AADL for robotics: a general approach for system architecture modeling and code generation

Gianluca Bardaro\textsuperscript{1} Andrea Semprebon\textsuperscript{1} Matteo Matteucci\textsuperscript{1}

\textsuperscript{1} Department of Electronics, Information and Bioengineering, Politecnico di Milano, Milano 20133, Italy

Abstract—Modern robotic systems are a combination of sophisticated software and hardware components and they offer complex functionalities. While (few) popular middlewares which promote component-level reusability, and assist development, exist, there are no established techniques or procedures that use a formal approach to robot system and architecture design yet. This work aims at the long term goal of model-based design and development of complex robot systems (and their software architectures), by surpassing current techniques based on personal expertise and best practices in favor of model-based approaches. Our contribution tackles the problem from the ground up by proposing a way to model ROS nodes, and robotic architectures in general, using the Architecture Analysis and Design Language (AADL) and by deriving, from these models, reusable templates to streamline the design of robotic systems and minimize their development time. The result is not bound to ROS, proven by outlining a model also for RoCK, and provides a general formal framework to describe complex robotic architectures suitable for automatic code generation and system verification.

Index Terms—robotics, middleware, model-based development, modeling, ROS

1 INTRODUCTION

Nowadays robotic applications have become more complex and popular, this is thanks to the advancement and popularity of component-based robotic middlewares. These middlewares (a survey here [1]), lead in popularity by the Robot Operating System (ROS) [2], free the developer from low-level component management (i.e., communication, synchronization, drivers) and create communities of users that share their work and help each other. Recently, this type of approach is going beyond the prototype applications and it is surging interest in those fields more close to traditional robotics, e.g., manipulators in the industrial sector [3] or teleoperated rovers in space. This new horizon requires an evolution in the development process of robotic applications, going from the current approach, mostly based on the expertise of the developer, to a more formal and model-based approach.

The effort in formalizing robot software development is as old as the introduction of the first middlewares, starting from Orocos [4] tentative to create a best-practice in robotics, followed later by the BRICS [5] project, and nowadays by the Rosin and RobMoSys European funded projects. To the best of our knowledge, BRICS was the first European effort aimed to structure and formalize the robot development process itself and to provide tools, models, and functional libraries, to help accelerating this process significantly. While the results of the project were remarkable and are used as a reference for this work, the provided tools, specifically the BRIDE [6] development environment, are not particularly popular and scarcely adopted, especially within the ROS community, which is currently the most active. A possible reason of this resides in the fact the community was not ready for this type of approach; when working with prototype or academic applications, the "make-it-work" approach is applied instead of a formal one. From this perspective, BRICS was too focused on the architectural view, using a top-down analysis of the system, creating an abstract representation not grounded in the application. This approach works well for traditional software engineering, but it is ill-suited for robotics, where the characteristics of the physical platform drive the implementation of the application.

The aim of this work is to enhance current techniques based on the expertise of the developer, or best practices, with model-based methodologies, but, differently from similar works, it follows a bottom-up approach, capitalizing on the existing resources provided by robotic middlewares (infrastructures and communities) and building on top of them reusable models with a consolidated modeling language. In particular, in this paper, ROS is used as a reference middleware to build these models, since it is the most popular and less restrictive
of all, and the Architecture Analysis and Design Language (AADL) [7] is used as the modeling language, the reasons of this choice are explained in Section 2.

While this work focuses on ROS, because we want to use at least one middleware as a reference, it is not strictly about ROS, since the modeling approach is general enough to be adapted, with few changes, to other component-based middlewares. The paper is organized as it follows, Section 2 introduces AADL and explains the reason of its choice, Section 3 presents how to model components and how to create template for them, using ROS nodes as a reference. Next, following the hierarchical structure of a system, architectures are modeled and templates are derived (Section 4). Section 5 presents how to automatically generate ROS C++ code from the architecture model. Section 6 presents how it is possible to adapt the concepts introduced in this work to model other robotic middlewares. Lastly Section 7 discusses possible extensions and presents related works and Section 8 draws relevant conclusions.

2 Architecture Analysis and Design Language

The core philosophy of this work is to use, as solid basis, consolidated technologies to capitalize on their popularity and exploit their existing functionalities. Therefore the Architecture Analysis and Design Language (AADL) was select as modeling language. While UML 2.0 is, by far, the most popular modeling language in software engineering (as shown in this survey [8]), the choice fell on AADL because, while being only the second most adopted, its characteristic are more suited for robotics applications, and because solutions using UML already exist (e.g., [9], [10], [11]) and they have not resonate with the robotic community.

AADL is an approved and published SAE standard [12] and is designed for the specification, analysis, automated integration and code generation of real-time performance-critical (e.g., timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems. It provides tools to allow analysis of system designs (and system of systems) prior to development and supports a model-based, model-driven development approach throughout the system life cycle.

AADL captures the architecture of complex systems as models that provide well-defined analyzable semantics, structured representation of the system and clear runtime architecture. These descriptions include identifying and detailing software and hardware components and their interactions in the form of interface specifications and implementation outlining, everything organized into packages. Packages represent libraries of components specifications that can be used in multiple architecture models, they support information hiding, with public sections, for specification available to other packages, and private sections, to protect details of component implementations. AADL models also support a variety of properties for each component and property sets through which the designer can introduce new user-defined properties.

Central to an AADL model are component types and component implementation declarations, together they define a component classifier. In the component type declaration the category and interfaces (features, in AADL terminology) of the component are defined. Categories are grouped into: application software, used to model the software element of the system (e.g., processes, threads, data, etc.), execution platform, representing the hardware aspects (e.g., processors, memory, buses, etc.), composite, the system type used to integrate software and hardware component in a distinct unit, generic, the abstract type defining a runtime neutral component that can be refined into another category. In the component implementation declaration, for a specific component type, the internal structure is described by using already defined (or generic) component type and implementations. Moreover, properties, both already available in AADL and user-defined, are specified to fully define a component’s runtime characteristics.

AADL provides various tools to facilitate code and model re-usability. Component types and component type declarations are organized in packages, and existing packages can be imported to use already defined component classifiers when describing a new model. Similarly to object oriented programming languages, AADL provides components inheritance. When declaring a new type or type implementation it is possible to use the keyword extends to include all the features (for a component type) and subcomponents (for a component type declaration) of an existing definition. Moreover, using the keyword refined to, it is possible to specify more details of an existing component, for example it is possible to define a generic thread in the parent and then refine it in a thread with a specific dispatch policy in the children. These features of the language make it possible to create generic models to be later specialized for a specific software application or hardware platform.

AADL is a formal declarative language described by a context-free syntax. The textual representation and well-defined semantics are important advantages of the language, but AADL also provides a standard graphical representation which corresponds to most of the textual one. OSATE [13] is an Eclipse plugin that offers an IDE for AADL development and to generate automatically the graphical representation from the textual one or vice versa.

3 Modeling Components

This section presents how to model the basic ROS component (i.e., a node) using AADL, focusing on the graphical representation provided by the language, when possible, and, if necessary, including some AADL code fragments. First, a base ROS node used as a foundation for any other node is presented, then a generic ROS node is modeled in an effort to capture the main elements of ROS components (i.e., publishers
Fig. 1. Graphical representation of the base node. Multiple level of detail coexist in the figure: the whole node (external container), subcomponents (main thread, state machine and internal state and detailed implementation (function inside the main thread).

and subscribers). Nodes presented in this section show how it is possible to enhance ROS software development, by adding elements normally not found in a ROS node, but common in other robotic middlewares. Namely, the model includes an internal state machine associated with a precise node life cycle, a strict distinction between the callbacks and the distinction between framework and logic implementation.

3.1 Base ROS node

Since AADL supports inheritance between models, each node is modeled starting from a base one that contains the key elements of a ROS node. Figure 1 shows an overview of the model using the AADL graphical representation. As visible from the model, everything is encapsulated inside an AADL process, this is a complete correspondence with the actual implementation, since in ROS any component is associated with a process.

As stated before and as visible in the graphical representation, the model of the base node contains a state machine. Four different states are modeled here: init, running, closing and error. In AADL each transition has to be triggered by an event; an event data port on the process models any external signal to force the shutdown of the node (i.e., SIGINT signal) and various ports on the main thread trigger the transition from a state to another. For each existing element of the model (i.e., threads, subprograms, connections, ports, etc.) it is possible to define in which state they are active, and this defines a specific work-flow and a strict initialization procedure of the node.

An AADL thread models the main execution flow of the node, main_thread in the figure, which is the essential component of each ROS node. A background thread, with no specific frequency, models it and it is mapped on a series of method calls and it uses an asynchronous spinner. Inside this main thread, four subprograms represent the four phases of the life cycle of the node. The prepare subprogram is for initialization and it is active in the init state. The running state enables the spin subprogram, here the callbacks are managed and it models the ROS spin function. When an external or internal signal triggers the closing state, the tear_down subprogram is enabled. It manages the termination of the node. In case of faulty behaviors, the state of the system change to error and the corresponding subprogram, error_handler, is enabled. Each subprogram has two ports: one event port, to trigger the transition in the state machine, and one data access port, to access the internal parameters of the node. The internal parameters are modeled using an AADL data component and can be accessed by any element of the node.

Since the base node has a specific life cycle, the model includes a subprogram access used to identify a ROS service. When the node changes its state, it can call a service to an overseer node which registers the current state of each active node.

3.2 Generic ROS node

By extending the base node presented in the previous section, it is possible to create any ROS node. In practice, it is possible to add functionalities by adding AADL threads. They are used to model any internal function of the node (i.e., callback), this is not a direct mapping with the ROS implementation, since in a node an independent thread does not manage a single callback. However, this represents a good approximation of the internal functioning of a node and provides an useful
logical representation; moreover this gives to the designer and
the developer a clear view of the structure of the module. Lastly, using AADL inheritance described in Subsection 3.3, it is possible to implement a callback-level re-usability. In the
generic ROS node in Figure 2 we modeled three different type
of functions:

- **Callback.** This models a generic and simple callback, the thread has an inbound event data port, connected to the corresponding one of the process. Using AADL the thread is defined as aperiodic (port-based and supports queues) dispatched by the incoming data on the port. Since this callback has no output the data collected are processed and discarded or used to modify the internal state of the node.

- **Callback with publisher.** This is an extension of the previous function, a callback which include a publisher, a typical implementation in ROS when the developer needs to process messages and republish them with the same frequency. As before, there is an inbound event data port to trigger the callback, additionally there is an outbound data port to model the publisher. This port is connected to the corresponding one on the external process. A data access port is available to modify the internal state of the node, not connected to internal state in the figure to show how a port can be declared independently to the existence of a connection.

- **Publisher.** A publisher used to generate messages or to republish messages from a subscriber at a different frequency. A thread with a specific period models this function, implemented in ROS using a callback triggered by a timer. Similar to the previous example an outbound data port connected to the external process represents the publisher and an optional data access port provides access to the internal state of the node.

Interfaces between processes, topics in the ROS implementa-
tion, are modeled using ports. A subscribed topic is repre-
sented with an inbound event data port, this is to capture the fact that connections will carry information (ROS messages) and trigger a callback, on the other side a published topic is modeled with an outbound data port, since it carries only data.

### 3.3 Templates

The true strength of the proposed node model is revealed when exploiting the inheritance capabilities of AADL. Any element described until now is defined independently in a library and they can be combined to create templates for commonly used nodes with no specific implementation. The granularity of the inheritance goes from the single thread (i.e., a callback or a publisher) to an entire node, or even further (described in Section 4).

Templatization was indirectly introduced in the previous sections; indeed the base node is also the minimal template for any node, since it capture the main and essential features of ROS nodes. Moreover, the elements defined as part of the generic node (i.e., callbacks and publishers) can be considered as independent and not connected to the description of the node itself. These components are defined separately and can be used together to compose nodes, more complex element may exists, for example a callback triggered by two synchronized topics, or can be defined and then used independently. Following a hierarchical interpretation, templates of components combined together and extended with the correct properties and connections can be used to design templates of nodes.
In the code snippet, it is visible how the template inherits the base node description presented in Subsection 3.1, automatically including all the essential characteristics of a ROS node (i.e., parameters, main loop and state machine). The template also takes advantage of the already defined callback with publisher element, making the description even more compact, understandable and standardized.

The transition from a template to a model is then straightforward. The template defines a component type in AADL and it can be used directly in a model, while properties can be used to specify the template for the current application. In the following listing it is possible to see how the property Source_text can be used to specify the actual implementation the node is going to use to transform the messages received.

The most important advantage of using template is to separate the design of the nodes form the implementation of the algorithms. Designer can create models of nodes or templates by starting from the minimal model and adding all necessary elements. Later, problem expert write the necessary algorithm and embed them in the model using properties. These model are then converted automatically in C++-based ROS code (more details in Section 5). Through accurate wrapping, it is possible to combine different programming languages, removing the necessity for the final developer to learn how to use the ROS framework and easing the integration process of existing algorithms into ROS nodes.

So far everything is a model and there is no actual implementation, it follows there is no strict difference between a template and the final model of a node, indeed a complete
model can be used as a template for future extension and a sufficiently complete template can be converted into a working node. The only limitation is in the use of abstract elements as placeholders, these can only be used in templates.

4 Modeling Architectures

This section continues the hierarchical representation of a system by presenting how to model architectures by combining together models of components. First, a minimal example containing most of the key elements of a robot architecture is presented, and then a more complete model is shown, including how to push the templatization of the system to the architectural level.

4.1 Minimal architecture

In robotics, a system architecture has to take into account both software and hardware components; sensors and actuators are indeed integral part of the system and often it is necessary to interact with them almost directly. Other than the models already described in the previous sections, the AADL language provides tools to model hardware components; especially useful are device and bus, which can be used to model sensors, actuators and connections. Considering the specific case of the ROS architectures, one of their most important features is the distribution on multiple machines, this can be captured in AADL using hardware binding on a specific processor as a way to specify how a portion of the architecture will be deployed on a specific platform.

Figure 4 shows a minimal example of a ROS architecture modeled using AADL. In this example, it is possible to see how different technologies (e.g., communication protocols, middleware implementations, hardware solutions, etc.) can be modeled in the same architecture: the joystick is an hardware component, modeled using an AADL device, joy_node is an existing ROS node, existing only as an interface, joy_to_cmd is a custom node, modeled starting from a transform template, ROS_bus is the representation of the ROS communication protocol, as a virtual bus, and usb_bus is the USB connection to the joystick, modeled using a physical bus.

At this level of detail only few elements are visible in the graphical representation, those relevant form an architectural point of view. As described in previous sections, nodes or templates for nodes are represented by processes and they expose as an interface only the external ports (i.e., topics) or accesses (i.e., services). Since there is no need, at this level of detail, to define the internal description of a node, even nodes from existing packages can be used in the architecture description, it is enough to define the necessary ports to model their correct interface; this procedure can be automatic if the documentation of the node is sufficiently detailed. Even abstract components can be used as placeholders for nodes, for example at the beginning of the design process when not all the interfaces are completely defined.

Hardware component are modeled directly in the architecture using devices, it is possible to specify various properties; from the frequency and type of communication to the actual type of data exchanged. AADL bus can be used to discern between physical connections (i.e., an USB connection or a serial port) and ROS topics by defining a physical bus and binding it to a specific connection, as shown in Figure 4.

Software components and hardware interfaces (i.e., devices) are all connected using AADL connections, they are defined in the container component, a system in this case, therefore they are independent from the definition of the subcomponent. Ports support multiple connections at the same time and they can model the publish/subscribe paradigm of ROS, if necessary; for different type of connections, it is possible to specify, using a property already available in AADL, the cardinality of the communication. In ROS, topics are identified by their
Fig. 5. Graphical representation of a hypothetical full robot architecture, where some of the elements are based on previous templates, while other are interfaces to be refined.

name, but in AADL each connection needs a unique identifier, therefore, to specify in the model the name of the topic, it is possible to define a new property for the connection. Following, we present a snippet of AADL code showing how the new property is defined and how it can be used when modeling a topic.

```aadl
property set topic_properties is
    Name: aadlstring applies to (connection);
end topic_properties;
-----

system implementation container.tlk_lis
    subcomponents
        tlk: process talker.impl;
        lis: process listener.impl;
        connections
            chatter: port tlk.chat -> lis.chat;
        properties
            topic_properties::Name => "/chatter"
            applies to chatter;
end container.tlk_lis;
```

4.2 General architecture

As already mentioned in Section 3, AADL inheritance can be used to create reusable templates at the architectural level. Various levels of details can coexist in the same architecture template, for example it is possible to use a completely defined node connected with an abstract component or a semi-defined node template connected with a model representing an existing node, therefore defining only their interface. Generic devices with only interfaces can be used as placeholders and later refined by adding relevant details.

Figure 5 shows how to define a generic architecture for a simple autonomous robot. Since the scope of this example is to show how to describe an hypothetical architecture and not to present the best possible approach, the model itself is an essential one used to capture the key elements of a robot architecture.

To maintain a general approach, devices are defined based on a simple template containing only an outbound (for sensor) or inbound (for the actuator) port, any specific hardware feature will be modeled accordingly to the actual hardware platform. If necessary even a more general approach is possible by considering all sensors as a single abstract element with numerous outbound connections and all actuators as an abstract component with various inbound connections. For this template we selected some common hardware components: GPS, IMU and odometer for sensing and a control unit which receives control setpoints to be applied on the robot. The nodes acting as drivers for the sensors are modeled referencing existing ROS nodes. The name of the process is the name that the node will have at runtime, while the component type (i.e., nmea_navsat_driver nmea_serial_driver) reflects the actual name of the package and the ROS node. This nodes will be included automatically in the architecture when building the launch files during the automatic code generation.

This architecture supports two different ways to control the robot: manual, controlled directly by the user using a joystick, and autonomous, the robot is piloted by a controller. This characteristics is typical for an autonomous robot and it requires a multiplexer node to direct the correct input to the motor, in the presented model the output_selector node is based on a template that uses two topics as input and decides which one has to be republished based on its internal state (determined by a service). Excluding the drivers, all the nodes in this architecture are modeled starting from the base node model described in Subsection 3.1, as visible by the
always present close port and notify state service. In an actual application a designer could create templates even for some of these nodes, for example for the control node where the interfaces are defined and only algorithm changes.

This general architecture also gives an overview of the various possible connection modeled by AADL. Previous sections already presented topics as connection between data ports and event data port. Moreover, as in other middlewares, ROS provides a synchronous communication using remote function calls, called services; in AADL subprogram access is used to model this type of behavior. In Figure 5 it is possible to see how different processes in a system are connected together by a client/server relationship using remote procedure call modeled as subprogram accesses, in particular how the node managing the state machine interacts with the multiplexer. More in detail, both the subprogram providing the service and the one using it are managed by their own thread, which has a subprogram as a subcomponent; subprogram access connections create a directed relationship between the user and the provider of the service. This architecture also provides a possible way to model ROS tf. In ROS, tf is a centralized container for all the reference frames of the system, therefore, an AADL data component at system level models it. Each node that want to access tf can do so by using a data access port, this is a bidirectional communication that can be used to update the frames (i.e., publish a new reference frame) or collect information (i.e., subscribe to a frame broadcaster).

This template includes also an high level state machine implemented inside a ROS node, this process can call a ROS service on the output_selector node to notify it of any change in state. Since ROS lacks an architectural view of the entire system, there is no straightforward way to implement a system-wide state machine, moreover the unreliability of ROS messages and services makes it impossible to implement a quick-reacting one, for instance, to switch from a running state to an emergency state in case of a fault. While AADL can be used to model state machines (as shown in Section 3), it is also able to integrate models from different languages, and in this case it is better to use a dedicated language to describe state machines, since they can become quite complex according to the functionalities of the robots.

5 Automatic code generation

The pipeline that goes from the an AADL model to the corresponding ROS C++ code is divided into two main stages. The first one is a model to model transformation from AADL to XML done extending Ocarina [14] (more on that in Subsection 5.1), while the second phase involves a code generator written from scratch in Python that parses the XML and generates the C++ ROS middleware code (explained in Subsection 5.2).

5.1 From AADL to XML

The first model to model translation has no correlation with any specific middleware; therefore, in a future development, this model to model transformation could be used as solid basis to build code generators for possibly any desired robotics middleware. Instead of building a complex AADL parser, which would have required the writing of a grammar for the entire language, our work leverages on Ocarina, a stand-alone AADL model processor written in Ada. Ocarina is used primarily in the project TASTE [15] [16], an open-source tool-chain for embedded software development, used and maintained by ESA (European Space Agency), and it is distributed under GPLv3 plus runtime exception [17]. Our AADL backend, having its foundation on Ocarina, has been written entirely in ADA. This opens to application domains where ADA characteristics are appreciated if not even mandatory such as space-oriented applications, where the entire toolchain needs to be certified with respect to some strict constraints.

Our model to model transformation goes from AADL to XML, the choice fell on XML being the dominant standard for interoperability and because it has dozens of parsers implemented in almost every programming language. Thanks to this, in possible future implementations, any preferred language could be chosen to write a software (e.g. a code generator) starting from the XML output of our backend without major difficulties. This opportunity has been already exploited by us as shown in Subsection 5.2 by writing a code generator from XML to C++ entirely in Python.

The parsing starting point is the identification of the main system, talking in AADL jargon. A system represents a physical robotic system or platform in the real world. In case multiple systems are designed and developed at the same time, we allow the possibility to provide no main system to the backend and let it scan all the files, looking for all system entries, and generate an XML file for each of them. This step, called system identification, launches a recursive procedure that visits all the system components and subcomponents, their features, properties, and connections. Each call to this procedure analyzes one element: for instance, each component is characterized by a name chosen by the developer, the type (e.g. publisher or subscriber process), a namespace, and a belonging category (useful to distinguish different AADL subcomponents). All these definitions are read and translated as for the example provide here related to the general properties:

AADL:

```plaintext
package example_node
public
system container
end container;

system implementation container.tlk_lis
end container.tlk_lis;
end example_node;
```
Resulting XML:

```xml
<system>
  <type>container_tlk_lis</type>
  <category>system</category>
  <namespace>example_node</namespace>
</system>
```

After reading the general properties, features, always speaking in AADL terms, associated with each element are analyzed; for each of them, in addition to the name, the type of port, the direction, and the correlated set of AADL properties are parsed and added to the XML file. Properties can be assigned not only to a feature, but also to the component itself; an example could be the source codes of application dependent functions or the frequency for a publisher thread. All these information are parsed and added to the XML file to be available to the next toolchain steps.

As for features and components, also connections have properties: for each connection information such as the source and destination port (along with their parents), port type, kind and category are extracted and provided, along with their properties if available. As said until now, each component can have features, connections, and its own properties, but it can also be made of subcomponents: the visiting procedure is recursively called passing each time, as argument, the various subcomponents to analyze the entire tree of dependencies.

At the end of this process, a well-formed XML file is produced together with a JSON file mapping each XML tag to its corresponding name: this is necessary because a name sensitive toolchain is not robust and desirable. In this way, any change occurred in this very first phase is automatically reflected along the toolchain.

### 5.2 From XML to ROS C++

Recalling what written in Subsection 5.1, the first toolchain phase is independent from the robotic middleware chosen for the final implementation; it can also be used as foundation to built a code generator to produce compilable code. The result of this is detailed in the following paragraph, describing the implementation of a code generator for ROS C++. Of course this is strictly related to the ROS middleware [18] so, in this work, the code generator has specifically been designed to support it.

XML is well supported by almost every programming language, so our decision to chose Python is not the only viable option, but it was driven by the easiness and flexibility of this specific language. First point for Python is that it is multi-platform and, secondly, the community is fresh and active, providing well tested and supported libraries (such as the XML parser used, namely `lxml` [19]).

The XML representation is a model itself, just like the corresponding AADL one, so the process follows a path similar to the one described for the AADL to XML backend; after some setup steps, alike the reading of the XML tag mapping to make the entire process name insensitive, code generation can start. In our model, each ROS node contained in a system is represented by a process; thus, the code generator goes through these processes and creates a different and distinct ROS node, with all the corresponding files, for each of them.

Each implementation of a ROS node derives from a class called `ROSNode` which implements the main functionalities for the node. Some of them are simply too ROS dependent to be expressed also in the AADL, but others can be represented and managed inside the AADL model itself. In particular, three significant functions are relevant to analyze:

- `prepare`: it contains all the operations needed to the node setup. For instance, it is in this function that a subscriber sets parameters such as the topic and the callback.
- `tearDown`: it is called right before a node is shut down.
- `errorHandling`: it works like an exception manager, called each time something does not work as expected.

These three functions are specified in the main thread of each AADL process and are also part of the ROS node: some properties can be linked in the model, like application dependent code which needs to be included also in the actual implementation. The parser identifies these three functions inside the main thread of each process and creates an instance for each of them; without them the parser stops his work and skips the node translation due to the lacks of mandatory features. Some other characteristics are extracted from the main thread, such as the runtime node name and the class name: all of them are used to built the basic component of a ROS node.

On top of this foundation, it is possible to add some blocks representing known patterns like the ones described in Subsection 3.1. They are modeled inside the AADL like `thread`: each of them is recognized by its namespace and name. When a known thread is found, its characteristics are added to the basic ROS node that will manage it. An AADL thread is composed, in C++, of variables, methods, parameters and instructions: for each of them the code generator creates an object, a sort of handler that manages all the various aspect. Take as example the creation of a publisher node publishing a message of type `std_msgs::String`: this type needs also the import of a library, `std_msgs/String.h`. All these major and minor details are managed by the specific class: each variable knows what it should import or not, the same for each method or parameter.

The gather a clearer view of the entire process, the pseudo code below can help:

```plaintext
1: Initialization step, reading JSON file
2: Read and parse the XML model
3: processes := all process element in the XML
4: for i = proc...processes do
5:   main_thread := locate the main thread
6:   error := main_thread elaboration, namely look for prepare, tearDown and errorHandling
```
The only unexplained thing is what happens during the second for loop, the one iterating over threads. Inside it the pattern is recognized and an instance of a specific class to handle it is created. Inside this class all the mandatory and optional methods and parameters are defined: for instance, recalling the example of the publisher node with a message, the name of the topic is specified in the AADL thread and read by the code generator only inside the specific class handler. It is important to note that no specialization of a node is possible without the skeleton of the node itself: that is why if an error is found in the main thread the entire node is skipped, while if an error is present inside the thread description only this specific thread is canceled, but the rest of the node with all the others specializations are maintained.

As also visible in the pseudo-code, at the end of every process elaboration, the actual code is generated and saved.

6 Non ROS Component Middlewares

In Section 1 we stated that the models presented in this work are flexible enough to be adapted to other middlewares with the appropriate modifications, but we mostly focused on ROS as target for our code generator. This choice was necessary, since the development of a complete toolchain from the model to the code requires a considerable amount of time and energy, and a reasonable knowledge of the target middleware; we could not face to handle other middleware code generators at the current time. However, this section outlines how the concepts presented so far could be adapted to model components of the Robot Construction Kit [20]  [21] (RoCK) middleware.

Figure 6a shows a graphical representation of two RoCK components implementing a basic producer/consumer pattern. In RoCK, components are defined as tasks, and each task has a series of ports that it can use to communicate with other components or to define the internal parameters of the task itself. Communication between components is direct and defined before runtime. Inside each task, the functionalities of the component are implemented in a library. Figure 6b shows a schematic description of a component, containing all the possible inbound and outbound ports of a task and the internal state machine.

The mapping between the RoCK components and the model is quite straightforward. Each task can be modeled using a process, in the same way a ROS node is modeled as a process, while connections, and their respective ports, are mapped directly on AADL connections and ports. The same data component used in ROS to model node parameters can be used in RoCK to map the parametrization of tasks. RoCK already separates the logic implementation from the framework implementation, this makes the inclusion of logic in the model using a subprogram even more straightforward with respect to the ROS components. Finally, Section 3 already shows how AADL can be used to model the internal state machine of a component, for a RoCK task it is enough to define the correct transition and include them in the model.

In synthesis, component based middlewares are all based on the concept of reusable component connected using a specific communication protocol, this similarities are enough to make the transition from one to the other simple, at least at the model level. The real complexity stands in the code generation, that needs to be implemented almost from scratch for each different middleware, although part of the process can be reused as shown in Subsection 5.1.

7 Discussion and Related Work

Modeling the nodes or the architecture of a robotic application using AADL lays the foundations for system analysis, model based development and runtime management. By using AADL to model the system and by following its formalism it is possible to exploit its already existing tools to analyze, for instance, the general correctness of the model, the latency of the system, error propagation (i.e., what happens to the controller if a sensor stops working) and, by detailing hardware components, resource utilization and management. From a more ROS specific point of view by having an architectural model it is possible to analyze beforehand properties normally not available until the end of the development cycle. Few examples are: a) topic compatibility and redundancy, when messages are replicated and provided under different forms instead of reusing the same topic; b) functions concentration, when a single node provides multiple unrelated functionalities; c) excessive distribution, the opposite problem where a simple task is decomposed in different nodes.

Starting from the model of the system, coupled with the use of a middleware, it is also possible to automatize most of the coding, as already presented in Section 5. It is also possible to extend the code generation even further, depending of the level of detail of the model, to the point that the developer can completely ignore the framework implementation and only focus on implementing the actual algorithms. Additionally, the code generation can be extended to support different ROS-based projects (e.g., ROS industrial) or different robotic middlewares (e.g., RoCK) with the appropriate model adaptation.
In order to simplify the inclusion of existing ROS nodes in the architecture, we are currently working on automatically generate AADL models from existing nodes exploiting the tools provided by ROS or the information obtained from the ROS wiki. The last possible step is to create a single toolchain composed by the model design and the automatic code generation and embed it in a IDE (i.e., Eclipse or RoboWare) giving developers a complete suite for robot software development.

Having the detailed model of the entire system available before execution make the runtime management easier, too. It is possible to select which subsections of the system are active in specific situation, for example when running a simulation or when working with a real robot, or to develop a more complex system and then deploy only a subset of the nodes on different platform. In ROS, specifically, the model of the architecture can be used to automatically generate launch files or to selectively run nodes.

As already mentioned in Section 1, model-based approaches are not new in robotics, however they are still scarcely developed and rarely adopted. For instance consider the main result of the already cited BRICS project: the Hyperflex toolkit. Hyperflex [9] uses a software product line approach and exploits models to symbolically represent variation points and variants in robot architectures and to automatically generate the configuration files for various software frameworks (i.e., OROCOS, ROS). While this toolkit offers a general approach, independent from the underlying middleware, it is useful only for configuration since it is limited to the architectural level, and it assumes that components are already available.

Moving at the component-level, some middleware exits which embraced the model-based approach such as SmartSoft [22], a service-oriented component-based approach for robotics software based on communication patterns. The development of components in SmartSoft is based on a DSL, derived from UML, which is used as a starting point to automatically generate various artifacts which lead to the complete components. The downside of this is the strong tie with the underlying middleware; indeed, with these models it is only possible to implement SmartSoft components, while a good modeling approach should be as independent as possible from the actual components implementation. Other middlewares follow the same strategy, like C-Forge [23] and RoCK, where languages with various levels of formality are used to describe components and facilitate automatic code generation. While this is quite useful for developers, as for the work presented in this paper, the model should be a starting point for something more than just code generation. This is one of the reasons of selecting AADL, i.e., to use it for further system analysis and design proofing.

Regarding the use of AADL in robotics, there are only few examples. In the context of the EROCOS project [24] funded by ESA, AADL is used indirectly to develop robotic application, since it is the backbone of the TASTE toolchain selected by the project. While this is a promising effort, the special requirement of space robotics and the tight connection with TASTE, make this approach unsuitable for general robotics. In [25], AADL is used a posteriori to model the architecture of an existing robot wheelchair and it is used to estimate the global latency on the control system in order to guarantee safety constraints. This is a proof of how AADL-based analysis can be used to enhance robot development and how code generation is just the first step of a more articulated solution.

8 CONCLUSION

This work presents how a model based approach to robot software development could enhance and compensate current techniques mostly based on the experience of the developer, expanding and capitalizing on existing methodologies and
tools. In particular it presents a way to use AADL as a modeling language to model a robotic system starting from the structure of the single component to its overall architecture. While the use of a model based approach is already an advancement given the current development practice in robotics, even more useful is to extend reusability outside the existing boundaries, this is the reason why templates have been introduced. They represent a way to reuse not only algorithms and implementations, but also useful designs. Thanks to the flexibility of AADL, models and templates are described with the same syntax and rules; the same concept can be used both at component and architectural level.

Given its popularity in the robotic community, ROS is used as a reference to build the models described in this work, but their structure is general enough to be adapted to other middlewares or future development of ROS itself, like ROS 2.0. Moreover, Section 5 introduces the first steps of automatic code generation from the model to ROS, showing how it is actually possible to use the model to simplify and speed up the development process by reducing the time spent implementing the framework code.

The long term aim of this work is to provide fully automatic and complete ROS, and eventually other middlewares, code generation starting from the AADL models, and to create libraries of component and architecture templates by analyzing common practices in robotic. This would allow to exploit as many AADL analysis and verification capabilities as possible and adapt them to ROS robotic applications too.

REFERENCES

Gianluca Bardaro is a third year PhD student at Politecnico di Milano, where he also attained a bachelor and master degree in computer engineering. Currently, he works with the Department of Electronics, Information and Bio-engineering and specifically in the Artificial Intelligence and Robotics Laboratory (AIRLab). Starting form his master thesis he worked on robotics, mostly on autonomous off-road vehicles, with a particular focus on software development and architectural aspect. His PhD work revolves around how to enhance robotic development by using model-based techniques, this, combined with code generation and system analysis, aims to streamline and formalize the development of robotic applications.
Andrea Semprebon is a Master Student at Politecnico di Milano, for his Master Thesis he has worked on model to model translation of systems described with the Architecture Analysis and Design Language (AADL) into an XML based description. He has also developed an automated ROS C++ code generator for AADL models based on the XML intermediate description. His research interests are in robotics, machine learning and computer vision applied to any field of robotics, from small robots to autonomous vehicles. He has participated to the ASP program, a school of excellence founded by Politecnico di Milano and Politecnico di Torino, it aims to provide society with high-profile graduates combining in-depth disciplinary knowledge with interdisciplinary skills, needed to work in a truly multidisciplinary environment.

Matteo Matteucci Laurea degree 1999, M.S. 2002, Ph.D. 2003 is Associate Professor at Politecnico di Milano. Divided between Robotics and Algorithms for Machine Learning, in 2002 he received a Master of Science in Knowledge Discovery and Data Mining at Carnegie Mellon University (Pittsburgh, PA), and in 2003 a Ph.D. in Computer Engineering and Automation at Politecnico di Milano. He is actually working in both Robotics and Machine Learning, mainly applying, in a practical way, techniques for adaptation and learning to autonomous robotics systems. He is part of the Technical Committee of AI and Robotics conferences, and reviewer for international journals on Robotics and Computational Intelligence. He got interested in model based robotics since his PhD studies and particularly in publish/subscribe middlewares for robotics. He has participated to the developed of a publish/subscribe middleware for the Politecnico di Milano RoboCup team, and to the design of R2P, an open source hardware and software framework for robot prototyping which exploits the publish/subscribe paradigm in embedded robotics applications development.