Model-driven Software Engineering in Robotics

Davide Brugali* *School of Engineering, Universita' Degli Studi di Bergamo, Dalmine, 24044 Italy

I. Introduction

A model is an abstract representation of a real system or phenomenon [1]. The idea of a model is to capture important properties of reality and to neglect irrelevant details. Which properties are relevant and which can be neglected depends on the purpose of creating a model. A model can make a particular system or phenomenon easier to understand, quantify, visualize, simulate, or predict.

Models and modeling are an essential part of every engineering endeavor. Using models to design complex systems (an edifice, an airplaine, a production plant) reduces the risk of making costly errors before undertaking the effort of its realization: models help us understand the various aspects of a complex problem and evaluate alternative solutions.

Two models of the same system or phenomenon may be essentially different depending on the properties they aim to capture. Specialized models are needed to visualize a bridge's structure or to evaluate an aircraft aerodynamic. Models might have different forms: graphical (i.e. the architecture of a building) or textual (i.e. the differential equations describing the effects of an earthquake on buildings)

Models are created by using modeling languages, i.e. artificial languages that defines symbols, keywords, and their semantic and syntactic rules. Mathematics, statistics, and logic are typical scientific and engineering modeling languages. Domain-specific languages (DSL), have been defined for a variety of technological domains, such as electrical circuits (i.e. the topological description language) [2], physical dynamic systems (i.e. the bond graph representation) [3], and manufacturing control (i.e. function blocks diagram) [4].

Software systems, which are often among the most complex engineering systems, can benefit greatly from using models and modeling techniques [5]. This is because a software model and the software system of which the model is an abstraction have the same nature. The transition from design to implementation is a sequence of refinement steps without semantic gaps. Backtracking is simple since there is no metal to bend and there are no circuits to solder.

The term Model-Driven Engineering (MDE) [6] is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations. A key premise behind MDE is that software models do not serve merely the purpose of documentation but that they enable (a) automatic assessment of system-level quality aspects, such as safety, correctness, and performance, way before any implementation, and (b) automatic generation of source code and system configuration parameters.

Manuscript received December 1, 2012; revised December 27, 2012. Corresponding author: D. Brugali (email: brugali@unibg.it).

During recent years, several approaches have been proposed in the literature that exploit MDE technologies in robotic software development. Most of these approaches use standard and general-purpose software modeling languages (e.g. UML [7]). Other approaches are based on robotic-specific languages. The aim of this tutorial is not to survey all of them, but to analyze the role and use of MDE technologies in robotic software engineering.

This tutorial is structured around three major phases of the software development process, namely architecture design and analysis (Section II), system implementation and integration (Section III), and system configuration (Section IV). Section V draws the relevant conclusions. The following subsection illustrates peculiar characteristics of software systems for autonomous robots.

A. Software Control Systems for Autonomous Robots

Autonomous robots are versatile machines equipped with a rich set of software functionalities, typically deployed on a distributed computing infrastructure with stringent resource constraints, for interacting purposefully and in real-time with an open-ended environment through sensors and actuators.

The use of models and modeling languages can have a high impact on the product life cycle of robotic software systems, mostly because they can embed the knowledge of experts in multiple scientific and technological domains (e.g. mechanics, control, cognitive sciences) and support the automatic transition from the problem space (the robotic requirements) to the solution space (the software implementation).

1) Real-Time Embedded Software Systems: A robot control application is an embedded software system that is specialized for the particular hardware it runs on and has time and memory constraints. Generally, for an embedded system, we are interested in modeling the interaction of the system with the surrounding environment. This interaction is characterized by the real-time execution of control activities that should react timely to sensory stimuli and produce commands to the actuators. Typically, in any large-scale embedded system, several control activities are executed concurrently on the same computational unit or on a distributed networked system. Concurrency implies communication among control activities, which exchange data and events, and requires careful design of their interleaving and synchronization in order to avoid anomalous behaviors.

2) Distributed Software Systems: The computational hardware of an autonomous robot is interfaced to a multitude of sensors and actuators, and has severe constraints on computational resources, storage, and power. Computational performance is a major requirement since autonomous robots process large volumes of sensory information and have to react in a timely fashion to events occurring in the human environment.

In order to meet these highly demanding requirements, the computing infrastructures of advanced autonomous robots have evolved from single processor systems to networks of general purpose computers, microcontrollers, smart networked sensors and actuators, introducing great flexibility in robot capabilities construction but, at the same time, making software development more challenging.

Contextually, a variety of software frameworks have been specifically designed for simplifying the implementation of robot control systems (see [8] for a survey). They offer mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management. These frameworks are supported by MDE toolchains that enables the sami-automatic transition from high-level design of the system functionalities to their actual implementation and deployment on top of specific runtime infrastructures.

3) Rich functionalities: Differently from other embedded systems (e.g. cars, medical devices, etc.), the control system of an autonomous robot is characterized by a large variety of functionalities (e.g. motion planning and control, perception, task planning, etc.) that together realize complex robot capabilities, such as navigation and manipulation.

Robot functionalities are conveniently implemented as software components that can be assembled in many different ways, like reusable building blocks, according to specific application requirements. A robot control system is composed of tens of components (e.g. see the ROS repositories [9]). For each component, tens of different implementations may be available (e.g. different algorithms for obstacle avoidance).

For these systems, a critical development phase is the design of the software architecture, which represents the partitioning of the control system into parts, with specific relationships among the parts, and makes the set of parts work together as a coherent and successful whole [10]. As such, the software architectures defines the rules and constraints that determine the overall behavior of the control system and that guarantee system dependability and safety.

MDE approaches 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 schedulability, performance, responsiveness, and fail-safe behavior.

4) Versatile machines: Robots are versatile machines that are increasingly been used not only to perform dirty, dangerous, and dull tasks in manufacturing industries, but also to achieve societal objectives, such as enhancing safety in transportation and reducing the use of pesticide in agriculture.

In this scenario, the cost of creating new robotics products is significantly related to the complexity of developing software control systems that are flexible enough to easily accommodate frequently changing requirements: more advanced tasks in highly dynamic environments, in collaboration with unskilled users, and in compliance with changing regulations.

Recent initiatives aim at developing MDE approaches that simplify the static and dynamic re-configuration of a robot control system according to specific application requirements and operational conditions.

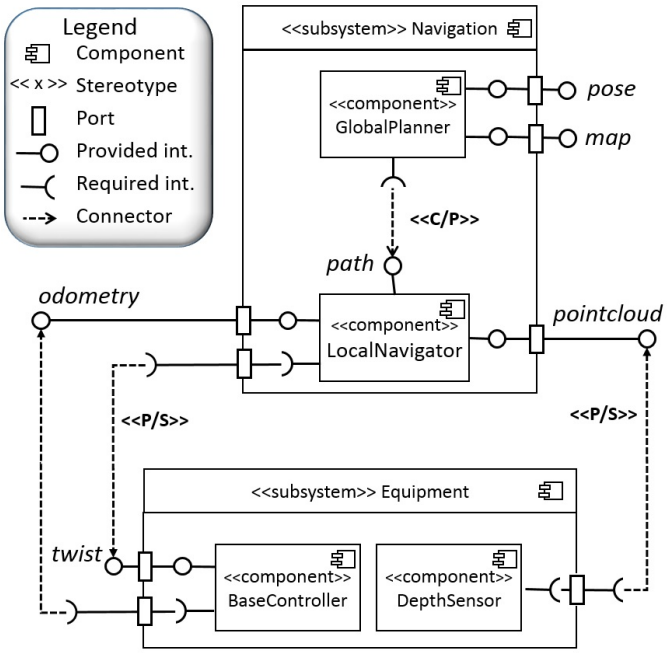

Fig. 1. The UML Component model of the Navigation architecture

II. System Architecture Modeling and Analysis

Software architecture design is a multi-dimensional decision-making process and different software models are needed to describe the system from multiple perspective, such as structure, behavior, and non-functional properties.

The Object Management Group (OMG) [11], a not-for-profit technology standards consortium founded in 1989, defines and maintains the specification of the Unified Modeling Language (UML) [7], a semi-formal general-purpose graphical language for modeling software systems. UML 2.5 is composed of fourteen standard diagram types, which are classified as *structure diagrams* or *behavior diagrams*. Structure diagrams show the static structure of a system in terms of parts and relationships among the parts on different abstraction and implementation levels. Behavior diagrams show the dynamic behavior of a system, i.e. how it changes over time.

In particular, the UML 2.5 *Component Diagram* defines a standard graphical notation for documenting architectural representations that emphasize the run-time computational elements (aka components) of a software systems and their communication channels (aka connectors). In [10] these representations are referred to as the "component and connector" architectural viewtype.

For example, Figure 1 depicts a simplified version of a software architecture for mobile robot navigation. It is composed of elemental components and composite components. Elemental components (e.g `GlobalPlanner`, `BaseController`) are graphically represented by boxes marked with the `<<component>>` stereotype. Similarly, composite components (e.g `Equipment`) are marked with the `<<subsystem>>` stereotype.

Each component is characterized by a set of *ports*, which represent distinct interaction points with other components.

Each port can be associated with a number of interfaces. A *provided interface* describes the features that constitute a coherent service provided by a component. Similarl y, a *required interface* describes the dependency of a component to a type of service that should be provided by another component. A *connector* represents a communication link between two or more components. Different types of connectors can be distinguished by labeling the association link with a stereotype. For example, in Figure 1, the stereotype `<<C/P>>` refers to the Caller/Provider communication paradigm (i.e. components interact through synchronous invocation of services), while the `<<P/S>>` refers to the Publisher/Subscriber communication paradigm (i.e. components interact through asynchronous messages). Guidelines to document the "component and connector" architectural viewtype in UML can be found in [12].

The main purpose of this kind of diagram is to document the software architecture from a functional point of view and serves as a vessel for communication between the varied worlds of often non-software-technical stakeholders on one hand, and software engineers on the other hand. Such a documentation promotes software reuse by exhibiting stable structures recurrent in many systems and facilitates maintenance by clarifying the impact of changes [13]. A large variety of UML-compliant software tools support diagram editing, differencing, merging for system design and documentation (see [14] for a survey).

Unfortunately, UML diagrams are not effective in capturing and representing non-functional properties of embedded, concurrent, and real-time software system, such as the timing constraints of system functionalities, the capabilities of the (often distributed) communication infrastructure, and the allocation of threads and processes to different processors.

Researchers have faced the limitations of the semi-formal notation of UML by defining specific extensions of the UML standard (called profiles) or specialized architectural modeling languages. These approaches are exemplified in the next two sections. A comparison of several modeling languages for embedded and real time systems can be found in [15].

A. UML profile for Embedded and Real-Time Systems

UML Profiles are an extension mechanism provided to allow adaptation and customization of the UML notation by adding ad-hoc semantic and constraints and introducing terminology that are specific to a particular domain, platform, or method.

In particular, OMG has developed the Modeling and Analysis of Real-Time Embedded Systems (MARTE) [16] profile, which focuses on performance and schedulability analysis and provides stereotypes for annotating architectural model and map them into corresponding analysis domain concepts.

The HLAM (High-Level Application Modeling) sub-profile defines a set of stereotypes to annotate the functional model with real-time features. For example, Figure 2 denotes a use of the `<<rtUnit>>` stereotype to annotate two computing units (i.e. *ObstacleAvoider* and *TrajectoryFollower*) of the *LocalNavigator* component, which perform concurrent activities, i.e. adapting the rover trajectory when an obstacle is detected and computing the twist to let the rover follow the trajectory.

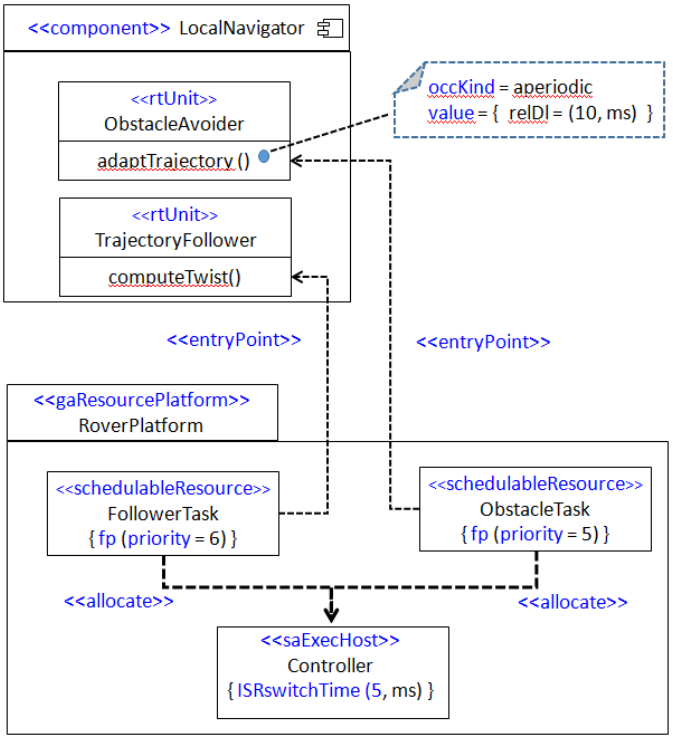

Fig. 2. Example of using MARTE stereotypes for schedulability analysis

An annotation specifies that the former activity is aperiodic and that its relative deadline is equal to 10 milliseconds.

The SRM (Software Resource Modeling) sub-profile provides modeling artifacts to describe software multi-tasking application programming interfaces (API). For example, Figure 2 denotes a use of the `<<schedulableResource>>` stereotype to annotate two concurrent tasks (i.e. *FollowerTask* and *ObstacleTask*) with the specification of their priority. The stereotype `<<entryPoint>>` indicates the routine (i.e operation) executed in the context of each task.

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, which executes functional steps) with non-functional properties, such as schedulability metrics, interrupt overheads and utilization of scheduling processing. In particular, in Figure 2 the stereotype `<<SaExecutionHost>>` represents any kind of processing resource (e.g. POSIX threads) and contains a property `ISRswitchTime` which can be used to represent the worst context switching time.

Once the application model has been annotated with MARTE stereotypes for real-time features, it needs to be converted in a software model that can be processed by tools for schedulability and performance analysis. In [17] the author proposes an automatic translation technique from MARTE models into input for MAST (Modeling and Analysis Suite for Real-Time Applications), which is a state-of-the-art schedulability analysis tool used in the academia. A list of tools related to MARTE can be found in [18].

A robotic example of using MARTE for schedulability and performance analysis can be found in [19].

Listing 1. Flow specification for the LocalNavigator

```
device laser_scanner
  features
    point_cloud: out data port;
  flows
    on_flow_src:flow source point_cloud
                {latency => 5 ms .. 5 ms;};
end laser_scanner;

process adapt_trajectory
  features
    point_cloud: in data port;
    trajectory: out data port;
  flows
    on_flow_path:flow path point_cloud->twist
                 {latency => 40 ms .. 60 ms;};
  properties
    Period => 100 ms;
end obstacle_avoider;

process compute_twist
  features
    odometry: in data port;
    trajectory: in data port;
    twist: out data port;
  flows
    on_flow_path:flow path trajectory->twist
                 {latency => 20 ms .. 30 ms;};
  properties
    Period => 50 ms;
end obstacle_avoider;

device rover
  features
    twist: in data port;
  flows
    on_flow_snk:flow sink twist
                {latency => 10 ms .. 10 ms;};
end rover;
```

B. Architecture Analysis and Design Language

For cyberphysical systems and, in particular, for robotic systems, response time analysis is of paramount importance. The response time of an embedded system depends on the latency between receiving an input from the sensors and producing an output to the actuators. For example, safety depends on the latency of responding to the detection of an obstacle along the robot path.

Such analysis requires to model an application in terms of both the computational units and of the communication channels. In particular, in a distributed embedded real-time (DERT) system, communication may be periodic or aperiodic, event-triggered or based on data sampling. Representing all these kinds of communications in MARTE is challenging.

In several application domains, the Architecture Analysis and Design Language (AADL) [20] has proven a good candidate as a modeling language for DERT systems.

AADL is a textual and graphical language with precise execution semantics for modeling the architecture of embedded software systems and their target platforms.

AADL provides modeling concepts to describe the runtime architecture of DERT systems in terms of components, connectors, concurrent tasks, their interactions, and their mapping onto an execution platform. Recently, OMG has provided the MARTE specification with guidelines to map its modeling entities to AADL concepts [21].

AADL is supported by the Open Source AADL Tool Environment (OSATE) [22], which comes with a suite of automated analysis tools. In particular, OSATE includes a flow latency analysis tool that automatically calculate end-to-end latency, i.e. the time required for a signal to travel from the source to the sink, and verify if latency requirements are satisfied.

Listing 1 exemplify the specification of a flow for the LocalNavigator component depicted in Figure 2.

Component interactions consists in directional *flows* from an event or data source (the *laser_scanner*), through a communication and processing path (*adapt_trajectory* and *compute_twist*), to a command sink (the *rover*). For each modeling element, the tag *features* specifies the component interface in terms of input and output ports.

The model specifies the latency time related to the acquisition of a point cloud, the adaptation of the current trajectory in order to avoid an obstacle, the generation of a twist command, and the transmission of the command to the rover actuators.

In [23] the authors use AADL to model the control system architecture for a safety-monitoring motorised wheelchair. End-to-end latency analysis is performed to determine such factors as the latency in responding to the appearance of an obstacle, the response time to a failure, and the suitability of the chosen micro-controller hardware.

C. Behavior Interaction Priority Framework

Expensive robot missions, like Mars exploration, and safety critical systems, like driverless cars, demand for formal proofs of their correct behavior, i.e. guarantees that the robot will not perform actions that lead to catastrophic consequences. Informally, a safety property stipulates that ”bad things” do not happen during execution of a program [24].

Usually, in the context of software engineering, discrete state approaches are applied to explain the behavior of a system with respect to safety. Examples are Statecharts, Petri Nets, and Labelled Transition Systems (LTS). In this context, a safety property asserts that the program does not exhibit bad behaviors, e.g. it never enters an undesirable state.

A relevant issue is the compositionality of the property ”safety” in concurrent and distributed systems. State-of-the art approaches are based on defining LTS models of concurrent processes, which are synchronized through actions sharing the same labels. For example, let *drop* represent the action in which a robot manipulator places an object in a bin transported by a mobile robot. The software process that controls the robot manipulator should issue the command to open the gripper only when both the arm and the rover are in the drop position. In terms of LTS, *drop* is modelled as a possible action in the standalone behavior of both the process that controls the robot manipulator and the process that controls the mobile robot. Its

execution then requires simultaneous participation from both processes.

The BIP (Behavior Interaction Priority) [25] framework is a architecture definition language for component-based systems, which uses LTS for modeling the behaviour (B) of elemental components and the interaction among components. The transitions between the discrete states of the component behaviour are triggered by events received through the component ports (action names). Atomic components can be assembled to form compound components by means of connectors, which specify possible interaction policy (I) between ports of atomic components. The behavior of a compound component is formally described as the composition of the behaviors of its atomic components. At the level of compound components a set of priority rules (P) describe scheduling policies for the interactions among atomic components.

The BIP framework is supported by a set of tools for offline analysis (e.g. the D-Finder tool [26]) and an engine for online monitoring of safety properties of a component-based system.

In [27] the authors present an approach to develop correct-by-construction component-based software controllers for autonomous robots by integrating the BIP framework with the LAAS architecture. The proposed approach consists in (i) developing the functional components of a robot control systems using the tools of the LAAS architecture, (ii) defining a corresponding BIP model for the functional components, (iii) adding safety constraints into the BIP model.

Safety constraints can be encoded as BIP connectors. As an example, let's consider a mobile robot that guides the patients of a hospital towards a medical office on demand. We want to express the constraint that the robot can execute only one guidance service at a time, i.e. it does not leave the patient in the middle of a corridor to serve an incoming request.

This can be done by defining a constraint on the values of the input and output ports of the *TrajectoryPlanner* component and the *LocalNavigator* component. The former receives *GoTo* requests and generates *NewTrajectories*. The latter notifies the *Status* of the current trajectory execution.

To enforce this constraint, if the *LocalNavigator* has not completed the current trajectory when a new request is received, the new *GoTo* request is rejected with a specific error message (e.g. *NOT-IDLE*). Listing 2 illustrates a simplified version of the BIP syntax for this constraint.

Listing 2. A BIP connector

```
connector RejectNavigationRequest

  on TrajectoryPlanner.GoTo,
     TrajectoryPlanner.NewTrajectory,
     LocalNavigator.Status

  provided LocalNavigator.Status.done

  do {TrajectoryPlanner.GoTo = NOT-IDLE}
```

At run-time, the BIP engine acts as controller of the functional components. It prevents the robot from reaching unsafe states, even if bugs exist at the decisional level of the control architecture, and reports faults to the decisional level.

III. System Implementation and Integration

MDE has become popular in many engineering domains because of its promise to bring benefits in software development, such as increased productivity and quality, thanks to automatic code generation from abstract models of an application. The idea behind is that the model is much simpler and thus easier to write and assess than the resulting code. This idea is valid in principle, but in practice it remains challenging to develop MDE environments, which fully automate the generation of code from models, as illustrated in [28].

Many state-of-the art MDE approaches (see [29] for a classification) consist in using domain-specific code generators for transforming abstract models into executable code. Typically, a code generator is specific for a modeling language and embeds the knowledge of a specific technological platform (e.g. a programming language, a middleware framework, a runtime infrastructure, a simulation environment).

Implementing code generators is hardly a core competence for most academic and industrial organizations that develop software technologies for specific application domains, such as robotics. Moreover, robotics is a highly change-centric domain and new technological platforms (software and hardware) are continuously developed. This means that code generator becomes quickly obsolete.

The Model Driven Architecture (MDA) [30] and other associated standards from the OMG are an attempt to reduce the impact of changes in technological platforms on the life cycle of software systems. The MDA enforces a clear separation of the functional architecture, called the Platform Independent Model (PIM) of a software system, from the technological details of the specific platform used to implement it. This is achieved using model transformation techniques that convert the PIM into one or more platform-specific models (PSM), which specify the details of how the functionality of the system use the capabilities of the software or hardware platforms to provide their operations.

Over the last few years, the software development industry has developed a significant variety of tools that automate model-to-model and model-to-code transformations. They use standard transformation languages, which provide constructs and mechanisms for expressing, composing, and applying transformations [31].

The subsequent transformation of the PSM into executable code requires to map the architectural model concepts to certain fixed code fragments (templates) provided by a domain framework and component library that represent the interface to the underlying platform (e.g. a distributed computing middleware).

Code templates need software developer to fill in details, i.e. the implementation of data structures and algorithms providing specific robotic functionalities. In some cases, this code can be generated automatically from behavioral models defined with general purpose modeling languages such as state charts and petri nets, or domain-specific modeling languages such as block diagrams and bond graphs.

The following sections exemplify three approaches to code generation, which are primarily concerned with reducing the

gap between the problem domain and the software implementation domain, by capturing different aspects of robotics systems.

A. SmartSoft

The SmartSoft project [32] has developed a software component framework, which aims at simplifying the development of real-time and distributed control systems by standardizing the component structure and connectors.

The framework provides mechanisms for:

- implementing software components according to the principles of Service Oriented Computing [33],
- interconnecting components by means of connectors that implement a limited set of communication patterns typically found in robotic control applications,
- dynamically reconfiguring the components behavior and interconnections according to the application task.

Each SmartSoft component can encapsulate one or several threads, which are executed in the context of a single process. Components provide services to and require services from other components. Component services exchange typed data called communication objects.

Two reference implementations of the SmartSoft component framework are currently available, one for real-time control systems based on Linux with the RTAI extension [34] and the other for distributed systems based on CORBA [35]. The SmartSoft framework is accompanied by a rich library of components, which implement the most common robot functionalities.

The SmartSoft MDE toolchain provides graphical editors for designing individual components, services, and interconnections and for specifying non-functional properties such as task execution time, period, and resource usage. The graphical editors are based on the UML profile for SmartSoft, which defines stereotypes for designing the PIM and PSM models.

Figure 3 shows a schematic representation of the model transformation supported by the SmartSoft MDE toolchain. The upper part of the figure depicts the *LocalNavigator* component as defined in the PIM. Here the software developer specifies the component interface in terms of communication objects and the component structure in terms of concurrent tasks. For each task (identified by the stereotype *¡¡SmartTask¿¿*) the non-functional properties (e.g. *period* and *wcet*) indicate the admissible values as specified by the application requirements (e.g. the *wcet* should not exceed 50 ms).

The trasformation of the PIM into the PSM (the central part of Figure 3) requires the specification of the software and hardware target platform. Here, the abstract tasks are mapped to Linux RTAI threads, as indicated by the stereotype *¡¡RTAITask¿¿*. The non-functional properties indicate actual values, such as the measured *wcet*.

Finally, the source code of the *LocalNavigator* is generated by customizing the template of the generic SmartSoft component with the information modeled in the PIM. The software developer needs to finalize the implementation of the component by hand coding the provided functionality.

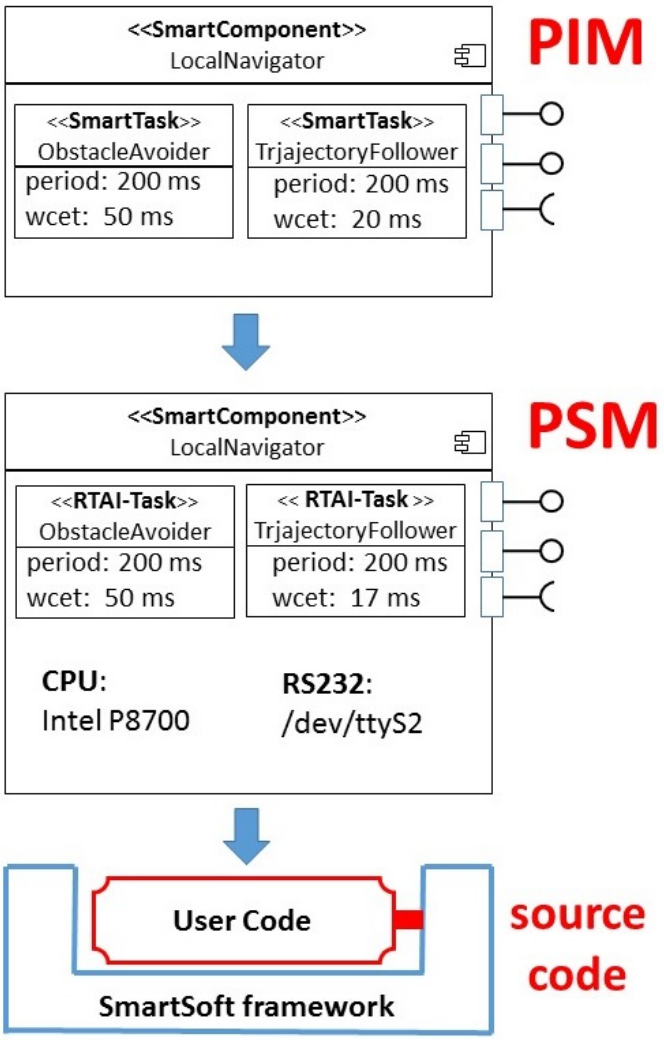

Fig. 3. Model transformations in the SmartSoft MDE Toolchain

In the simplest case, the generated code can be used as a wrapper of an existing software library.

The components interconnections are specified at design-time in the PIM, but can be modified at run-time in order to adapt the robot behavior to the actual operational conditions.

For example, the *ObstacleAvoider* task requires sensory information (e.g. a point cloud) to detect obstacles. At run-time different sensors can provide the point cloud based on environment illumination (e.g. a laser scanner or an RGB stereo camera).

The SmartSoft MDE includes the *Task Coordination Language* (SmartTCL) for specifying the high-level application tasks (e.g. fetch coffee to visitors) that the robot is able to carry out and how these tasks are refined during task execution in terms of services provided by individual components (e.g. “navigate towards the coffe machine”). The SmartSoft framework provides the *SmartTCL engine* that is in charge of orchestrating the control system by activating, deactivating, and interconnecting components according to the action plan.

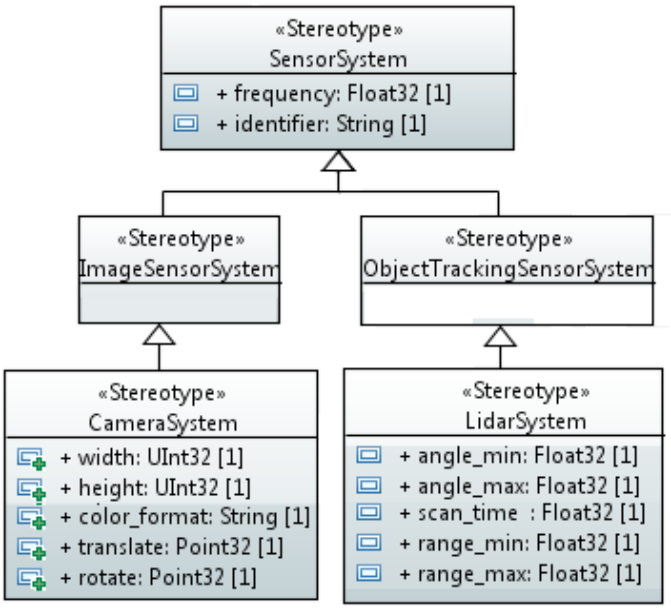

Fig. 4. The RobotML profile for robot sensors

B. Proteus

The Proteus [36] project (Platform for RObotic Modelling and Transformation for End-Users and Scientific communities) is an initiative of the French robotic community (GDR Robotique) that has developed a MDE toolkit for robotic system design and implementation.

The distinctive feature of the Proteus Toolkit is the use of a set of ontologies [37] for representing knowledge about robotic systems, operational environments, and applications.

The Proteus ontologies are an attempt to formalize the vocabulary of robotic system engineers, in order to allow them to model robot control architectures directly in terms of domain concepts (e.g. sensor, motion planner, rover, etc.) and not only in terms of software concepts (e.g. component, port, algorithm, etc.) Similar initiatives are sponsored by the IEEE Standard Association, which have established the Ontologies for Robotics and Automation working group.

The Proteus ontologies are the basis of the Proteus Domain Specific Language (*RobotML*) [38], which includes modeling entities for specific architectural elements (e.g. *Robot*, *SensorSystem*, *ActuatorSystem*, *LocalizationSystem*) and architectural styles (e.g. reactive, deliberative, hybrid).

Similarly to SmartSoft (see Section III-A), the Proteus toolchain is based on a UML profile for defining the PIM of the robot functional architecture. As an example, Figure 4 shows an excerpt of the RobotML profile for robot sensors, where the modeling entities *CameraSystem* and *LidarSystem* are defined as stereotypes that can be used to annotate the components of the robot control architecture.

For code generation, the elements of the PIM model need to be allocated to an execution platform, such as a middleware and a simulator. For examples, the element representing robot functionality and control activities are allocated to a component-based middleware, while the element representing the robotic equipment are allocated to a simulator.

C. Modeling the components behaviour

Code generation from behavior models is a growing area of interest due to its benefits of verifying models by simulation and reducing error-prone hand-coding efforts. Several tools generate source code form UML specifications to mainstream languages, such as C, C+, Java and to simulation languages such as SystemC. Typically, MDE environments supporting component-based architectural modeling (see Section II) also provides languages for modeling the discrete behavior of individual components, such as UML Statecharts and Petri Nets.

When developing embedded control software, control systems engineers model both the control algorithm and the system to be controlled, the so-called plant, together to ensure optimal performance of processes with continuous dynamics. [39]. Typically, the robot control functionalities are conveniently modeled as *Hybrid systems*, since they can specify continuous change of the system state as well as discrete transition of states. Continuous behavior can be specified using differential as well as algebraic equations.

Code generation from hybrid-systems models eventually involves simulating continuous change of a variable by stepwise update of the variable based on numerical methods. This requires the model designer to assign a rate by which the continuous state evolves.

Several modeling and simulation environments for embedded systems (e.g. Matlab/Simulink [40] and 20-sim [41]) support code generation from continuous and hybrid models, defined with domain-specific languages, such as *Bond Graphs*, *Block Diagrams*, and *Modelica*. These tools allow to define custom templates for code generation in order to simplify the integration of behavioral code with component frameworks.

In [42] the authors exemplify the modeling, code generation, and integration process for the motion control system of the KUKA youBot mobile manipulator [43]. The algorithm design requires a model of the system that will be controlled. They used the 20-sim toolchain to model the youBot kinematic chain as a bond graph and the BRIDE model-based toolchain [44] to model the component-based control system.

The control algorithm design is performed in parallel with the design of the component architecture. These two design phases have mutual dependencies and require an iterative approach.

The algorithm is partitioned in blocks of elemental functionalities with well-defined interfaces, in such a way that they can be modified independently (e.g. for improving their performance) without affecting the design and implementation of the rest of the system.

On the other side, message passing between components can introduce message losses and time delays in updates and communication between sensors, controller, and actuators that need to be explicitly represented in the model of the control algorithm. Unreasonable communication requirements (such extreme band-with or unachievable small latency) can lead to restructuring the component architecture.

In [45] a criterion for faithful implementation of hybrid-systems models in a concurrent and distributed system is presented.

IV. System Configuration

Current practice in software engineering for robotics consists in developing fine-grain software components, which implement single robotic functionalities. This approach is embodied by a repeated mantra among ROS developers [9]: "We don't wrap your main". The strength of this approach is the possibility to develop a large variety of different control systems by composing in multiple ways reusable software building blocks. Its weakness is the lack of support to the reuse of effective solutions to recurrent architectural design problems.

Consequently, application developers and system integrators have to solve architectural design problems always from scratch. The difficult challenge consists in selecting, integrating, and configuring a coherent set of components that provide the required functionality taking into account their mutual dependencies and architectural mismatches.

This challenge is exacerbated by the peculiarity of the robotics domain: robots can have many purposes, many forms, and many functions. Consequently, each robotic systems has to be configured with a specific mix of functionalities and that strongly depends on the robot mechanical structure (a rover with zero or multiple arms), the task to be performed (cleaning a floor, rescuing people after a disaster), and the environmental conditions (indoor, outdoor, underground).

In various application domains, software product line (SPL) development has proven to be the most effective approach to face this kind of challenges. A SPL is a family of applications (products) that share many (structural, behavioral, etc.) commonalities and together address a particular domain [46].

Typically, a SPL is a strategic investment for an engineering company that wants to achieve customer value through large commercial diversity of its products (e.g. different control systems for warehouse logistics) with a minimum of technical diversity at minimal cost.

The core of an SPL is a stable software architecture that clearly separates common features from the variations reflected in the products and prescribes how software components can be assembled to derive individual products. Each new application is built by configuring the SPL, i.e. by selecting the variants (e.g. functionalities, software resources) that meet specific application requirements.

MDE environments simplify system configuration by providing domain specific languages to model robot variability and model-to-model transformations to resolve the variability in the software control system.

The following sections illustrate two MDE approaches for robotic systems configuration. The former focuses on modeling variability in robot functionality, the latter on modeling variability in robot resources.

A. Software Product Lines for Robotics

The HyperFlex [47] toolchain is a MDE environment for the development and configuration of Software Product Lines (SPL) for Robotics.

It provides domain-specific languages and graphical editors for the definition of three types of models.

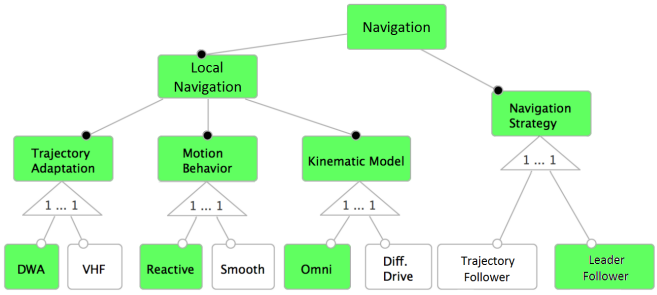

Fig. 5. The HyperFlex Feature Model of the Navigation system

The *Template Architectural Model* represents the software architecture (as discussed in Section II) of a family of similar software systems. As such, it explicitly represents all the possible components (variants) that implement a given functionality (variation point). For example, the obstacle avoidance functionality might be provided by two alternative components implementing the *Vector Histogram* approach or the *Dynamic Window* approach.

The *Feature Model* [48] symbolically represents the variant features of a control system; symbols may indicate individual robot functionality (e.g. marker-based localization) or concepts that are relevant in the application domain, such as the type of items that the robot has to transport (e.g. liquid, fragile, etc.), which affect the configuration of the control system.

The *Resolution Model* define model-to-model transformations, which allow to automatically configure the architecture and functionality of a control system based on required features. In this context, architecture configuration means to resolve the functional variability of a system by selecting one variant for each variation point.

A model-to-text transformation generates the configuration file (launch file) corresponding to the architecture of the configured control system Currently HypeFlex supports the configuration of component-based systems based on the ROS and OROCOS frameworks.

Let's consider an example of SPL related to the simple system depicted in Figure 1. The *LocalNavigator* compound component encapsulates the *TrajectoryFollower* component (see Figurefig:UML-MARTE), which implements the algorithm for generating *twist* commands to the *BaseController* component in order to follow a given trajectory.

Let us now consider a different application in which the robot has to follow a moving target (e.g. another robot). The *TrajectoryFollower* component is replaced with the *LeaderFollower* component that periodically generates *twist* commands for the robot according to the position of the target estimated by the sensors.

HyperFlex allows to represent the navigation strategy as a variation point that can be resolved by selecting at deployment time either the *LeaderFollower* component or the *TrajectoryFollower* component. Figure 5 shows the Feature Model of the Navigation system. Green boxes represent the selected features. The triangles below the boxes indicate the cardinality of the variation point. In particular, it indicates that for each variation point only one variant can be selected. Black circles

indicate that the child feature is mandatory.

The selection of desired features triggers model-to-model transformations that configure the template architecture. Several types of transformations can be defined, such as removing a component, changing the properties of a component, and changing the connections between components.

For example, the selection of the feature *LeaderFollower* triggers a transformation that replaces the *TrajectoryFollower* component with the *LeaderFollower* component and configures the connectors.

The HyperFlex toolchain offers the possibility to compose hierarchically Feature Models in such a way that higher-level Feature Models abstract the subsystems functional features. Here the idea is to model features at different levels of abstractions for different types of users.

Typically, the expert in robotic functionalities is interested in a representation of the functional features that highlights the different algorithms implemented in the robot control system.

On the contrary, the application domain expert is interested in a representation of the control system capabilities that highlights the application requirements. For example, the robot is able to navigate in an environment, which might be static or populated by moving obstacles (e.g. people). It might be an open space with wide passages or a small room crowded with tiny obstacles.

The selection of a feature corresponding to an application requirement triggers the automatic selection of features corresponding to robot capabilities. For example, if the environment is a static and crowded space, the robot should be configured with a complete (and likely slow) motion planner; instead, in dynamic environments, a fast and approximate motion planner is more effective.

B. Modeling robotic resources

A model of the robot embodiment (the kinematic structure, the device characteristics, etc.) and a model of the operational environment (the objects to recognize, avoid, grasp, etc.) are crucial for object manipulation and navigation. They act as shared resources among multiple robot functionalities, such as perception, planning, and control. Most algorithms that implement these functionalities are parametric with respect to the embodiment and world models. This means that the robot control system can be configured at deployment time according to the specific application requirements and can be reused without modification for different tasks and different robotic systems.

In [49] the authors present a textual domain specific modeling language for modeling the structural aspects of the robot embodiment and environment as a *Scene Graph*, i.e. a description of relevant objects and the relations among them. These relations are organized in a *Direct Acyclic Graph* (DAG). The Scene Graph can express prior semantic knowledge about a scene, such as the morphology and appearance of an object (e.g. a table) that the robot should recognize and locate in the environment.

Another example of textual modeling language is presented in [50]. It allows the description of differential constraints typically found in motion planning problems for mobile robots. Listing 3 shows an example of differential constraints for a differential drive rover, where `ul` and `ur` are the input angular velocities of the left and right wheels respectively (the action vector), `x`, `y`, and `theta` represent the robot configuration (the state vector), `r` is the wheel radius, and `L` is the distance between the two wheels.

Listing 3. The specification of a differential constraint

```
BEGIN DifferentialDrive
  ACTION : ul, ur;
  PARAM : L, r;
  CONFIG : x, y, theta;

  d(x) = r / 2 * (ul + ur) * cos(theta);
  d(y) = r / 2 * (ul + ur) * sin(theta);
  d(theta) = r / L * (ur - ul);
END;
```

These models are widely used in simulation algorithms (that, given the starting configuration and the action vector, compute the final configuration of the robot) and sampling-based motion planning algorithms (that sample collision free configurations that need to be compatible with the differential constraints). In order to compute final configurations it is necessary to solve the differential equations and this can be done by means of solvers, which use numerical approximation techniques.

V. Conclusions

MDE is often considered to be synonymous with code generation and, frequently, research and development efforts in MDE for Robotics are motivated by the objective of simplifying application development by robotic experts with limited software engineering skills. In particular, the availability of open source tools (e.g. Eclipse), that greatly simplify the development of speciaized MDE environments, stimulated the definition of domain specific languages for a large variety of robotics concerns. Two recent surveys can be found in [51] and [52].

Interestingly, a study on the state of practice in MDE in industry reports that productivity gains due to automatic code generation are not considered significant enough to drive an MDE adoption effort, due to increased training costs and substantial organizational changes [6]. It turns out that the main advantages are in the support that MDE provides in defining the architecture of a software system.

For this reason, this tutorial focused on the architectural model as the central artifact of almost all software development activities (analysis, design, implementation, configuration, and documentation). Intentionally, this tutorial did not illustrate MDE solutions for two other key activities in software development for robotics, namely simulation and model definition for run-time system configuration.

Simulation is such a wide field of research, that the analysis of MDE techniques for simulation could be the topic of a new tutorial.

Run-time configuration is a new trend in the development of self adaptive systems. In this context, the software control

system is equipped with mechanisms for automatic interpretation of architectural variability models and for dynamic reconfiguration, based on context awareness, of its resources and functionalities. MDE techniques (e.g. Dynamic Software Product Lines [53]) have been developed for several software-intensive application domains, but their applicability to the development of self adaptive robot control system is still a research issue.

REFERENCES

[1] H. Schichl, "Models and the history of modeling." in *Modeling Languages in Mathematical Optimization*, J. Kallrath, Ed. Kluwer, 2004, pp. 25–36.

[2] F. Cellier, "Principles of active electrical circuit modeling," in *Continuous System Modeling*. Springer New York, 1991, pp. 201–249.

[3] J. F. Broenink, "Bond-graph modeling in modelica," in *In Proceedings of ESS97 - European Simulation Symposium*, 1997.

[4] R. Lewis and I. of Electrical Engineers, *Modelling Distributed Control Systems Using IEC 61499: Applying Function Blocks to Distributed Systems*, ser. IEE control engineering series. Institution of Engineering and Technology, 2001. [Online]. Available: http://books.google.it/books?id=m3LaTv7VefwC

[5] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231146

[6] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *Software, IEEE*, vol. 31, no. 3, pp. 79–85, May 2014.

[7] OMG, "Unified modeling language (uml)," http://www.omg.org/spec/UML/, 2015.

[8] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)[tutorial]," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 1, pp. 100–112, 2010.

[9] S. Cousins, B. Gerkey, K. Conley, and W. Garage, "Sharing software with ros [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 17, no. 2, pp. 12–14, 2010.

[10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2003.

[11] O. M. Group, http://www.omg.org/, 2015.

[12] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and O. Silva, "Documenting component and connector views with uml 2.0," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2004-TR-008, 2004. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7095

[13] D. Garlan, "Formal modeling and analysis of software architecture: Components, connectors, and events." in *SFM*, ser. Lecture Notes in Computer Science, M. Bernardo and P. Inverardi, Eds., vol. 2804. Springer, 2003, pp. 1–24. [Online]. Available: http://dblp.uni-trier.de/db/conf/sfm/sfm2003.htmlGarlan03

[14] H. Eichelberger, Y. Eldogan, and K. Schmid, "A comprehensive survey of uml compliance in current modelling tools." in *Software Engineering*, ser. LNI, P. Liggesmeyer, G. Engels, J. Mnch, J. Drr, and N. Riegel, Eds., vol. 143. GI, 2009, pp. 39–50. [Online]. Available: http://dblp.uni-trier.de/db/conf/se/se2009.htmlEichelbergerES09

[15] K. D. Evensen and K. A. Weiss, "A comparison and evaluation of real-time software systems modeling languages," in *In Proceedings of AIAA Infotech@Aerospace*, Altanta, GA, 2010.

[16] OMG, "Modeling and analysis of real-time and embedded systems," http://www.omg.org/omgmarte/, 2015.

[17] J. Medina and A. Garcia Cuesta, "Model-based analysis and design of real-time distributed systems with ada and the uml profile for marte," in *Reliable Software Technologies - Ada-Europe 2011*, ser. Lecture Notes in Computer Science, A. Romanovsky and T. Vardanega, Eds. Springer Berlin Heidelberg, 2011, vol. 6652, pp. 89–102.

[18] OMG, "Tools related to marte," http://www.omgmarte.org/node/31, 2015.

[19] S. Demathieu, F. Thomas, C. Andr, S. Grard, and 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.

[20] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[21] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, "Marte: Also an uml profile for modeling aadl applications," in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, July 2007, pp. 359–364.

[22] "Osate: Open source aadl tool environment," http://www.aadl.info, 2015.

[23] G. Biggs, K. Fujiwara, and K. Anada, "Modelling and analysis of a redundant mobile robot architecture using aadl," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, D. Brugali, J. Broenink, T. Kroeger, and B. MacDonald, Eds. Springer International Publishing, 2014, vol. 8810, pp. 146–157.

[24] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, mar 1977.

[25] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2006.27

[26] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5643. Springer, 2009, pp. 614–619.

[27] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A verifiable and correct-by-construction controller for robot functional levels," *Journal of Software Engineering for Robotics*, vol. 2, no. 1, pp. 1–19, 2011. [Online]. Available: http://dx.doi.org/10.4018/ijismd.2014070103

[28] R. France, B. Rumpe, and M. Schindler, "Why it is so hard to use models in software development: observations," *Software Systems Modeling*, vol. 12, no. 4, pp. 665–668, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0383-z

[29] Y. Zheng and R. N. Taylor, "A classification and rationalization of model-based software development," *Softw. Syst. Model.*, vol. 12, no. 4, pp. 669–678, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0355-3

[30] OMG, "Model driven architecture," http://www.omg.org/mda/, 2015.

[31] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, Sep. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231150

[32] A. Lotz, J. F. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, and C. Schlegel, "Towards a stepwise variability management process for complex systems: A robotics perspective," *Int. J. Inf. Syst. Model. Des.*, vol. 5, no. 3, pp. 55–74, Jul. 2014. [Online]. Available: http://dx.doi.org/10.4018/ijismd.2014070103

[33] M. Huhns and M. Singh, "Service-oriented computing: key concepts and principles," *Internet Computing, IEEE*, vol. 9, no. 1, pp. 75–81, Jan 2005.

[34] "Rtai - the realtime application interface for linux," https://www.rtai.org/, 2015.

[35] O. M. Group, "Common object request broker architecture," http://www.corba.org/, 2015.

[36] "Plateforme pour la robotique organisant les transferts entre utilisateurs et scientifiques (proteus)," http://www.anr-proteus.fr, 2015.

[37] M. R. Genesereth and N. J. Nilsson, *Logical Foundations of Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[38] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds. Springer Berlin Heidelberg, 2012, vol. 7628, pp. 149–160.

[39] J. Palczynski and S. Kowalewski, "Early behaviour modelling for control systems," in *Computer Modeling and Simulation, 2009. EMS '09. Third UKSim European Symposium on*, Nov 2009, pp. 148–153.

[40] "Simulink, simulation and model-based design," http://it.mathworks.com/products/simulink/, 2015.

[41] "20-sim," http://www.20sim.com/, 2015.

[42] Y. Brodskiy, R. Wilterdink, S. Stramigioli, and J. Broenink, "Fault avoidance in development of robot motion-control software by modeling the computation," in *4th International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR 2014*, ser. Lecture notes in computer science, D. Brugali, J. Broenink, T. Kroeger, and

B. MacDonald, Eds., vol. 8810. Springer International Publishing, October 2014, pp. 158–169.

[43] R. Bischoff, U. Huggenberger, and E. Prassler, "Kuka youbot - a mobile manipulator for research and education," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 1–4.

[44] "Brics integrated developement environment," http://www.best-of-robotics.org/bride/, 2015.

[45] M. Anand, S. Fischmeister, Y. Hur, J. Kim, and I. Lee, "Generating reliable code from hybrid-systems models," *IEEE Transactions on Computers*, vol. 59, p. 1281–1294, July 2010. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5453343

[46] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[47] L. Gherardi and D. Brugali, "Modeling and Reusing Robotic Software Architectures: the HyperFlex toolchain," in *IEEE International Conference on Robotics and Automation (ICRA 2014)*. Hong Kong, China: IEEE, May 31 - June 5 2014.

[48] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[49] S. Blumenthal and H. Bruyninckx, "Towards a Domain Specific Language for a Scene Graph based Robotic World Model," in *4th International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-13)*, Tokyo University, Japan, Nov. 2013.

[50] M. Guarnieri, E. Magri, D. Brugali, and L. Gherardi, "A Domain Specific Language for Modeling Differential Constraints of Mobile Robots," in *International Conference on Autonomous Robot Systems and Competitions (Robotica 2012)*, Guimares, Portugal, 2012. [Online]. Available: http://hdl.handle.net/1822/18887

[51] A. Nordmann, N. Hochgeschwender, and S. Wrede, *A Survey on Domain-Specific Languages in Robotics*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8810, ch. 17, pp. 195–206.

[52] A. Ramaswamy, B. Monsuez, and A. Tapus, "Model-driven software development approaches in robotics research," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, ser. MiSE 2014. New York, NY, USA: ACM, 2014, pp. 43–48. [Online]. Available: http://doi.acm.org/10.1145/2593770.2593781

[53] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Corts, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: Observations from research and industry," *Journal of Systems and Software*, vol. 91, no. 0, pp. 3 – 23, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214000119