A modeling framework for software architecture specification and validation

Nicolas Gobillot Charles Lesire David Doose

ONERA – the French Aerospace Lab
2 avenue Edouard Belin, FR-31055 TOULOUSE, FRANCE
email: firstname.lastname@onera.fr

Abstract. Integrating robotic systems into our everyday life needs that we prove that they will not endanger people, i.e. that they will behave correctly with respect to some safety rules. In this paper, we propose a validation toolchain based on a Domain Specific Language. This DSL allows to model the software architecture of a robot using a component-based approach. From these models, we provide tools to generate deployable components, as well as a two-step validation phase. This validation first performs a real-time analysis of the component architecture, leading to an evaluation of the software architecture schedulability. Then we can check the validity of some behavioral property on the components.

1 Introduction

Nowadays, computer-based systems occupy an increasing place in our everyday or professional life. Robots for instance were absent of our houses in the 80's but today tasks performed by such machines are increasing. In the early days, only simple tasks were given to robots due to different limitations: robots were mechanically limited by their heavy materials, their sensors and actuators were big and inaccurate and their processors were slow (HERO[1] or PUMA[2]). Thanks to miniaturisation, mechanical parts are lighter, electronic circuits are smaller and processors are powerful enough to perform complex tasks (Da Vinci Surgical System[3] or Curiosity[4]). To make these robots usable in our everyday life, we need to ensure that they respect some safety rules, especially regarding their damaging capabilities. Safety has been considered regarding several aspects of robotics:

- *collision avoidance*, adapting the robot movements in presence of obstacles (e.g., [14] generates safe velocity bounds based on environment geometry; [17] conceives a mechanical system able to detect contact at an early stage);
- *human interaction*, where human behaviors are anticipated to avoid collisions while interacting with people (e.g., [24] in manufacturing places, [29] that tracks pedestrian behaviors);

[1] http://www.hero-1.com/Broadband/
[2] http://www.digplanet.com/wiki/Programmable_Universal_Machine_for_Assembly
[3] http://www.davincisurgery.com/index.php
[4] http://mars.jpl.nasa.gov/msl/mission/rover/

- *fault detection and tolerance*, where software/hardware reconfigurations or mode changes are controlled depending on what happens in the environment (e.g., [16] use invariant monitoring and change robot's mode accordingly; [20] uses a hierarchical decomposition of actions to switch between alternatives);
- *controller synthesis*, where the robot behavior is guaranteed by construction of the movement or action policy (e.g., [9] for continuous control of non-linear robots; [21] that verifies an action policy while learning it).

Another specificity in robot development is its fast evolution. This fast evolution leads to short development cycles of several month unlike in aeronautics or in the nuclear field which have development cycles of tenth of years. Due to this we need fast and accurate methods and tools to guarantee that the robots will always have a safe behavior. In order to have fast these methods and these tools, we need to reuse hardware and software parts between robots and design these parts with maximum modularity.

To help the software robot developer, modern designs are made of two parts: a middleware and a component-based architecture. The middleware provides operating system and hardware abstractions. The middleware typically proposes an Application Programming Interface to develop and deploy tasks and threads without taking into account the operating system and thus the hardware specificities. Among robotic-oriented middlewares, we can cite OROCOS [27] as a real-time focused middleware, ROS [22] that provides a large amount of already developed components, and GenoM3 [19] that provides a component generator with a component modeling language.

A component-based design pattern allows the software architect to build a robotic architecture by assembling existing software components (see [5, 6] for a survey on component-based software engineering in robotics). These software components are made of two parts: their communication interface and their internal behavior. Communications are driven by connecting ports or by service calls between two or more components. The component's behavior is often defined by a state-machine, that allows to define several operational modes, including some degraded mode to be robust to sensor or software failures.

In this paper, we propose an evolution of the Mauve Domain Specific Language (DSL) [18] to specify robotic architectures through an extensive use of models. These models are then used to generate the executable codes run on the real robots. From this DSL, we then provide methods and tools to analyze and check the validity of functional and temporal properties leading to robot safety.

2 Experimental setup

We will illustrate our approach using a concrete robotic experiment. This case study uses a Pioneer 3-DX robot (P3DX) from Adept Mobile Robots (Fig. 1). The P3DX is a wheeled robot equipped with an internal computer which serves as a controller interface. Its stock capabilities allows it to move around through its wheel speed controller and avoid some obstacles using its sonar range finders.

Its engines are sufficiently powerful to move around outdoors on reasonably rough terrains and its small size allows it to find its way into corridors. Our P3DX platform is equipped with a Hokuyo UTM-30LX laser scanning range finder and an Asus Xtion Pro Live depth and color camera. In this case study, our robot has to navigate safely in unknown and dynamic environments. We then need some navigation functions (including localization and mapping), a path planning algorithm to compute paths to follow, and a control function to follow correctly this path. We also want our robot to detect and track specific objects identified by color patterns.

Fig. 1. The P3DX platform equipped with a Hokuyo laser and a laptop for processing

The objective of this work is to propose first a design process for software architectures of autonomous robots, based on a DSL (section 3), and then to prove some aspects of the robot safety using tools leaning upon this DSL (section 4).

3 The Mauve DSL

This section presents the Mauve DSL for specifying and conceiving software architectures for autonomous robots. This DSL is an extension of [18]. It is based on four layers: *codels*, that correspond to the computational aspects of the software, *components*, that are elementary blocks of the software, *architecture*, where components are instantiated and connected, and *deployment*, where the execution policy of the architecture is defined depending on the target.

3.1 Codels

A *codel* (term taken from [19]) stands for an *elementary code* and represents any computational part of a component. The Mauve DSL does not provide any way to implement the codel (which could be implemented in any language; for code generation and analysis, we only support C and C++). The Mauve DSL provides instead a language for specifying the codels, that will then be called from components. Listing 1.1 shows the specification of codels implementing detection of an object of interest (`Detect`), and tracking of this object on images (`Track`). They both take an image as input, and provide the pose of the object.

Code 1.1. Codels from an image detection and tracking algorithm

```
codel Detect(img: Img): Pose
codel Track(img: Img): Pose
```

3.2 Components

According to [28], *a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Therefore, in order to help composition and modularity, we decompose the specification of a component into a *shell* (or interfaces) and a *core* (or behavior). In order to perform some validation of the component and architecture behavior, we also define some contracts that indicate the conditions of use of the component.

Component's Shell The shell of a component defines its interface, i.e. its inputs and outputs. We propose three types of interfaces: *properties*, which are component parameters, generally set at instantiation or deployment time (e.g., the max velocity of a platform); *data ports*, similar to the *push* pattern of [25], are used to publish data from/to a component; they are typed and oriented; *operations*, similar to the *query* pattern of [25], are used to call functions or send requests to components.

Specifying a shell in then done by listing the properties of a component (with a type and possibly a default value), its ports (with a type and a direction – in or out), and its operations (same way as codel signature). Values of properties, as well as connections of ports and operations, are not done at the moment of specifying a component shell. Instead, they are defined when instantiating and connecting components (*architecture* specification step).

Listing 1.2 shows the shell specification of the detection and tracking component. It has one property, `cameraType`, used to configure properly the component, e.g. by mapping the camera type value to resolution and focal length of the sensor. Default value refers to a Asus Xtion depth sensor. The component also has one input port `imgPort` to get an image stream (typically from a sensor component) and one output port *objectPort* exposing the pose of the detected and tracked object.

Code 1.2. Shell of the detection and tracking component

```
shell DT_Shell {
  property cameraType: string default "Xtion"
  input port imgPort: Img
  output port objectPort: Pose
}
```

Component's Core The core of a component defines its behavior. It can be described on two ways: first by mapping provided operations to codels, and second, by specifying a state-machine. Using a state-machine representation has

some advantages, among which clearly separating the different functionalities of the component, and providing some states to handle errors, then increasing the robustness or reconfiguration skills of the architecture. A state-machine is defined by a set of states and a set of transitions connecting the states. Each state represents a step in the functional behavior. The whole functionality is achieved by a sequence of states connected with transitions. The execution of each state is decomposed into several blocks, following the same approach than in UML state charts for instance:

- `entry`, defining instructions called only when a state is entered;
- `run`, called each time a state is active;
- `exit`, called when a state is exited (through a transition);
- `handle`, called when no transition is taken;

In each block, it is possible to write some instructions provided by Mauve, such as reading or writing on ports, calling codels, or calling remote operations. In each state, possible transitions are specified by a label, a guard, and a destination state. Listing 1.3 shows the core of the detection and tracking component. Its state machine is made of four states:

- `Initialize` (line 3), in which some algorithm data structures are prepared;
- `Cleanup` (line 6), in which data are cleaned;
- `Detecting` (line 9), in which the input port is read, the detecting algorithm is called, and the resulting pose is published on the output port;
- `Tracking` (line 18), similar for the tracking algorithm.

Transitions are guarded by events that will come either from the component itself (`[pose]` on line 16 checks that a pose has been returned by `Detect`) or triggered from other components.

Code 1.3. State-machine example using the detection and tracking codels

```
1  core DT_Core (DT_Shell) {
2    statemachine {
3      initial state Initialize {
4        transition toDetecting [not initialize] -> Detecting
5      }
6      state Cleanup {
7        transition toInitialize [not cleanup] -> Initialize
8      }
9      state Detecting {
10       run {image = read imgPort}
11       handle {
12         pose = Detect(image);
13         write pose in objectPort
14       }
15       transition toCleanup [cleanup] -> Cleanup
16       transition toTracking [pose] -> Tracking
17     }
18     state Tracking {
19       run {image = read imgPort}
20       handle {
21         pose = Track(image);
22         write pose in objectPort
23       }
24       transition toDetecting [pose] -> Detecting
25     }
26   }
27 }
```

Component's Contracts Contracts are meant to represent the condition of use of a component, and the result or behavior we could expect when executing this component. The shell of the component already specifies a contract: it declares the inputs needed by the component, and the type of data published by the component. The Mauve DSL provides complementary instructions to specify functional properties of the components. These properties represent an abstraction of the behavior of the component. For instance, the role of the guidance component is to follow a path while avoiding obstacles. For this component the most important feature regarding safety is to avoid collisions. Listing 1.4 shows how this property is expressed as a contract on the guidance component: the robot has to stop (speed command sent on `speedPort` must be equal to 0) whenever something is detected (read from the `scanPort`) within a safety distance (defined as a component property).

Code 1.4. Contract on the Guidance component

```
shell Guidance_Shell {
  codel minRange(scan: Scan): double
  property robotType: string default "unicycle"
  property safetyMargin: double default 0.5
  input port scanPort: Scan
  output port speedPort: Speed

  contract emergencyStop:
    [minRange(read scanPort) < safetyMargin ⇒ write 0 in speedPort]
```

3.3 Architecture

Creating a functional software layer for autonomous robots settles on reusing and composing basic component blocks. The *architecture* design step consists in instantiating some components (defined using the previous language instructions), possibly define some properties. Then these components are connected together, specifying the configuration of the communication. For instance, it is where we can specify that a connection between two ports is buffered, and specify the management policy of the buffer (size, circular or not).

We had developed a Navigation, Guidance and Control component-based architecture meant to be run on mobile robots [13], that we modified, improved, and adapted to the match the Mauve DSL presented in this paper. It results in the architecture of Fig. 2, made of eight components (drawn as circles): components for sensor (`Camera` and `Laser`) and actuator (`Robot`) interfaces; the `Navigation`, `Guidance` and `Control` components managing movements; a `SLAM` component to build a map and navigate in it; and finally the `Detection and Tracking` component.

This architecture is specified using the Mauve DSL. Listing 1.5 shows a piece of the architecture specification where two components are instantiated: `camera` and `detectTrack`, and the ports of the two components are connected.

Code 1.5. Part of the architecture specification

```
instance camera: Camera {}
instance detectTrack: DT_Core {}
port camera.imgPort data detectTrack.imgPort
```

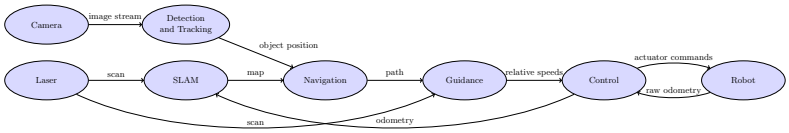

Fig. 2. Simplified component-based architecture running on the robot

3.4 Deployment

The deployment is the target-specific part of a real-time software development. It will map the components and architecture specification to the final target (environment, platform) in order to be executed. The deployment is decomposed into several layers: the *hardware*, also called target, corresponds to the actual platform used for experiments, i.e. sensors, computer units, etc. Using the Mauve DSL, architecture components properties can be set to indicate which platform is used, e.g., by defining the device port on which the camera is connected; the *middleware* is a layer over the operating system providing a set of features that makes development easier. For the moment, the only supported middleware is Orocos [27], and therefore we do not provide any mean to choose the middleware.

In order to be able to analyze the deployed architecture, and more specifically its real-time characteristics, we impose that each component is mapped to at most one thread. When deploying the components, we then associate to each component an activity indicating the execution behavior of the component. This activity allows to define the period of the component, its priority, and its deadline. These properties will then be mapped to Orocos components activities, resulting in properties of the corresponding OS threads. We can also set the affinity of a component, i.e. the core on which the component will run if executing on a multi-core platform.

Listing 1.6 shows a part of a deployment where the `robot` component has a period and a deadline of 100ms, a priority of 0. Furthermore, we can specify the execution time of codels (codel `command` takes 16ms to execute), used for real-time analysis (see section 4.1).

Code 1.6. Part of a deployment specification

```
1 deployment {
2   command = 16..16
3   activity robot {
4     priority = 0
5     period   = 100
6     deadline = 100
7   }
```

3.5 Execution

Along with the Mauve DSL, we provide a code generation toolchain that, for each Mauve component, generates the code of an Orocos component linked with the corresponding codels library; and for each Mauve architecture plus deployment,

generates the code of an Orocos script that deploys the architecture (loads the components, instantiate them, connects them, ...) The result is then directly executable on the specified platform.

3.6 Why a new DSL?

A lot of software modeling DSLs for robotics can be found in the literature. In [15] the robot's software architecture is modeled in three main layers: the *functional architecture*, the *component architecture* and the *runtime architecture*. The *functional architecture* expresses the functionality needed in an architecture. The *component architecture* details the software implementation of the architecture's functionality. Lastly the *runtime architecture* defines the deployment of the components on the robot's operating system. This framework also generates *roslaunch* files for ROS [22]. [10] uses an UML-based language called RobotML, based on an ontology to reuse as much as possible robotic knowledge. This knowledge is split in five packages: *robotic system*, *system environment*, *data types*, *robotic mission* and *platform*. Afterward the robotic architecture is defined using the RobotML packages, and it is possible to generate executable code for several robotic middlewares, among which Orocos. V^{3}CMM [1] separates the robotic architecture in three abstraction levels: *Computation Independent Models*, *Platform-Independent Models* and *Platform-Specific Models*. Once the architecture is set-up, a model-to-model transformation is used to provide UML models, and then executable code through a model-to-text transformation to the Ada 2005 programming language. In this paper, we have described the Mauve DSL. Mauve relies on more or less the same concepts than other DSLs. The aim of Mauve is to not only provide architectural abstraction, simple software design and code generation but also to systematically perform validation and analysis based on these models. It hence seemed difficult to reuse an existing DSL, as we needed to specify new concepts (or to prevent the use of existing concepts).

Two major works are using DSLs for architecture specification along with a validation process: SmartSoft [26] models the architecture and the components through model layers. It provides real-time specific parameters to allow a static analysis of the architecture through CHEDDAR [11]. Regarding the real-time analysis, we propose to reason on the component models to have a more accurate estimation of architecture schedulability. Moreover, we are concerned with other analyzes than just real-time analysis. BIP [2] is a modeling language that comes with safety properties and deadlock freedom analysis tools. The safety properties ensure that no unexpected behavior will ever happen thanks to global state exploration. The deadlock analysis go through a structural analysis to guarantee the software will keep its nominal execution. The major drawback of using this paradigm is that generated code is tied to their own execution engine (behaving as a scheduler), and real-time analysis is not dealt with.

4 Validation tools

The main objective of our work is to perform some validation of the executed software architecture for an autonomous robot. We therefore lean upon the modeling framework presented above in order to reason on the architecture behavior. Regarding the presented experimental case study, it is for instance important for the P3DX robot to ensure that it will not collide into any object in the environment. Proving this safety constraint relies on several concerns: we first need to prove that a logical property is true, by reasoning on the contracts provided by the components. Second, we need to check that all component will be able to execute in time. This last property is called schedulability.

4.1 Real-time analysis

The aim of the real-time analysis is to *a priori* check the schedulability of the architecture on a specific system, before deploying it on the real system. Usually the schedulability of a system is defined by the schedulability of all the tasks involved in the system. We do not detail in this paper all the models and computations for the analysis to happen, as it is a bit out of the scope of the paper, but we give a brief explanation of the whole process.

When specifying the deployment of the architecture using the Mauve DSL, we map components to real-time tasks on the system, with tasks parameters such as period, deadline, and priority. Classical schedulability methods directly use these parameters, along with an estimation of the computation time of each task, to compute the Worst Case Response Time (WCRT) of a task. If the WCRT of a task is lesser than its period, the task is said schedulable.

We have adapted this process to use models of components, in order to perform a more accurate computation of the WCRT. We transform component models into Periodic State-Machines (PSM), on which each transition is labeled with the time taken to go from one state to another, computed from the state blocks, the execution times of codels, and the interactions between components (e.g., when a component calls an operation on another component). The execution time of each codel is obtained using Worst Case Execution Times (WCET) analysis. We tried two approaches depending on the codels: a static analysis, using Otawa [23], where the binary code is directly analyzed to estimate the number of cycles a function will take to execute; and a statistical analysis based on execution runs of the component. For now we made only basic statistics to deduce an experimental WCET, but we are currently working on using extreme-value theory to have a mathematically sound estimation (with a given probability) of the WCET. We then modified the classical WCRT computation algorithm to take into account tasks state-machines (PSMs) and components interactions. It results in a new evaluation, called WCRT+, which is still pessimistic (hence safe) but more accurate then the classical approach.

Table 1 presents the results of the schedulability analysis of our robotic architecture. The lower the priority value, the higher the priority of the component. The WCET value corresponds to the WCET of the most time-consuming

transition of the PSM. The WCRT and WCRT+ columns are respectively the "classical" and our state-machine based evaluations. The WCRT+ computation method proves the schedulability of the architecture whereas the typical method indicates the `Navigation` component may not be schedulable.

Table 1. Real-time characteristics of the architecture's components

component	priority	WCET	WCRT	WCRT+	deadline
Robot	0	16	16	16	100
Control	1	3	19	19	100
Guidance	2	12	31	31	100
Laser	3	22	53	53	150
SLAM	4	30	83	83	150
Camera	5	10	93	93	250
Detection and Tracking	6	30	237	237	250
Navigation	7	30	**338**	**297**	**300**

4.2 Checking behavioral properties

Previous section shows how we used the component and architecture models to accurately compute the deadline (and the schedulability) of the deployed components. Enforcing these deadlines is needed for a good behavior of the robot. However it is not sufficient to guarantee the correctness of this behavior. In this section, we propose to analyze the correctness of this behavior by studying the evolution of the components (and their state machines) along the time.

For the moment, the properties that we are able to manage rely on observation points: we can specify instructions or states on which we want to elaborate a property. For instance, we can express, using these observation points, that a component will eventually enter in state A, or that when component *guid* enters state *running*, then component *control* will eventually enter state *running* before 10 time units (see listing 1.7).

Code 1.7. Specification of a property

```
1 property latency = guid.running leadsto control.running within [0, 10]
2 assert latency
```

Formally, these properties are expressed using a temporal logic that accepts Dwyer's patterns [12] extended with timed data. Verifying such properties is quite complex in general, as we must analyze both the control flow of the architecture but also the interactions between components (leading to task preemption). We then developed a specific validation process that directly uses temporal information coming from the real-time analysis presented in section 4.1. For that, the Mauve model of the components and architecture, along with the properties we want to analyze, are transformed into Timed Petri Nets using the RT-Fiacre language [3]. The resulting model is then analyzed using Tina [4], which either validates the property or provides a counter example as a timed execution of the system.

5 Conclusion and perspectives

In this paper, we have presented a robotic architecture modeling framework based on the Mauve DSL. It allows to model robotic software architectures from the algorithms to a real-time deployment thanks to four layers: the *codels* specification, the *component* modes, the specification of the *architecture* and, finally, the *deployment*. From these models, Mauve is able to generate C++ executable code designed for Orocos-RTT [27]. Along with the Mauve DSL, we have provided tools to first analyze the real-time correctness of the architecture, and second to check the validity of some behavioral properties.

For future developments, we plan to improve the validation toolchain by analyzing not only behavioral properties but also property that contains data, such as the contract defined in listing 1.4. To do that, we will rely on well known tools for codel analyzes, such as Frama-C [8] and Coccinelle [7]. Finally, we plan to apply our design and validation process to other kind of robots, like hybrid leg-wheel robots and quadcopters.

References

1. Alonso, D., Vicente-chicote, C., Ortiz, F., Pastor, J., Alvarez, B.: V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. Journal of Software Engineering for Robotics (JOSER) 1, 3–17 (2010)
2. Basu, A., Gallien, M., Lesire, C., Nguyen, T.H., Bensalem, S., Ingrand, F., Sifakis, J.: Incremental Component-Based Construction and Verification of a Robotic System. In: ECAI. Patras, Greece (2008)
3. Berthomieu, B., Bodeveix, J., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: Fiacre: an intermediate language for model verification in the TOPCASED environment. In: Embedded Real Time Software and Systems (ERTSS). Toulouse, France (2008)
4. Berthomieu, B., Vernadat, F.: Time Petri Nets Analysis with TINA. In: Int. Conf. on Quantitative Evaluation of Systems (QEST). Riverside, CA, USA (2006)
5. Brugali, D., Scandurra, P.: Component-Based Robotic Engineering. Part I: Reusable Building Blocks. IEEE Robotics and Automation Magazine 16(4) (2009)
6. Brugali, D., Shakhimardanov, A.: Component-Based Robotic Engineering. Part II: Systems and Models. IEEE Robotics and Automation Magazine 17(1) (2010)
7. Brunel, J., Doligez, D., Hansen, R.R., Lawall, J.L., Muller, G.: A foundation for flow-based program matching using temporal logic and model checking. In: ACM Symposium on Principles of Programming Languages. Savannah, GA, USA (2009)
8. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, A Software Analysis Perspective. In: Int. Conf. on Software Engineering and Formal Methods (SEFM). Thessaloniki, Greece (2012)
9. DeCastro, J.A., Kress-Gazit, H.: Guaranteeing reactive high-level behaviors for robots with complex dynamics. In: IROS. Tokyo, Japan (2013)
10. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In: SIMPAR. Tsukuba, Japan (2012)
11. Dissaux, P., Singhoff, F.: Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures. In: Embedded Real Time Software and Systems (ERTSS). Toulouse, France (2008)

12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Software Engineering. Los Angeles, CA, USA (1999)
13. Gobillot, N., Lesire, C., Doose, D.: A Component-Based Navigation-Guidance-Control Architecture for Mobile Robots. In: ICRA – SDIR workshop. Karlsruhe, Germany (2013)
14. Haddadin, S., Khoury, A., Rokahr, T., Parusel, S., Burgkart, R., Bicchi, A., Albu-Schaffer, A.: A truly safely moving robot has to know what injury it may cause. In: IROS. Vila Moura, Portugal (2012)
15. Hochgeschwender, N., Gherardi, L., Shakhirmardanov, A., Kraetzschmar, G.K., Brugali, D., Bruyninckx, H.: A model-based approach to software deployment in robotics. In: IROS. Tokyo, Japan (2013)
16. Jiang, H., Elbaum, S., Detweiler, C.: Reducing failure rates of robotic systems though inferred invariants monitoring. In: IROS. Tokyo, Japan (2013)
17. Lens, T., von Stryk, O.: Investigation of safety in human-robot-interaction for a series elastic, tendon-driven robot arm. In: IROS. Vila Moura, Portugal (2012)
18. Lesire, C., Doose, D., Cassé, H.: MAUVE: a Component-based Modeling Framework for Real-time Analysis of Robotic Applications. In: ICRA – SDIR workshop. Saint-Paul, MN, USA (2012)
19. Mallet, A., Pasteur, C., Herrb, M.: GenoM3: Building middleware-independent robotic components. In: ICRA. Anchorage, AK, USA (2010)
20. Nakamura, A., Nagata, K., Harada, K., Yamanobe, N., Tsuji, T., Foissotte, T., Kawai, Y.: Error recovery using task stratification and error classification for manipulation robots in various fields. In: IROS. Tokyo, Japan (2013)
21. Pathak, S., Pulina, L., Metta, G., Tacchella, A.: Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub. In: IROS. Tokyo, Japan (2013)
22. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software. Kobe, Japan (2009)
23. Rochange, C., Sainrat, P.: OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS). pp. 35–46. Waidhofen, Austria (2010)
24. Rybski, P., Anderson-Sprecher, P., Huber, D., Niessl, C., Simmons, R.: Sensor fusion for human safety in industrial workcells. In: IROS. Vila Moura, Portugal (2012)
25. Schlegel, C.: Communication Patterns as Key Towards Component-Based Robotics. International Journal of Advanced Robotic Systems 3(1) (2006)
26. Schlegel, C., Steck, A., Brugali, D., Knoll, A.: Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering. In: SIMPAR. Darmstadt, Germany (2010)
27. Soetens, P., Bruyninckx, H.: Realtime hybrid task-based control for robots and machine tools. In: ICRA. Barcelona, Spain (2005)
28. Szyperski, C.: Component Software : Beyond Object-Oriented Programming. Reading, MA : Addison-Wesley (2002)
29. Tamura, Y., Le, P.D., Hitomi, K., Chandrasiri, N.P., Bando, T., Yamashita, A., Asama, H.: Development of pedestrian behavior model taking account of intention. In: IROS. Vila Moura, Portugal (2012)