Multi-Paradigm Modeling for early Analysis of ROS-based Robotic Applications using a Library of AADL Models

Eric Senn
Lab-STICC
Université de Bretagne Sud
Lorient, France
eric.senn@univ-ubs.fr

Lucie W. J. Bourdon
Lab-STICC
Université de Bretagne Sud
Lorient, France
lucie.bourdon@univ-ubs.fr

Dominique Blouin
LTCI, Telecom Paris,
Institut Polytechnique de Paris
Palaiseau, France
dominique.blouin@telecom-paris.fr

Abstract

ROS, the Robot Operating System, is a middleware that eases the programming of robotic applications significantly, bringing standard communication and synchronization mechanisms to a wide variety of operating systems and computers or embedded computer boards. However, a robotic application is a complex set of many services, with many relations between them, and multiple choices have to be made regarding the software and the hardware architectures. To rely on a formal representation of those two, from which early analysis can be performed, is extremely beneficial since it allows detection of design errors early in the process. This is what we present in this paper. Our approach is based on the Architecture Analysis and Design Language (AADL) and on a set of AADL models to represent both the application and the robot, including its embedded computers and its many devices. Those models are gathered into an open source AADL library from which ROS developers can largely benefit.

CCS Concepts

• **Computer systems organization → Embedded systems**; **Robotics**; • **Software and its engineering → System modeling languages**.

Keywords

ROS, AADL

ACM Reference Format:
Eric Senn, Lucie W. J. Bourdon, and Dominique Blouin. 2022. Multi-Paradigm Modeling for early Analysis of ROS-based Robotic Applications using a Library of AADL Models. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3550356.3563129

1 Introduction

Mobile robots for smart factories are complex systems. On the hardware side, they gather many different parts, including high-tech mobile mechanisms, electro-mechanical devices, high precision sensors and actuators, with associated electronic interfaces and power electronic converters for control and regulation. On the software side, they usually have to perform different tasks at the same time. Beside their main mission, obviously reaching their destination, they have to avoid unscheduled obstacles, to ensure safety for every person on their path, and to interact with high-level control software or supervisor. The sole navigation task involves several essential services among which global and local path-planning, localization, mapping, possibly target tracking, etc. In addition, a set of low-level tasks, one for each device attached to the robot, will be running and communicating with the associated hardware drivers.

In the case of ROS [16], a specific service is associated with all of these tasks. The role of this service is to transform low-level messages to or from the device, to the asynchronous publish and subscribe based message passing mechanism implemented in ROS. There, messages travel through topics, complying with a well-defined standard format. This allows deploying services onto different platforms, as long as a ROS distribution is available for them. In fact, ROS was initially built on top of Linux/Ubuntu. It is now supported by different operating systems and associated computers.

Another benefit of using ROS is to gain access to a large set of dedicated commands and tools for development, debugging, monitoring, and configuration, as well as accurate simulators like the multi-physical GAZEBO. As a result, ROS is widely used in industry, significantly reducing the time-to-market of many new robot development projects.

But using ROS does not come without drawbacks. In fact, running a middleware, and especially, in the case of ROS2, on top of another middleware, namely DDS (Data Distribution Service) to tackle the publish and subscribe communication scheme, consumes processing power. Embedded systems being naturally limited in such power, we quickly reach their limits and what we observe are robots executing slower than expected, missing their deadlines, or completely failing their mission. In a navigation stack for instance, when the costmap2D service fails delivering an obstacle map in time, the robot stalls. Speed and accuracy are often the first criteria to evaluate the overall performance, as used in [9] for trajectory planning, or [6] for an automated vehicle application. While investigating the reasons behind such performance degradation, we observe, for a full navigation stack like in [12], high CPU loads and slowed down communications. We also observe more deadlines misses for a number of tasks.

In those situations, either the hardware does not meet the application requirements, even if this may be arranged by tweaking

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3563129

the application parameters, or the application is poorly deployed onto the hardware resources as shown in [8]. Determining which of the two reasons is the cause of the problem, and finding a fix to this problem is not straightforward. We clearly need some help to understand what is going on and what has to be done, and this is where model based engineering comes in. To clarify, we need:

- A comprehensive view of the whole application, including:
  - The software as a set of interacting ROS nodes,
  - The hardware including the robot, its sensors and actuators and embedded computer boards,
  - The binding of software services to hardware resources.
- A tool to conduct performance analysis such as:
  - CPU load,
  - BUS load,
  - Timing and in particular schedulability and latency.

We also want to be able to conduct those analyses as early as possible in the development cycle due to time-to-market and therefore, the modeling of the application should be fast, and the analysis should be quick to perform and accurate.

The *Architecture Analysis and Design Language* (AADL) [5, 10, 11] quickly appeared as an excellent choice to cover our needs. It covers the domain of Cyber-Physical Systems (CPS, including robotics) with a focus on real-time embedded systems including:

- Software components (process, thread, data, port, etc.),
- Hardware components (processor, bus, memory, devices),
- Deployment specification with bindings indicating to which hardware component a software component is bound to.

Besides, AADL embeds in its heart several paradigms, making it a multi-paradigm modeling language allowing to cover several parts of CPSs; it is Object-Oriented (OO), which is very helpful in building component libraries, Synchronous Data Flow (SDF) through its *data port* construct, and Discrete Event Dynamic Systems (DEv) through its *event data port* construct and its DEVS Annex (DA) [1], among others.

AADL is supported by different tools, among which the Open Source AADL Tool Environment (OSATE) [20], which out of the box provides the analysis capacities we are seeking. In fact, analysis from AADL models relies on dedicated properties which values have to be set. This involves the profiling and benchmarking of both software and hardware components, on which we rely to develop the AADL component library we present in this paper.

Our paper is structured as follows. We first present the workflow of our approach, followed by the library of components created and used within this workflow. We then show how performance analysis can be done from the complete model of a robotic application. Then we present the related work and conclude the paper.

2 The workflow

The workflow we propose is depicted in Figure 1 in a notation similar to the Process Model part on an FTG+PM (Formalism Transformation Graph + Process Model) [15]. Activities are depicted as blue rounded corner rectangles and the models they process as green rectangles. Dashed arrows link the models with the activities that require them while solid arrows represent control flows between activities. All these activities are performed manually for the time being, although we plan to automate some of these activities such as code generation in future work.

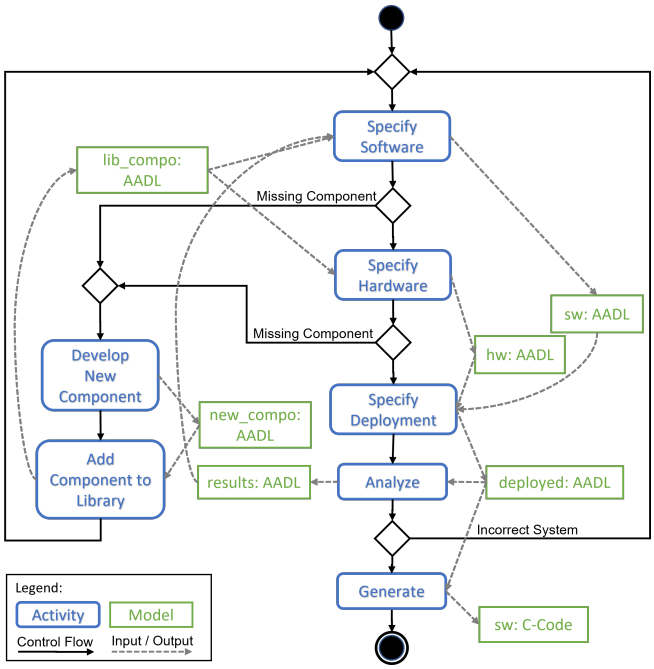

Figure 1: Workflow model.

The starting point is to specify the software application and the robot (hardware) that it drives in terms of software and hardware AADL models obtained from our library of components. While this is being done, there are two possible situations. First, all the components already exist in our library, which means that the complete robot can already been modeled including the computer boards it carries. Then we simply instantiate and interconnect the components from the library and them we specify the deployment of the software application onto the execution platform of the robot, describing how every software component is deployed onto components of the hardware. This is achieved using binding properties to create a deployment model.

From this deployed system model, we can then run different analyses depending on the properties that have actually been set in the components models. We typically analyze the performance of this model in order to find a better deployment of the application on the robot hardware, or to find what is the reason for observed failures.

These activities are performed iteratively, taking as input results from the analyses to modify the initial models so that different specifications and deployments can be evaluated, as well as different sensors or computer boards if no solution satisfies the desired system properties. Once a correct model has been developed, we generate the software code and assemble the selected hardware to implement the real system.

In case a component is not included in our library, we build a model of this component according to our modeling approach for ROS nodes describing its inputs, outputs and the different threads it

will be running. This includes profiling the performance of our new node on the hardware it might be deployed onto. From the results of this profiling, AADL properties will be defined and added to the component model to characterize its performances. If it is a new piece of hardware that we are dealing with, we need to benchmark its capabilities with respect to the analyses we intend to perform.

The profiling of ROS nodes with measurements has been discussed in former publications, specifically for CPU load analysis [18] and bus load analysis [17]. It relies on standard tools and commands from both Linux and ROS. Hardware counters are read from Linux (cf *perf, mpstat, etc.*), while the CPU and operating system are properly configured (cf *cpufreq, taskset, chrt, etc.*). Additional classical developer tools like the *valgrind* suite may also be used. In addition, ROS tools allow monitoring the application behavior, message rates, bandwidth requirements (see *rostopic, rosnode, etc.*).

Once developed, the new component is added to our library and the process of specifying the system is repeated until no more new components are required.

Our approach, being simple, is also fast. A simple node can be profiled in less than half an hour. Results obtained from the analyses that follow are relatively accurate, with an average error under 5%. Adding a new implementation of a ROS node model to the library is straightforward: an existing implementation of the associated AADL process and threads has to be duplicated, and required properties values have to be refined for the new component.

Table 2 shows an extract from our catalog of AADL models for ROS nodes. For each node, measurements are made to determine the values for properties that allow to check if the way the application is deployed on the hardware resources enables the robot to behave as expected. Overloaded CPU or bus will lengthen the compute execution times and deadlines will be missed. Even with a soft real-time middleware like ROS, performances will be degraded and mission objectives may not be achieved.

Performance actually depends on the robot's embedded computer system. Changing the hardware could help solve problems, and our library of models precisely allows us to check that. Indeed, measurements are made for different single board computers, as shown in Table 2. The model for a ROS node comes with several implementations, carrying different set of analysis related parameters. Once a board is chosen, and integrated in the robot's complete model, deployment solutions can be evaluated.

3 The Library

3.1 Overview

A typical robotic application involves several services : some are deployed on the robot with its embedded computer board(s). Others might be deployed on a remote computer for control purpose. In ROS, a service is implemented as a *node*. Nodes communicate through virtual channels called *topics*. A node may publish on a topic, subscribe to a topic, or do both at the same time.

The model of a ROS-based application must reflect this set of nodes with their interconnections. Hence we provide a set of packages with models for different robotic services, aka nodes in ROS. Figure 2 shows the model of a typical target tracking application using these packages. An AADL model is not limited to a graphical representation of an assembly of components, since it provides both a graphical and a textual notations so that users can use whatever notation best fits their needs. Every component, connection, and port can be typed and the compatibility of connecting one component with another is constantly checked while building the system. Moreover, different parameters or properties can be added to the models, being easier to visualize with the textual notation. Such properties allow for performing the different analyses that will be presented in the following section.

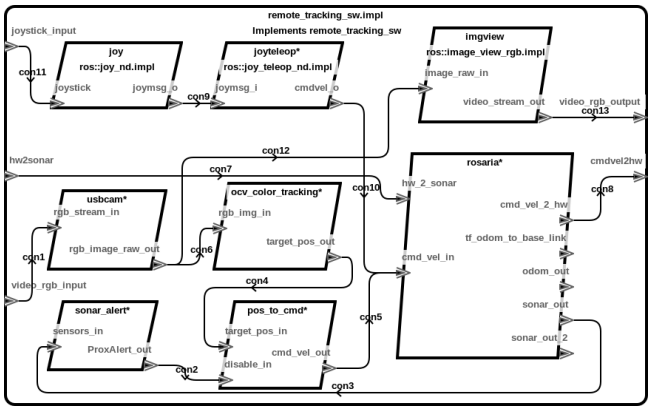

Figure 2: The AADL model of a tracking application: ROS nodes are modeled as process components.

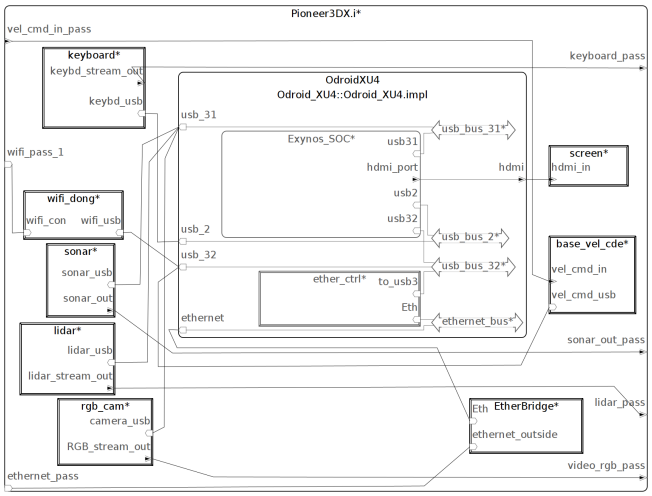

Figure 3: Pioneer 3DX model.

Figure 3 shows an excerpt of the model of the Pioneer 3DX robot from *Generation Robots* that we use in our lab, and for which we have developed a complete model. In the center of this figure is a component for the Odroid XU4 Single Computer Board (SBC) from *HardKernel*. The XU4 contains different components that we have included inside its model, but that are not displayed here to keep the figure small. Among those components is the Exynos 5422 from *Samsung*, which is a heterogeneous octo-cores System on Chip (SoC), included in our library of hardware components.

Like the Pioneer 3DX, a mobile robot generally includes two electronic boards. A powerful enough SBC is in charge of the heavy processing services, typically the navigation stack in a mobile and autonomous robot. It runs ROS and every needed services as ROS nodes. Another board, often built around a micro-controller, for instance the STM32 in Arduino boards, deals with tight control of the actuators. Specific ROS bridge nodes are used to exchange messages from the application on the SBC to the controller board.

Table 1 provides a list of robots and SBC for which we have developed models. Any combination of boards and robots might be tried to analyze how well the software fits on it. In addition to this list, different SoPC have also been modeled: Virtex 5 and Virtex 2 Pro from Xilinx, and Cyclone 3 LX from Altera. Those programmable and re-configurable circuits include one or several CPUs in the form of soft or hard cores (Naios, Micro-Blaze, Power-PC).

Our library is organized as follows.

- **A core *ros* package:** our core *ros* package contains the declaration of every component type and the associated implementation that will be used in our library of ROS nodes. Indeed, according to the AADL modeling style, inputs and outputs of any component are declared in component types and the internal structure of the component is declared in component implementation(s) of a type.
- **A collection of packages for different ROS nodes:** one AADL package is provided for each node we want to include in our library. A node package includes the node declaration as an AADL *process*, defining its inputs and outputs, and its platform-independent implementation where its internal structure is defined. In this implementation, every thread in the node is declared as an AADL *thread subcomponent*, together with its connections to other threads and to the node's input and outputs.
  A node package also includes several platform-dependent implementations for every thread in the node. Those implementations carry AADL property values related to the thread performances measured on different embedded computer boards. For one computer board, properties might differ depending on the actual CPU cores on which the thread is running, getting as many different implementations as needed.
- **A package for ROS messages:** This package will contain the declaration of types and associated implementations for every kind of messages that the nodes in our library can exchange.
- **A set of packages for modeling the hardware:** here we find packages for different robots, as well as for single computer boards and the system on chip or multiprocessor CPU they carry.

In fact, our AADL library does not include a model for all nodes that can be found in a given ROS distribution. To obtain such an exhaustive library, several tens of nodes would need to be modeled while we only use a fraction of them in our applications. As explained in section 2, our approach is incremental: we add a new AADL component to our library every time we use a new node. It is the same for new hardware parts or single board computers.

As an example, Figure 4 shows an excerpt of the packages tree used in the model of a robotic application running on the LeoRover robot with a RaspBerry Pi4B SBC. The whole system is modeled in the *sys_rover* package. It uses components from the *sw_rover* and the *leo_rover* packages. This last one itself uses components from the *USB*, *ETHERNET*, and *Raspberry_PI4B* packages.

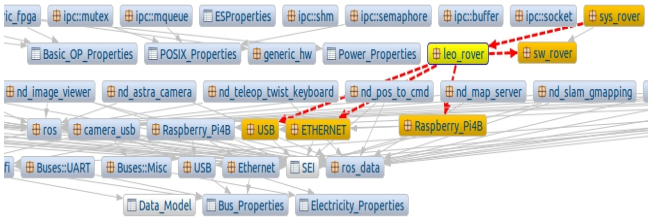

Figure 4: Packages tree.

3.2 Modeling ROS Nodes

A robot software based on ROS is a constellation of nodes communicating through topics. As mentioned earlier, a dedicated package is created for every node that we want to include in our library. Every node in those packages will inherit from the high-level node component type and implementation defined in our core *ros* package. A ROS node is modeled as an AADL *process* component that includes several threads, mimicking the actual ROS node code. The typical threads running in a ROS node are :

- The main thread in every node.
- The receiver, or subscriber, thread listens for incoming messages and pushes them on the callback queue after de-serializing them. Indeed, this thread is associated with a callback function called to deal with the data received on the topic listened to. The callback function is placed in the FIFO callback queue of the node.
- The spinner thread keeps taking the callbacks from the callback queue and executing them one by one in an infinite loop acting as a "spinning" thread. The spinner thread can run periodically or every time a message is received. One ROS node has one spinner thread by default. However, several spinners could be spawned to allow for multiple concurrent executions of callback functions [21].
- A thread type to implement callback functions in nodes (from [3]).
- A thread type to implement publishers in nodes. The publisher thread takes messages from the FIFO publisher queue, which is filled every time a call to *publish()* is made.
- A thread to provide a ROS *service* to a requesting node.
- A timer thread to implement a ROS *timer* in nodes.
- A thread type to implement a broadcaster of Transformation of Frames (TF), i.e. the position of one frame relative to another, in nodes. One frame is associated with each part of a robot. Keeping track of the position of every frame is one hidden but essential service in a robotic application.
  A dedicated node named tf will publish on the /tf topic with many others in general in a robot application. Unlike in [3], where /tf is modeled as an independent bus to which

Table 1: Hardware models: Robots & SBC

robots	maker	SBC	maker	SoC	maker	Architecture
Pioneer 3DX	Generation Robots	Odroid XU4	Hardkernel	Exynos 5422	Samsung	Octo core ARM15 / ARM7
Rover Pro	Rover Robotics	Jetson Xavier	Nvidia	Carmel/Volta	Nvidia	6-core NVIDIA Carmel ARM
TurtleBot 2	ClearPath Robotics	RaspBerry Pi4B	RaspBerry	BCM2711	Broadcom	Quad core Cortex-A72
LeoRover	Kell Ideas	RaspBerry Pi4B	"	"	"	"
WifiBot	Nexter Robotics	LE-376	Commel	Atom D510	Intel	Dual core Atom

different nodes require an access, we model it as a regular topic that will be bound to the ROS virtual bus.

Like with AADL processes used to model ROS nodes, an AADL thread component has one declaration where its inputs and outputs are specified, and a "parent" implementation which defines its inner structure. As many child implementations as needed will inherit from this parent implementation to carry properties reflecting the performance of the thread on different hardware targets and CPU cores. The AADL OO paradigm and its *extends* mechanism is used there. New AADL *properties* values will be declared with a new set of performances for a thread. Those properties, namely *MIPS-Budget* and *Compute_Execution_Time*, are directly issued from the measurements presented in [18] and [17].

As an example, we provide an extract of the AADL model for the usb_cam node found in many ROS distributions. This node takes the video stream from a RGB camera plugged in the USB bus of the computer board and publishes the video frames on a dedicated ROS topic. The first part of our nd_usb_cam AADL package includes the PIM definition: the node input and output in a process component, and its internal structure in the associated implementation.

```
process usb_cam_nd extends ros::node
 features
  rgb_stream_in: in event data port ros_data::video_stream.rgb;
  rgb_image_raw_out: out event data port ros_data::Image.rgb;
 end usb_cam_nd;

process implementation usb_cam_nd.impl
 subcomponents
  image_broadcaster: thread imagePublisher.impl;
  usbSpinner: thread usbcam_spinner.impl;
 connections
  con1: port image_broadcaster.pub_msg -> rgb_image_raw_out;
  con2: port rgb_stream_in -> usbSpinner.rgb_stream_in;
 end usb_cam_nd.impl;

thread imagePublisher extends ros::publisher
 features
  pub_msg: refined to out event data port ros_data::Image.rgb;
 end imagePublisher;

thread implementation imagePublisher.impl
 properties
  Period => 33333 us;--@ 30 images/s
 end imagePublisher.impl;
```

The second part (PDM) shows the platform and CPU-dependent implementation for the Odroid XU4 (from HardKernel) board, which includes ARM A15 and A7 cores. We chose to put the node Period on the PIM implementation of the node, since it does not depend on the hardware target, but on the RGB camera we use. The Compute_Execution_Time property is set on the PDM side.

```
process implementation  usb_cam_nd.xu4_a15 extends usb_cam_nd.impl
 subcomponents
  image_broadcaster: refined to thread imagePublisher.xu4_a15;
 properties
  SEI::MIPSBudget => 141.0 MIPS;--(197 MIPS)/(1.4 IPC)
 end usb_cam_nd.xu4_a15;

thread implementation imagePublisher.xu4_a15 extends
      imagePublisher.impl
 properties
  Compute_execution_time => 2319 us .. 2319 us;
  Queue_Size => 512 applies to pub_msg;
 end imagePublisher.xu4_a15;
```

Table 2 shows an extract from our catalog of AADL models for ROS nodes.

4 Analyses from AADL Models

Before beginning any analysis from our models, we need to specify how the software is deployed on the hardware. This consists of first specifying on which processor or set of processors every node of the application will be executed.

The Actual_Processor_Binding property is used, for example here to bind the previously mentioned usb_cam node to CPU #1 in the cluster of ARM15 processors in the Exynos 5422 SoC of the Odroid XU4 board:

```
Actual_Processor_Binding => (reference (p3DX.OdroidXU4.Exynos_SOC.
      big_procs_cluster.big_proc1)) applies to rem_trk_sw.usbcam;
```

Next, we must indicate which bus will transfer the data of every connection between two nodes. This is specified with the Actual_Connection_Binding property in the AADL model of the deployed implementation of our complete system; e.g. for the output of our usb_cam node:

```
Actual_Connection_Binding  => (reference (ROSbus)) applies to
      rem_trk_sw.con6;
```

Those bindings properties are added to the high-level implementation model of the system, which includes the application model and the robot model as subcomponents. From them, the following analyses can be performed, using standard tools available in OSATE.

Analyze Resource Allocations (Bound): The CPU load is computed from the Compute_Execution_Time and Period properties for each bounded thread. The MIPS demand for a thread is the MIPSCapacity of the CPU (defined in the SBC / SoC model) times the load. The total MIPS demand for the processor is the sum of every bounded thread demand. Whenever this demand exceeds the total capacity of the processor, an error is reported.

Analyze Resource Budgets (Not Bound): The MIPSBudget (SEI standard) property must be set for the process. The tool adds the MIPS demands for every thread inside the process and checks if the total does not exceed the budget. An error is reported if so.

Analyze Bus Load (Bound): Bus load analysis is performed from the size of the messages to be transmitted and their frequency. The first one is the Data_Size property, which is defined in our *ros_data* package. We give below the definition for the 640x480 RGB images, 8 bits per channel, 3 channels, message stream from the RGB camera that the usb_cam node of our former example is

Table 2: Catalog of models (extract); *bandwidth in* $KBytes/s$*, execution time in* μs

					Odroid XU4		JETSON XAVIER NX	RaspBerry Pi4
			CPU		ARM A15	ARM A7	NVIDIA Carmel	ARM A72
			MIPSCapacity		2000	1100	2400	1500
			frequency (GHz)		2	1.4	1.9	1.5
node	measure	computed	AADL properties	PIM				
usb_cam			Period (µs)	33333				
		IN: 640x480x3 Bytes	Data_Size (bytes)	921000				
	MIPS (raw)				197.00	256	212.62	350
	IPC				1.40	0.33	2.14	0.90
			MIPSBudget		141.00	776.00	99.36	388.89
			Compute_execution_time		2319.00	18500	2529	8411
		CPU_load %			6.96	55.50	7.59	25.23
		MIPS_demand			139	610	182	378
		Bandwidth_demand		27630				
pos_to_cmd			Period (µs)	100000				
			Data_Size (Kbyte)	48				
	MIPS (raw)				1.03	1.12	1.00	1.57
	IPC				0.12	0.12	0.13	0.35
			MIPSBudget		8.62	9.34	7.69	4.49
			Compute_execution_time		700	1060	288	104
		CPU_load %			0.70	1.06	0.29	0.10
		MIPS_demand			14.00	11.66	6.91	1.56
		Bandwidth_demand		0.48				
ocv_color_tracking			Period (µs)	33333				
			Period OUT(µs)	100000				
		OUT : Point.impl	Data_Size (bytes)	24.00				
	MIPS (raw)				4300	1450	370	321
	IPC				1.95	0.85	2.34	0.95
			MIPSBudget		2205	1705	158	338
			Compute_execution_time		37000	40000	4127	7283
		CPU_load %			111	120	12.38	21.85
		MIPS_demand			2220	1320	297	328
		Bandwidth_demand		0.24				

processing. In the PIM model of the thread imagePublisher given earlier, we find that the format of the *out event data port* is *refined to* this particular data size:

```
data implementation Image.rgb extends Image.impl
  properties
  --640x480x3=921600 Bytes #0.92MBytes
  Data_Size => 921 KByte;
  end Image.rgb;
```

The frequency of the message is the Period property of the thread component that issues the message. From the period and data size, the tool calculates the bandwidth demand for a publisher thread on the bus onto which its output connection is bound. In our example, that would be 921 KBytes× $1/33333\mu s$ = 27.63 MBytes/s.

Bind and Schedule Threads: This tool from OSATE performs a static scheduling analysis of the set of tasks in the application. It uses the following few properties to be set for every thread we want to consider:

```
properties
    Dispatch_protocol => periodic;
    period => 125 ms;
    compute_execution_time => 1 ms .. 25 ms;
    deadline => 125 ms;
    priority => 7;
    POSIX_Scheduling_Policy => SCHED_FIFO;
```

Real-time Scheduling Analysis: A dynamic scheduling analysis is achieved with the help of the dedicated plugin *Cheddar 3.x* [19]. This plugin, which might be run as an independent external tool, also considers possible accesses to shared data and the associated priority inheritance mechanism if used. Then it simulates the scheduling of the whole set of tasks, exhibiting possible deadline misses although the system is found statically schedulable.

```
data data_rw
  properties
    Priority => 6;
    Concurrency_Control_Protocol => Priority_Inheritance;
  end data_rw;
```

5 Related Work

Different works have recently studied the use of AADL for checking robotic applications. Latencies are analyzed in [4], however regardless of CPU or bus load, which actually impacts a robot reaction time. Besides, the approach is not dedicated to ROS-based application. The authors in [14] focus on ROS components but have to develop their own modeling language. They propose a modeling of task chains to evaluate response times, but tasks are only assigned to one CPU core and communication needs are not checked against bus capacities.

The benefit of modeling and developing ROS-based robotic application with AADL has been presented in [2] where an automatic generation of ROS code from the model [3] is proposed. The AADL model is only built for the software while hardware performances are not considered.

In [13], the author tackled, as we do, AADL modeling, hardware profiling and deployment analysis at the same time. His approach for measuring the compute execution time is however far more complex since it involves modification and rebuilding of the software. This is definitely something we do not want for our own code, and even less for entire ROS on-the-shelf packages we use in our applications. Besides being time consuming, this is also an intrusive approach that impacts performances and should be evaluated. Nodes are also considered independently, whereas we observe a modification of performances when they are connected to each others. Hence the necessity to profile a node in its proper usage context. Moreover, MIPS budget properties are not determined, which prevent from checking MIPS demand at the process level.

6 Conclusion

We have presented a multi-paradigm modeling approach and its library of AADL components to allow early analysis of robotic applications based on ROS. This library has been built during the past two years but is far from being completed. We are indeed constantly adding new models to our library to include freshly available nodes for a given ROS distribution that we want to use in our existing applications and for those we are currently developing. Also, a new implementation of a node is appended whenever a new computer board is evaluated.

Existing components might also be upgraded with new properties to allow further analysis that are not supported by our approach yet: for instance we intend to add signal flows for each component on a critical path of an application, and to profile reaction times onto the hardware target it might be deployed onto.

Our intention is to share our library and allow multiple users to extend it according to their needs. A link will be advertised on our web pages to allow public access to our models.

Regarding future work, we have already begun to extend our library to ROS2, the new version of ROS where the communication mechanism has evolved to use DDS. Other short term perspectives also include automatic code generation for ROS from our AADL models, by developing an extension of the RAMSES (Refinement of AADL Models for Synthesis of Embedded Systems) [7] automatic code generation tool[1].

References

[1] Ehsan M. Ahmad and Hessam Sarjoughian. 2019. A Behavior Annex For AADL Using The DEVS Formalism. In *2019 Spring Simulation Conference (SpringSim)*. 1–12. https://doi.org/10.23919/SpringSim.2019.8732894

[2] G. Bardaro and M. Matteucci. 2017. Using AADL to Model and Develop ROS-Based Robotic Application. In *2017 First IEEE International Conference on Robotic Computing (IRC)*. 204–207. https://doi.org/10.1109/IRC.2017.59

[3] G. Bardaro, A. Semprebon, A. Chiatti, and M. Matteucci. 2019. From Models to Software Through Automatic Transformations: An AADL to ROS End-to-End Toolchain. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 580–585. https://doi.org/10.1109/IRC.2019.00118

[4] Geoffrey Biggs, Kiyoshi Fujiwara, and Keiju Anada. 2014. Modelling and Analysis of a Redundant Mobile Robot Architecture Using AADL. In *Proceedings of the 4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots* (Bergamo, Italy).

[5] Dominique Blouin and Etienne Borde. 2020. *AADL: A Language to Specify the Architecture of Cyber-Physical Systems.* Springer International Publishing, Cham, 209–258. https://doi.org/10.1007/978-3-030-43946-0_8

[6] Dipak Bore, Amit Rana, Nilima Kolhare, and Ulhas Shinde. 2019. Automated Guided Vehicle Using Robot Operating Systems. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. 819–822. https://doi.org/10.1109/ICOEI.2019.8862716

[7] Fabien Cadoret, Etienne Borde, Sébastien Gardoll, and Laurent Pautet. 2012. Design Patterns for Rule-Based Refinement of Safety Critical Embedded Systems Models. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. 67–76. https://doi.org/10.1109/ICECCS20050.2012.6299202

[8] J. Cano, A. Bordallo, V. Nagarajan, S. Ramamoorthy, and S. Vijayakumar. 2016. Automatic configuration of ROS applications for near-optimal performance. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2217–2223. https://doi.org/10.1109/IROS.2016.7759347

[9] Fagner de Assis Moura Pimentel and Plinio Thomaz Aquino-Jr. 2019. Performance Evaluation of ROS Local Trajectory Planning Algorithms to Social Navigation. In *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*.

[10] Peter Feiler and David Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley Professional.

[11] P. H. Feiler, B. A. Lewis, and S. Vestal. 2006. *The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems.* 1206–1211.

[12] S. Gatesichapakorn, J. Takamatsu, and M. Ruchanurucks. 2019. ROS based Autonomous Mobile Robot Navigation using 2D LiDAR and RGB-D Camera. In *2019 First International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP)*. 151–154. https://doi.org/10.1109/ICA-SYMP.2019.8645984

[13] Morten Larsen. 2016. *Modelling field robot software using AADL.* Electrical and Computer Engineering Technical report ECE-TR-25. Aarhus University.

[14] Alex Lotz, Arne Hamann, Ralph Lange, Christian Heinzemann, Jan Staschulat, Vincent Kesel, Dennis Stampfer, Matthias Lutz, and Christian Schlegel. 2016. Combining robotics component-based model-driven development with a model-based performance analysis. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 170–176.

[15] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. 2013. FTG+PM: An Integrated Framework for Investigating Model Transformation Chains. In *SDL 2013: Model-Driven Dependability Engineering*, Ferhat Khendek, Maria Toeroe, Abdelouahed Gherbi, and Rick Reed (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–202.

[16] Open Robotics. 2022. *ROS - Robot Operating System.* Retrieved July 12, 2022 from https://www.ros.org/

[17] Eric Senn. 2022. ROS communications profiling for bus load analysis from AADL. In *ERTS 2022, 11th European Congress on Embedded Real Time Systems* (Toulouse, France).

[18] Eric Senn and Lucie Bourdon. 2021. Introducing CPU load Analysis from AADL Models for ROS applications : a use case. In *FDL 2021, Forum on specification & Design Languages* (Antibes, France).

[19] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. 2004. Cheddar : a Flexible Real Time Scheduling Framework. *ACM SIGAda Ada Letters* 24, 4 (December 2004).

[20] Carnegie Mellon University. 2022. *OSATE 2.11 Documentation.* Retrieved July 05, 2022 from https://osate.org/

[21] Nicolo Valigi. 2019. Concurrency in ROS1 and ROS2. In *ROSCon 2019, ROS developers CONference* (Macau / China).

[1] https://mem4csd.telecom-paristech.fr/blog/index.php/ramses/