Adaptive Deployment of Safety Monitors for Autonomous Systems

Nico Hochgeschwender(✉)

German Aerospace Center (DLR),
Simulation and Software Technology, Cologne, Germany
nico.hochgeschwender@dlr.de

Abstract. This article discusses the problem of deploying safety-critical software for an autonomous system, namely a collaborative robot operating in domestic environments. We present a deployment infrastructure to enhance both humans and robots in carrying out their deployment activities. We develop means to enable humans to explicitly specify the requirements of the software to be deployed, along with the resources of the robot platform on which the software will be executed. In addition, we propose an architecture which enables robots to autonomously re-deploy their software at run-time in order to account for changing requirements imposed by their task, platform and environment. We show how the architecture enables a collaborative robot to autonomously re-deploy *safety monitors* for detecting in-hand slippage often occuring in human-robot handover tasks. By doing so, the robot autonomously maintains a certain safety level as the functioning of the monitor depends on both selecting and deploying the correct monitoring strategy for the situation at hand.

Keywords: Runtime AI safety monitoring · Model-based engineering approaches to AI safety

1 Introduction

Autonomous systems such as collaborative robots (see Fig. 1) are expected to carry out many different tasks in challenging environments not only over a long period of time, but also in a trustworthy manner. To do so, robots are equipped with a wide variety of software components ranging from solving functional problems such as planning, perception and control to safety-critical components required for execution monitoring, diagnosis and fault detection and isolation [14]. To achieve a certain level of autonomy robots are required to autonomously plan and execute actions and at the same time to cope with varying requirements for the software. Many of those variations are difficult to predict as they are induced by changing tasks, goals and environmental features. It is important to emphasize that the changing requirements are not only influencing core functional components, but also safety-critical modules such as *functional safety* features which aim to control hazards such that risks are mitigated.

A. Romanovsky et al. (Eds.): SAFECOMP 2019 Workshops, LNCS 11699, pp. 346–357, 2019.
https://doi.org/10.1007/978-3-030-26250-1_28

Therefore, deployment in robotics can be considered as an ongoing activity as different software components both functional and safety-critical are likely to be re-deployed at run-time in order to fulfill varying requirements. As we aim for truly autonomous systems robots themselves should be endowed with means to deploy their software. By doing so, the need for human intervention is reduced which paves the way for long-running robot applications.

Even though, in robotics deployment information is already available in terms of architectural and platform models [4,12] and configuration files [3,15] we identified in [6] that the majority of deployment approaches tend to be inflexible as they do not cover use cases where there is a need to respond to run-time changes. To this end, we generalized and described in [6] a *reference architecture* for deploying component-based robot software. In this article we instantiate this architecture (see Sect. 4) for a collaborative robot application (see Sect. 2) and thereby validate its applicability. We show how the architecture and corresponding deployment algorithms enable a collaborative robot to autonomously re-deploy safety monitoring strategies. By re-deploying the monitors we can assure that in-hand slippage is detected even in the presence of varying requirements induced, for example, by the robots task.

Fig. 1. The collaborative Care-O-bot 3 robot in a domestic environment.

2 Motivating Example

We consider collaborative robots as those shown in Fig. 1 to exemplify our work on deploying safety monitors. Collaborative robots are becoming more and more widespread not only in industrial and factory-like applications [16], but also in domestic scenarios. While traditionally in industrial scenarios robots are separated from human workers by fences collaborative robots share their workspace with humans. In those workspaces collaborative robots are expected to carry out a wide variety of tasks simultaneously or even in cooperation with humans. The close interaction with humans requires to adequately address safety concerns over the complete life-cycle of a robotic application. By conforming to standards such as ISO TS 15066 [10] and ISO 12011 [9] the collaborative workspace and type of tasks are specified which enables safety engineers to perform risk assessment. In the context of this work we consider the Care-O-bot 3 (see Fig. 1) as a collaborative robot capable of preparing, transporting and handing over coffee mugs to inhabitants of an apartment. For this application the risk assessment identified a unacceptable risk, namely that the robot could scald a person by dropping a mug with hot coffee while handing it over to a user. Thus, the risk

assessment recommends to implement risk reduction techniques, namely safety monitoring which is described in the following section.

2.1 Safety Monitor for Detecting In-Hand Slippage

To safely execute manipulation tasks (e.g. object picking, placing and human-robot handover) collaborative robots need to be equipped with means to detect slippage. That is, whether an object is moving within the robot's grasp. To detect in-hand slippage on the Care-O-bot 3 service robot (see Fig. 1), Sanchez *et al.* [1] proposed three different types of slip detectors based on tactile (exteroceptive) and force (proprioceptive) measurements and as a fusion of these (see Fig. 2). The force slip detector assumes a slip occurs whenever a force is exerted in the right direction (e.g. downwards with respect to the grasp frame). The tactile slip detector estimates the tangential force on the sensor caused by a sliding pressure (e.g. a grasped object slipping). The combined slip detector fuses both slip signals from the tactile slip detector and the force slip detector in a rule-based manner where experimentally obtained threshold values for the tactile and force slip detector are compared with each other. From a safety perspective the three slip detectors represent an active safety feature, namely a *safety monitor* which aims to preserve safety by checking safety-relevant information and possibly altering the robots behavior. Safety monitors are well-developed techniques in robotics which generally aim to prevent an autonomous robot of performing unsafe actions [14]. In our scenario, an unsafe action could be dropping a coffee mug while handing it over to a user. The slip detector can be used to prevent dropping the mug by detecting slippage and by exerting a higher force on the grasped mug. However, the performance of each slip detector varies considerably depending on the action, for example, the tactile slip detector outputs a slip whenever grasping an object. Contrary, the force slip detector achieves perfect accuracy for detecting actual slips, however its performance is poor when no slippage occurs, for example, when the robot base is moving. Thus, the evaluation of the approach carried out on the Care-O-bot 3 platform [1], highly suggests that the actions and motions performed by the robot during grasping should be taken into account for improved safety performance. Therefore, it is necessary to adapt the safety monitor to the current robot's actions at run time.

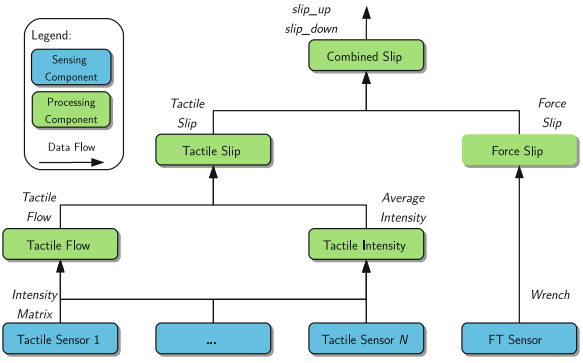


Fig. 2. The high-level functional architecture of the force, tactile and combined slip detector.

2.2 Deployment Requirements of the Safety Monitor

To this end, it is not only crucial to select the correct monitoring strategy for the task at hand, but also to deploy the safety monitor on the best matching platform. That is, a platform which fulfills the deployment requirements of the selected safety monitor. In the context of this work the Care-O-bot 3 robot is a distributed system with several computational platforms, thus several deployment options. For the described slip detectors we can formulate the following deployment requirements:

R1: The force slip detector should be deployed on a platform to which the force sensor is connected.
R2: The tactile slip detector should be deployed on a platform to which all tactile sensors are connected.
R3: The combined slip detector should be deployed on a platform with at least 250MB working memory.

Those requirements are based on experimentally obtained timing results of different deployment options.

3 Specifying Safety Monitors

We employ the textual, Ruby-based domain-specific languages `RPSL` (Robot Perception Specification Language) and `DepSL` (Deployment Specification Language) [8] to model architectural and deployment concerns of safety monitors (see Fig. 3). With `RPSL` one can model multi-stage slip detectors as those introduced in Sect. 2 as directed acyclic graphs composed of sensor components (e.g. force sensors) and processing components. In `RPSL` those graphs are called *perception graphs* and represent an executable and deployable unit. `DepSL` is used to attach platform requirements to each each safety monitor (cf. Sect. 2.2). To this end, `DepSL` supports the following requirement types proposed by the OMG deployment specification [13]:

Quantity. This requirement allows to express a certain number of required elements. For example, a certain number of tactile sensors connected to a platform (cf. **R2**).
Capacity. This requirement allows to express a certain capacity of a platform resource which can be consumed by one or more perception graphs. For example, the size (capacity) of working memory (cf. **R3**).
Minimum. This requirement allows to express an acceptable lower bound of a platform property. For example, maximum latency of a networking connection.
Maximum. This requirement allows to express an acceptable upper bound of a platform property. For example, the maximum latency of a networking connection.

```
rpsl.perception_graph do
   name "force_slip_detector"
   connect "force_sensor", "out_port",
           "slip_detection", "in_port"
   ...
end

depsl.deployment_specification do
   name "force_slip_detector"
   add_constraint :attribute, :platform_has, "force_sensor"
   ...
end
```

Fig. 3. An excerpt of the `RPSL` domain model of the force-based slip detector. `DepSL` is employed to specify deployment constraints for the `force_slip_detector` perception graph, namely that the platform where the graph will be eventually deployed has a force sensor attached to it.

Attribute. This requirement allows to express the existence of certain platform properties. For example, a certain hardware version of a sensor or a specific operating system installed on the platform (cf. **R1**).

Selection. This requirement allows to express a set of elements where one or more should be available on the platform. For example, different sensors of the same modality, but from different manufacturers.

Those requirements are used by the deployment architecture (see Sect. 4) to identify suitable platforms for each safety monitor.

4 Deploying Safety Monitors

The deployment architecture depicted as a component-based diagram in Fig. 4 conforms to the reference architecture specified in [6]. The basic idea of the architecture is to separate the concern of *what* should be deployed from the *where* it should be deployed. In the following sections we focus on the latter. Note, as the deployment architecture has been developed in the context of `RPSL` and as the safety monitors are specified with `RPSL` we will use the term *perception graph* interchangeably with the term *safety monitor*.

4.1 Context Monitoring

One or more context monitoring components are composed in the deployment architecture in order to provide the contextual information needed to select (see Sect. 4.3) and deploy (see Sect. 4.4) safety monitors. To this end, context monitors collect – hence requiring additional interfaces (see Fig. 4) – and interpret all the measurements required to infer the current state of robots' environment, platform (e.g. sensors, actuators and computational elements) and tasks and skills. Context monitors make this information accessible to other components by inserting them in the repository (see Sect. 4.2). In the context of the case study the main objective of the context monitor

is to retrieve the current action performed by the Care-O-bot 3. To this end, three different actions are detectable by the context monitor. First, whether the fingers of the gripper are closed to hold an object (cf. **grasp**). Second, whether the robot's base moves while holding an object (cf. **move_base**). Third, whether the fingers of the gripper are open to hand-over an object (cf. **release**).

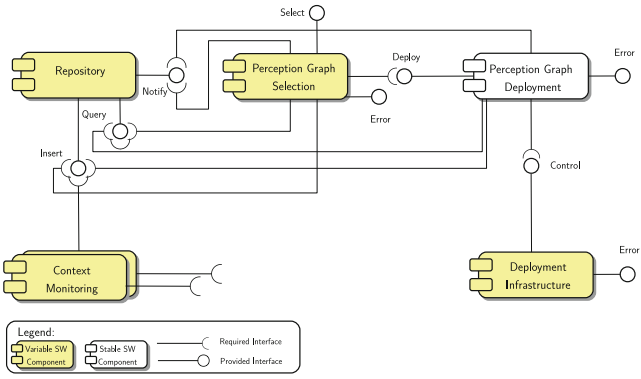
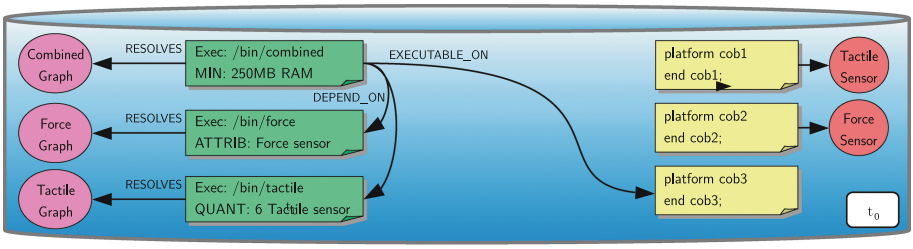


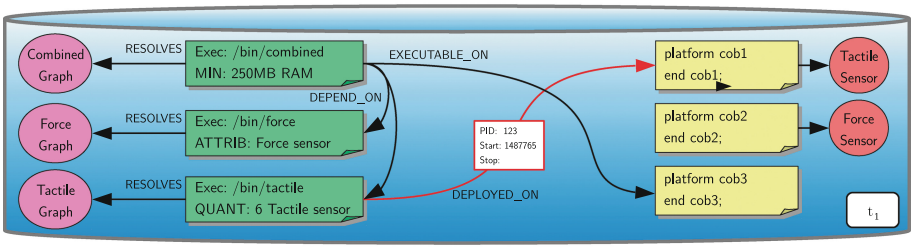
Fig. 4. The architecture for deploying the safety monitors depicted as a UML-like component diagram.

4.2 Repository

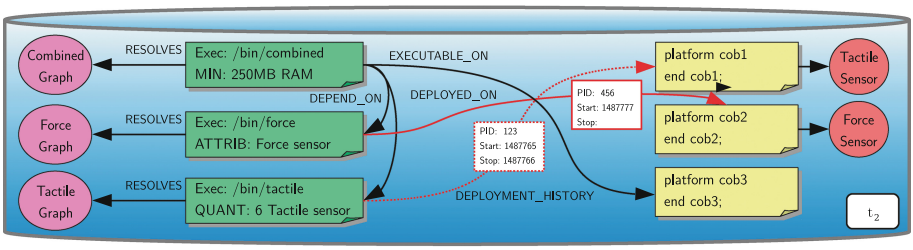
The repository plays a central role in the deployment architecture as it contains all the knowledge required for carrying out the deployment activities. Broadly speaking, the knowledge can be classified in design time and run time knowledge. Examples of the former are domain models expressing knowledge about the safety monitors (see Sect. 2) and the robot platform, and examples for the latter are information about the current memory usage or the availability of sensors required, for example, by the safety monitors. The repository component provides three interfaces, namely `Insert`, `Query` and `Notify`. The `Query` interface is used to retrieve information about design and run time knowledge. The `Insert` interface is used to create and update information in the repository, and the `Notify` interface is employed to inform other components about those changes in the repository. In the context of the case study both the `RPSL` and `DepSL` (see Fig. 3) were employed to create domain models representing the knowledge relevant for deploying the safety monitors. Those domain models represent not only the three different slip detectors, but also their associated deployment descriptions and the computational hardware of the Care-O-bot 3 robot. It is important to note that the repository in the context of this case study is realized as a graph database [7]. Thus, both `RPSL` and `DepSL` domain models have been translated to a labeled property graph representation. As demonstrated in [7] this representation paves the way to execute semantic queries to retrieve implicitly defined information required to infer on which platform, for example, the safety monitor should be deployed. As shown in Fig. 5a–d excerpts of the graph expressing the case study are depicted.



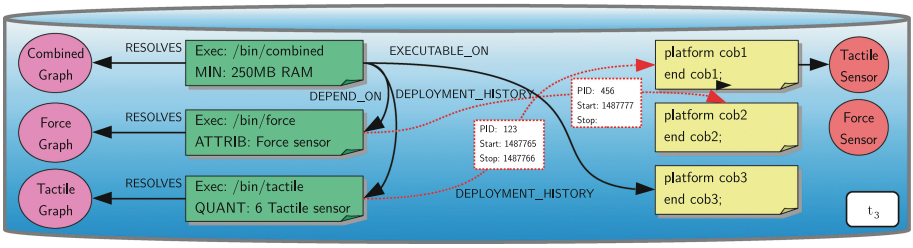
(a) An excerpt of the repository at time t_0.



(b) An excerpt of the repository at time t_1.



(c) An excerpt of the repository at time t_2.



(d) An excerpt of the repository at time t_3.

Fig. 5. Snapshots of the graph-based repository during the case study. Note, the snapshots are an excerpt of the complete graph database and focus on how the links between the deployment description (and their corresponding requirements) and the available platforms are established during the case study.

4.3 Selection of Safety Monitors

The perception graph selection component (see Fig. 4) is in charge of selecting one or more perception graphs suitable for the task at hand. To this end, activation is either triggered – in a reactive manner – by changes of context conditions using the `Notify` interface or by higher-level components via the `Select` interface provided by the component. In order to select a safety monitor which is appropriate for the current action context a simple, yet powerful rule-based approach is applied. During design time a set of of decision rules have been devised. Here, the action context is part of the rule condition and the selection of a safety monitor is part of the rule body. The rules are based on the experiments described in [1].

Algorithm 1. Finding a platform satisfying the deployment requirements.

```
1: function Deploy.perceptionGraphs(G)  ▷ G is the set of graphs to be deployed.
2:     for each g_i ∈ G do
3:         d_i ← Query.getDeploymentInformation(g_i)
4:         if d_i ≠ ∅ then
5:             if Query.hasFixedDeployment(d_i) then
6:                 p ← Query.getFixedPlatform(d_i)
7:                 if Control.start(g_i, p) then
8:                     return success
9:                 else
10:                    Error.deploymentFailed(g_i, p)
11:                    return error
12:            else
13:                C ← Query.getConstraints(d_i)   ▷ C, the deployment constraints.
14:                P ← checkValidity(C)            ▷ P, the acceptable platforms.
15:                if P ≠ ∅ then
16:                    if not Query.isDeployed(g_i, P) then
17:                        if Control.start(g_i, p_k ∈ P) then
18:                            return success
19:                        else
20:                            Error.deploymentFailed(g_i, p_k ∈ P)
21:                            return error
22:                else
23:                    Error.noAcceptablePlatforms(g_i)
24:                    return error
25:        else
26:            Error.deploymentInformationMissing(g_i)
27:            return error
```

4.4 Adaptive Deployment of Safety Monitors

The perception graph deployment component (see Fig. 4), or just deployer, is responsible for deploying one or more perception graphs. To do so, the deployer provides a `Deploy` interface which is used by the selector to inform the deployer

which perception graphs have to be deployed. After receiving such a request, Algorithm 1 is used to find those platforms which meet the deployment requirements for the given graphs. In case no platform is suitable, for example, if no platform satisfies the memory requirements, an error is reported via the `Error` interface. The implementation of the `Error` interface depends significantly on *(a)* how the reference architecture is realized in an application context, and *(b)* how the overall error management is implemented (see e.g. Garcia *et al.* [5] for a survey). The deployment algorithms shown in Algorithms 1 and 2 are explained in the following paragraphs by making use of the case study described in Sect. 2. We assume that at time t_0 the robot is located in the kitchen and has a mug in its hand. At this point in time the repository (see Sect. 4.2) is composed of nodes and edges as shown in Fig. 5a. Subsequently, a user requests the robot to deliver the mug to the living room. At time t_1 the robot starts moving it's base, hence the context monitor (see Sect. 4.1) detects the `move_base` action context and updates the repository. Based on the context update, the selector (see Sect. 4.3) component requests the tactile slip detector to be deployed. Thus, the selector calls the `perceptionGraphs()` method of the `Deploy` interface provided by the deployer.

As shown in Algorithm 1 for each selected perception graph g_i corresponding deployment information d_i is retrieved. That is, the node which resolves the perception graph is retrieved. In case no deployment information for g_i is available an error is reported. Subsequently it is checked whether or not a fixed deployment is given. That is, it is checked whether d_i has an edge to a platform which is labeled `:EXECUTABLE_ON`. As shown in Fig. 5a no fixed deployment for the tactile slip detector is provided. Thus, all the deployment requirements of d_i are retrieved in order to find an acceptable platform meeting the requirements. The `checkValidity()` method takes the requirements and returns those platforms P satisfying them. In the context of this example, `checkValidity()` checks which platform provides nine tactile sensors as those are required for the tactile slip detector. Basically two situations can occur, namely no platform is meeting the requirements or one or more platforms meet the requirements. In the former case an error is reported and for the latter case it is checked whether or not g_i is already deployed on one of the acceptable platforms. If g_i is not yet deployed on one of the acceptable platforms the deployer calls the `start()` method of the `Control` interface provided by the infrastructure component[1] in order to request the execution of g_i on $p_k \in P$. Having successfully deployed g_i on p_k the infrastructure component creates an edge labeled `:DEPLOYED_ON` from the tactile deployment node to the platform node (see Fig. 5b).

At time t_2 the robot reaches the living room and hands-over the mug to the user. The robot opens the fingers of the gripper to release the mug. Thus, the context monitor detects the release action. Subsequently, the selector chooses an appropriate slip detector for the observed context, namely the force slip detector. The selector requests the deployer to stop the current slip detector and to

[1] The infrastructure component abstracts the concrete runtime environment, e.g. a robot software framework.

deploy the force slip detector. Depending on the implementation of the perception graphs it would be also possible to simply send a pause signal to the slip detector.

As shown in Fig. 5c once the tactile slip detector is stopped, the edge from the deployment to the platform node is updated, namely the label is changed from `:DEPLOYED_ON` to `:DEPLOYMENT_HISTORY`. Like at time t_1 deployment requirements are checked and the force slip detector is deployed on the platform to which the force sensor is connected. At time t_3 the force sensor breaks and no force signal is provided anymore. The context monitor detects this failure and updates the corresponding platform model, namely the edge from the platform node to the sensor/device node is removed (see Fig. 5d). The repository notifies the deployer about those changes. Subsequently, the deployer executes the `checkDeployment()` method shown in Algorithm 2. The main objective of Algorithm 2 is to ensure that deployments remain valid in the presence of platform changes. To this end, each active deployment on the updated platform p_i is checked whether or not the requirements are met (cf. Algorithm 1). Three situations can occur, namely *(a)* no platform meets the requirements, *(b)* p_i meets the requirements, or *(c)* other platforms than p_i meet the requirements. In the context of the case study no platform satisfies the requirements, thus, the force slip detector is stopped.

Algorithm 2. Checking whether or not deployments are valid.

```
1: function checkDeployment(p_i)
2:     D ← Query.getActiveDeployments(p_i)      ▷ D, the active deployments on p_i.
3:     for each d_i ∈ D do
4:         g_i ← Query.getPerceptionGraph(d_i)          ▷ g_i, the perception graph.
5:         C ← Query.getConstraints(d_i)                ▷ C, deployment constraints.
6:         P ← checkValidity(C)                         ▷ P, the acceptable platforms.
7:         if P = ∅ then
8:             Error.noAcceptablePlatforms(g_i)
9:             if Control.stop(g_i, p_i) then
10:                return success
11:            else
12:                Error.stoppingFailed(g_i, p_i)
13:                return error
14:        if p_i ∈ P then
15:            return success
16:        if not P = ∅ and p_i ∉ P then
17:            if Control.start(g_i, p_k ∈ P) then
18:                return success
19:            else
20:                Error.deploymentFailed(g_i, p_k ∈ P)
21:                return error
```

5 Related Work and Discussion

Software deployment for autonomous systems and robotics in particular is usually achieved by some kind of deployment infrastructure provided by the underlying robot software framework. For example, the `roslaunch` deployment tool of the popular ROS [15] framework takes a XML-based description of the ROS architecture as an input and initiates the deployment according to it. To this end, components in ROS also known as nodes are started, stopped, parameters are set and so forth. Another notable deployment approach in robotics is proposed by Ando *et al.* [2]. Here – in the context of the OpenRTM robot software framework – deployment is considered as a part of component and system lifecycle management. The approach mainly deals with implementation-level details, for example, how manager services interact and how components are instantiated. Like in ROS, the OpenRTM deployment infrastructure relies on dedicated deployment files expressing crucial deployment information such as the location of an executable and so forth. Although these approaches help to automate the deployment task they are limited as they are not capable of expressing and resolving deployment requirements as presented in this paper. The deployment architecture proposed in this paper is inspired by the MAPE-K [11] reference architecture for self-adaptive software systems as it contains similar building blocks as those proposed in MAPE-K such as monitoring, knowledge storage, analysis and so forth. However, the introduced deployment architecture is more fine-grained as, for example, a stepwise deployment is supported. Here, the selector (see Sect. 4.3) deals with what should be deployed and the deployer (see Sect. 4.4) deals with how and where it should be deployed. It is important to note that currently, all deployment requirements are treated equally by Algorithm 1 as no preferences, weights or the like are given. Thus, if a platform is not meeting all the requirements it is not in the set of acceptable platforms P (cf. Algorithm 1). In addition, the current implementation ensures that the deployment requirements are not modified at run time. However, supporting dynamic, modifiable requirements could be feasible in even more dynamic situations such as in the context of cloud-robotics where resources are requested on demand.

6 Concluding Remarks

This article presented an approach for deploying safety monitors at run-time even in the presence of varying requirements. The presented work has been developed and integrated on a real robotic system which demonstrates its applicability to re-deploy safety monitors at run-time.

References

1. Sanchez, J., Schneider, S., Hochgeschwender, N., Kraetzschmar, G.K., Plöger, P.G.: Context-based adaptation of in-hand slip detection for service robots. IFAC-PapersOnLine **49**(15), 266–271 (2016)

2. Ando, N., Suehiro, T., Kotoku, T.: A software platform for component based RT-system development: OpenRTM-Aist. In: Carpin, S., Noda, I., Pagello, E., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2008. LNCS (LNAI), vol. 5325, pp. 87–98. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89076-8_12
3. Bruyninckx, H., Soetens, P., Koninckx, B.: The real-time motion control core of the OROCOS project. In: Proceedings of the IEEE International Conference on Robotics and Automation (2003)
4. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR 2012. LNCS (LNAI), vol. 7628, pp. 149–160. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34327-8_16
5. Garcia, A.F., Rubira, C.M., Romanovsky, A., Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software. J. Syst. Softw. **59**(2), 197–222 (2001)
6. Hochgeschwender, N., Biggs, G., Voos, H.: A reference architecture for deploying component-based robot software and comparison with existing tools. In: 2018 Second IEEE International Conference on Robotic Computing (IRC), pp. 121–128, Jan 2018
7. Hochgeschwender, N., Schneider, S., Voos, H., Bruyninckx, H., Kraetzschmar, G.K.: Graph-based software knowledge: storage and semantic querying of domain models for run-time adaptation. In: 2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), pp. 83–90 (2016)
8. Hochgeschwender, N.: Model-based specification, deployment and adaptation of robot perception systems. Ph.D. thesis, University of Luxembourg, Luxembourg (2017)
9. Safety of machinery - General principles for design - Risk assessment and risk reduction. Standard, International Organization for Standardization, Geneva, CH (2010)
10. Robots and robotic devices - Collaborative robots. Technical specification, International Organization for Standardization, Geneva, CH (2016)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
12. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. J. Softw. Eng. Rob. **7**(1), 75–99 (2016)
13. Object Management Group: Deployment and configuration of component-based distributed applications specification (2004). http://www.omg.org/spec/DEPL/4.0/. Accessed 05 May 2019
14. Pettersson, O.: Execution monitoring in robotics: a survey. Robot. Auton. Syst. **53**(2), 73–88 (2005)
15. Quigley, M., et al.: ROS: an open-source robot operating system (2009)
16. Villani, V., Pini, F., Leali, F., Secchi, C.: Survey on human-robot collaboration in industrial settings: safety, intuitive interfaces and applications. Mechatronics **55**, 248–266 (2018)