

# Modeling Variability in Self-Adapting Robotic Systems

Davide Brugali

*DIGIP, University of Bergamo*

*viale Marconi 5, Dalmine, Italy*

---

## Abstract

Autonomous robots operating in everyday environments, such as hospitals, private houses, and public roads, are context-aware self-adaptive systems, i.e. they exploit knowledge about their resources and the environment to trigger runtime adaptation, so that they exhibit a behavior adequate to the current context. For these systems, context-aware self-adaptation requires to design the robot control application as a dynamically reconfigurable software architecture and to specify the adaptation logic for reconfiguring its variable aspects (e.g. the modules that implement various obstacle detection algorithms or control different distance sensors) according to specific criteria (e.g. enhancing robustness against variable illumination conditions). Despite self-adaptation is an intrinsic capability of autonomous robots, ad-hoc approaches are used in practice to design reconfigurable robot architectures. In order to enhance system maintainability, the control logic and the adaptation logic should be loosely coupled. For this purpose, the adaptation logic should be defined against an explicit representation of software variability in the robot control architecture. In this paper we propose a modeling approach, which consists in explicitly representing robot software variability with the *MARTE::ARM-Variability* metamodel, which has been designed as an extension of the UML MARTE profile. We evaluate the applicability of the proposed approach by exemplifying the software architecture design of a robot navigation framework and by analyzing the support provided by the ROS infrastructure for runtime reconfiguration of its variable aspects.

*Keywords:* robot architectures, software variability, ROS

---

## 1. Introduction

Autonomous robots are versatile machines that act deliberately in order to fulfill their missions [1] in everyday environments, such as hospitals, private houses, and public roads. They are situated agents and robot situatedness refers to existing in a complex, dynamic, and unstructured environment that strongly affects the robot behaviour. Situatedness implies that the robot is aware of its own internal state (e.g. resource availability) as well as of its temporal and spatial interactions with the environment thanks to its sensors and actuators. Sensing, actuating, and control capabilities may be subject to hardware failures, computational overload, and changes in the environmental conditions. For this reason, autonomous robots exploit knowledge about their resources and the environment, which all together constitute the system's context [2], to trigger runtime adaptation so that they exhibit a behavior adequate to the current context.

Robot capabilities are typically implemented as concurrent activities, which execute feedback control loops with real-time constraints and continuously adapt the behaviour of the robot so that it can maintain a safe, robust and efficient interaction with the surrounding environments. Concurrent activities are typically implemented either as loosely-coupled executable components that interact through a middleware infrastructure, or as tightly-coupled cooperating threads that share the resources of an executable component [3].

For these systems, context-aware self-adaptation consists of being able to dynamically reconfigure the software architecture [4, 5] by activating/deactivating components, changing their connections, and replacing functionality in order to exploit at best the robot resources in every operational condition.

For example, a mobile robot for logistics at hospitals can navigate autonomously for transporting blood samples, medicine, bed covers, and food. The robot should be able to complete a navigation task even if the ambient illumination becomes suddenly scarce. In this case, the camera cannot detect

30 obstacles and the robot activates a laser scanner in its place. Switching between two sensors requires the activation/deactivation of the software components that interface with them and the replacement of the obstacle detection algorithm.

Dynamic reconfiguration of component-based systems is an emerging topic 35 in robotics and ad-hoc approaches are mostly used in practice [6] [7] [8]. Typically, the design of the robot control system encodes the adaptation logic in the behavior of individual components, which imposes co-evolution constraints on different levels of abstraction and across components [9].

In order to enhance system maintainability, the control logic and the adap- 40 tation logic should be loosely coupled.

This means that revising the adaptation logic (e.g. which sensor data to use in different environmental conditions) does not require a change to the control logic, i.e. how robot functionality are implemented. Similarly, revising the control logic (e.g. how the navigation task is decomposed in a sequence 45 of control activities) does not affect the implementation of the reconfiguration mechanisms, i.e. how to switch between alternative implementations of the same functionality.

For this purpose, self-adaptive systems are conveniently structured as a two-layer feedback-control loop [10], where the *Managed Subsystem* comprises the 50 control logic that provides the system functionality (e.g. robot navigation) and the *Managing Subsystem* comprises the adaptation logic that deals with one or more concerns (e.g. improving robot robustness, optimizing resource usage, etc.). This clear separation between the two layers requires a threefold approach.

The software architecture of the *Managed Subsystem* (i.e. the component- 55 based system that implements the robot capability) should explicitly model software variability in terms of variant features [11]. For example, a variant feature is the module that implement the localization algorithm and that can be replaced with compatible modules implementing different algorithms.

The *Managed Subsystem* should be developed on top of a runtime infras- 60 tructure (e.g ROS [12], SmartSoft [13], Orocos [14], Yarp [15]) that supports

various reconfiguration mechanisms (e.g. dynamic libraries and plugins) [16] for the implementation of variant features.

The reconfiguration logic of the *Managing Subsystem* should be specified against an abstract interface of the *Managed Subsystem* [17], i.e. its variant  
65 architectural features.

The aim of this paper is to present a variability Meta-Model that abstracts the reconfiguration mechanisms supported by robotics middlewares. It allows to annotate the software architecture of the managed subsystem with the appropriate reconfiguration mechanism for each variant feature. In this way, it represents  
70 the interface between the *Managing Subsystem* and the *Managed Subsystem*.

The proposed meta-model, called *MARTE Autonomous Robot Modeling - Variability (MARTE::ARM-Variability)*, has been defined as an extension of the Unified Modeling Language (UML) [18] profile for the Modeling and Analysis of Real Time Embedded Systems (MARTE) [19], which adds capability to UML for  
75 specification and analysis of non functional requirements in real time embedded systems, such as performance, schedulability, reliability, safety, etc.

In a previous paper [16] we have analyzed the software reconfiguration mechanisms supported by ROS and we have exemplified their exploitation for the design of a reconfigurable robot navigation system. In this paper, we show how  
80 the architecture of the navigation system should be annotated with the proposed *MARTE::ARM-Variability* metamodel.

The remainder of this paper is organized as follows. Subsection 1.1 presents the research questions and the methodology followed in this study to answer them. Section 1.2 motivates the adoption of the Unified Modeling Language  
85 (UML) [18] for the proposed approach and provides background information on its extension mechanisms with a running example of UML MARTE annotations. Section 2 discusses related work on architectural models for runtime adaptation. Section 3 analyses the architectural requirements for modeling the variability in self-adaptive robot control systems and the reconfiguration mech-  
90 anisms supported by popular robotics middlewares. Section 4 illustrates the proposed *MARTE::ARM-Variability* profile. Section 5 presents a case study of

a robot navigation framework to evaluate the proposed approach and illustrates the threats to its validity. Finally, Section 6 presents some concluding remarks and directions for future work.

### 95 1.1. Methodology

To address the earlier defined issues on the proper separation of Managing and Managed Subsystems in self-adapting robotic systems, the objective of this study was to design a variability Meta-Model for annotating the variant features of a robot software architecture according to the reconfiguration mechanisms  
100 supported by robotics middlewares.

According to the ACM SIGSOFT Empirical Standards [20] we follow a methodology (*Engineering Research*) that consists in proposing and evaluating a technological artifact (the variability Meta-Model). As later described in Section 5, the evaluation consists in designing an *Exploratory Case Study* in  
105 Robot Navigation and using a state of the practice standard framework, i.e. ROS, as *Benchmark* of the proposed Meta-Model.

To fulfill the stated objective, the following research questions were defined.

- **RQ1:** What is the granularity of the variant features in typical robot software architectures?
- 110 • **RQ2:** What are the commonalities and differences in the reconfiguration mechanisms supported by robotics middlewares?

In order to answer research question RQ1, we have analyzed some well known robot control architectures (i.e. 3T Architecture [21], ClaraTy [22] and LAAS Architecture [23]). The result of this analysis is documented in Section 3.1.

115 In order to answer research question RQ2, we have analyzed some of the most popular robotic middlewares (i.e. Orocos [14], Yarp [15], and SmartSoft [13]) following the classification schema for component-based robotic frameworks introduced in [24]. The result of this analysis is documented in Section 3.2.

The answers to the two research questions have allowed us to design the  
120 variability Meta-Model documented in Section 4, where architectural features

are modeled at various granularity levels (i.e. component, activity, functionality, property, connectors) and are associated to various reconfiguration mechanisms (e.g. YARP plugins, SmartSoft communication patterns, Orocos Services, ...).

The proposed approach has been validated with a case study, where the architectural features of a robot navigation system are modeled with the proposed  
125 variability Meta-Model (see Section 5.2) and their implementation is exemplified with a robotic middleware (i.e. ROS) that has not been considered for answering research question RQ2 (see Section 5.3).

### 1.2. Background

130 Modeling languages are classified in two broad categories: general-purpose (e.g. UML [18], AADL [25]) and domain-specific (e.g. Matlab Simulink [26] for control systems).

The strength of UML is its extension mechanism called *Profile* that allows adaptation and customization of the general-purpose UML notation by adding  
135 ad-hoc semantic and constraints and introducing terminology that are specific to a particular domain, platform, or method. A profile is a collection of domain-specific tags, called stereotypes, that can be applied to the graphical elements of a UML diagram. Tags are used to annotate a software model with information that are relevant in a particular domain. In particular, OMG has developed the  
140 Modeling and Analysis of Real-Time Embedded Systems (MARTE) [19] profile, which focuses on performance and schedulability analysis.

The UML extension mechanism allows the seamless collaboration of experts with different background. The software engineer uses the standard UML notation for modeling the high level software architecture and the design of individual  
145 modules. The domain expert (e.g. the control engineer) uses specific profiles for modeling different concerns of a complex software system, such as its dynamic behavior. For this purpose, UML profiles have a formal semantic [27], which allows the automatic conversion of UML models into domain-specific models, such as Matlab Simulink [28] or Petri Nets [29]. A robotic example of using  
150 MARTE for schedulability and performance analysis can be found in [30].

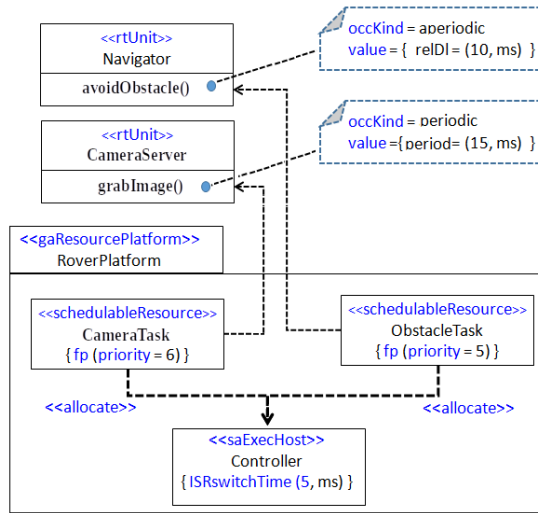


Figure 1: An example of architecture annotation with *UML MARTE* stereotypes.

In order to exemplify the concept of model annotation, Figure 1 shows a simplified UML model of the hardware computational resources and software components of a robot navigation system.

The HLAM (High-Level Application Modeling) sub-profile defines a set of stereotypes for real-time features. For example, the `<<rtUnit>>` stereotype is used to annotate two software components (i.e. *Navigator* and *CameraServer*), which perform concurrent activities. The annotations inside the dashed boxes specify that the former activity is aperiodic (relative deadline equal to 10 milliseconds) and the latter activity is periodic (period equal to 15 milliseconds). HLAM defines other stereotypes, such as `<<rtBehavior>>`, which specifies properties of the behaviors owned by an `RtUnit`.

The SRM (Software Resource Modeling) sub-profile is used in Figure 1 to specify the priority of concurrent tasks (i.e. *CameraTask* and *ObstacleTask*) with the `<<schedulableResource>>` stereotype.

The SAM (Schedulability Analysis Modeling) sub-profile defines the stereotypes to annotate the elements of the platform model (e.g. a CPU or other device). In particular, in Figure 1 the `<<SaExecHost>>` stereotype specifies the

ISRswitchTime which represents the worst context switching time.

Another sub-profile relevant for the approach presented in this paper is the  
170 MARTE GeneralComponentModel (GCM), which provides a common denomi-  
nator among various component models. It defines the «AssemblyConnector»  
stereotype for representing a specific interaction between communication ports.

## 2. Related work

A recent mapping study on variability modeling in software architectures [31]  
175 has analyzed a variety of approaches covering the period from 90's to nowadays.  
Most of them are based on the semi-formal UML, and some approaches present  
UML profiles for modeling architectural variability with generic stereotypes,  
such as «listValueVariant», «typeValueVariant», and «derivedValueVariant» in  
[32] and «variant» in Kobra [33]. As indicated in [31], one of the main limita-  
180 tions of these approaches is the lack of a proper support to bind variants into  
variation points at runtime.

A novel aspect of the approach presented in this paper is that it defines  
specific stereotypes for modeling variability in architectural elements of con-  
current and distributed component-based systems and these stereotypes can be  
185 associated to the reconfiguration mechanisms of the most popular robotics mid-  
dlewares. Another novel aspects of the proposed approach is that it is based on  
the UML MARTE profile, which formally describes the real-time and embed-  
ded features of a system. A significant benefit of this choice is the possibility  
to exploit and customize available modeling tools that support UML MARTE,  
190 such as MODELIO [34] and Eclipse Papyrus [35].

Architectural models for runtime adaptation have been studied extensively  
in the context of dynamic architectures (see the survey [36]) and a variety of  
Architecture Description Languages (ADLs) have been defined for modeling self  
adaptive systems (e.g. [37]), in particular for Component&Connector systems  
195 (see the survey [9]). Typically, these ADLs support architecture reconfiguration  
in terms of the addition and removal of components and in terms of the addition

and removal of connectors. They differ in their support for modeling e.g. the events that trigger the reconfiguration (i.e. the external environment that interacts with the application, or one of its internal functionality), the reconfiguration mechanisms (imperative vs. declarative), and the reconfiguration possibilities (a closed set of configurations defined at design time, or an extensible set of new configurations added after deployment). In *Fractal* [38], components can expose elements of their internal structure for introspection and may contain various controllers to add and remove subcomponents as well as connectors, and to configure component properties. Different than the *MARTE::ARM* meta-model, *Fractal* is targeted at Java-based system development and does not explicitly model concurrent activities.

The self-adaptive community has extensively used the MAPE-K loop [39] as the central mechanism to analyze, plan and trigger the adaptations required by a system. In particular, in [40] the authors show how MAPE-K loops for self-adaptation can be naturally specified in an abstract stateful language like Abstract State Machines. In the context of dynamic software product lines (e.g. [41] and [42]), architectural and component models have been defined to support runtime variability management. For example, the *MADAM* component model and middleware [43] for mobile computing uses component frameworks to describe a composition of component types. Variability is achieved by plugging in different component implementations whose externally observable behavior conforms to the type. Each component plugged into a component framework may be an atomic component, or a composite component built as a component framework itself. Different from *MADAM*, *MARTE::ARM* allows to model components as White-box Object Oriented Frameworks [44], which rely heavily on OO language features like inheritance and dynamic binding to achieve extensibility and reconfigurability.

The authors of [45] use a robotic navigation scenario to validate an algorithm for modifying the variability model of dynamic software product lines at runtime and check the feature constraints on the fly. The functional variability of the robot control system is represented by a Feature Model [46], a hierarchical dia-

gram that visually illustrates the features (e.g. different algorithms for obstacle avoidance) of a software application. This approach is complementary to the  
230 approach presented in this paper. As described in Section 4, *MARTE::ARM* is used to model the architectural variation points of a robot control system, while Feature Models are used to symbolically represent their admissible functional variants.

During recent years, several approaches have been proposed that exploit  
235 Model-Driven Engineering (MDE) MDE technologies for the development of autonomous robotic systems [47]. They support architecture design by automating some complex and error-prone tasks, such as editing diagrams, reverse-engineering legacy systems and, more importantly, validating assumptions made by system engineers and control engineers about system properties such as  
240 schedulability, performance, responsiveness, and fail-safe behavior. Most of these approaches are based on robotic-specific languages (see [48] for a survey).

The approach presented in [49] and [50] specifically focuses on runtime variability management in robot control systems. For this purpose it proposes to use two different Domain Specific Languages: the *Task Coordination Language*  
245 (SmartTCL) for modeling variability in service operation as hierarchical task decompositions, and the *Variability Modeling Language* (VML) [51] for modeling variability in quality of service. By modeling the two concerns separately, the approach enables variability management by two orthogonal mechanisms: (i) sequencing the robot's actions to handle variability in operation, and (ii)  
250 optimizing the non-functional properties for variability in quality.

A similar approach is presented in [52], where the variability in service operation is modeled as extended Behavior Trees (BT) [53], where variant sub-trees are explicitly represented and selected at runtime according to context information and non functional requirements.

255 Different from the *MARTE::ARM* profile proposed in this paper, SmartTCL, VML, and the extended BTs do not model the variability of the robot software architecture. As described in Section 1.2, we believe that the UML profile extension mechanism is particularly beneficial for coherently modeling different

concerns (e.g. architecture, dynamic behavior, non functional properties) in a  
260 multidisciplinary domain such as robotics.

The RobMosys EU project [54] has developed a meta-model and toolchain  
for component-based robotics software development based on the concept of  
composable software models. In particular, *MROS* (Metacontrol for ROS2 sys-  
tems) [55] is derived from the RobMosys and uses a combination of domain  
265 specific languages to model functional variability in ROS-based robotic applica-  
tions. Different from *MROS*, the *MARTE::ARM* meta-model allows to model  
architectural variability at different granularity levels, from coarse-grained com-  
ponents to fine-grained functionalities.

### 3. Requirements and technologies for self-adaptive robotic systems

270 In this section we first analyze the granularity of software building blocks  
typically found in robot control architectures and the requirements for their  
runtime reconfiguration, and then the mechanisms provided by popular robotics  
middlewares for implementing and reconfiguring those building blocks.

This analysis is at the basis of the definition of the *MARTE::ARM-Variability*  
275 profile, which allows to annotate the architectural model of a robotic system  
with stereotypes that correspond to the various types of architectural variability  
and that can be mapped to specific reconfiguration mechanisms in robotics  
middlewares.

#### 3.1. Component granularity in robot software architectures

280 A robot control architecture is typically organized as a hierarchy of control  
layers. Higher layers include components that request services from the  
components of the lower layers and that receive feedback data and events from  
them.

Well known examples of hierarchical control architectures, such as the 3T  
285 Architecture [21], ClaraTy [22], and the LAAS Architecture [23], distinguish  
three main layers. The (lower) functional layer includes all of the basic built-in

robot action and perception capabilities. These processing functions and control loops (e.g., image processing, obstacle avoidance, motion control, etc.) are encapsulated into software components providing services, which can be activated by the decisional level. The functional layer is made of a network of communicating components. For example in the LAAS architecture communicate asynchronously through a non-blocking client-server protocol. The (upper) decisional layer includes the capability of producing the task plan for a robot mission. The (intermediate) execution control layer controls the execution of the functional services according to the task plan and enforces safety constraints. Our study focuses on the functional and execution control layers, which include two types of variant components.

Components that interface with sensors and actuators implement stateless functionalities. Actuator components receive desired velocities or positions, implement various stacks of nested control loops that regulate the wheel motors of a rover or the joints of a manipulator arm, and write motion commands on a communication port, such as USB, CANBUS, or Ethercat. Similarly, sensor components read data on a communication port periodically, perform some basic computation, such as data filtering and type conversion, and make data available to other components through a provided interface. This type of component may implement a simple state machine that distinguishes between an initial setup and calibration phase and an operational phase. Runtime reconfiguration consists of adapting the value of some attributes, such as the sensor scan rate or the parameters of the motor control stack or in activating or deactivating the component.

Components that provide robotic capabilities typically implement stateful functionalities. For example, the *navigator* component of the ClaraTy framework aggregates an *action\_selector* module and a *locomotor* module. Two interchangeable variants of the *action\_selector* use either a grid-based representation of the world or a vector-based one [22]. A stateful component should not be terminated and restarted at runtime unless the current state can be stored and restored and even migrated to a different component. For this reason, it is con-

venient to manage the variability of a stateful component within the component itself. Replacing the implementation of a stateful functionality does not require  
320 to deactivate and replace the entire component, but to reconfigure its internal structure.

Architectural reconfiguration of robot control systems occurs at different levels of granularity. Let's consider the situation where ambient illumination becomes suddenly scarce and prevents the use of RGB cameras to detect ob-  
325 stacles. In this case, the robot swaps between the RGB camera and another distance sensor, such as a laser range finder or an infrared depth camera. In this scenario, reconfiguration consists of (1) activating and deactivating the executable components that interface to the robot devices and periodically acquire sensory data; (2) redirecting the data flow and control flow between components  
330 (e.g. camera images require additional intermediate processing, such as noise reduction); (3) replacing the algorithm that processes the measures (e.g. for updating the obstacle map); and (4) adapting the value of some global parameters, such as the robot's maximum speed, in order to take into account differences in sensors performance.

335 Typically, these four types of architectural reconfiguration are interdependent. Replacing the software module that implements a functionality affects the control system response time, since different algorithms might have different computational times. As discussed in [11], the control system response time affects the robot's stopping distance and thus the robot maximum speed needs  
340 to be adapted accordingly.

### 3.2. *Runtime Reconfiguration in Robotics Middlewares*

Typically, robotics middlewares (i.e. Orocos [14], Yarp [15], and SmartSoft [13]) support runtime reconfiguration of software control architectures at various levels of granularity, as discussed in the previous section. In this section  
345 we identify their commonalities and differences. We illustrate them following the classification schema for component-based robotic frameworks introduced in [24], which distinguishes the following four design concerns (the four C) for

software components as defined in [56].

**Computation** is concerned with the execution of concurrent activities, which are encapsulated in software components at different levels of granularity [24]: *Sequential Components* encapsulate data structures and operations that implement specific processing algorithms. *Service Components* represent sequential execution threads of control. *Container Components* provide the runtime environment for sequential and service components. A *Component Assembly* encapsulate the other types of components in executable units.

**Communication** deals with the exchange of data [56] and in robot control systems can be imperative, as in the case of a command issued by one component to another, or reactive, as in the case of event notification [24].

**Configuration** is concerned with the runtime infrastructure that supports the dynamic reconfiguration of components at various levels of granularity.

**Coordination** deals with the state of the computation of the entire system at any moment in time in terms of the current internal state of each component and state transitions [57].

### 3.2.1. YARP

The YARP middleware [15] is a multiplatform and multiprotocol communication framework for robotic research. Available protocols in YARP are tcp, udp, multicast, shared memory, and video compression.

A YARP application consists of a set of component assemblies that are executable c++ programs. YARP provides several container components, such as the PolyDriver class for dynamically loading plugins, and the ResourceFinder class for parsing the configuration files and initializing component parameters. Concurrent activities are implemented as service components that specialize the RFModule class and override two abstract methods: `configure()` is executed at startup for loading command line parameters and initializing the communication ports; `updateModule()` periodically executes the control algorithm. The RFModule class also provides methods for dynamically start, interrupt, and terminate the execution of the periodic activity. YARP supports the defini-

tion of sequential components as plugins that implement a variety of software interfaces to hardware devices.

380 YARP RModules exchange data through communication ports that support both synchronous and asynchronous communication. Connections between ports are established dynamically at runtime by specifying the names of the ports, which are resolved by a central nameserver.

In YARP the coordination of concurrent activities and the runtime reconfig-  
385 uration policies are specified as Behavior Trees (BTs) [58]. A BT organizes tasks in a hierarchical tree structure. The leaf nodes of a BT are either Conditions nodes or Actions nodes. A BT engine continually evaluates the status of its nodes to find the actions to be executed or aborted.

YARP provides tools for managing the system configuration. Components  
390 are started using the yarprun service, which keeps a list of components it has spawned. The same service can be used to monitor and send termination signals to individual components.

### 3.2.2. *Smartsoft*

SmartSoft [13] is a service-oriented component-based framework that is char-  
395 acterized by a library of pre-defined communication and deployment patterns for components interconnection.

Similar to YARP, a SmartSoft assembly component is an executable c++  
program that encapsulates concurrent activities and communication ports. Smart-  
Soft container component is represented by the SmartACE::SmartComponent  
400 class, which needs to be instantiated in the main() function of each executable program and made accessible to concurrent activities and communication ports. SmartSoft does not clearly distinguish between service components and sequential components. Concurrent activities are implemented by extending the SmartACE::ManagedTask class and overriding the on\_execute() function, which is  
405 executed periodically.

SmartSoft defines several component interfaces, called communication patterns, with strictly defined interaction semantic. These include publisher/sub-

scriber (*push newest, push timed*) and client/server (*send, query, event*).

The SmartSoft framework provides the SmartTCL language for the coordi-  
410 nation of robotic tasks. With SmartTCL, tasks are defined hierarchically where  
high-level tasks are refined into lower level tasks, which can be recursively re-  
fined until a sufficient level of detail is reached such that the task can be mapped  
directly to service components. The Sequencer Component is the central coordi-  
415 nation and configuration entity, which orchestrates the system by executing  
the robotic behavior models defined with SmartTCL.

SmartSoft concurrent activities have predefined states and might have a  
set of parameters. The default states are Neutral, Alive, and Fatal, used to  
pause, resume, and interrupt the activity. Additional user-defined states can be  
added. The Sequencer Component can change the state of a component activ-  
420 ity at runtime using the SmartSoft's State deployment pattern and can directly  
set its parameters value using the SmartSoft's Parameter deployment pattern.  
Ports interconnection can be establish at startup time using the SmartMDSD  
toolchain or at runtime using the SmartSoft's Dynamic Wiring deployment pat-  
tern.

### 425 3.2.3. OROCOS

OROCOS [14] is one of the oldest open source framework in robotics, under  
development since 2001. The focus of OROCOS is to provide a hard real-time  
capable component framework, the so-called *Real-Time Toolkit* (RTT).

OROCOS does not distinguish between component assemblies and container  
430 components. Indeed, an application consists of a deployer console tool (the  
OCL::DeploymentComponent), which is responsible for instantiating, connect-  
ing, and executing service components. The user can enter commands to ex-  
plore, debug, and interact with them at runtime.

OROCOS does not clearly distinguish between service components and se-  
435 quential components, which are implemented as subclasses of the RTT::TaskContext  
class by overloading five standard functions that are called when TaskContext's  
state changes: configureHook(), startHook(), updateHook(), stopHook(), and

cleanupHook(). Their implementation can vary from mere C/C++ functions over real-time program scripts to hierarchical state machines. The RTT::TaskContext  
440 class encapsulates a RTT::ExecutionEngine object, which is executed in a single thread. It guarantees thread-safe time determinism, processes incoming messages, and (a)periodically calls the updateHook() function.

Several instances of the RTT::TaskContext class can be deployed within an OCL::DeploymentComponent. As they share the same address space, they  
445 can communicate according to the caller/provider paradigm using pointers to other RTT::TaskContext objects for accessing public properties and operations. For data-flow communication the OCL::DeploymentComponent establish connections between ports of RTT::TaskContext objects as specified in a program script or XML file. A distributed application may consists in several  
450 OCL::DeploymentComponent running on networked machines. In this case, remote RTT::TaskContext objects communicate through the CORBA middleware.

OROCOS provides the *restricted Finite State Machine (rFSM)* language [59] to model coordination of robotic tasks. It has been designed as a minimal  
455 subset of UML 2 and Harel statecharts consisting of only three model elements: states, transitions and connectors. An rFSM execution engine can be embedded in RTT::TaskContext objects to implement the Coordinator component, which receives events from and sends commands to other RTT::TaskContext objects.

The rFSM language is used in OROCOS control applications to model also  
460 the configuration logic according to the *Coordinator-Configurator* pattern presented in [60], where the Coordinator remains in charge of commanding and reacting but a distinct Configurator component performs actions such as starting or stopping components, creating or destroying connections, invoking component services, or configuration of parameters.

#### 465 4. A Metamodel for Architectural Variability

In this section we propose a meta-model for autonomous robots that has been defined as an extension of the UML MARTE *Generic Component Model* profile [19]. The proposed profile, called *MARTE::ARM-Variability* (ARM stands for Autonomous Robot Modeling), is intended for annotating UML design models of robot software architectures with information about their *variation points*.  
470 In a previous paper [61] we have presented the *MARTE::ARM-Safety* subprofile for modeling robotic non-functional requirements, such as navigation safety.

A *Variation Point* is defined in [62] as a particular place in a software system where choices are made as to which variant feature to use. A *Variant Feature*  
475 is an abstraction for a set of related features (optional or mandatory) [63]. As there are several definitions of the term *Feature*, in the context of this paper we found particularly relevant the definition given in [64]: a feature represents a logical unit of behavior that is specified by a set of functional and quality requirements.

480 For example, the localization functionality is a *Variant Feature* of a mobile robot, and the Adaptive Montecarlo Localization (AMCL) algorithm is a possible *Feature*. The AMCL algorithm is implemented as a dynamic link library that is loaded at runtime by a *Localization* module, which represents a *Variation Point* in the robot control architecture.

485 Features can be implemented in a multitude of ways, using a range of different software entities, such as components, sets of classes, single classes or lines of code. Bosch and his colleagues have analyzed and classified a variety of runtime variability realization techniques systematically in [65, 62]. The systematic literature review in [66] classifies variability in different dimensions  
490 including artifacts like Architectures and Components.

The proposed *MARTE::ARM* profile defines a set of stereotypes, which represent different types of *Variant Feature* typically found in a robot software architecture. They map the four levels of component granularity in architectural reconfiguration identified in Section 3.1 to the reconfiguration mechanisms

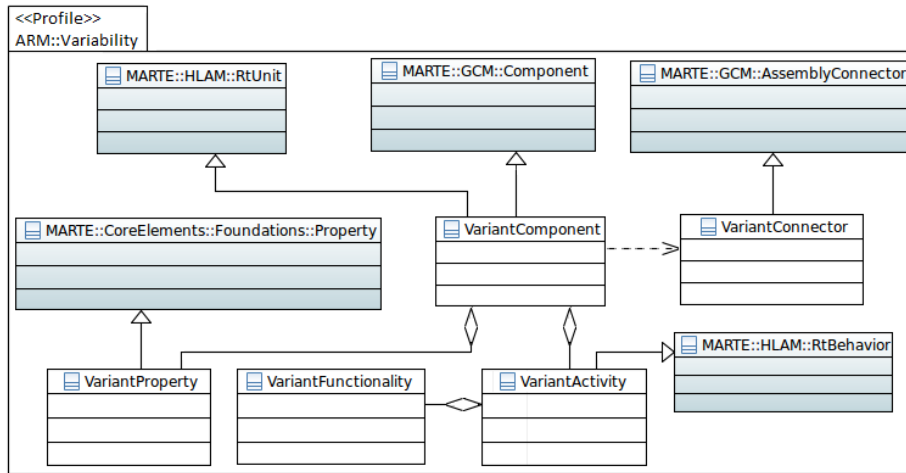


Figure 2: The *MARTE::ARM-Variability* profile for modeling variability.

495 of robotics middlewares analyzed in Section 3.2.

Figure 2 depicts the *MARTE::ARM-Variability* profile. The main entity is the *MARTE::ARM::VariantComponent*, which extends the stereotype *MARTE::GCM::Component* to support system design according to the Component&Connector architectural style. A component can implement a variant feature, since it can  
 500 be created or destroyed at runtime. It extends the *MARTE::HLAM::RtUnit* stereotype to represent an executable program that performs computation for providing a robot capability. A *VariantComponent* feature can be implemented as a component assembly in YARP and SmartSoft and as a *DeploymentComponent* in OROCOS.

505 Components may encapsulate multiple concurrent activities represented by the *MARTE::ARM::VariantActivity* stereotype, which extends the *MARTE::HLAM::RtBehavior* stereotype. Activities capture the dynamic of the component and are schedulable resources, triggered at runtime by the receipt of messages on a dedicated message queue or by timing events. Activities can be started and  
 510 stopped at runtime and access shared data structures encapsulated in the *VariantComponent* and annotated with the *MARTE::HLAM::PpUnit*. A *VariantActivity* feature can be implemented as a *RFModule* object in YARP.

Each activity encapsulates a functionality that takes a set of inputs, updates its state, and generates a set of outputs. A functionality is represented  
515 by the *MARTE::ARM::VariantFunctionality* stereotype and might be provided by a variety of interchangeable algorithms, which differ for some non functional properties, such as performance, completeness, robustness, etc. A *VariantFunctionality* feature can be implemented as a plugin in YARP. Since SmartSoft and OROCOS do not clearly distinguish between service and sequential components  
520 (see Section 3.2), SmartACE::ManagedTask objects and RTT::TaskContext objects implement variant features that are annotated with both the *VariantActivity* and the *VariantFunctionality* stereotypes.

Components and activities encapsulate attributes, whose value can be modified at runtime. For this purpose, the *MARTE::ARM::VariantProperty* stereotype  
525 extends *MARTE::CoreElements::Foundations::Property*. An OROCOS TaskContext may have any number of attributes of any type and they are accessible by external tasks through their interface. Similarly, SmartSoft allows to define parameters that can be used to configure a component at deployment and at run-time.

530 Components can be interconnected to each other to form systems by means of connectors, which allow components to exchange data and events according to various communication paradigms (e.g. publisher/subscriber and caller/provider). Connectors can be created, destroyed, and replaced at runtime. For this reason, they are modeled by the *MARTE::ARM::VariantConnector* stereotype that  
535 extends the *MARTE::GCM::AssemblyConnector* stereotype. YARP, Orocos, and SmartSoft provide mechanisms for establishing various types of connectors among components at runtime.

The *ARM-Variability* profile does not include stereotypes for annotating explicitly the variants of a feature (e.g. the various algorithms for obstacle  
540 avoidance or the algorithms for path planning). Indeed, we follow an orthogonal modeling approach [67]: the *ARM-Variability* profile is used to annotate the structural variability of a robot software architecture (i.e. the variation points), while the functional variability of the robotic domain is represented in a separate

Feature Model [46], which specifies dependencies and constraints among the  
545 features (e.g. the AMCL algorithms requires a laser range finder sensor). We  
have followed a similar approach for the development of the HyperFlex toolchain  
[68], which allows to connect the features of a Feature Model to the variation  
points of an architectural model. A runtime variability algorithm is presented  
in [45] that checks the constraints between features at runtime.

550 The *MARTE::ARM* profile has been defined starting from robotic-specific  
requirements but it does not capture any robotic-specific knowledge. This de-  
sign choice favors its adoption in conjunction with different robotics frameworks,  
middlewarees and development tools currently available. Typically, robotic do-  
main knowledge is captured by the plethora of robotic domain-specific languages  
555 (see [69] for a survey). For example, *RobotML* [70] allows users to design a robot  
control application by interconnecting components selected from a repository of  
domain specific entities, such as *MappingModule*, *LocalizationModule*, *StereoVi-  
sion*. These components can be annotated with the *MARTE::ARM* profile and  
implemented using a variety of robotic frameworks.

## 560 5. Evaluation

In Section 1 we stated that system maintainability of self-adaptive robot  
control systems can be enhanced by clearly separating the reconfiguration logic  
from the control logic of a robot control system. This separation requires (i)  
to explicitly model software variability in the architecture of the robot control  
565 system (the *Managed Subsystem*), (ii) to implement its variant features on top of  
a Middleware Infrastructure that supports various reconfiguration mechanisms,  
and (iii) to specify the reconfiguration logic of the reconfiguration manager (the  
*Managing Subsystem*) against an abstract interface of the *Managed Subsystem*.

In this section we evaluate the applicability of the proposed approach to the  
570 design of robot control architectures with a case study in robot navigation. We  
have conducted this experiment in two main phases as follows.

In the first phase, the variability of navigation systems is captured and the

reusable assets needed for developing specific systems are identified. For this purpose, in subsection 5.1 we analyze some well know libraries for robot navigation in order to identify commonalities in their software architectures and configuration requirements, and in Section 5.2 we define the architecture of a component framework that captures their recurrent variable features. This part of the experiments demonstrates that these variable features can be adequately annotated with the stereotypes of the *MARTE::ARM-Variability* profile.

In the second phase, the architectural model of the navigation component framework works as a basis for changes performed by the runtime reconfiguration middleware. Subsection 5.3 shows strengths and weaknesses of the ROS framework for the design and implementation of reconfigurable robot control systems. This part of the experiment demonstrates that the stereotypes of the *MARTE::ARM-Variability* profile, devised in Section 3.2 from the analysis of three popular robotics frameworks (i.e. Orocos [14], Yarp [15], and SmartSoft [13]), can be associated to variant feature implemented with the reconfiguration mechanisms provided by a different middleware, i.e. ROS.

We conclude this section with a discussion on the threats to validity of the proposed approach.

### 5.1. Navigation libraries

The *ROS Navigation Stack* contains the software resources needed to let a robot plan and follow a collision-free path in both known and unknown environments using the ROS middleware [12]. The main component is the *move\_base* Node (an executable component in ROS terminology), which encapsulates concurrent activities for (i) updating a local and a global map of the environment (called *local\_costmap* and *global\_costmap*) with sensory data, (ii) for obstacle avoidance and motion control (the *local\_planner*), (iii) for global planning obstacle-free paths (the *global\_planner*), and (iv) for executing *recovery\_behaviors*. The *local\_planner* can be configured with the *Trajectory Rollout* or *Dynamic Window* approaches to local robot navigation. Similarly, the *global\_planner* can be configured with several path planning algorithms,

such as  $A^*$ , *Dijkstra's*, and *Navigation Function*.

The *RobMosys Flexible Navigation Stack* is based on the SmartSoft [13] component-based framework. The obstacle avoidance and trajectory following functionality are implemented as a single activity of the *ObstacleAvoidance* component, which uses the Curvature Distance Lookup (CDL) algorithm to compute a collision-free path to a local goal. Similar to the ROS *local\_costmap*, the *Mapper* component implements the obstacle detection activity and the obstacle map data structure. The stack provides distinct executable components implementing the interface to the mobile platform, the map-based localization functionality, and the global planning functionality using a grid map of the environment.

The *Clarity Framework* [22] has been developed at NASA JPL in collaboration with other research institutions for controlling the large variety of mobile robots for Mars exploration. The framework includes the *Navigator* component, which receives the desired path as a sequence of waypoints, computes the trajectory to the next waypoint and keeps track of the progress of the robot as it progresses towards it. It encapsulates distinct modules for motor control (called *Locomotor*) and for obstacle avoidance (called *Action Selector*). A variety of implementations of the *Locomotor* are available for rovers with different kinematics structures. The *Action Selector* uses a variety of algorithms for computing the rover trajectory, including the *Traversability Analyzer*, the *Local Cost Function*, and the *Global Cost Function*. The obstacle map can be either a simple binary occupancy grid or a more complex grid with statistical evaluation of the terrain.

## 5.2. A component framework for robot navigation system

Figure 3 depicts the UML model of the navigation control system, which is part of a software product line (SPL) for mobile robotic applications [71] that has been initially designed in the context of the EU FP7-ICT BRICS project and further extended (e.g. a SPL for robot perception system [72]) by refactoring open source robotic libraries developed in a variety of research projects. The component framework consists of four components, which are annotated with

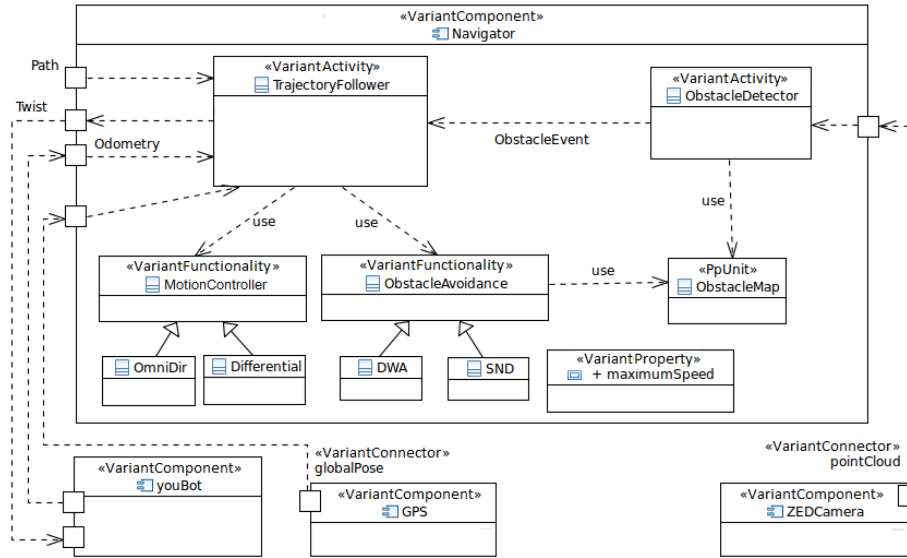


Figure 3: The annotated UML model of the Navigator component framework.

the *MARTE::ARM::VariantComponent* stereotype (see Section 4).

The *youBot* component implements the motion controller for the youBot  
 635 rover. It receives *Twist* messages that specify the desired longitudinal and  
 rotational speed of the robot and regulates the wheel motor speed accordingly. It  
 reads the motor encoders to estimate the wheels speed and generates *Odometry*  
 messages that indicate the current robot speed and position with regards to the  
 initial pose reference frame.

640 The *GPS* component interfaces with the Global Positioning System (GPS)  
 sensor device and generates *globalPose* messages periodically that indicate the  
 robot global pose. The *ZEDCamera* component interfaces a stereo camera and  
 generates *pointCloud* messages periodically.

The *Navigator* component encapsulates the control activities for driving the  
 645 robot towards a desired location along a precomputed path. They are annotated  
 with the *MARTE::ARM::VariantActivity* stereotype.

The *TrajectoryFollower* activity receives the odometric position (*Odome-*  
*try*) of the robot from the *youBot* component and generates velocity com-

mands (*Twist*) in order to follow the path received on the corresponding input port and to avoid unexpected obstacles. It encapsulates the functionality *MotionController* and *ObstacleAvoidance*, which are annotated with the *MARTE::ARM::VariantFunctionality* stereotype.

Two variants of the motion control algorithm are available: the *Omnidir* generates velocities for rovers with an omnidirectional kinematic, while the *Differential* is for rovers with a differential drive kinematic. Two algorithms for obstacle avoidance can be used alternatively, namely the Smooth Nearness-Diagram (SND) [73], which is the best choice for robot navigation in narrow environments, and the Dynamic Window Approach (DWA) [74], which is recommended for environments with wider passages and dynamic obstacles.

Unlike the ROS *local\_planner* and the RobMoSys *ObstacleAvoidance*, the *TrajectoryFollower* activity in Figure 3 separates the motion control functionality from the obstacle avoidance functionality and allows to replace them independently. The *Claraty Framework* adopts a similar solution.

The *ObstacleDetector* activity receives sensory measures (*pointCloud*) from a distance sensor and updates a local obstacle map. When an obstacle is detected, it generates an *ObstacleEvent* message to the *TrajectoryFollower*, which reacts by stopping the robot if the obstacle cannot be avoided or by adapting the path otherwise. The *Navigator* component specifies the attribute *maximumSpeed*, which indicates the maximum rover speed and is annotated with the *MARTE::ARM::VariantProperty* stereotype.

Unlike the ROS *move\_base* Node and similar to the *RobMosys Flexible Navigation Stack*, the *Navigator* component of Figure 3, receives the desired path from a separate executable component. This design solution enhances flexibility since the global planning functionality is a variant feature: the desired path could be computed using completely different types of environment map, e.g. a topological map, or generated on the fly when the robot follows a moving object (e.g. a person).

Figure 4 shows a portion of the Feature Model that specifies the variability in the navigation functionality of a mobile robot control system. It has been

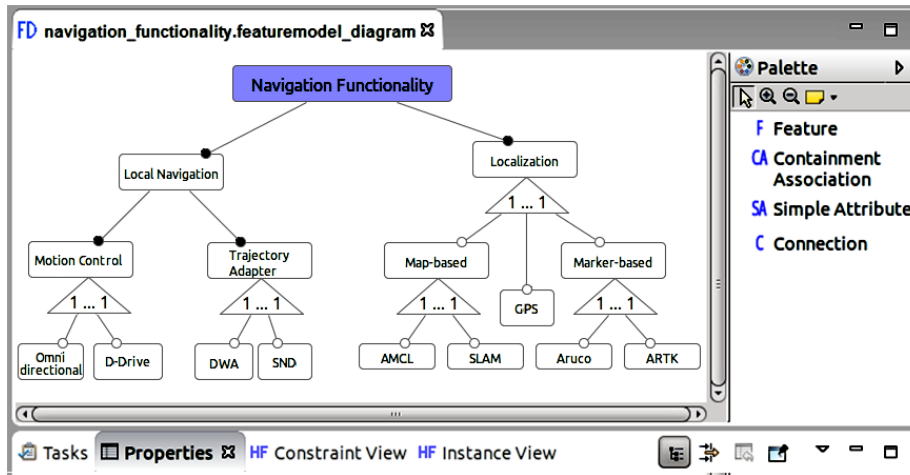


Figure 4: An example of Feature Model created with the HyperFlex graphical editor.

680 created with the HyperFlex graphical editor [68]. As an example, it defines a mutual exclusion constraint between the features corresponding to the motion control functionality, which are implemented by the *OmniDir* and *Differential* classes in the Navigator component in Figure 3. More specific *require* and *exclude* constraints among features can be specified with the HyperFlex editor, 685 such as *DWA requires Omnidirectional*. As described in [11], the Feature Model is used at deployment time for manual configuration and at runtime for dynamic reconfiguration of the robot control system.

### 5.3. Implementing the variability profile in ROS

In this section we illustrate how architectural elements annotated with the 690 *MARTE::ARM-Variability* profile can be implemented using the built-in mechanisms of the ROS infrastructure and for each architectural element we analyze the flexibility of the ROS mechanisms with regards to the possibility to modify the adaptation and control logic independently.

For this purpose, we use the Navigator component framework as a case study 695 (see Section 5.2).

The Robot Operating System (ROS) [12] is an open-source middleware for robots. It provides a message-based peer-to-peer communication infrastructure,

called *roscore*, supporting the integration of independently developed software components, called ROS nodes, that are organized into a graph.

700 In this case study, we assume that the reconfiguration of the Navigator component framework is managed by an external component, which is part of the *Managing Subsystem* (see Section 1), called *ConfigurationManager*.

Typically, the reconfiguration is triggered by changes in the operational context and consists in replacing features for the architectural variation points.

705 In [11] we have presented a case study where the reconfiguration logic of a mobile robot navigation system enforces safety requirements. A change in the operational context (e.g. scarce illumination) leads to the hardware and software reconfiguration of the robotic system, which reduces the system response time. In order to enforce safety requirements, the maximum value of the robot speed  
710 (i.e. a *VariantProperty*) is also reduced.

For example, the context change *scarce illumination* triggers a reconfiguration, which consists in deactivating the variant component *StereoCamera* and activating the variant component *LaserScanner*. Contextually, the variant connector between the detection sensor and the obstacle avoidance activity is re-  
715 configured from *2DPointCloud* to *3DPointCloud* and a different plugin provides the variant functionality for obstacle avoidance. The new software configuration determines an increase in the system response time due to the higher computational cost of the obstacle detection functionality. In order to enforce the navigation safety requirement, as documented in [61], the value of the variant  
720 property *MaximumRoverSpeed* is recalculated by solving a constraint satisfaction problem that expresses the relationships between the rover speed and the hardware and software configuration.

For defining the reconfiguration logic of the *ConfigurationManager*, the ROS framework provides a Python library, called SMACH, to build executable hierar-  
725 chical state machines. In SMACH a *state* corresponds to the system performing some activity implemented as a user-defined class, which executes actions and, at the end of the execution, returns an outcome, which is used to specify state transitions. SMACH actions can reconfigure the variant features of a control

architecture using the ROS mechanisms as described in the following.

730 **VariantComponent.** The *Navigator* is implemented as a ROS Node, i.e. a C++ main program that accesses the *roscore* through the *NodeHandle* class. The *Navigator* is started using two command-line tools called *rosvrun* and *rosvlaunch*. Their execution is terminated when the *ros::shutdown()* method is invoked. A custom SIGINT handler can be implemented to execute some final actions before shutting down when *Ctrl-C* is pressed. The *ConfigurationManager*, can start and stop the the *Navigator* component and every other component at runtime by calling *rosvrun* and *rosvlaunch* programmatically and sending shutdown requests.

New components can be added to the control architecture (e.g. a component that interfaces a new sensor used for detecting obstacles) without the need to modify the other components of the system. This is possible in ROS because components interact according to the publish/subscribe communication paradigm by exchanging typed messages on topic-specific communication channels: the components that provide data and services (the publishers) and the components that receive and use them (subscribers) have no visibility of their communicating peers [75].

Only the state machine of the *ConfigurationManager* needs to be updated, by adding a new state that calls the launch file of the new component. In order to allow dynamic replacement of an existing component with the new component, they should have compatible interfaces, i.e. they exchange messages of the same type (e.g. *PointCloud*) on the same communication channel (e.g. *laser\_scan*).

**VariantProperty.** The ROS framework offers the *Parameter Server*, a shared, multi-variate dictionary that is accessible via network APIs, to store and retrieve parameters at runtime. It is globally viewable so that nodes can easily inspect parameters values and modify them if necessary. It is mostly used for loading component parameters at start-up.

For dynamic reconfiguration of parameters at runtime without having to restart the node, ROS provides the *dynamic\_reconfigure* package (still exper-

imental) [76]. In order to manage the internal configuration, each component  
760 implements the *dynamic\_reconfigure()* callback, which is called at runtime when  
the component parameters are updated by the *ConfigurationManager*.

The domain of admissible values for a given *VariantProperty* are defined at  
deployment time, while the actual value is computed at runtime according to the  
reconfiguration logic defined in the state machine of the *ConfigurationManager*.

765 **VariantConnector.** Nodes exchange messages according to the publish/sub-  
scriber communication paradigm. Messages are typed data structures organized  
into topics. Each topic corresponds to a distinct message queue where nodes  
push and pull messages they are interested in.

Publisher and subscribers are typically defined and initialized in the main  
770 function of a node, which creates a spinner thread for processing incoming mes-  
sages and invoking the associated callback functions.

The topics names of publishers and subscribers can be stored as *Variant-*  
*Properties* that can be reconfigured at runtime. Publisher and subscribers need  
to be destroyed and reinitialized in the *dynamic\_reconfigure()* callback when  
775 their topic names are updated by the *ConfigurationManager*.

New connections among components can be established or removed by sim-  
ply updating the reconfiguration logic of the *ConfigurationManager*, which sets  
the same topic name for pairs of publishers and subscribers that have to ex-  
change messages.

780 **VariantActivity.** The ROS framework allows a node to create any number of  
threads that can publish messages, check for incoming messages, and execute  
the associated callbacks.

ROS allows users to create any number of callback queues, each one associ-  
ated to a specific set of message topics. Each callback queue can be associated  
785 to a dedicated thread that checks for incoming messages and executes the asso-  
ciated callbacks.

For this purpose, ROS provides the *ros::MultiThreadedSpinner* library and  
in particular the *ros::AsyncSpinner* class, which spins asynchronously and can

be started and stopped dynamically at runtime. Therefore, it can be used to  
790 implement instances of *VariantActivity*.

A boolean parameter indicating the state of a *VariantActivity* (i.e. *started* or *stopped*) can be stored as a *VariantProperty* that can be reconfigured at runtime. The *VariantActivity* is started or stopped by the *dynamic\_reconfigure()* callback when its state is updated by the *ConfigurationManager*.

795 **VariantFunctionality.** The ROS framework includes the *pluginlib* package that allows ROS nodes to open at runtime a shared library containing exported classes (called plugins) without having any prior awareness of the library or the header file containing the class definition. This mechanism is used for runtime reconfiguration of functional classes annotated with the *VariantFunctionality*  
800 stereotype.

In order to load a plugin class at runtime, the ROS framework requires to explicitly list the registered plugins in a *plugin description file*. This file specifies the classes corresponding to the admissible variants of a *VariantFunctionality*.

For loading a new variant of a functionality at runtime, the *dynamic\_reconfigure()*  
805 callback reads the name of the plugin class from a *VariantParameter*, which is updated by the *ConfigurationManager*.

### 5.3.1. New features in ROS 2

ROS 2 revises and extends the ROS development environment in several ways (see [77] for the new use cases). In particular, it prescribes some design  
810 patterns for structuring robot control systems.

While in ROS a Node is implemented as a C++ main program that wraps robotic software libraries, in ROS 2 Nodes are implemented as shared libraries that are loaded at runtime by a container process.

The container offers the ROS 2 service *load node*, which can be called pro-  
815 grammatically to dynamically load a Node specified by the passed package name and library name.

This mechanism allows to activate and deactivate Nodes at runtime and

improves the reconfiguration mechanism of the *VariantComponent* architectural elements.

820 Moreover, in ROS 2 Nodes are discoverable at runtime when their libraries are being loaded into a running process. This allows to establish appropriate connections among running Nodes. This mechanism can be exploited to improve the reconfiguration mechanism of *VariableConnector*.

#### 5.4. Threats to validity

825 One possible threat to validity is the ability of the proposed UML profile to adequately represent variability in robot software architectures. To mitigate this issue, we have analyzed the characteristics of some of the most popular robot architectures in order to identify reconfiguration approaches and requirements. It should be noted that these architectures have been used by international  
830 research teams to develop a variety of robotic systems. Moreover, we have recently conducted a thorough study on the drivers of variability in autonomous robots [78].

Nevertheless, we are aware that the design of robot software architectures is a vibrant research topic and further investigation of how variability is managed in  
835 application-specific robot architectures is needed to provide objective evidence about the generalizability of the proposed approach.

Another possible threat to validity is the ability of the proposed UML profile to represent the reconfiguration mechanisms of robotics middlewares. To mitigate this issue, we have analyzed the commonalities of three popular robotics  
840 middleware (i.e, YARP, Orocos, and SmartSoft) and we have then exemplified how to use the proposed UML profile with a different middleware (i.e. ROS). Nevertheless, we are aware that other robotic middlewares exist, such as OpenRTM-Aist [79]. In order to mitigate this potential limitation, we are planning to verify the applicability of the proposed approach to other robotic  
845 middlewares as future work.

The proposed UML profile has been formalized by only one researcher but we attempted to avoid any bias in its definition by leveraging the experience of

numerous researchers and collaborators in designing real world robotic applications, which has been documented in a number of editorial projects lead by the  
850 author on software engineering for robotics, such as [80], [81], [82], [83].

## 6. Conclusion and Future Work

In this paper we have presented the *MARTE::ARM-Variability* UML profile for modeling the variability of robotic component-based systems that rely on a middleware infrastructure for concurrent execution of control activities and for  
855 message-based distributed communication.

Our work is motivated by the goal of enhancing maintainability of context-aware self-adaptative robotic systems by clearly separating the adaptation logic from the control logic through a well defined interface that specifies the software variability of the robot control system.

860 *MARTE::ARM-Variability* has been designed as an extension of the UML MARTE profile. This choice seemed natural to us for modeling robot control systems, which are characterized by the execution of several concurrent activities with real-time constraints. Indeed, UML MARTE provides generic stereotypes for annotating architectural models with information related to periodic activities, their timing, priorities, and required computational resources.  
865 *MARTE::ARM-Variability* specializes these stereotypes to model concurrency at various granularity levels and to specify architectural variability as typically found in robot control systems. For this purpose, the requirements for *MARTE::ARM-Variability* has been devised from the analysis of well-known  
870 robot software architectures and of popular robotic middlewares.

To the best of our knowledge, existing approaches based on UML have been developed for generic self-adaptative systems, but they do not take into account robotic-specific requirements.

The choice of extending the UML MARTE profile has the benefit of favoring  
875 the seamless collaboration of experts focusing of different aspects of a complex robot control system: the software engineer uses the standard UML notation for

modeling the high level software architecture and the design of individual modules, the control engineer uses specific profiles for modeling the system dynamic behavior with real-time constraints, the robotic engineer uses *MARTE::ARM-*  
880 *Variability* to model the application variability, which typically depends on the robot task, the robot embodiment, and the operational environment [84].

Moreover, the UML MARTE profile is supported by a variety of Model-driven Engineering (MDE) tools, which allow to graphically model the robot software architecture and to automatically transform the architectural model in  
885 more convenient domain-specific models (e.g. for schedulability analysis, performance analysis, etc.). The work presented in this paper adopts an orthogonal modeling approach: *MARTE::ARM-Variability* is used to annotate the structural variability of a robot software architecture (i.e. the variation points), while the functional variability of the robotic domain is represented in a separate Fea-  
890 ture Model, which specifies dependencies and constraints among the features. Our previous work related to MDE tools has developed the HyperFlex toolchain [85] for linking Feature Models and architectural models. As a future work, we plan to extend HyperFlex to exploit the *MARTE::ARM-Variability* profile.

To demonstrate the applicability of the proposed approach, we have shown  
895 that *MARTE::ARM-Variability* can be used to annotate the architecture of a robot navigation component framework that generalizes the design of well known navigation libraries. We have also shown that the variability elements of the proposed *MARTE::ARM-Variability* profile are not explicitly supported by the built-in mechanisms of the ROS infrastructure, but they can be implemented by  
900 exploiting several ROS packages (e.g. *dynamic\_reconfigure*, *ros::MultiThreadedSpinner*, *pluginlib*) appropriately. This means that the implementation of variable architectural elements in self-reconfigurable robotic systems requires detailed knowledge of these ROS packages and is error-prone.

In our future work, we will address this limitation by extending the Hyper-  
905 Flex toolchain for the automatic generation of ROS source code from an architectural model of component-based robotic systems annotated with *MARTE::ARM-Variability*. Similarly, HyperFlex will support the automatic generation of

SMACH executable hierarchical state machines that implement the reconfiguration logic.

910 In the Navigation case study, we adopted the *Closed Dynamic Variability* approach [86], which supports the dynamic activation and deactivation of features that have been predefined in advance in the runtime variability design. This means that the architectural variation points and variants of the Navigator component framework are defined at design-time.

915 Nevertheless, the metamodel and runtime mechanisms presented in this paper naturally support also the *Open Dynamic Variability* approach [86], which allows the addition, removal, and modification of features dynamically, facilitating system reconfiguration in unforeseen scenarios in a controlled manner [87].

920 This is possible because both the Feature Model of Figure 4 and the architectural model of Figure 3 build on formal metamodels, which are machine-readable and can be processed automatically by a reconfiguration algorithm.

As an example, the *Marker-based* localization functionality in Figure 4 could be removed in case the camera is faulty and cannot be used even when the environmental conditions allow it. Contextually, the ZedCamera *VariantComponent* 925 and the connected *VariantConnector* would be removed from the UML model of Figure 3.

As future work, we plan to extend the navigation case study in order to demonstrate the structural reconfigurability of the navigation framework (i.e. 930 adding/removing features and updating the reconfiguration logic). This capability will help the robot learn reconfiguration strategies at runtime. In addition, another possible line of future work includes further validation activities using different robotic middlewares and different use cases and the performance evaluation of the reconfiguration mechanisms presented in this paper.

935 **References**

- [1] F. Ingrand, M. Ghallab, Deliberation for autonomous robots: A survey, *Artificial Intelligence* 247 (2017) 10–44, special Issue on AI and Robotics.
- [2] K. Mens, R. Capilla, H. Hartmann, T. Kropf, Modeling and managing context-aware systems’ variability, *IEEE Software* 34 (6) (2017) 58–63.
- 940 [3] D. Brugali, A. Brooks, A. Cowley, C. Côté, A. C. Domínguez-Brito, D. Létourneau, F. Michaud, C. Schlegel, Trends in component-based robotics, *Springer Tracts in Advanced Robotics* 30 (2007) 135 – 142.  
URL [https://www.scopus.com/inward/record.uri?eid=2-s2.0-34247254869&doi=10.1007%2f978-3-540-68951-5\\_8&partnerID=](https://www.scopus.com/inward/record.uri?eid=2-s2.0-34247254869&doi=10.1007%2f978-3-540-68951-5_8&partnerID=40&md5=da9b6ff1942b5965e9ff06a498b7463d)  
945 [40&md5=da9b6ff1942b5965e9ff06a498b7463d](https://www.scopus.com/inward/record.uri?eid=2-s2.0-34247254869&doi=10.1007%2f978-3-540-68951-5_8&partnerID=40&md5=da9b6ff1942b5965e9ff06a498b7463d)
- [4] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: *Future of Software Engineering (FOSE '07)*, 2007, pp. 259–268.
- [5] B. Selić, Specifying dynamic software system architectures, *Software and Systems Modeling* 20 (3) (2021) 595–605.
- 950 [6] Z. Yu, I. Warren, B. Macdonald, Dynamic reconfiguration for robot software, in: *2006 IEEE International Conference on Automation Science and Engineering*, 2006, pp. 292–297.
- [7] D. Sanjay, M. C. Chinnaiah, T. S. Savithri, P. R. Kumar, A survey of reconfigurable service robots, in: *2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS)*, 2016, pp. 1–4.  
955
- [8] A. K. Srivastava, S. Kumar, Dynamic reconfiguration of robot software component in real time distributed system using clustering techniques, *Procedia Computer Science* 125 (2018) 754–761.
- 960 [9] A. Butting, R. Heim, O. Kautz, J. O. Ringert, B. Rumpe, A. Wortmann, A classification of dynamic reconfiguration in component and connector

- architecture description, in: Proceedings of MODELS 2017 Satellite Event, Austin, TX, USA, September, 17, 2017., ACM/IEEE, 2017, pp. 10–16.
- [10] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K. Goeschka, On Patterns for Decentralized Control in Self-Adaptive Systems, in: Software Engineering for Self-Adaptive Systems II, Vol. 7475 of Lecture Notes in Computer Science, Springer, 2013, pp. 76–107.
- [11] D. Brugali, R. Capilla, R. Mirandola, C. Trubiani, Model-based development of qos-aware reconfigurable autonomous robotic systems, in: 2018 Second IEEE International Conference on Robotic Computing (IRC), IEEE, 2018, pp. 129–136.
- [12] S. Cousins, B. Gerkey, K. Conley, W. Garage, Sharing software with ros [ros topics], Robotics Automation Magazine, IEEE 17 (2) (2010) 12–14.
- [13] C. Schlegel, Communication patterns as key towards component interoperability, in: Software engineering for experimental robotics, Vol. 30 of Springer Tracts in Advanced Robotics, Springer, 2007, pp. 183–210.
- [14] H. Bruyninckx, Open robot control software: the orocos project, in: Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164), Vol. 3, 2001, pp. 2523–2528 vol.3.
- [15] G. Metta, P. Fitzpatrick, L. Natale, Yarp: Yet another robot platform, International Journal of Advanced Robotics Systems 3 (1) (2006) 43–48.
- [16] D. Brugali, Runtime reconfiguration of robot control systems: a ros-based case study, in: 2020 Fourth IEEE International Conference on Robotic Computing (IRC), 2020, pp. 256–262.
- [17] F. Steimann, P. Mayer, Patterns of interface-based programming., Journal of Object Technology 4 (2005) 75–94.
- [18] OMG, Unified modeling language, <http://www.uml.org/> (2022).

- [19] OMG, Marte, <http://www.omg.org/omgmarte/> (2022).
- [20] A. SIGSOFT, Empirical standards, <https://acmsigsoft.github.io/EmpiricalStandards/about/> (2023).  
990
- [21] R. P. Bonasso, D. Kortenkamp, D. P. Miller, M. Slack, Experiences with an architecture for intelligent, reactive agents, in: M. Wooldridge, J. P. Müller, M. Tambe (Eds.), *Intelligent Agents II Agent Theories, Architectures, and Languages*, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 187–  
995 202.
- [22] I. A. D. Nesnas, The claraty project: Coping with hardware and software heterogeneity, in: *Software Engineering for Experimental Robotics*, Springer, Berlin, Heidelberg, 2007, pp. 31–70.
- [23] R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand, An architecture  
1000 for autonomy, *The International Journal of Robotics Research* 17 (4) (1998) 315–337.
- [24] D. Brugali, A. Shakhimardanov, Component-based robotic engineering (part ii), *IEEE Robotics & Automation Magazine* 17 (1) (2010) 100–112.
- [25] P. H. Feiler, D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st Edition, Addison-Wesley Professional, 2012.  
1005
- [26] Simulink, simulation and model-based design, <http://it.mathworks.com/products/simulink/> (2015).
- [27] J. Tatibouët, A. Cuccuru, S. Gérard, F. Terrier, Formalizing execution semantics of uml profiles with fuml models, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), *Model-Driven Engineering Languages and Systems*, Springer International Publishing, Cham, 2014, pp. 133–148.  
1010

- 1015 [28] Y. Vanderperren, W. Dehaene, From uml/sysml to matlab/simulink: Current state and future perspectives, in: Proceedings of the Design Automation & Test in Europe Conference, Vol. 1, IEEE, 2006, pp. 1–1.
- [29] I. Bicchierai, G. Bucci, L. Carnevali, E. Vicario, Combining uml-marte and preemptive time petri nets: An industrial case study, *IEEE Transactions on Industrial Informatics* 9 (4) (2013) 1806–1818.
- 1020 [30] S. Demathieu, F. Thomas, C. Andr  , S. G  rard, F. Terrier, First experiments using the uml profile for marte., in: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA, IEEE Computer Society, 2008, pp. 50–57.
- 1025 [31] A. P. Allian, R. Capilla, E. Y. Nakagawa, Observations from variability modelling approaches at the architecture level, in: Software Engineering for Variability Intensive Systems - Foundations and Applications, Auerbach Publications Taylor & Francis, 2019, pp. 41–56.
- 1030 [32] A. Wichmann, R. Maschotta, F. Bedini, S. J  ger, A. Zimmermann, A uml profile for the specification of system architecture variants supporting design space exploration and optimization, in: MODELSWARD, 2017.
- [33] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. W  st, J. Zettel, Component-based Product Line Engineering with UML, Addison-Wesley, 2002.
- 1035 [34] Modelio, <https://www.modelio.org/> (2022).
- [35] Eclipse papyrus, <https://www.eclipse.org/papyrus/> (2022).
- [36] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger, A survey of self-management in dynamic software architecture specifications, in: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems, WOSS '04, ACM, New York, NY, USA, 2004, pp. 28–33.
- 1040

- [37] S.-W. Cheng, D. Garlan, Stitch: A language for architecture-based self-adaptation, *Journal of Systems and Software* 85 (12) (2012) 2860–2875.
- [38] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems, *Software: Practice and Experience* 36 (11-12) (2006) 1257–1284.
- [39] J. Kephart, D. Chess, The vision of autonomic computing, *Computer* 36 (1) (2003) 41–50.
- [40] P. Arcaini, E. Riccobene, P. Scandurra, Modeling and analyzing mape-k feedback loops for self-adaptation, in: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015, pp. 13–23.
- [41] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, M. Hinchey, An overview of dynamic software product line architectures and techniques: Observations from research and industry, *Journal of Systems and Software* 91 (2014) 3–23.
- [42] J. D. A. S. Eleuterio, C. M. F. Rubira, A comparative study of dynamic software product line solutions for building self-adaptive systems, *Tech. rep.* (2017).  
URL <http://www.ic.unicamp.br/~reltech/2017/17-05.pdf>
- [43] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjørven, Using architecture models for runtime adaptability, *IEEE Software* 23 (2) (2006) 62–70.
- [44] M. Fayad, D. C. Schmidt, Object-oriented application frameworks, *Communications of the ACM* 40 (10) (1997) 32–38.
- [45] A. Valdezate, R. Capilla, J. Crespo, R. Barber, Ruva: A runtime software variability algorithm, *IEEE Access* 10 (2022) 52525–52536.

- [46] K. Kang, Feature-oriented domain analysis (FODA) feasibility study, Tech. rep., DTIC Document (1990).
- 1070 [47] D. Brugali, Model-driven software engineering in robotics, *IEEE Robotics & Automation Magazine* 22 (3) (2015) 155–166.
- [48] A. Nordmann, N. Hochgeschwender, S. Wrede, A Survey on Domain-Specific Languages in Robotics, Vol. 8810 of *Lecture Notes in Computer Science*, Springer International Publishing, 2014, Ch. 17, pp. 195–206.  
1075 doi:10.1007/978-3-319-11900-7\\_17.
- [49] A. Lotz, J. F. Inglés-Romero, C. Vicente-Chicote, C. Schlegel, Managing run-time variability in robotics software by modeling functional and non-functional behavior, in: S. Nurcan, H. A. Proper, P. Soffer, J. Krogstie, R. Schmidt, T. Halpin, I. Bider (Eds.), *Enterprise, Business-Process and Information Systems Modeling*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 441–455.  
1080
- [50] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. Inglés-Romero, C. Vicente-Chicote, Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot, Vol. 57, 2013, pp. 85 – 98.
- 1085 [51] J. Inglés-Romero, A. Lotz, C. Vicente-Chicote, C. Schlegel, Dealing with run-time variability in service robotics: Towards a dsl for non-functional properties, in: *3rd International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob-12)*, 2012.
- [52] A. Romero-Garcés, R. Salles De Freitas, R. Marfil, C. Vicente-Chicote, J. Martínez, J. F. Inglés-Romero, A. Bandera, Qos metrics-in-the-loop for endowing runtime self-adaptation to robotic software architectures, *Multi-media Tools Appl.* 81 (3) (2022) 3603–3628.  
1090
- [53] M. Colledanchise, P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*, 1st Edition, Addison-Wesley Professional, 2017.

- 1095 [54] RobMoSys, Composable models and software for robotic systems, <https://robmosys.eu> (2022).
- [55] D. Bozhinoski, E. Aguado, M. Oviedo, C. Hernandez, R. Sanz, A. Wasowski, A modeling tool for reconfigurable skills in ros, in: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE),  
1100 IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 25–28.
- [56] M. Radestock, S. Eisenbach, Coordination in evolving systems, in: O. Spaniol, C. Linnhoff-Popien, B. Meyer (Eds.), Trends in Distributed Systems CORBA and Beyond, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 162–176.
- 1105 [57] G. A. Papadopoulos, F. Arbab, Control-driven coordination programming in shared dataspace (1997) 247–261.
- [58] M. Colledanchise, P. Ogren, Behavior Trees in Robotics and AI: An Introduction, CRC Press, 2018.
- [59] M. Klotzbücher, H. Bruyninckx, Coordinating robotic tasks and systems  
1110 with rfsm statecharts, Journal of Software Engineering for Robotics 1 (3) (2012) 28–56.
- [60] M. Klotzbücher, G. Biggs, H. Bruyninckx, Pure coordination using the coordinator-configurator pattern, CoRR abs1303.0066 (2013).  
URL <http://arxiv.org/abs/1303.0066>
- 1115 [61] D. Brugali, Modeling and analysis of safety requirements in robot navigation with an extension of uml marte, in: Proceedings of the 2018 IEEE Int. Conf. on Real-time Computing and Robotics, RCAR '18, IEEE, 2018, pp. 439–444.
- [62] M. Svahnberg, J. van Gurp, J. Bosch, A taxonomy of variability realization  
1120 techniques: Research articles, Software: Practice and Experience 35 (8) (2005) 705–754.

- [63] M. L. Griss, J. Favaro, M. d'Alessandro, Integrating feature modeling with the rseb, in: Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203), IEEE, 1998, pp. 76–85.
- 1125 [64] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [65] J. V. Gorp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, WICSA '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 45–.
- 1130 [66] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in software systems — a systematic literature review, IEEE Transactions on Software Engineering 40 (2014) 282–306.
- 1135 [67] N. Loughran, P. Sánchez, A. Garcia, L. Fuentes, Language support for managing variability in architectural models, in: C. Pautasso, É. Tanter (Eds.), Software Composition, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 36–51.
- [68] D. Brugali, M. Valota, Software variability composition and abstraction in robot control systems, in: Computational Science and Its Applications – ICCSA 2016, Springer International Publishing, Cham, 2016, pp. 358–373.
- 1140 [69] N. Arne, H. Nico, W. Dennis, W. Sebastian, A survey on domain-specific modeling and languages in robotics, Journal of Software Engineering for Robotics 7 (2016) 75–99.
- 1145 [70] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, M. Ziane, Robotml, a domain-specific language to design, simulate and deploy robotic applications, in: Simulation, Modeling, and Programming for Autonomous Robots, Springer, 2012, pp. 149–160.

- [71] D. Brugali, L. Gherardi, A. Biziak, A. Luzzana, A. Zakharov, A reuse-oriented development process for component-based robotic systems, in: Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR'12, Springer, Berlin, Heidelberg, 2012, pp. 361–374.  
1150
- [72] D. Brugali, N. Hochgeschwender, Software product line engineering for robotic perception systems, International Journal of Semantic Computing 12 (01) (2018) 89–107.  
1155
- [73] J. W. Durham, F. Bullo, Smooth nearness-diagram navigation, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 690–695.
- [74] D. Fox, W. Burgard, S. Thrun, The dynamic window approach to collision avoidance, IEEE Robotics Automation Magazine 4 (1) (1997) 23–33.  
1160
- [75] P. T. Eugster, P. A. Felber, R. Guerraoui, A. Kermarrec, The many faces of publish subscribe, ACM Computing Surveys 35 (2) (2003) 114–131.
- [76] OpenRobotics, The dynamic reconfigure package, [http://wiki.ros.org/dynamic\\_reconfigure](http://wiki.ros.org/dynamic_reconfigure) (2022).  
1165
- [77] OpenRobotics, Robot operating system 2.0 design, <http://design.ros2.org/> (2022).
- [78] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, T. Berger, Software variability in service robotics, to appear in Empirical Software Engineering (2022).  
1170
- [79] N. Ando, T. Suehiro, T. Kotoku, A software platform for component based rt-system development: Openrtm-aist, in: Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR '08, Springer-Verlag, Berlin, Heidelberg, 2008, p. 87–98.  
1175

- [80] D. Brugali, M. E. Fayad, G. Menga, R. Volz, Guest editorial on object-oriented methods for distributed control architectures, *IEEE Transactions on Robotics and Automation* 18 (4) (2002) 407–408.
- [81] D. Brugali, I. A. Nesnas, Guest editorial on software development and  
1180 integration in robotics, *International Journal of Advanced Robotic Systems* 3 (1) (2006) 1.
- [82] D. Brugali, *Software Engineering for Experimental Robotics*, Vol. 30 of Springer Tracts in Advanced Robotics, Springer Berlin Heidelberg, 2007.
- [83] D. Brugali, E. Prassler, Guest editorial on software engineering for robotics,  
1185 *IEEE Robotics & Automation Magazine* 16 (1) (2009) 9–15.
- [84] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Schillinger, P. Pelliccione, T. Berger, Variability modeling of service robots: Experiences and challenges, *VAMOS '19*, Association for Computing Machinery, New York, NY, USA, 2019.
- [85] D. Brugali, L. Gherardi, Hyperflex: A model driven toolchain for designing  
1190 and configuring software control systems for autonomous robots, *Studies in Computational Intelligence* 625 (2016) 509–534.
- [86] O. Aguayo, S. Sepúlveda, Variability management in dynamic software  
1195 product lines for self-adaptive systems. a systematic mapping, *Applied Sciences* 12 (20) (2022).
- [87] D. Brugali, R. Capilla, M. Hinchey, Dynamic variability meets robotics, *Computer* 48 (12) (2015) 94–97.