Towards Rule-Based Dynamic Safety Monitoring for Mobile Robots

Sorin Adam[1], Morten Larsen[1], Kjeld Jensen[2], and Ulrik Pagh Schultz[2]

[1] Conpleks ApS, Struer, Denmark
[2] University of Southern Denmark, Odense, Denmark

Abstract. Safety is a key challenge in robotics, in particular for mobile robots operating in an open and unpredictable environment. To address the safety challenge, various software-based approaches have been proposed, but none of them provide a clearly specified and isolated safety layer. In this paper, we propose that safety-critical concerns regarding the robot software be explicitly declared separately from the main program, in terms of externally observable properties of the software. Concretely, we use a Domain-Specific Language (DSL) to declaratively specify a set of safety-related rules that the software must obey, as well as corresponding corrective actions that trigger when rules are violated. Our prototype DSL is integrated with ROS, is shown to be capable of specifying safety-related constraints, and is experimentally demonstrated to enforce safety behaviour in existing robot software. We believe our approach could be extended to other fields to similarly simplify safety certification.

1 Introduction

Safety is a key challenge in robotics, in particular for domains such as precision agriculture where large, mobile robots operate in an open and unpredictable environment [1]. Safety is typically addressed by a combination of physical safety systems [2], the use of a safety-aware control algorithm [3], and the use of a software architecture that maps safety-critical program parts to a specific subsystem [4]. In an effort to address the safety challenge, various software architectures have been suggested for agricultural robotic vehicles [5, 6], but none of them provide specification and isolation of the safety-critical parts of the software. This increases the risk that programming errors will cause violations of those safety properties of the robot that are dependent on the correctness of the software. Moreover, faulty or erratically behaving hardware poses an additional safety risk: software built on implicit assumptions regarding the reliability of the hardware must monitor the system to ensure that these assumptions remain valid, failure to do so may compromise safety.

Mainstream robotic middleware such as Orocos [7] and ROS [8] allows software to be built in terms of reusable and individually tested components that can be deployed in separate execution environments, but do not provide any explicit means of expressing safety-related concerns. Model-driven software development approaches allow controllers to be automatically assembled from well-specified

D. Brugali et al. (Eds.): SIMPAR 2014, LNAI 8810, pp. 207–218, 2014.

components with explicit invariants that can be monitored at runtime, but typically provide a component-centric view that does not address the performance and safety of the system as a whole [9, 10]. Specific components can include invariants that specify assumptions about the hardware, but there is no comprehensive, implementation-independent specification of the hardware platform. In the specific case of safety, we observe that safety concerns may cross-cut the component structure of the system, for example enforcing a stop after a bump sensor has triggered could involve different software components (one for the sensor, one for the motion actuators).

We propose that safety-critical concerns regarding the robot software be explicitly declared separately from the main program, in terms of the overall functionality of the software. Rather than addressing the individual functionality of specific components, we address the functionality of the system as a whole in terms of externally observable properties of individual components, their communication, and the state of the surrounding execution environment.The main contribution of our work is the proposal and proof-of-concept experiments of a simple yet expressive rule-based language for enforcing safety constraints on existing ROS-based software. This paper presents the initial language design and prototype implementation, and experimentally documents the effectiveness of the solution through a series of experiments that test safety-oriented scenarios both involving software and hardware failures.

The rest of this paper is organised as follows: Section 2 discusses robot safety and model-driven software development, after which Section 3 presents our main contribution, Section 4 documents the experimental validation of our approach and discusses limitations, last Section 5 concludes.

2 Robot Safety and Modeling

2.1 Robot Safety

A robot has to be safe and reliable [11]. In the context of our work, *robot safety* concerns all elements of the robot and its immediate environment in which one or more errors may constitute a threat to nearby humans, animals, facilities and the robot itself. Faults in control architectures for mobile robots can be categorised as [12]: Environment (such as unpredictable environmental changes); Environmental Awareness divided into sensing faults (due to sensor or perception algorithm limitations) and action faults (unexpected outcomes of actuations); Autonomous System divided into decision making faults (lack of knowledge leading to inadequate decision making), hardware faults (sensors, actuators, embedded hardware) and software faults (with regards to software design, implementation and runtime execution). From a technical point of view, we aim to provide a system-wide supervision system that dynamically detects software faults; detection of hardware faults is supported to the extent that the fault is detectable from software. In this respect, our approach is similar to Blanke et al, where manually implemented supervision modules are used at different levels to increase safety and reliability in an autonomous robot conducting maintenance

tasks in an orchard [13]. In our work, we aim to automatically implement all parts of the supervision infrastructure based on declarative rules, but currently limit the supervision to deal with safety (not reliability). Unlike more general-purpose runtime monitoring systems based on temporal logic, we focus on providing a simple specification language easily accessible to non-experts.

2.2 Commercial Applications and Legal Regulations

We are interested in commercial applications of mobile robots within the agricultural sector. In Europe, from the regulatory point of view, robots are *currently* treated like any other commercial machine and thus have to comply with three European Directives: 2006/95/EC, 2001/95/EC, and 2006/42/EC. One group of the stated requirements concerns safety, usually evaluated by performing a safety risk assessment and reduction by using a standard like ISO 12100:2010. Therefore, the safety risk assessment is the primary source for the safety requirements of the robot. Standards like ISO 13849 provide guidance for establishing safety performance levels (PL) for the safety-related controllers. However, the safety PLs refer only to qualitative aspects of the software development, and are not concerned with quantitative aspects like latency, performance or reaction time. The standards demand, for higher safety PLs, increased software quality, and thus extensive code reviews, testing and documentation, all adding up to the project cost and delaying the release date. Moreover, the safety certification requirements can make the effort of releasing a new revision comparable with that of releasing of a new product.

2.3 Software Safety

Safety-critical software can be implemented in a general-purpose language and then verified automatically, and fault-tolerance can be improved using traditional techniques for software reliability, such as n-version programming [14]. Alternatively, using model-driven software development driven by a metamodel, the software can be specified in a high-level formalism from which an implementation satisfying the required properties can be automatically derived [15, 16]. Formal modeling enables analysis of more abstract properties, such as the safety of a robot, to be formally verified [17]. Automatic generation has the added benefit of accelerating software development [18–20, 10]. In this work we use a simple metamodel to describe existing ROS software, enabling both static analysis of the integrity of the software [21], as well as correctly programming in a high-level domain-specific language that targets this existing software, which is the subject of this paper.

2.4 Analysis

A commonly attributed reason for the popularity of ROS is the large amount of freely available software for the platform, in the form of reusable components (nodes). Indeed, the use of components as reusable building blocks is

fundamental to many approaches for model-driven software development for robotics [20, 10]. To ensure correct runtime functionality in a component, its execution can be monitored according to predefined invariants that essentially specify a contract for the dynamic behaviour of the component [9]. In all cases, the required safety-related behaviour may be specific to the application (e.g., the maximal speed at which the robot may move), may concern system-wide properties (e.g., a correlation of sensor values from multiple sensors), and may entail system-level reactions (e.g., an emergency stop of the robot). Since robot safety ultimately is a system-level property, we believe it is essential to enable the programmer to specify safety in terms of the robot software as a whole. Making this safety specification separate from the functionality facilitates verifying that the safety specification conforms to safety requirements, provided we guarantee that the robot software always follows the safety specification. In this paper, we propose to program the functionality-providing part of the software using standard ROS nodes, and to automatically program the safety-enforcing part based on declarative rules.

3 Rule-Based Dynamic Safety Monitoring

We propose a software architecture for implementing the safety-related functionality of the robot software separated from the main functionality, driven by a domain-specific language (DSL) for declaratively specifying the safety requirements. In more detail, safety rules, for example identified when performing the risk assessment, are described at a high level using the Rule-Based Safety Specification (RuBaSS) DSL. RuBaSS provides a simple and declarative syntax, making the task of implementing the safety-related requirements more accessible to robotics experts with a lower degree of software engineering expertise. The risk of errors is reduced, as the RuBaSS declaration drives the automatic generation of all safety-related code. Our approach directly enables an implementation-independent reuse of the safety-related part of a robot controller between different releases, since the RuBaSS declaration does not need to change when the underlying software changes (except that names shared between RuBaSS rules and component interfaces must be kept consistent). Moreover, the infrastructure can be reused in a range of products: the code generator can be directly reused whereas low-level interfaces to sensors and actuators will be specific to each robot. Safety-related customisation for the products is thus mainly achieved at the higher level, using the safety language.

The implementation of the low-level hardware interfacing and the code generation part of RuBaSS is naturally the responsibility of a skilled software development team, this division of roles is a normal consequence of introducing a more structured approach to robotics software development [20]. To further enhance robustness of the safety layer, and hence the overall safety of the robot, development of the code generator and execution supporting platform could be done by separate teams, targeting different programming languages. The decision for

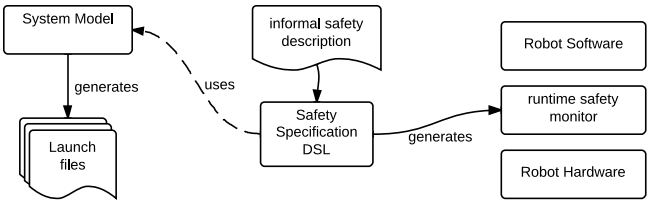

Fig. 1. Process overview

the implementation languages will normally be platform dependent, so different robotics platforms could favour certain languages. For example, for ROS-based robots, the safety-related code generation could target C++ and Python; in this paper, for simplicity we only implement a single prototype based on Python, in the future we expect to also support C++.

3.1 Overview

We consider that safety in robotic systems is a cross-cutting feature that interacts with many different parts of the system, so we propose to specify safety-related concerns in a separate declaratively programmed subsystem. This approach enables runtime isolation of the safety-related part from the rest of the robot application: although not currently implemented, the safety-related constraints can execute in a different context, for example using off-the-shelf virtualisation techniques, or on different hardware. Fig. 1 shows the overall workflow of using RuBaSS. The developer derives a RuBaSS from an informal safety specification and can access information from a system model which provides information on components (topics and nodes), thereby providing static consistency checks of the specification. The RuBaSS compiler generates a runtime safety component which monitors the specified properties of the software system.

The use of a DSL enables the total system cost to be optimised, since the same specification can be automatically redeployed by using different code generators. For example, the safety-related functionality may be executed as a regular ROS node during development, and then during field testing be redeployed to run on a dedicated, high-reliability hardware platform with modest processing power requirements, while the rest of the robot software executes on an inexpensive, less reliable hardware platform. The safety-related language we propose relies on the modularity offered by frameworks like Orocos and ROS, where the software functionality is divided into several intercommunicating parts implemented in dedicated components. We currently only support components that communicate using topic-based publish-subscribe, support for other communication patterns is considered future work. Monitoring of internal component state is not supported, if needed we expect that an approach similar to Lotz et al could be used [9].

```
1 action primitive stop;
2 entity drive_system : encoder_node, actuator_node; {
3  rules: // simple example comprising one actuator
4   actuator_erratic:
5    ((topic /cmd_vel_left.linear.x > 0.02m/s
6      or topic /cmd_vel_left.linear.x < -0.02m/s)
7     and topic /encoder_left.data == 0) for 0.4sec; }
8 if (drive_system.actuator_erratic for 1 sec) then { stop; };
```

Fig. 2. RuBaSS example (prettyprinted): enforcing stop on erratic behaviour

3.2 The Language

A simple example of the RuBaSS language is presented in Fig. 2 (a more extensive set of rules is used for the experiments later in the paper). The listing demonstrates the main features of the language. RuBaSS code is written in a safety description file containing two key parts: entities that group nodes together based on their functionality, and the behaviour section where the safety requirements are described. The safety-related actions the robot can execute can be triggered from the behaviour section; in this example the primitive action `stop` is declared, a complete set of actions can be imported from a robot-specific library. If the `stop` action is invoked by the rules, the robot stops (primitive actions are implemented in the underlying robot firmware). The entity section describes the `drive_system` as being composed of two nodes: the `encoder_node` and the `actuator_node`. For this entity only one rule is exemplified: `actuator_erratic`. This rule collects three different conditions under a common name using logical operators expressed in words. The rule is fulfilled when the actuator is unresponsive. RuBaSS also accepts temporal conditions, e.g., a logical expression is assessed and has to remain true for a continuous period of time. The behavioural section is where robot actions are associated with selected event occurrences. In the example, the actuator is declared to be "erratic" if there is a command and the encoders are not reporting any movement. Since this condition can occur during normal operation, e.g., due to reaction delays, a temporal condition is added specifying that the command and lack of movement must be present for more than 0.4 seconds. In summary, the entity rules define concrete safety-related events, the behavioural part of the specification concerns what action to take when based on combinations of these events.

In general, RuBaSS supports multiple entities and multiple compound rules defined inside every entity. The rules can be constructed around nodes or topics. Nodes can be supervised in terms of liveness, CPU and memory usage, whereas topics can be supervised in terms of publishing frequency and constraints on the data exchanged. The behaviour section associates actions to logical combinations of rules. All conditions, both for rules and behaviours, can be time-quantified using the `for` operator, and all constants can include physical units; units are currently only for documentation, statically checking their consistency using a component model that annotates physical units to components is future work.

3.3 Target Platform and Code Generation

A proof of concept of RuBaSS, generating Python code, has been implemented. Python has been chosen due to its ease of use and previous experience with Python and ROS; for production code C++ would be a better choice for target language, since it can both execute on a standard laptop and be executed on an embedded system with few resources. (Since having multiple implementations is advantageous for safety-critical systems, the Python-generated code could run on a PC-class controller together with the embedded code on a low-end controller.)

In order to ease the code generation, a Python library implementing classes for representing rules, entities, topic and node monitoring has been developed. A rule object stores the result of each rule evaluation in a timestamped buffer, making the result of the rule accessible for a given time interval. The topic monitoring class also stores the received ROS messages in a timestamped buffer. Each of the classes implement a loop method allowing the rule to be evaluated, and in the case of a topic, the average frequency to be calculated. The loop method is called on each object at a specific interval, currently 0.1 seconds (10 Hz), but can be set as high as the computer performance allows.

The need to access past data of the left hand side of the operator, complicates the generation of the code for the `for` operator. The solution used is to implement a buffering scheme in the rule and topic monitoring classes. However, the expression may be composed of a number of sub-expressions, for which past results should be kept. To solve this problem, we analyse the RuBaSS program before code generation and replace the left-hand side expression of a `for` with a reference to an intermediate rule containing the original left-hand side expression. Each time a rule is evaluated, the timestamped result is stored in a buffer used to remove old rule results outside of the time limit of the `for` expression.

4 Experiments

Experiments have been conducted using two different robots running different software: a physical *Frobit* robot and the standard, simulated iClebo *Kobuki* robot from the ROS distribution. The Frobit [22] is a small, low-cost robot designed for rapid prototyping; it is differentially steered and the low-level interface resembles that of many tracked and wheeled field robots; it is running the standard deployment of the FroboMind ROS-based software framework for field robots [5] on an i3 PC with 3Gb RAM running Ubuntu 12.04. The Kobuki is similar to the Frobit but has a different low-level interface and runs the standard, different and independently developed ROS-based control software, demonstrating the generality of our approach.

The experiments have been conducted using a specially written ROS test node, illustrated in Fig. 3. The node consists of a common part implementing the launch of the test cases and another containing the test-case-specific code for test environment (physical or simulated). For the Frobit, the test node is able to physically interrupt control lines of the wheels encoders and motors

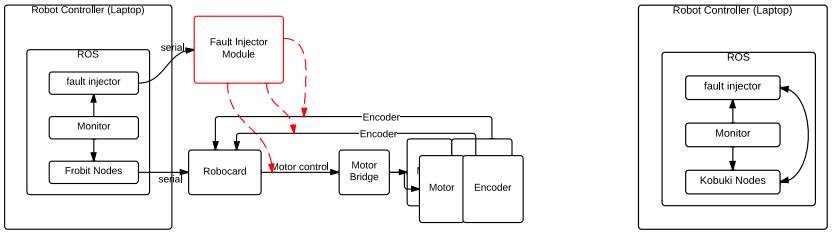

Fig. 3. Robot setup. Frobit hardware setup (left) and Kobuki setup (right).

through an Arduino-type board communicating with the test program via a serial connection. As the Arduino board is able to control the motors, the RuBaSS `stop` command is implemented using this board. For the Kobuki, the velocity commands coming from the robot's controller are altered by the test program to simulate a misbehaviour. In all cases we measure the time between when a fault has been introduced and when the safety-related node sends the stop command. All experiments have been repeated ten times.

A set of safety rules extracted from the risk assessment performed for commercial robots developed at Conpleks[1] have been implemented using the DSL. For the Frobit experiments, RuBaSS was used to implement rules supervising the wheels encoders, and limiting the linear speed of the robot (in total 3 entities and 9 rules). For the Kobuki experiments, the rules enforce a maximum linear and spinning speed for the robot, a maximum processor load for the ROS node controlling the robot, as well as the area the robot is allowed to move in (in total 1 entity and 4 rules).

4.1 Hardware Failure Experiments

For the Frobit, a number of experiments have been performed simulating a single fault or combination of wheels encoders not working or motors misbehaving. In all tests, the robot has been manually controlled using a remote control (a Nintendo Wii game controller) and specific faults, such as an encoder failure, were manually triggered using the test node. The experimental scenarios, implemented in the dedicated ROS test node, were running on the same hardware platform as the rest of the robot.

We tested combinations of the following failures: (1) Left or right wheel encoder not working (denoted E_x): The control line for the left and/or right wheel of the quadrature encoder has been interrupted. (2) Left, right, or both motors running at full speed (denoted M_x): The H-bridge controller of one or both motors have been wired in such a way that the motor has been forced to run at full speed, but the direction of movement (forward or backwards) was still under the control of the robot. (3) Combined simulated faults, with left or right motor full speed and right or left wheel encoder not working.

[1] The GrassBots grass-cutting robot [23] and the FIXFeeder mink-feeding robot [24] from Conpleks ApS.

Table 1. Experimental results. All times are in seconds. E indicates encoder failure, M indicates motor speed exceeded, in both cases left or right.

Frobit	Ideal	Avg	SD	Min	Max
E_R	1.40	2.07	0.35	1.41	2.66
E_L	1.40	1.87	0.24	1.60	2.34
M_R	0.50	0.82	0.13	0.61	1.02
M_L	0.50	1.71	0.30	0.92	2.13
M_{LR}	0.50	0.86	0.31	0.58	1.56
$M_L + E_R$	0.50	0.72	0.10	0.58	0.93
$M_R + E_L$	0.50	0.74	0.18	0.57	1.11

Kobuki	Ideal	Avg	SD	Min	Max
Boundary	0.00	0.07	0.02	0.03	0.09
Max speed	1.00	1.23	0.42	0.91	2,00
Max spin	0.70	8.15	1.89	4.66	10.32
(Exp=0.2)	0.20	0.17	0.03	0.12	0.21
(Exp=0.0)	0.00	0.17	0.04	0.11	0.22
CPU	1.50	1.71	0.14	1.51	1.95

4.2 Simulated Experiments

Several experiments have been performed using the simulated Kobuki robot: leaving a predefined area, exceeding the linear or spinning speed, and ROS-node CPU overload. All the tests were executed by indicating a target goal for the robot, leaving the planner to decide a route and control the robot. The same dedicated test ROS node has been used as for the Frobit, interfacing to the simulator by having the node monitor and modify the velocity and odometry information exchanged between the simulator and the robot.

We performed the following experiments: *(1) Boundary exceeded:* The robot target position has been set outside the predefined safety area, forcing the controller to violate the safety rules when driving the robot towards the target. The time interval between the moment when the border has been touched by the robot and when the stop command was issued by the safety ROS node has been measured. *(2) Maximum linear or spin speed exceeded:* The velocity commands issued by the controller have been modified by the test node: with every new velocity command received, an increasingly drifting amount has been added to the linear or spin velocity field value of the commands sent to the simulated robot. The test case measured the delay between the moment when the sent velocity command exceeds the maximum allowed linear or spin speed defined by the safety rules and when the stop command have been sent by the safety node. *(3) CPU overload:* The test node simulated a temporary software misbehaviour of one of the ROS nodes by executing a CPU-intensive loop for a determined period of time. The test case measured the time between when the CPU intensive loop was entered and when the safety node issued the stop command.

4.3 Results

The data obtained are shown in Table 1 for both the Frobit and Kobuki experiments. The columns in the table refer to the ideal minimum expected time according to the safety rules (Ideal), average time over the 10 repeated experiments (Avg), standard deviation (SD), minimum (Min) and maximum (Max) reaction time of the safety-related node measured during the experiments. For the Kobuki, the "maximum spinning speed exceeded" experiment has been repeated with the RuBaSS rules reacting to the speed being continuously exceeded for 0.7 seconds, 0.2 seconds and 0 seconds (instant reaction).

4.4 Discussion

All the tests performed were successful: the robot stopped when expected to stop. However, for the Frobit experiments, in the cases of right and left encoder error, the average reaction time was significantly higher than the ideal one. The reason is the way the temporal rule works: the condition has to be true continuously for the specified amount of time. Any change in the monitored values will reset the rule, and thus will delay triggering the action. In case of the wheel encoders, one line of the quadrature encoders was interrupted which generated some noise at the output. Similarly, there is a significant difference in reaction time when the right or left motor was forced to run at full speed. Even though the robot was not able to control the speed of one of the wheels, the rotation direction was still left under the control of the robot. In the left motor running at full speed experiments, the control algorithm of the robot changed the rotation direction of the left wheel several times, inducing a brake effect on the wheel, and delaying the fulfilment of "continuous speed exceeded for 0.4 seconds." In the simulator experiments the minimum measured reaction time is, in some cases, lower than the minimum ideal reaction time; The reason is the lack of real-time in event publishing and event handling, and the unsynchronised messages published by the ROS nodes. Conversely, the results of the tests when maximum spinning speed is exceeded are far from the minimum ideal value. Here, the robot has been instructed to spin at a value different from the one calculated, diverting from the planned route, and triggering the controller error correction mechanisms. The rule used requires the condition to be true continuously for 0.7 seconds, so even if the maximum spinning speed was exceeded momentarily, the triggering condition was not fulfilled as the controller tried to compensate. The recorded time presented in the table includes successful robot controller error corrections, and therefore is much longer that the ideal minimum one. A similar behaviour was seen in the case of the linear speed being exceeded, but at a much lower scale, giving a lower standard deviation. When the reaction time was set to 0.2 seconds, the robot did not have time to react to the introduced spinning error: the topics were publishing new messages every 0.1 seconds (10 Hz), so two consecutive cases of exceeding the maximum spinning were enough to trigger the safety rule. In the case when the safety rule was changed to react to any single case of exceeding the maximum spinning speed, due to the fact that the robot software is not reacting in real-time to the events but with a delay of approximately 0.1 seconds, the end results of the experiment are not significantly different from the case when 0.2 seconds reaction time was tested. The CPU load safety rule implementation detects if ROS nodes running in separate threads are overloaded. The usage of this kind of rule is limited, since the same processor is used for both assessing the rule conditions and running the ROS node code. To address those limitations we plan to monitor a heartbeat signal in the supervised ROS nodes and measure its frequency on another processor [25].

In the experiments only a complete stop action has been used as the reaction to any safety rule violation. That was done for simplicity of the implementation, but is obviously not the best action in all real-life scenarios. An improved

fault-handling based on diagnosis and fault isolation will be addressed in a future work together with improving the RuBaSS language to statically detect overlapping safety rules or potential contradictions. We note that for the implementation of safety-related rules using our DSL to take place without modifying the existing source code, we are dependent on the interfaces of the robot (e.g., the exposed interfaces between the different ROS nodes). If the needed information is not available on the exposed interfaces, the appropriate ROS nodes of the robot would have to be modified to publish it.

5 Conclusion

We have shown that it is possible to use RuBaSS to generate the implementation of the safety rules identified during a safety risk assessment, covering both hardware faults (e.g., encoders or motors not working) and misbehaviour of the software controlling the robot (e.g., the robot leaving the designated working area or CPU overload). RuBaSS has a simple syntax, making it easy to express the safety rules, and enabling the generation of runtime safety monitoring code, as our initial proof-of-concepts experiments demonstrate. Moreover, based on our initial experience, we find that addressing safety issues of robots with RuBaSS is efficient and easily customisable, even with partial access to source code. A systematic and realistic validation of RuBaSS is considered future work, we expect that the language, the software architecture and the implementation need to be significantly extended to be useful for realistic scenarios.

References

1. Kohanbash, D., Bergerman, M., Lewis, K.M., Moorehead, S.J.: A safety architecture for autonomous agricultural vehicles. In: American Society of Agricultural and Biological Engineers Annual Meeting (July 2012)
2. Griepentrog, H., Andersen, N., Andersen, J., Blanke, M., Heinemann, O., Madsen, T., Nielsen, J., Pedersen, S., Ravn, O., Wulfsohn, D.L.: Safe and reliable: further development of a field robot. In: Henten, E., Goense, D., Lokhorst, C. (eds.) Precision Agriculture 2009, pp. 857–866. Wageningen Academic Publishers (2009)
3. Bouraine, S., Fraichard, T., Salhi, H.: Provably safe navigation for mobile robots with limited field-of-views in unknown dynamic environments. In: 2012 IEEE International Conference on Robotics and Automation (ICRA), pp. 174–179 (May 2012)
4. Griepentrog, H., Jæger-Hansen, C., Ravn, O., Andersen, N., Andersen, J., Nakanishi, T.: Multilayer controller for field robots - high portability and modularity to ease implementation. Paper presented at LAND.TECHNIK - AgEng 2011 (2012)
5. Jensen, K., Bøgild, A., Nielsen, S., Christiansen, M., Jørgensen, R.: Frobomind, proposing a conceptual architecture for agricultural field robot navigation. Paper presented at CIGR 2012 (2012)
6. Nebot, P., Torres-Sospedra, J., Martnez, R.J.: A new hla-based distributed control architecture for agricultural teams of robots in hybrid applications with real and simulated devices or environments. Sensors 11(4), 4385–4400 (2011)
7. Bruyninckx, H.: Open robot control software: the orocos project. In: IEEE ICRA 2001 Proceedings, vol. 3, pp. 2523–2528 (2001)

8. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3(2) (2009)
9. Lotz, A., Steck, A., Schlegel, C.: Runtime monitoring of robotics software components: Increasing robustness of service robotic systems. In: Proceedings of the 15th International Conference on Advanced Robotics, pp. 285–290. IEEE (2011)
10. Gherardi, L., Brugali, D.: Modeling and reusing robotic software architectures: the hyperflex toolchain. In: IEEE International Conference on Robotics and Automation (ICRA) (to appear, 2014)
11. Dhillon, B.S.: Robot reliability and safety. Springer (1991)
12. Crestani, D., Godary-Dejean, K.: Fault tolerance in control architectures for mobile robots: Fantasy or reality? In: 7th National Conference on Control Architectures of Robots, Nancy, France (2012)
13. Blanke, M., Blas, M.R., Hansen, S., Andersen, J.C., Caponetti, F.: Autonomous robot supervision using fault diagnosis and semantic mapping in an orchard. In: Fault Diagnosis in Robotic and Industrial Systems, pp. 1–22. iConcept Press Ltd. (2012)
14. Powell, D., Arlat, J., Deswarte, Y., Kanoun, K.: Tolerance of design faults. In: Jones, C.B., Lloyd, J.L. (eds.) Dependable and Historic Computing. LNCS, vol. 6875, pp. 428–452. Springer, Heidelberg (2011)
15. Schlegel, C., Steck, A., Brugali, D., Knoll, A.: Design abstraction and processes in robotics: From code-driven to model-driven engineering. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS (LNAI), vol. 6472, pp. 324–335. Springer, Heidelberg (2010)
16. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
17. Yakymets, N., Dhouib, S., Jaber, H., Lanusse, A.: Model-driven safety assessment of robotic systems. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS (2013)
18. Bordignon, M., Stoy, K., Schultz, U.: Generalized programming of modular robots through kinematic configurations. In: Proceedings of IROS 2011: The 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3659–3666 (2011)
19. Schultz, U., Bordignon, M., Stoy, K.: Robust and reversible execution of self-reconfiguration sequences. Robotica 29(1), 35–57 (2011)
20. Steck, A., Lotz, A., Schlegel, C.: Model-driven engineering and run-time model-usage in service robotics. In: Proceedings of Generative Programming and Component-Based Engineering (GPCE). ACM (2011)
21. Larsen, M., Adam, S., Schultz, U., Jørgensen, R.N.: Towards automatic consistency checking of software components in field robotics. In: RHEA 2014: Second International Conference on Robotics and Associated High-technologies and Equipment for Agriculture and Forestry (May 2014)
22. Larsen, L.B., Olsen, K.S., Ahrenkiel, L., Jensen, K.: Extracurricular activities targeted towards increasing the number of engineers working in the field of precision agriculture. In: XXXV CIOSTA & CIGR V Conference, Billund, Denmark (July 2013)
23. Conpleks ApS: Grassbots, `https://www.youtube.com/watch?v=KMjEUrB5C5I`
24. Conpleks ApS: Fixfeeder, `https://www.youtube.com/watch?v=q8h63rYoNQ0`
25. Jensen, K., Larsen, M., Nielsen, S.H., Larsen, L.B., Olsen, K.S., Jørgensen, R.N.: Towards an open software platform for field robots in precision agriculture. Robotics 3(2), 207–234 (2014)