum=-1 QuantumDuration=0
_______________________________________________________
Tick root.process1 : 255
EPO light.orange
EPO light.green
EPO light.red
THR process1.light
EPI pieton.btn_pieton
EPO pieton.orange
EPO pieton.red
THR process1.pieton

process1(RATE_MONOTONIC_PROTOCOL):
 pieton:
    DispatchProtocol=SPORADIC      State=THREAD_STATE_SUSPENDED
    CET=25 DispatchOffset=0 Period=10 Priority=0.0 UserPriority=-1.0 Deadline=10 RealDuration=0 Quantum=-1 QuantumDuration=0
 light:main
    DispatchProtocol=PERIODIC      State=THREAD_STATE_RUNNING
    CET=40 DispatchOffset=0 Period=40 Priority=1.0 UserPriority=-1.0 Deadline=40 RealDuration=16 Quantum=-1 QuantumDuration=0
_______________________________________________________
Tick root.process1 : 256
```

The example above shows how a traffic light could be controlled by a Marzhin simulation described in AADL. The Different color signals are connected to *Data Port* outputs. It is possible for a virtual pedestrian to stop the cars by clicking on a red button in the 3D environment, this request is connected to an *Event Port* input of the AADL model. The interactive 3D objects (sensors and actuators) can be implemented by AADL *Device* components in order to well separate the control system and its environment.


tram_monoproc_monopart_v2 :
cpu :
ram :
car_detector : detector.car
new
tram_detector : detector.tram
new
pedestrian_detector : detector.pedestrian
new
boat_detector : detector.boat
new
bridge_crossing :
new_car
new_tram
new_pedestrian
new_boat
red
yellow
green
traffic_lights :
red
yellow
green

This approach provides a concrete solution for the integration of software models and virtual digital models in 3D. Although these two kinds of models are currently developed in separate design tracks for a same system, such an integration can benefit to both of them.

For the real-time control software design team, having the ability to perform realistic simulations in the early stages of the modeling process can help in ensuring requirements traceability as well as performing design trade-offs with future users.

For the 3D model design team, the use of actual models of the future software to animate the automated parts of the simulation will provide a more realistic and intuitive behavior of the system when used by its virtual users (training activities for instance).

However, our goal is not to develop an all-in-one integrated modeling framework that would encompass all the embedded software and 3D design activities of a complete system. On the contrary, we intend to focus on the specification of a library of reusable components representing the sensors and actuators that are the only entities that need to be described by both a real-time behavior and a 3D animation.

One possible approach to reach this goal would be to enrich the definition of the AADL *Device* components that are representing these sensors and actuators with a 3D annex.

5. Related Works

In this part, we present several works which also use a Multi-Agent approach in the context of real-time and/or scheduling systems, either for validation or implementation goals.

In order to validate agent based models, [5] proposes a framework called VOMAS (Virtual Overlay Multi-Agent), where a virtual overlay Multi-Agent system can be used to validate simulation models. While a simulation is executed, the overlay system performs monitoring and logging functions, and then checks constraints given by the system designer. Although VOMAS has not been especially design for verifying real-time architecture, this approach could be effectively applied in the context of SMART project, in order to validate Marzhin simulations driven by some constraints issued from the Cheddar analysis.

In [6], the authors address the problem of dynamic scheduling of resources for multiple projects in real-time. Agents are used to represent projects, composed of several tasks, and resources. Projects have scheduled work to be done by different resources. Resources are endowed with some capabilities that are requested to do the work. In order to visualize the current state, a monitoring agent is created, and allows to have a global behavior of the system. The implementation is based on a basic Liu and Layland [2] model of the application, while, in our works, an AADL model defines the simulation parameters.

Beyond the use of Multi-Agent framework for simulation and validation aims, the agents may be integrated for implementing the system itself. A real-time agent architecture is described in [16], and implements on-line scheduling algorithms, like EDF for instance. In the context of multi-core architectures of real-time systems, [15] combines a AMAS (Adaptive Multi-Agent Systems) with an affinity-based processor scheduling. The approach consists to define for each processor one agent, Processor Agent, and a central Middle Agent which interacts with the scheduler; the Middle Agent acts as a storage space for faster scheduling.

As shown in this section, various other projects have done the choice of using Muti-Agent technology within system and software modeling frameworks. However, none of these works have considered a close integration of an existing generic and lightweight Multi-Agent kernel and a standardized real-time system modeling language (AADL).

Conclusion and Future Work

This paper describes an outcome of the SMART collaborative project. The purpose of this work is to extend the capabilities of a real-time software simulator called Marzhin and to compare its results with those produced by Cheddar in order to verify that they are consistent.

After having given some elements showing this consistency, the paper introduces another benefit brought by the Multi-Agent approach by describing how such a simulator can also be used to animate a 3D graphical model of the system for which the real-time software is being developed.

The results presented in this paper correspond to the outcomes obtained at the middle of the project. We plan to provide a more complete solution at the end of the project in September 2014.

These future works will in particular address the following topics and apply them to case studies:

- Inter-*Processors* communication by *Bus*
- Impact of the use of multi-core *Processors*
- Enriched support of the AADL *Behavior Annex*
- Definition of reusable real-time 3D components
- UML/MARTE models import front-end

Acknowledgments

This work has been performed in the context of the SMART project with financial support from the Council of Brittany, Council of Finistère, BMO and BPI France.

References

[1] P. Feiler, B. Lewis and S. Vestal, "The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering", Workshop on Model-Driven Embedded Systems, 2003.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environnment", Journal of the Association for Computing Machinery, vol. 20, n° 1, pp 46-61, 1973.

[3] F. Singhoff, J. Legrand, L. Nana, L. Marcé. "Cheddar: a Flexible Real-Time Scheduling Framework", ACM SIGAda Ada Letters, 24(4):1-8, ACM Press. 2004.

[4] P. Dissaux, J. Legrand, A. Plantec, M. Kerboeuf, F. Singhoff, "AADL Design Patterns and Tools for Modelling and Performance Analysis of Real-Time Systems", Embedded Real-time Software and Systems Conference (ERTS), 2010.

[5] M. Niazi, A. Hussain and M. Kolberg , "Verification and Validation of Agent-Based Simulation using the VOMAS approach", Proceedings of the Third Workshop on Multi-Agent Systems and Simulation (MAS&S), 2009.

[6] J. Alberto, A. Arauzo, J. Pavon, A. Lopez-Paredes and J. Pajares, "Agent based Modeling and Simulation of Multi-project", Proceedings of the Third Workshop on Multi-Agent Systems and Simulation (MAS&S), 2009.

[7] L. C. DiPippo, E. Hodys and B. Thuraisingham. "Towards a real-time agent architecture - a whitepaper", Fifth International Workshop on Object-oriented Real-Time Dependable Systems (WORDS), pp 59-64. 1999.

[8] N. R. Jennings, K. Sycara and M. Wooldbridge, "A Roadmap of Agent Research and Development", IAutonomous Agents and Multi-Agent Systems, n° 1, pp 275-306, Kluwer Academic Publishers, 1998.

[9] K. Decker, M. Williamson and K. Sycara. "Matchmaking and Brokering", Second International Conference on Multi-Agent Systems (ICMAS), 1996.

[10] T. Finin, R. Frtzson, D. McKay and R. McEntrire, "KQML as an Agent Communication Language", Third International Conference on Information and Knowledge Management (CIKM), ACM Press, 1994.

[11] J. Graham and K. Decker, "Towards a Distributed, Environment – Centered Agent Framework", Workshop on Agent Theories, Architectures and Languages (ATAL), 1999.

[12] The Foundation for Intelligent Physical Agents, http://www.fipa.org/specifications/index.html.

[13] D. Benech and T. Desprats, "A KQML-CORBA based Architecture for Intelligent Agents Communication in Cooperative Service and Network Management", IFIP/IEEE International Conference on Management of Multimedia Networks and Services, pp 8-10, 1997.

[14] "CORBA Services: Common Object Services Specification. Vol 1.", OMG Specification, 1996.

[15] G. Muneeswari and K. L. Shunmuganathan, "A Novel Hard-Soft Processor Affinity Scheduling for Multicore Architecture using Multi-agents", European Journal of Scientific Research, vol 55, n° 3, pp 419-429, 2011.

[16] J. H. M. Lee and L. Zhao, "A Real-Time Agent Architecture: Design, Implementation and Evaluation", Intelligent Agents and Multi-agent Systems. Lecture Notes in Computer Science. Volume 2413, pp 18-32, 2002.

[17]Ellidiss download site:
http://www.ellidiss.com/download