A self-adaptation framework based on functional knowledge for augmented autonomy in robots

Carlos Hernández [*], **Julita Bermejo-Alonso** [†] **and Ricardo Sanz** [†]

[*]TU Delft Robotics Institute, and [†]UPM Autonomous Systems Laboratory

Submitted to Integrated Computer-Aided Engineering, IOS Press

Robot control software endows robots with advanced capabilities for autonomous operation, such as navigation, object recognition or manipulation, in unstructured and dynamic environments. However, there is a steady need for more robust operation, where robots should perform complex tasks by reliably exploiting these novel capabilities. Mission-level resilience is required in the presence of component faults through failure recovery. To address this challenge, a novel self-adaptation framework based on functional knowledge for augmented autonomy is presented. A metacontroller is integrated on top of the robot control system, and it uses an explicit run-time model of the robot's controller and its mission to adapt to operational changes. The model is grounded on a functional ontology that relates the robot's mission with the robot's architecture, and it is generated during the robot's development from its engineering models. Advantages are discussed from both theoretical and practical viewpoints. An application example in a real autonomous mobile robot is provided. In this example, the generic metacontroller uses the robot's functional model to adapt the control architecture to recover from a sensor failure.

Keywords | autonomy | resilience | self-adaptation | functional modeling

Introduction

Thanks to the continuous advances in computational intelligence, robotics is entering a golden age where smart and flexible robots are being deployed to perform complex missions. Robots are required to be easily re-taskable and deployable in uncertain environments, able to deal with unexpected changes and disturbances.

The research on autonomous robotics has been focusing typically on the development of the base capabilities needed for this advanced behavior, such as path planning, dynamical control, or trajectory tracking Wang et al. (2016). However, future robots should add the overall capability of performing their missions with *dependable autonomy*, handling disruption and recovering after a fault. Robots need *resilience* to recover base capabilities after failure, e.g., an autonomous robot shall recover navigation capability after the failure of a critical sensor.

Current methods to deal with abnormal scenarios rely on fault-tolerant solutions at the component level. However, these cannot account for emergent, systemic failures, for which ad-hoc, case-based solutions are typically hard-coded. This result in solutions that are expensive, non-scalable and hard to maintain. New mechanisms for *self-adaptation* are needed to provide adequate levels of dependability. Moreover, a more general and extensible approach to build them, supported by engineering tools and reusable assets, is required.

As example, consider the increasing demand of more flexible and agile robot solutions in manufacturing and warehouses, where a variety of heterogeneous components and techniques are involved Hernández et al. (2017) (3D object detection and localization, online planning or grasp synthesis), in addition to navigation in the case of mobile manipulators. The increased internal complexity, together with an open environment, leads to a potential increase of run-time emergent failures. Likewise, robots used in catastrophic situations require resilience, since disturbances and unpredictable environments are common. As Murphy describes (Murphy et al., 2016, p.41) "an Unmanned Ground Vehicle (UGV) (in a disaster situation) fails 10 times as frequently as the same robot in a laboratory setting and that UGVs have a mean time between failures (MTBF) in the field of 6-20 hours".

To engineer this dependable autonomy, the main issue is that the knowledge linking the robot design with the mission specification is only in the mind of engineers, not available in the run-time system once it is built. When a problem happens at runtime, this knowledge is necessary to understand why a specific system feature was placed there, to devise an adequate solution or walk-around. Hence, the traceability from requirements to implementation is lost in the development-runtime gap.

To overcome this issue, we consider the formal capture of the robot design (the reified knowledge from the engineers) related to the mission (the requirements traceability information), to support run-time meta-reasoning Russell et al. (1989). The robot will use this knowledge at runtime to reason about its mission, and how its elements and control architecture contribute to its base capabilities to perform it.

This article describes how *augmented autonomy* can be achieved thanks to a *metacontroller* that exploits *explicit functional models* sustained by *ontologies*, resulting in a run-time *self-adaptation* capability. The focus is on *resilience* for base capabilities, by exploiting *functional models* for diagnosis and reconfiguration. The proposed *architectural framework* provides a domain-independent metacontrol solution for component-based control systems. It has been implemented and demonstrated in the control architecture of an autonomous mobile robot, implemented in the Robot Operating System (ROS)Quigley et al. (2009) platform.

The article is organized as follows: Section explains our design principles, compared with previous approaches; Section describes the architectural framework that reifies those principles; Section 4 shows its application in a real autonomous mobile robot. Finally, Section 9 discusses the benefits and limitations of our approach and draws concluding remarks.

Background ideas and related work

This section describes the *design principles* behind the work presented in this article, as compared to similar research on the domain of autonomous systems engineering.

Related work on fault-tolerance and run-time adaptation. Diverse efforts have addressed fault diagnosis and recovery in robotic systems. Fault Tolerant Control (FTC) is used to address faults and perturbations out of the range of operation

About ASLab

The Autonomous Systems Laboratory is a research group focused on the generation of science and technology for robust autonomy. This is a technical property of systems that makes them capable of sustained provision of a particular service even in the presence of major uncertainties in the service demand, the context of service execution and the disturbances that the system may be suffering from outside or inside. Find us at www.aslab.org.

of the controller. Blanke et al. Blanke et al. (2006) have proposed an architecture for autonomous supervision in fault-tolerant controllers. It relies on "analytical redundancy" in the system to replace faulty components with alternatives in the system that can perform a similar role, as alternative to physical redundancy (e.g. redundant sensors). A "supervisor" acts as a resident engineer, using a model of the control system to diagnose any deviation from the expected behavior, and to determine the appropriate corrective reconfiguration. Gehin et al.Gehin et al. (2012) have used functional analysis to obtain a declarative model of the system suitable to develop FTC supervisors. Asato et al. Asato et al. (2016) have proposed a domain-independent fault-detection framework for robot middleware based on a layered architecture. The DyKnow stream reasoning framework by Leng at al. de Leng and Heintz (2016) extends ROS with support for reconfiguration, which opens more opportunities for FTC and knowledge based self-adaptation in ROS-based robotic systems.

Jiang et al. Jiang et al. (2017) have presented a solution based on automated invariant inference and monitoring to detect faults in ROS-based robotic systems. Their monitors could be used to instrument FTC supervisors such as the metacontroller presented here.

NASA's Remote Agent (RA) Muscettola et al. (1998); Rajan et al. (2000) proved that mission flexibility and resilience for spacecraft control can be achieved using a controller that uses declarative models to plan the goals for the flight software, and to identify operation modes and reconfigure its components for failure diagnosis and recovery. The continuation of that work in the NASA's Autonomous Sciencecraft Experiment Chien et al. (2005), successfully flying the Earth Observing One Spacecraft during the 17 years of the mission, confirms the benefits of using the benefits of using explicit models for run-time adaptation, a line that is further explored in the work presented here.

Metacontrol to bridge the development-runtime gap. Traditional systems engineering methods have been designed with a particular and static set of requirements in mind Balmelli et al. (2006). However, modern life-cycle models show that engineering and operation are concurrent activities in long-lived, adaptive, high value systems. Re-engineering tasks will necessarily happen at runtime for adaptive systems.

However, there is usually a big difference between the design models that engineers use to build the technical artifact, and the run-time models that some systems capable to some extent of reflection may use during their operation Blair et al. (2009). In other words, a development-time/run-time gap appears when following traditional systems development methods.

In our view, extending the *use of the design models as run-time self-models* can eliminate this gap, to leverage the full potential of model-driven development Sanz et al. (2009).

Our approach is based on an analogy between control systems and autonomous systems engineering. In the same way that a disturbance invalidates the open-loop control strategy, traditional engineering fails when the system needs to cope with situations not anticipated or considered during design, such as unexpected failures. Our *autonomy loop* solution Hernández et al. (2015) bridges the development-runtime gap with a *metacontroller* that closes a loop on the *mission requirements* to maintain the desired system behavior even in the presence of failures (Fig.). When the system's behavior deviates from the mission, the metacontroller performs a re-design to adapt the controller configuration at runtime.

To execute the appropriate re-design actions, the metacontroller needs the knowledge of the engineers. Our solution proposes to extend the use of engineering models to run-time models exploited by the metacontroller, effectively bridging the gap between the development and run-time stages of the robot life-cycle.

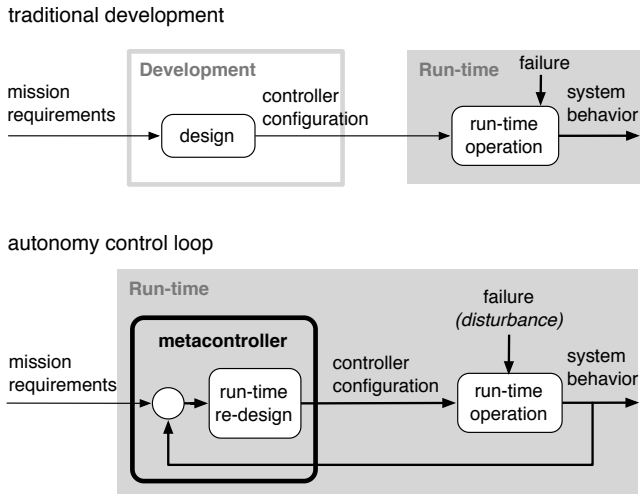

Figure 1 – The autonomy loop involves a metacontroller closing a loop at the mission level to achieve the desired behavior in the presence of failures.

Functional models for run-time adaptation. *Function* is a core concept in the approach described in this article. Functions map the domain of the stakeholder needs with the domain of the system realization. This is what enables addressing mission requirements by means of system reconfiguration. Our objective as designers is for the autonomous robot to provide the functionality required (i.e. displaying a certain behavior), even in the presence of disturbances such as unforeseen environmental conditions or internal emergent failures. The autonomy loop implementation presented here focuses on the internal emergent failures. It takes as reference the mission requirements, i.e. the functions needed. In the presence of a failure, this autonomy loop actuates on the system's structure –i.e. configuration–, to continue providing the required functions. Therefore, our *metacontroller operates in the domain of the functions* of the autonomous system.

Functional modelling addresses the formal representation of the relation between the mission requirements considered to design a system, and its engineered structure that achieves them during run-time operation Lind (1994). The idea is to specify: 1) the intention of the designer and the system's overall goal; 2) the functions that the system must perform to achieve this overall goal, and 3) the interaction between the structural elements to achieve this goal in terms of behavior of the physical structure (as variable interactions and component relationships) Chandrasekaran and Josephson (2000).

The approach described in this article offers a grounding of functional modeling by means of a *functional ontology* Hernández (2013), that conceptualizes the autonomous robots design knowledge. This ontology provides a consistent and shareable description of functional concepts, acting as support for the functional modeling activities involved in engineering such systems.

These functional models are the cornerstone of the *metacontroller*: control loops that exploit the robot's functional models to implement *run-time self-adaptation mechanisms*.

Ontologies to support model-based metacontrol. A knowledge-based approach in which the model used by the metacontroller is explicit, i.e. separated from the reasoning engine, allows the reuse of the reasoning software across applications. Knowledge-based control comprises general processes perception, understanding (i.e. evaluation, reasoning, prediction) and action Sanz et al. (2014); Hernández et al. (2013) around the run-time model.

The development and exploitation of explicit models that capture functional knowledge for our metacontrol solution is supported by ontologies. Schmill et al. Schmill et al. (2007) proposed a metacognitive loop (MCL) on top of the system that performs automated reasoning, supported by ontologies, to cope with failures. Alvares et al. Alvares et al. (2017) showed how domain-specific languages can support knowledge-based self-adaptive components, for improved base capability robustness.

An ontology defines a set of ontological elements as representational primitives that can be used to model a domain of knowledge Gruber (2009); Guarino et al. (2009). Ontologies have been used to represent the knowledge the robot need to perform its tasks or to interact with humans. As example, KnowRob Tenorth and Beetz (2013). It is a knowledge-based framework, where a set of ontologies provide a common vocabulary about robot actions, event, objects, etc Tenorth and Beetz (2015). A core ontology for robotics and automation has been published as a standard by the IEEE Standards Association IEEE Standards Working Group Ontologies for Robotics and Automation ORA (2015), to be used as a reference for knowledge representation and reasoning in robots. This core ontology is to be extended with robot task representation Balakirsky et al. (2017) and autonomous robot features Bayat et al. (2016), Fiorini et al. (2017).

Our approach using ontologies is twofold. Firstly, the robot models (aka knowledge to behave) can be obtained in a consistent, meaningful and shareable way using the ontologies developed to describe the system. Secondly, the ontologies serve as support to obtain the engineering models for the metacontroller.

Models constitute one of the main components of our knowledge-based strategy for autonomy. Knowledge in the form of *explicit models supported by ontologies and metamodels* grounds advanced and robust robot capabilities. Having an autonomous systems reference ontology Bermejo-Alonso et al. (2011) and related patterns becomes a significant advantage to obtain the autonomous system's models Bermejo-Alonso et al. (2016), compared to developing them in an ad-hoc or case by case manner.

Moreover, *ontologies guide the flow of the knowledge through the robot life-cycle and engineering* Sanz et al. (2017). They constitute analysis metamodels that define the different entities and relationships participating in the autonomous robot engineering and its operation.

We extend the use of ontologies and metamodels not only to develop the system design models, but also to support the model-driven engineering (MDE) of the robotic system. This *MDE process produces the model used by the robot at runtime through a model-to-model transformation from the engineering model.* This transformation is captured in the *Deep Model Reflection pattern*, saving efforts and reducing errors by automating the building of models. The run-time model must conform to a metamodel which is part of the model transformation definition. This shared metamodel between the engineering and run-time phases is what the functional ontology provides Hernández et al. (2013).

Model-based functional metacontrol. The former ideas are reified in the *Metacontrol Design pattern* Hernández et al. (2013), that splits the control system in two (see central part of Fig. 6). The standard *domain controller* is responsible for sensing, computing and acting on the robot to achieve a target reference that is typically the value of a variable in the robot's domain of operation, e.g. a position, a velocity, a trajectory. The *metacontroller* controls the former through an interface provided by its implementation component platform. The metacontroller's references are the system's mission requirements, which belong to the mission's domain and not to the robot's domain. This metacontroller follows the *Functional Metacontrol pattern* to target functional aspects. This pattern defines a layered structure for the metacontroller consisting of two nested loops (see OM Metacontroller in Fig. 6): 1) the *Components Loop* controls the configuration of the components of the controller; 2) the *Functional Loop* controls the performed functions. Hence, the functional and structural concerns are explicitly represented.

Our metacontroller is an FTC supervisor. However, it uses knowledge about *how its inner organization supports the mission*, to provide analytic fault-tolerance at the mission-level. This is above the system realization level that most FTC systems do offer Rodríguez et al. (2013). Likewise, while redundancy-based fault-tolerance mechanisms usually keep system organization, ours tries to *keep system function.* Our metacontroller rejects disturbances modifying the robot controllers to maintain its function from a mission perspective. If there is enough analytical redundancy in the robot and its controller, the metacontroller can exploit multiple options.

The framework presented in this article also follows similar ideas to NASA's RA, but offers a mechanism that explicitly addresses the match between the system and the mission by exploiting *functional models.* This explicitness enables the achievement of the objective of domain and mission neutrality for the metacontrol architecture.

An architectural framework for augmented autonomy robot control

Our approach to develop improved autonomy controllers for robots is architecture-centric and pattern-based. Focusing on the system's architecture is focusing on the structural properties that constitute the more pervasive and stable properties of the system. Architectural aspects are what critically determine the final capabilities of any information processing technology, such as robotic systems.

Functional system models are the cornerstone of the process, serving both as assets for model-based engineering and knowledge bases for cognitive control of the robot. This can be achieved because knowledge about the relation between the mission goals and the robot components and base capabilities are explicitly captured in the model. In this software-intensive approach, ontologies and metamodels act as backbones, to develop the system models in a consistent, meaningful and shareable way.

The *Operative Mind (OM) Architectural Framework* is the specific architectural solution that combines and integrates the design principles described in Sec. .

These design principles have been reified as a set of elements at different levels, to ensure general applicability, regardless of the application domain and the implementation technology:

- At the modeling level: a metamodel to specify the functional model of autonomous systems. This *functional*

metamodel captures both the system's functional *specification at design time and its runtime realization* in the control components.

- At the run-time operation level: a *reference architecture* for robot control with a *metacontroller* for augmented autonomy, following the metacontrol and functional metacontrol *patterns*.
- At the engineering process level: the implementation of the deep model reflection pattern by defining an *MDE process* to obtain the run-time functional model that the metacontroller exploits for self-adaptation.

TOMASys. The *Teleological and Ontological Model for Autonomous Systems (TOMASys)* is the metamodel developed to provide the concepts for modeling the functional knowledge of autonomous systems. It constitutes thus a functional ontology, and it is based on the Ontology for Autonomous Systems (OASys) Bermejo-Alonso et al. (2013, 2011). The TOMASys model of a robotic system allows automated reasoning on how the current robot's controller is achieving the mission objectives, so that a metacontroller can decide appropriate reconfiguration actions in an analogous way to how an engineer would do.

TOMASys's concepts (Fig. 3) have been specified using a UML-based notation, where each element is captured as a class with a set of properties (and relations). A complete specification can be found in Hernández (2013).

TOMASys captures not only functional matters at design-time as other functional modelling frameworks do, but also the instantiation of functions as component configurations during run-time operation. A TOMASys **Function**[1] represents a capability that has been designed in the system, for example `Localization` in an autonomous mobile robot (see Fig. 4), or `Sense 2D Obstacles`. At design-time, engineers typically create solutions for the robot's capabilities. These are modelled in TOMASys as **Function Designs**, and there can be several built-in the robot architecture for the same function/capability. Internally, a function design prescribes a certain structure that **solves** the function, i.e. it maps functionally to system structure, through **specifications** of components and their interconnection. For example to **solve** localization an Adaptive Monte Carlo (`amcl`) algorithm can be used with odometry information (`odom`), sensed 2D obstacles (`2D obs`) and a `map` as in the function design `Localization V1` in Fig. 4. This solution requires accurate information of 2D obstacles in a wide range, which is modelled in the specific `accu. 2D obst` objective instance of the sense 2D obstacles function/capability, and solves the localization function with maximum reliability (**confidence** = 1). An alternative that requires a less restrictive sense 2D obstacles objective is `Localization V2`, which integrates the odometry information (`odom`) with orientation from a digital `compass` through an extended Kalman filter (`ekf`); but this design is less reliable (**confidence** = 0.9). To solve the `sense 2D obstacles` function one possible design (`Scan_V1` in Fig. 4) is using a laser range scan sensor.

Part of TOMASys elements constitute a component model that represents both the static *design-time knowledge* about the system components, and the *instantaneous run-time information* of the their state. The design knowledge about the robot's components and their properties (such as fault behavior) is captured through **Component Classes**, their **Parameter Profiles** and **internal failure models** (**ifm()**), amongst other TOMASys elements (see `laser` and `amcl` classes in Fig. 4). The instantaneous state of these instances is captured by **Component States**, containing information about their internal state and specific configuration

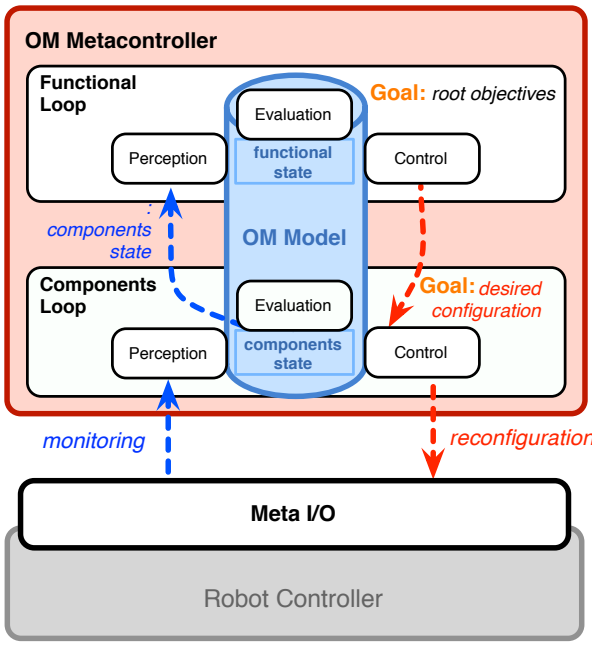

Figure 2 – The main elements of the internal structure of the OM Metacontroller.

for the mission, e.g. **Parameter** values. For example, in Fig. 4 `amcl_node` and `sicklms` capture the run-time information about the instances of `amcl` and `laser`.

A capability demanded at runtime by the robot's operation is represented by an **Objective** of the type of the function that represents that capability. This concrete instance can include specific requirements, e.g. a certain accuracy for the localization, or range and density for the scan of sensed 2D obstacles. The instance of the function design that is realised at runtime to achieve the objective is a **Function Grounding**, and it binds the roles defined by the specifications of the function design to the runtime components that realize them (see function groundings `A` and `B` in Fig. 4).

The OM Architecture. The *OM Architecture* is a reference architecture for the development metacontrollers to augment robot autonomy through self-adaptation. The OM Architecture defines the operation and structure of such a metacontroller and its integration with the robot controller, as proposed by the Metacontrol principle discussed in Sec. . The main elements of the OM Architecture are: i) the **OM Model**, which is model of the functional architecture of the robot controller, together with an instantaneous estimation of its structural and functional status, all specified with TOMASys, and ii) the **OM Metacontroller** which exploits the OM Model to diagnose and reconfigure –if necessary– the robot control architecture.

The OM Metacontroller is organized as a two-layered controller (see Fig. 6) that follows the Functional Metacontrol pattern explained in Sec. .

The lower **Components Loop** is continuously *monitoring* the status of the components of the robot controller using the reflection mechanisms available in its implementa-

[1] Note about notation: general elements of the framework are indicated in **bold** font upon first appearance, whereas specific instances in the mobile robot example are indicated with `courier`.

[2] This means that the improved resilience mechanisms described in this article can only be deployed over controllers built upon an infrastructure that provides reflection. This is a minimal but strong requirement that is however widely fulfilled by modern software frameworks.

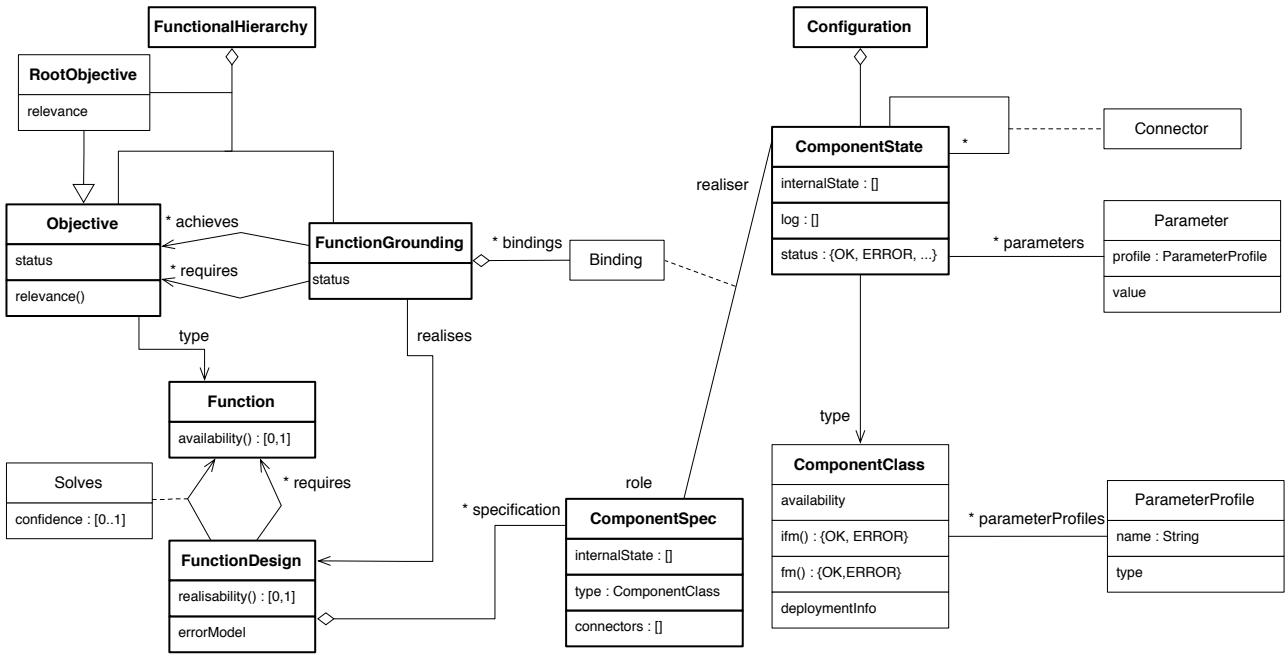

Figure 3 – The main elements in TOMASys that capture functional/structural and design/runtime information of an autonomous system.

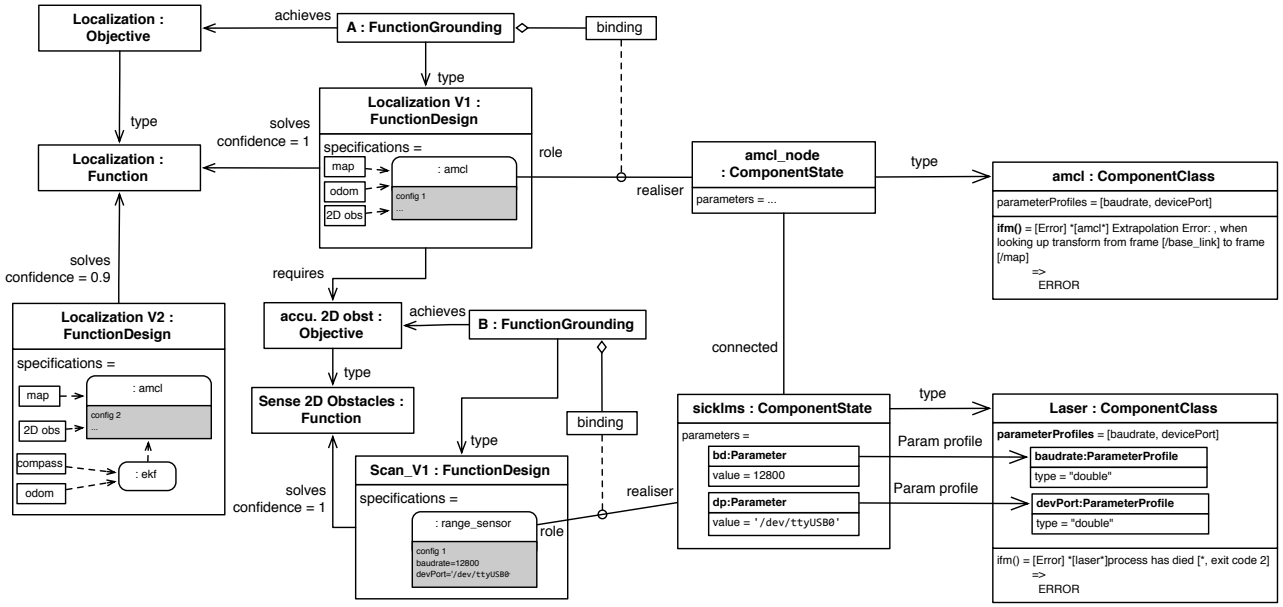

Figure 4 – Example of the TOMASys modelling of different functions and components of a mobile robot.

tion platform. The estimation of the current state of the components (**configuration**) is evaluated against the desired configuration (the reference goal for this loop), to determine any required *reconfiguration* action, executed again through the available reflection mechanisms[2]. Reconfiguration actions may include activation and de-activation of components, reconfiguration of their parameters, or re-connecting components.

If the reconfiguration actions do not succeed achieving the desired configuration, the OM Metacontroller operation escalates to the upper **Functional Loop**. The unsuccessful evaluation of the current *configuration* is used to update the **functional hierarchy**, up to the topmost objectives. This update is performed using the function designs contained in the **OM Model**, which define the requirement to achieve each objective. When any of the topmost objectives is not achieved, the OM Metacontroller looks for alternative function designs that are currently realisable. It seeks component classes available to fill in the roles defined by their specifications. A new functional hierarchy is obtained by computing all the function groundings and objectives that realise the selected function designs. Finally, all the new component specifications are

[3] OMJava is available as open source at

gathered in the new *desired configuration* sent to the Components Loop.

A complete description of the OM Metacontroller operation is provided in Hernández (2013), and the relevant details of its operation are described along the example case in Sec. 4.

The OM Engineering Process. The *OM Engineering Process (OMEP)* is the method proposed to develop the metacontroller for autonomous robots with the OM Architecture. OMEP divides the development of an autonomous robot application in two main activities (see left side of Fig. 6):

Control Development refers to the development of the robot's domain controller, developing the base capabilities required by the mission (e.g. task planning, navigation, motion, vision, etc.), for which state of the art techniques in robotics shall be used.

OMEP differs from other robot developments in the definition of *alternative solutions* for at least some of the capabilities, for the sake of analytical redundancy. Each alternative design for a capability is a variance point in the spectrum of possible functional architectures of the system, so that the total number of alternatives for the control architecture of the complete robot is the product of the number of alternatives for each capability. OMEP will thus produce a model for these architectural alternatives.

This model will be converted in the OM Model, so it has to be captured with a conceptualization that can be mapped to TOMASys elements. Applying Model-Based Systems Engineering with the OASys-driven Engineering MethodologyBermejo-Alonso et al. (2016), which is based on the same underlying ontology than TOMASys, facilitates this.

Metacontrol Development is the process to create the metacontroller and integrate it into the robot control architecture. Thanks to the model-based approach, this process consists on instantiating the OM Metacontroller, which can be reused across robot systems an applications, and creating the OM Model of the autonomous robot application.

For the creation of the **OM Metacontroller**, different elements were developed through the progressive platform-specific refinement by applying model weaving Assmann et al. (2006) (Fig. 4). This allows for the maximum reusability. The first step was to design the OM Architecture, a platform-independent solution, from TOMASys, which is a computation independent model. Then, from the OM Architecture a generic **OMJava** library was developed to implement OM-based metacontrollers[3]. Java was selected in order to provide a portable implementation, so that the **OMJava OMmetacontroller** can be easily integrated in the specific platform of the robot's domain controller. In the final refinement, the platform specific model has to be integrated in the platform of the robot control architecture (e.g. ROS). The details about this final integration are given in Sec. 4.

The OM Model is obtained by first parsing the functional model of the robot control into a TOMASys formulation, and then implementing it into the run-time executable OM Model by using the classes in **OMJava. metacontrol.knowledge**.

Example case: fault-tolerant mobile robot

The OM Architecture has been tested in the control architecture of an autonomous mobile robot, to improve its resilience to failures through the capability of reconfiguring its control architecture.

Autonomous navigation is representative of the current challenges in the manufacturing sector, where autonomous mobile solutions are envisioned for intralogistics, and mobile robotic platforms are also explored for mobile manipulation. These challenges encompass both operation in an open environment and internal complexity, thus offering opportunities to explore robustness to both external and internal disturbances.

The application involved a real mobile robot moving autonomously in an unstructured environment. The robot had to navigate in the Autonomous System Laboratory following a set of waypoints (see the plant in Fig. 6) to accomplish a surveillance mission. The robot consisted of a differential Pioneer 2AT8 platform that has internal encoders for odometry, additionally equipped with a laser sensor, a 3D Kinect camera and an electronic compass.

The goal was to implement and test a generic metacontroller capable of agilely adapting the robot control architecture to unforeseen events, such as a sensor failure, which could result in a critical failure in some of the base capabilities needed for the mission. Experiments involved different failure scenarios in simulation and with the real robot.

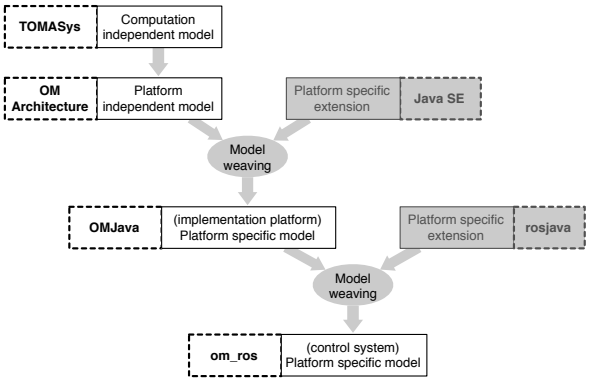

Figure 5 – The different elements developed in this work for ROS-based robots by refining MDE.

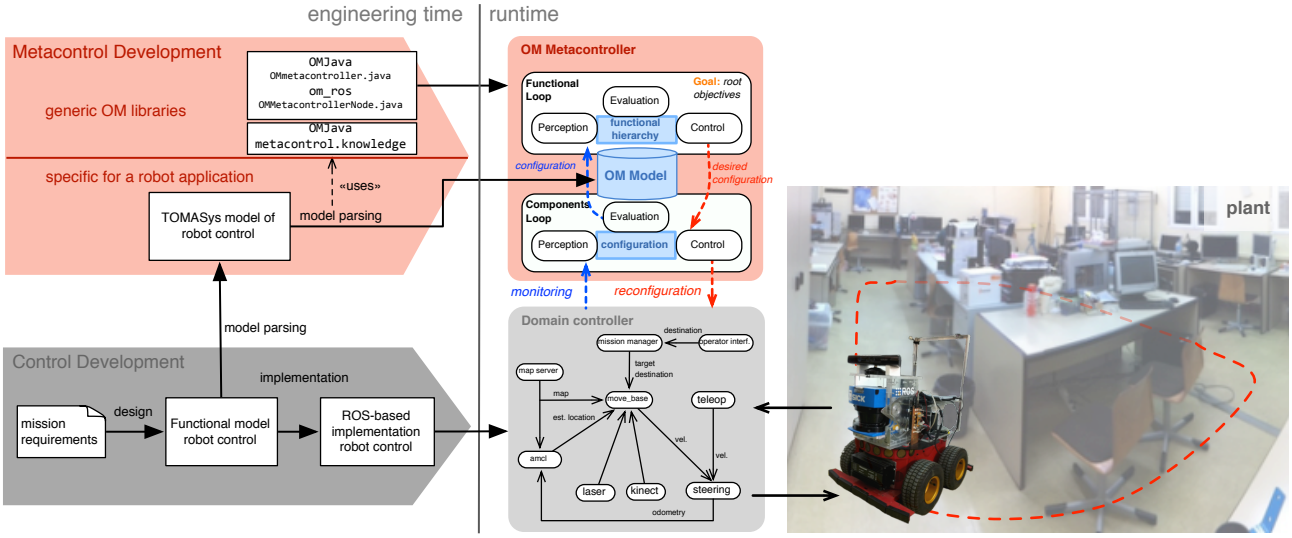

Figure 6 – Schematic overview of the application of the OM Architectural Framework following OMEP (left part), to develop the OM Metacontroller (central part) for an autonomous mobile robot resilient to failures (right side).

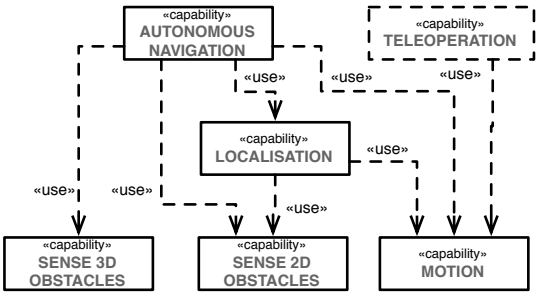

Figure 7 – Reduced version of the functional breakdown of the mobile robot.

Mobile robot control development. Following the OM Engineering Process, the robot control architecture was designed and modeled using the ISE&PPOOA methodology Fernández et al. (2016) for Model-Based Systems Engineering. Fig. 4 shows a simplified version of the base capabilities in the mobile robot. For the sake of simplicity, in this paper the focus is on the robot control architecture to achieve the *autonomous navigation* capability. The solution implemented is based on that of Marder-Eppstein et al. Marder-Eppstein et al. (2010) publicly available as a ROS open-source library. ROS is a component-based platform for robotics, so any robot controller developed with it is automatically suitable for the OM Architectural Framework.

Following OMEP guidelines, analytical redundancy was added by defining alternative function designs for the `Localization` and `Navigation` capabilities, using different components and configurations to use the available sensory information. Fig. 8 shows two alternative architectures, with their functional breakdowns and the configuration of components (represented as blobs) that realize the different functions (represented as rectangular areas).

In Architecture 1 (left part of Fig. 8), the laser readings are used by the `Scan V1` function design to achieve the capability/function `sense 2D Obstacles`. The density of the obstacle information provided by the laser implementation allows a good performance of the `Localization V1` solution for the `Localization` function that uses directly the ded reckoning information provided by the `robot driver` component. The `Navigation V1` design uses dense 2D Obstacles and sparse 3D Obstacles, and a high speed scale factor for the velocity commands.

In Architecture 2 (right part of Fig. 8), the laser sensor is not used. Instead `Localization V2` uses more sparse and noisy 2D obstacle sensing provided by `Scan V2`, which uses the Kinect and a conversion from PointCloud to range scan readings to `Sense 2D Obstacles`, compensated with more precise

Table 1 – Average time and standard deviation for the robot to navigate the route and reconfigure its control architecture over 6 trials for each of the following scenarios: i) using Architecture 1, ii) using Architecture 2, iii) when the metacontrol adapts the architecture from 1 to 2 to recover a permanent laser failure.

Scenario	Navigation (avg. & σ)	reconf. (avg. & σ)
Arch. 1	177.2 ± 26.4 s	
Arch. 2	379.0 ± 101.3 s	
metacontrol	256.2 ± 49.7 s	11.4 ± 0.7 s

[4] **om_ros** is available as open source at

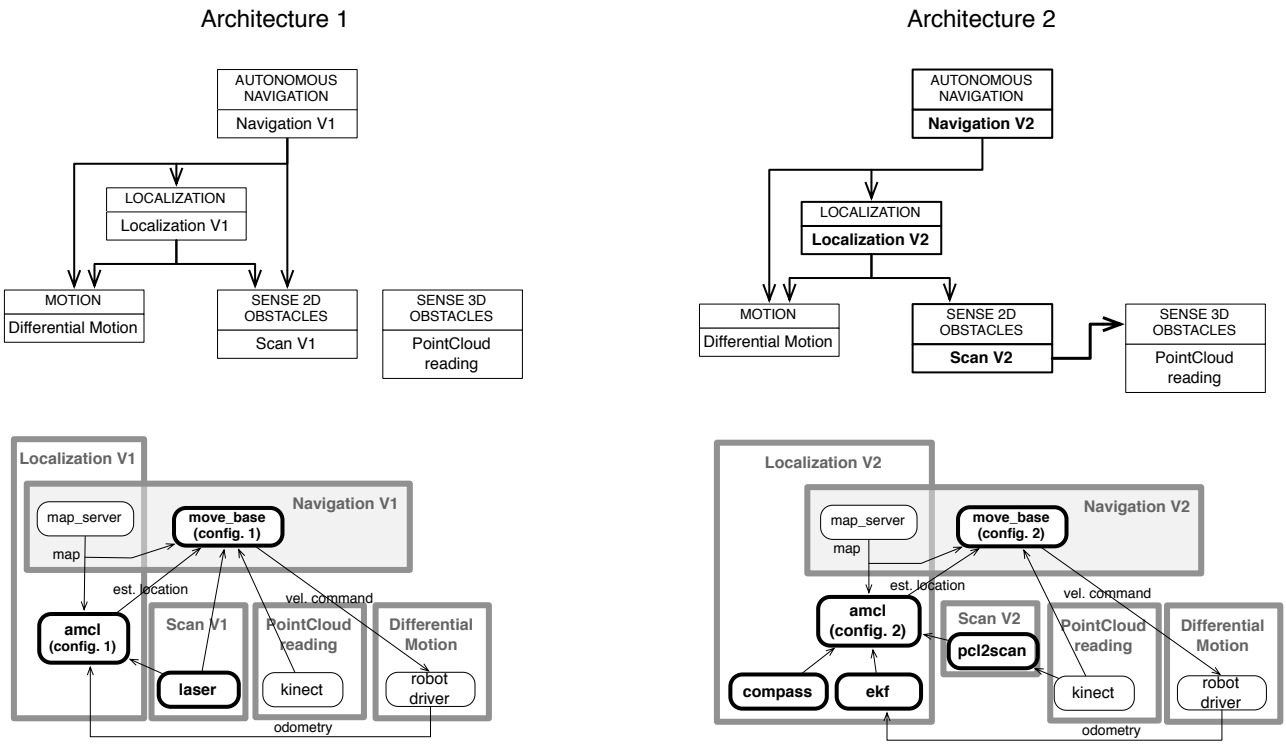

Figure 8 – Alternative architectures for the mobile robot's domain controller.

orientation information by integrating the compass readings with the odometry from the robot driver through a Kalman filter (implemented by the `ekf` component). `Navigation V2` uses a lower speed scaling factor (configuration 2 of the `move_base` component) to compensate for the lower performance of `Localization V2` and the more sparse obstacle information using only the 3D information from the Kinect, which result in slower robot motions than Architecture 1 (see Table 4), but accomplishes the objective of autonomous navigation without using the laser sensor.

Mobile robot metacontroller development. The Metacontrol Development for the mobile robot required the integration of the (platform independent model of the) metacontroller in the ROS platform of the robot, and the creation of the OM Model.

The **om_ros**[4] library was developed to integrate the OMJava OMmetacontroller in any ROS-based application. Its **OMMetacontrollerNode.java** class wraps the OMJava OMmetacontroller.java as a ROS node, and the **meta_sensor** and **meta_actuator** ROS nodes implement the monitoring and reconfiguration mechanism using the introspection services available in ROS.

Thanks to our model-based approach, the only Metacontrol Development effort specific to the mobile robot case was the creation of the the OM Model. For this purpose, first the TOMASys model was manually obtained from the functional architecture of the robot that was modelled during the Control Development phase. Then the OM Model was implemented by using the classes in OMJava.metacontrol.knowledge package.

Experimental Results. The capabilities of the proposed metacontroller were tested in different experiments in which the robot controller had to adapt to unforeseen events, such as simulated failures in different components, and in particular a sensor failure in the real robot.

Here, two of these fault-tolerance scenarios are described. One consists of a transient failure due to an error in the laser driver, to demonstrate fault-tolerance at the component level. The second scenario involves a permanent failure of the laser, and demonstrates mission resilience by reconfiguring the robot control architecture to overcome the problem.

In the mobile robot control Architecture 1 (see Fig. 8), the `laser` component is used to obtain 2D obstacle information, objective needed by `Navigation V1` to achieve `Autonomous navigation`, and by `Localization V2` to achieve the `Localization` objective.

Component-level resilience scenario

In this scenario, the metacontroller provides for *standard fault-tolerance at the component level*. Initially the robot is in normal operation with the OM Metacontroller maintaining the functional hierarchy corresponding to Architecture 1, since it is the most performing to address the mission objectives. This means that the estimated component state in the OM Metacontroller corresponds to the configuration of components that realises that hierarchy.

In this situation, a software failure of the ROS laser driver occurs. The bottom part of Fig. 9 shows the main reasoning steps and the OM Model elements involved in the metacontrol operation. The failure is detected by the `meta_sensor` node that monitors the log messages produced by the control components in the ROS system. The operation of the `meta_sensor` node is driven by the internal failure modes of the robot components captured in the component classes of the OM Model, which for ROS components consist of patterns in the log reported by the components.

At the Components Loop, this monitoring information is used to update the components state. The subsequent error

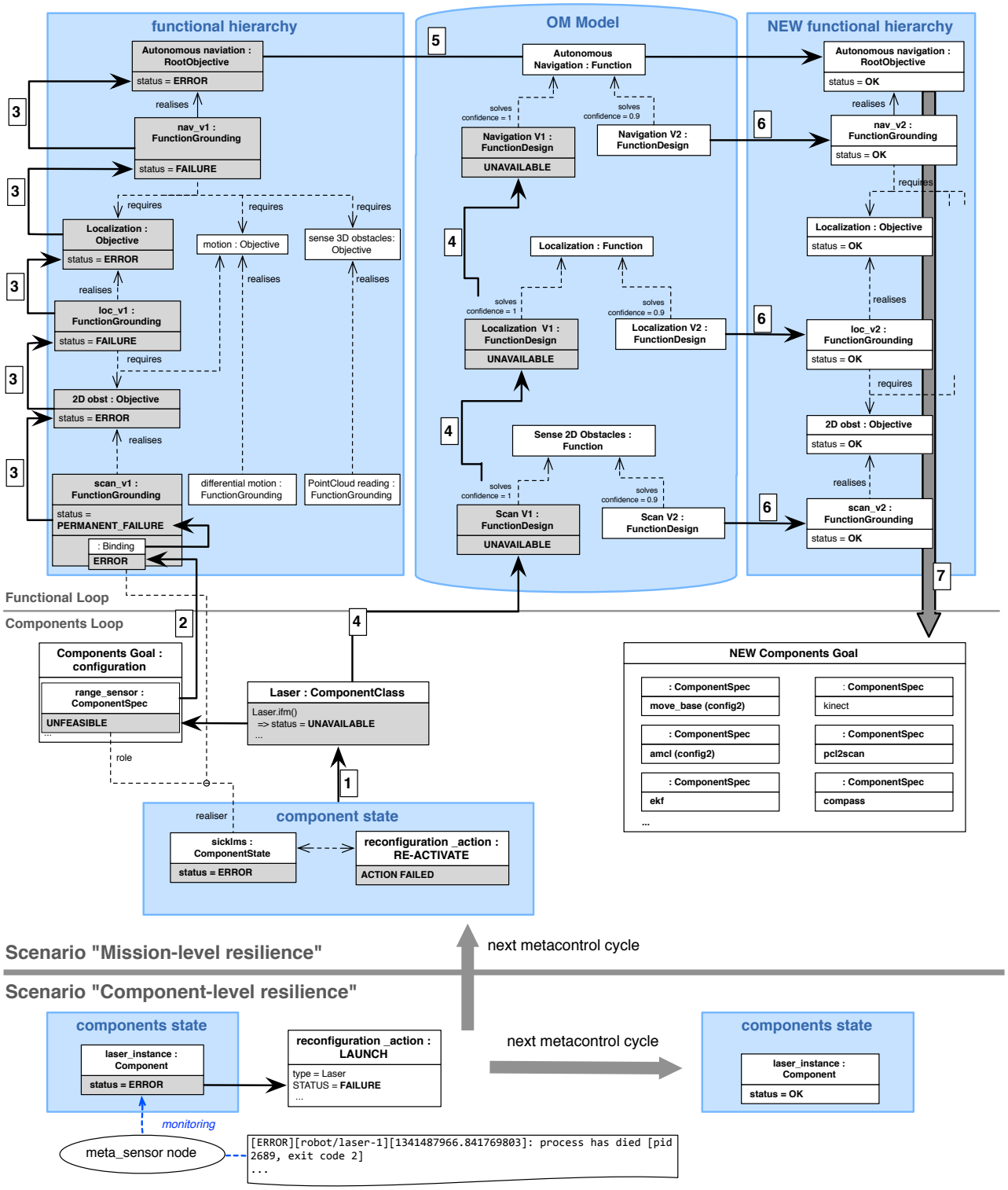

Figure 9 – Schematic representation of the OM Metacontroller operation for the two reconfiguration scenarios of the mobile robot, indicating the main reasoning steps and the elements of the OM Model involved. Elements for which an error state is diagnosed are depicted in gray.

status of the `laser` component is evaluated using the failure model of the `laser` component class, and the OM Metacontroller uses it to and decide a `re-start` reconfiguration action to recover from the error. This reconfiguration is executed by the meta_actuator node, which restart the ROS laser driver. The metacontroller process runs at 0.3Hz for the mobile robot case, to guarantee that any reconfiguration action is seen as instantaneous to the metacontroller. This metacontrol frequency has to be determined for each robotic application.

Since the laser failure is due to a transient fault, the laser recovers its normal operation, and so do the Localization and Navigation functions, so that the robot can continue performing its mission. It is important to remark here that in the example, the components in the robot architecture for navigation are loosely-coupled. This means that instantaneous (according to the metacontrol frequency) interruption of components does not critically affect the behavior of the robot, apart from a resulting temporary pause, in the pursue of its mission. For other applications it might be necessary to take additional measures upon reconfiguration, such as putting the system or specific components in a safe state. These requirements, and associated metacontrol actions, can be added to the OM Model, either at the component level or at the functional level as required, making use of the corresponding internal failure models for component, or errorModels for function designs.

One of the benefits of our metacontrol approach over ad-hoc fault-tolerance mechanisms is that it is easily scalable. To include fault-tolerance mechanisms for a new component, or to incorporate the new failure information about an existing component, all that is needed is to update the OM Model of the robot control architecture, by respectively updating to the new failure models in the component classes, or including the new component classes in the model; no changes are needed in the OM Metacontrol.

Mission-level resilience scenario

In this scenario, the metacontroller provides for *mission-level resilience by dynamically reconfiguring the robot control architecture*. The initial situation is the same than in the previous case, with Architecture 2 deployed in the robot. In this scenario, though, the laser sensor becomes permanently unavailable due to a permanent fault (e.g. due to physical damage).

For the OM Metacontroller, this scenario is an extension of the previous one, as shown in the upper part of Fig. 9. The reconfiguration action described before does not solve the problem and further metacontrol action ensues.

In the next metacontrol cycle at the components loop, the monitoring information reports the failure of the reconfiguration action to `re-activate` the laser driver component. This failure implies, according to the simple internal failure model of the `Laser` component class, the unavailability state of that component (arrow 1). The OM Metacontroller evaluation of the current desired configuration uses the general knowledge contained in TOMASys to determine that the Component Specification for a laser component in the system is `unfeasible`, since the component class is unavailable.

The subsequent unfeasibility of the goal of the Components Loop is therefore escalated to the Functional Loop, as the role required for the `Scan V1` function has no realiser (arrow 2). In general, when the Components Goal becomes unfeasible, meaning that the desired configuration is not recoverable by the Components Loop, the problem escalates to the Functional Loop as binding errors for those functions in which the corresponding roles become unfulfilled.

Then, the evaluation process updates the state of the robot functional hierarchy up to the topmost Objectives (arrows 3). It is a reasoning process that follows the functional dependencies bottom-up, using the error models for each function to determine the state of the function groundings, and the general model included in TOMASys that an objective is in error if its realising function grounding is in a failure status. The default errorModel for functions in TOMASys state that function groundings are in PERMANENT_FAILURE status if one of their required roles has no available realiser (the case for the `Scan V1` function), or in FAILURE status if one of their depending objectives are in ERROR (the case for the rest of the function groundings in the mobile robot hierarchy). Note that specific error models can be included in the OM Model of the robot application, by incorporating them to the corresponding functions designs. As a result of the evaluation of the functional hierarchy, a critical ERROR state for the autonomous navigation topmost objective is determined.

In addition to the evaluation of the functional hierarchy (instantaneous information), the general knowledge about the functions availability is also updated in the robot OM Model using the error models. In this case (arrows 4), the unavailability of the laser component determines that the function design `Scan V1` is unavailable, and the reasoning proceeds following the required dependencies amongst function designs. In this case, only the function design corresponding the current grounding in the functional hierarchy become unavailable, but it could be the case that some additional possible function designs, not currently grounded, would be updated to unavailable, directly due to a missing component class required, or indirectly due to some function being unavailable.

To solve the unmet autonomous navigation objective, the OM Metacontroller computes an alternative functional hierarchy for it (arrow 5) by selecting and grounding function designs (arrows 6). The selection is first based on their availability, and then on their performance, represented by the confidence to solve their respective functional objectives. This confidence is part of the OM Model, and can be tuned for an application based on experimental results. Gathering all the component specifications to fulfill the required roles, a new desired configuration for the components in the robot controller is obtained, and send as the new goal to the Components Loop (arrow 7). Back at the component level, the OM Metacontroller executes the component reconfiguration action required to obtain the desired components configuration. Note that some of these actions, such as re-activating a component, can require additional knowledge for the metacontroller, e.g. an internal state in the component to recover. A basic internal state is already supported in TOMASys Component Class, which is not described here for simplicity.

As a result, the OM Metacontroller reconfigures the robot control architecture grounding the functional hierarchy that corresponds to Architecture 2 in Fig. 8, which uses the Kinect camera both to `sense 3D obstacles` for the autonomous navigation function, and to `sense 2D obstacles` for the localization function, and incorporates a Kalman filter to improve the accuracy of the odometry information with the orientation information from the compass.

Discussion

The experiments with the mobile robot show the feasibility and benefits of the metacontrol solution for enhanced resilience in autonomous robots proposed here.

Benefits of the approach. Given the generic model-based OM Metacontroller developed, the development effort to enable the self-adaptations required to recover from a failing laser sensor was limited to creating the OM Model of the robot control system. The same OM Model also provides from component-failure recovery behavior in any of the other components of the system, thanks to the generic default failure model included in TOMASys. The Components Loop in the metacontroller is easily scalable for improved fault-tolerance, e.g. with new knowledge from failure analysis of the robot's components. All that is needed is to update the component

classes in the robot's OM Model. This is a clear advantage over hard-coded fault-tolerance methods.

The main current benefit of the OM Architectural Framework is to provide capability resilience at the mission level, when the autonomous system is faced with unforeseen internal emergent failures. In the mobile robot case, to enable resilience for the capabilities affected by the laser sensor, alternative solutions had to be developed for them. This development consisted of: i) designing additional architectural configurations to realise the sense 2D obstacles, localization and autonomous navigation functions, which mainly involved different component's configuration and connection of additional available components (compass, ekf), ii) including those alternatives in the OM Model, by encoding the corresponding TOMASys function designs and component classes. Note that no overall architectural solutions where specified, only partial designs for different functions in the system. The overall final architecture after reconfiguration is the result of the OM Metacontroller operation to achieve the topmost objectives. The metamodelling approach for the run-time reconfiguration knowledge, with the OM Model based on the TOMASys metamodel, makes the metacontrol solution for the mobile robot immediately scalable. If new capabilities were added to the robot, e.g. object recognition, enabling component fault-tolerance simply requires defining the component classes and failure model for the new elements in the system, whereas capability resilience can be enabled by creating alternative designs and defining them by extending the OM Model of the system with new function designs. Note that all these modifications reuse the initial OM Model of the mobile robot. Component classes and function designs can be reused for other missions, and even for other robots, that use the same control components. More importantly, the same OM Metacontroller can be used in any other cases, since it is a general reasoning engine. All the metacontrol development effort is limited to creating the OM Model of the autonomous robot application.

Limitations of the current solution. We discuss here the limitations identified so far in the use of the OM Architectural Framework. One is that TOMASys only captures static considerations about the systems capabilities, and not dynamic considerations: temporary objectives, and how availability of functions impact mission planning/re-scheduling. Adding the temporal dimension in the metamodel would allow the framework to also address mission scheduling issues, tackling dynamic objectives. More critically, the OM Metacontroller does not account for the temporal nature of the reconfiguration, discretizing the adaptation process to 3 steps: failure, reconfiguration (during which domain operation is paused) and return to normal operation.

Another current limitation of the OM Architectural Framework is that it only addresses unforeseen situation arising from internal emergent failures. However, disturbances due to unexpected changes in the environment are not yet considered. An example could be smoke rendering a camera in a robot useless. This external disturbance will not be properly identified with the current features of the framework, preventing the metacontroller from adjusting to the new navigation conditions. As a consequence, the robot would not detect obstacles and keep bumping into them. In Hernández (2013) a solution was already drafted, consisting of adding observer mechanisms to update of the status of the objectives –e.g. obtain RGB images— in the functional hierarchy. Frequently, such a mechanism is already available in the components of the system –e.g. some camera drivers detect smoke–. An alternative to this direct observation is to build a perception model of the status of the objective that uses other related information in the system. For example, a perception model could be added to the TOMASys model of the 'obtain RGB image' objective to detect when an image does not have sufficient quality.

The assignment of components to functions at runtime is currently a challenge in situation more complex than one-to-one. TOMASys metamodel does not specify any cardinality for the bindings between functions and components. This means that it is possible in principle to have several allocations of the same component to different functions. It can be the case that a component cannot be assigned to more than 'n' functions, and/or that some rules may apply depending on the operational context for the assignments. For example, a while a camera can perform a role in different functions, a gripper can generally be used for one function at a time. Currently TOMASys does not model cardinality rules for component bindings, but it is possible to extend it to do so, for example through a 'cardinality' property of the ComponentClass, and 'cardinality constraints' in the Roles of the Function Design.

Finally, a restriction of the proposed framework is that it is only applicable to component-based systems, which can be modelled with TOMASys. However, advanced robotic applications are componentized Brugali and Scandurra (2009), and the OM Architectural Framework structure separating functional and structural concerns allows to reuse the functional elements for the metacontrol of control architectures whose structure is not currently suitable for the OM Architectural Framework presented in this article. TOMASys structure allows to modify its component model while reusing the functional elements, and while that change in TOMASys component model requires a re-implementation of the Components Loop in the OM Metacontroller, the preservation of the functional elements allows to reuse the Functional Loop for the new metacontroller.

Concluding remarks. Achieving high levels of autonomy for robots requires complex controllers that can provide sophisticated fundamental capabilities. But this is not enough. Autonomous robots must also be able to be resilient and recover after failure affecting their capabilities. This article has described a model-based approach that can support the engineering of metacontrollers for robots to improve their autonomy concerning resilience. The approach has been reified in the OM Architectural Framework, whose essential elements are i) a metamodel that underlies the construction of a functional model of the robotic system under control; ii) a reference architecture to implement the dual component/function metacontrol strategy that performs mission-oriented reconfiguration of the robot; and iii) reusable assets to integrate these metacontrollers in ROS-based autonomous robots. This methodology and elements have been demonstrated with a mobile robot that shows a capability for reconfiguration without ad-hoc mechanisms.

The OM Architectural Framework has three characteristics that make it especially interesting:

- It is scalable, i.e. new missions or components in the robot only can be accounted for by extending the functional model, without changes in the metacontroller (i.e. the shelf-adaptation mechanisms).
- It is unified, i.e. it provides support for fault-tolerance at the component level and resilience at the mission level using a single architecture for metacontrol.
- It is universal, i.e. it can be applied to any kind of system that fulfills a minimal set of requirements (in essence, be componentized and both observable and controlable, e.g. through reflection mechanisms).

Unification and universality are desirable properties for any kind of technology. It is considered that they set the foundation for any future theory of autonomous systems and any engineering methodology based on it.

Acknowledgments

This research has been supported by the Spanish Ministry of Education and Science through FPU grant AP2006-02778. The authors also gratefully acknowledge the financial support by the European Union through Seventh Framework Programme's projects ICEA (IST-027819-IP), HUMANOBS (contract no. FP7-STREP-231453), and Factory-in-a-day (grant no. FP7-609206), and HORIZON 2020 Programme's project ROSIN (grant no. 732287)

References

Alvares, F., Rutten, E., and Seinturier, L. (2017). A domain-specific language for the control of self-adaptive component-based architecture. *Journal of Systems and Software*, 130:94 – 112.

Asato, T., Suga, Y., and Ogata, T. (2016). A reusability-based hierarchical fault-detection architecture for robot middleware and its implementation in an autonomous mobile robot system. In *2016 IEEE/SICE International Symposium on System Integration (SII)*, pages 150–155.

Assmann, U., Zschaler, S., and Wagner, G. (2006). Ontologies, meta-models, and the model-driven paradigm. In Calero, C., Ruiz, F., and Piattini, M., editors, *Ontologies for Software Engineering and Technology*. Springer.

Balakirsky, S., Schlenoff, C., Fiorini, S., Redfield, S., Barreto, M., Nakawala, H., Carbonera, J., Soldatova, L., Bermejo-Alonso, J., Maikore, F., Goncalves, P., de Mori, E., Ragavan, S. V., and Haidegger, T. (2017). Towards a robots task ontology standard. In *Proc. ASME 2017 International Manufacturing Science and Engineering Conference*, Los Angeles, CA, USA.

Balmelli, L., Brown, D., Cantor, M., and Mott, M. (2006). Model-driven systems development. *IBM Systems journal*, 45(3):569–585.

Bayat, B., Bermejo-Alonso, J., Carbonera, J., Facchinetti, T., Fiorini, S., Goncalves, P., Jorge, V., Habib, M., Khamis, A., Melo, K., Nguyen, B., Olszewska, J., Paull, L., Prestes, E., Ragavan, V., Saeedi, S., Sanz, R., Seto, M., Spencer, B., Vosughi, A., and Li, H. (2016). Requirements for building an ontology for autonomous robots. *Industrial Robot: An International Journal*, 43(5):469–480.

Bermejo-Alonso, J., Hernández, C., and Sanz, R. (2016). Model-based engineering of autonomous systems using ontologies and metamodels. In *2016 IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–8.

Bermejo-Alonso, J., Sanz, R., Rodríguez, M., and Hernández, C. (2011). An ontological framework for autonomous systems modelling. *International Journal on Advances in Intelligent Systems*, 3(3):211–225.

Bermejo-Alonso, J., Sanz, R., Rodríguez, M., and Hernández, C. (2013). Engineering an Ontology for Autonomous Systems - The OASys Ontology. In Fred, A., Dietz, J. L. G., Liu, K., and Filipe, J., editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management 2013*, volume 348 of *Communications in Computer and Information Science*, pages 47–58. Springer.

Blair, G., Bencomo, N., and France, R. (2009). Models@run.time. *Computer*, 42(10):22–27.

Blanke, M., Kinnaert, M., Lunze, J., and Staroswiecki, M. (2006). *Diagnosis and Fault-Tolerant Control*. Springer-Verlag Berlin.

Brugali, D. and Scandurra, P. (2009). Component-based robotic engineering (part i) [tutorial]. *IEEE Robotics Automation Magazine*, 16(4):84–96.

Chandrasekaran, B. and Josephson, J. R. (2000). Function in device representation. *Engineering with Computers*, 16(3-4):162–177.

Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castaño, R., Davies, A., Mandl, D., Frye, S., Trout, B., D'Agostino, J., Shulman, S., Boyer, D., Hayden, S., Sweet, A., and Christa, S. (2005). Lessons learned from autonomous sciencecraft experiment. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 11–18, New York, NY, USA. ACM.

de Leng, D. and Heintz, F. (2016). Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 55–60.

Fernández, J., López, J., and Gómez, J. P. (2016). Feature article: Reengineering the avionics of an unmanned aerial vehicle. *IEEE Aerospace and Electronic Systems Magazine*, 31(4):6–13.

Fiorini, S. R., Bermejo-Alonso, J., Gonçalves, P., de Freitas, E. P., Alarcos, A. O., Olszewska, J. I., Prestes, E., Schlenoff, C., Ragavan, S. V., Redfield, S., Spencer, B., and Li, H. (2017). A suite of ontologies for robotics and automation. *IEEE Robotics Automation Magazine*, 24(1):8–11.

Gehin, A.-L., Hu, H., and Bayart, M. (2012). A self-updating model for analysing system reconfigurability. *Engineering Applications of Artificial Intelligence*, 25(1):20 – 30.

Gruber, T. (2009). Ontology. In Liu, L. and Ozsu, M. T., editors, *Encyclopedia of Database Systems*, pages 1963–1965. Springer US.

Guarino, N., Oberle, D., and Staab, S. (2009). *Handbook on Ontologies*, chapter What is an ontology?, pages 1–17. Springer Berlin Heidelberg.

Hernández, C. (2013). *Model-based Self-awareness Patterns for Autonomy*. PhD thesis, Universidad Politécnica de Madrid, ETSII, Dpto. Automática, Ing. Electrónica e Informática Industrial, José Gutierrez Abascal 2, 28006 Madrid (SPAIN).

Hernández, C., Bermejo-Alonso, J., López, I., and Sanz, R. (2013). Three patterns for autonomous robot control architecting. In Zimmermann, A., editor, *PATTERNS 2013, The Fifth International Conferences on Pervasive Patterns and Applications*, pages 44–51. IARIA.

Hernández, C., Bharatheesha, M., Ko, W., Gaiser, H., Tan, J., van Deurzen, K., de Vries, M., Mil, B. V., van Egmond, J., Burger, R., Morariu, M., Ju, J., Gerrmann, X., Ensing, R., van Frankenhuyzen, J., and Wisse, M. (2017). Team Delft's Robot Winner of the Amazon Picking Challenge 2016. In Behnke, S., Sheh, R., Sarıel, S., and Lee, D. D., editors, *RoboCup 2016 Proceedings*, volume 9776 of *Lecture Notes in Computer Science*, pages 613–624. Springer.

Hernández, C., Fernandez-Sánchez, J., Sánchez-Escribano, G., Bermejo-Alonso, J., and Sanz, R. (2015). Model-based metacontrol for self-adaptation. In Liu, H., Kubota, N., Zhu, X., Dillmann, R., and Zho, D., editors, *Intelligent Robotics and Applications (ICIRA 2015)*, volume 9244 of *Lecture Notes in Artificial Intelligence*, pages 643–654. The 8th International Conference on Intelligent Robotics and Applications (ICIRA2015), Springer International Publishing.

IEEE Standards Working Group Ontologies for Robotics and Automation ORA (2015). *1872-2015 IEEE Standard Ontologies for Robotics and Automation*. IEEE Standards.

Jiang, H., Elbaum, S., and Detweiler, C. (2017). Inferring and monitoring invariants in robotic systems. *Autonomous Robots*, 41(4):1027–1046.

Lind, M. (1994). Modeling goals and functions of complex industrial plants. *Applied Artificial Intelligence*, 8(2):259–283.

Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., and Konolige, K. (2010). The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 300 –307.

Murphy, R. R., Tadokoro, S., and Kleiner, A. (2016). Disaster robotics. In Siciliano, B. and Khatib, O., editors, *Springer Handbook of Robotics*, chapter 60, pages 1577–1604. Springer International Publishing.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. C. (1998). Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1):5 – 47. Artificial Intelligence 40 years later.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source robot operating system. In Hirukawa, H. and Knoll, A., editors, *ICRA Workshop on Open Source Software*.

Rajan, K., Bernard, D., Dorais, G., Gamble, E., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Nayak, P., and Rouquette, N. (2000). Remote agent: An autonomous control system for the new millennium. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 726–730. IOS Press.

Rodríguez, M., de la Mata, J. L., and Díaz, I. (2013). Fault-tolerant self-reconfigurable control system. In Kraslawski, A. and Turunen, I., editors, *Computer Aided Chemical Engineering. 23rd European Symposium on Computer Aided Process Engineering*, volume Volume 32, pages 901–906. Elsevier.

Russell, S., Wefald, E., Karnaugh, M., Karp, R., Mcallester, D., Subramanian, D., and Wellman, M. (1989). Principles of metareasoning. In *Artificial Intelligence*, pages 400–411. Morgan Kaufmann.

Sanz, R., Bermejo, J., Morago, J., and Hernández, C. (2017). Ontologies as backbone of cognitive systems engineering. In *Proceedings of AISB CAOS 2017: Cognition And OntologieS*, Bath, UK.

Sanz, R., Hernández, C., and A. Hernando, J. Gómez, J. B. (2009). Grounding robot autonomy in emotion and self–awareness. In *Proceedings of the FIRA RoboWorld Congress 2009 on Advances in Robotics*, Lecture Notes in Computer Science, pages 23–43. Springer-Verlag Berlin Heidelberg.

Sanz, R., Hernández, C., Bermejo, J., and Rodríguez, M. (2014). Improved resilience controllers using cognitive patterns. In *Proc of the 19th World Congress of the International Federation of Automatic Control (IFAC)*, pages 683–688.

Schmill, M. D., Josyula, D., Anderson, M. L., Wilson, S., Oates, T., Perlis, D., Wright, D., and Fults, S. (2007). Ontologies for reasoning about failures in AI systems. In *Proceedings from the Workshop on Metareasoning in Agent Based Systems at the Sixth International Joint Conference on Autonomous Agents and Multiagent Sytems*.

Tenorth, M. and Beetz, M. (2013). KnowRob - a knowledge processing infrastructure for cognition-enabled robots. *International Journal of Robotics Research*, 32(5):566–590.

Tenorth, M. and Beetz, M. (2015). Representations for robot knowledge in the knowrob framework. *Artificial Intelligence*, 24(7):151–169.

Wang, X., Zhang, G., Neri, F., Jiang, T., Zhao, J., Gheorghe, M., Ipate, F., and Lefticaru, R. (2016). Design and implementation of membrane controllers for trajectory tracking of nonholonomic wheeled mobile robots. *Integrated Computer-Aided Engineering*, 23(1):15–30.