Journal of Software Engineering for Robotics 7(1), July 2016, 120-141
ISSN: 2035-3928

Rule-based Dynamic Safety Monitoring for Mobile Robots

Sorin Adam[1,2] Morten Larsen[1,3] Kjeld Jensen[2] Ulrik Pagh Schultz[2]

[1] Conpleks Innovation, Struer 7600, Denmark
[2] The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense 5230, Denmark
[3] Department of Engineering, Aarhus University, Aarhus 8000, Denmark

Abstract—Safety is a key challenge in robotics, in particular for mobile robots operating in an open and unpredictable environment. Safety certification is desired for commercial robots, but no existing approaches for addressing the safety challenge provide a clearly specified and isolated safety layer, defined in an easily understandable way for facilitating safety certification. In this paper, we propose that functional-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 DSL, 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.

Index Terms—Functional safety, DSL, robot, code generation, model driven development, compiler.

1 Introduction

Safety is a key challenge in several robotics domains, including field robotics where mobile robots operate in an open and unpredictable environment [1]. Here, safety is typically addressed by a combination of physical safety systems [2], the use of safety-aware control algorithms [3], [4] and the use of software architectures that map safety-critical program parts to a specific subsystem [5], [6], [7]. In an effort to address the safety challenge, various software architectures have been suggested e.g., for agricultural robotic vehicles [8], [9], 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 safety requirements. Moreover, faulty hardware imposes 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 [10] and ROS [11] 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 allows controllers to be automatically assembled from well-specified components with explicit invariants, but typically provides a component-centric view that does not address the performance and safety of the system as a whole [12], [13]. Moreover, the safety concerns may cross-cut the component structure of the system, for example enforcing a stop after a bump sensor being triggered could involve different software components (one for the sensor, one for the motion actuators).

Functional safety is the aspect of safety that aims to avoid unacceptable risks. Functional safety can be addressed with software-based systems, which is the approach taken in this paper. Neither the robot's reliability to fulfil the missions, nor the hardware and software reliability, are focus points for this paper. Although all of them are important topics in robotics, this paper investigates how to obtain dependable software code for the functional safety part, and where this code should be placed in the robot's software architecture. We see this work as a step towards both safer robots and easier safety certification.

We propose that safety-critical concerns regarding the robot software be explicitly declared separately from the main program. Rather than addressing the individual functionality of

Regular paper – *Manuscript received September 1, 2015; revised January 12, 2016.*

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 experimental demonstration of a rule-based language for enforcing safety constraints on ROS-based software. This paper presents the language design and the C++-based implementation, and experimentally documents the effectiveness of the solution through a series of experiments that test functional-safety-oriented scenarios.

The rest of this paper is organised as follows: Section 2 discusses related work in the area, after which Section 3 analyses key challenges in safety for robotics. Then Section 4 presents our main contribution, Section 5 describes the language implementation, Section 6 presents the usage in a real life scenario of the language, Section 7 documents the experimental validation of our approach and discusses limitations, last Section 8 concludes.

1.1 Relation to previous work

This paper is an enhancement of our previous work [14] in several key aspects: improved language design, new DSL for ROS node code generation, complete new code generation and architecture for the safety-related DSL, and experiments with real robots. The outcome is a significantly faster response time of the safety ROS node and a program with a higher level of readability compared to the previous work.

2 Background and Related work

We now discuss related work, starting with safety and functional safety (2.1), then legal aspects of safety for robotics (2.2), fault analysis (2.3), work specific to software safety (2.4), safety in robot control architectures (2.5), and ending with DSL languages and languages expressiveness (2.6).

2.1 Safety and functional safety standards

From the linguistic point of view, *safety* is defined by the Oxford Advanced Learner's Dictionary as *"the condition of being protected from or unlikely to cause danger, risk, or injury"*. It is a central subject in several international or national standards, like IEC 61508, ISO 13849, ISO 13482, etc.. According to IEC 61508-4: 2010, safety is *"freedom from unacceptable risk"*, while Avižienis et al [15] see safety as an attribute of dependability (together with the availability, reliability, integrity and maintainability), and define it as the *"absence of catastrophic consequences on the user(s) and the environment"*. As illustrated by those definitions, safety is a broad concept covering all possible hazard sources and all technically possible ways to avoid or minimise them. In order to narrow down the domain, IEC 61508-4:1998 introduced the functional safety term as *"part of the overall safety relating to the [equipment under control] EUC and the EUC control system that depends on the correct functioning of the [electrical and/or electronic and/or programmable electronic] E/E/PE safety-related systems and other risk reduction measures"*. Functional safety, as well as safety in general, should not be confused with dependability, it is only an attribute of it. Dependability is defined as *"the ability to avoid service failures that are more frequent and more severe than is acceptable"* [15], implying (in addition to the absence of catastrophic consequences) readiness and continuity of correct service delivery. Therefore, a safe robot, at least from the functional safety point of view, should not harm the humans or the environment when it operates, meaning that the mission success (fulfilment of the ordered task) is not seen as being part of the functional safety. However, a safe robot unable to perform tasks autonomously in a reliable way will not be useful, therefore a balance between the level of functional safety achieved and the impact of this on the robot's functions is needed [16]. Recognising that, the ISO 13482 safety standard addresses the functional safety of the personal care robots in the context of its overall functionality.

In the robotics domain, the number of existing safety standards is lower than for other engineering domains. Currently, the main safety standards specific for the robotics field are ISO 10218 [17] and ISO 13482 [18]. Other standards are under preparation, such as ISO/TS 15066 (Safety of collaborative robots) and ISO/DIS 18497 (Agricultural machinery and tractors - Safety of highly automated machinery) for the agricultural domain. The ISO 10218 standard deals with industrial robots confined in working cells, and therefore presents limited interest for mobile robots in contrast with ISO 13482. Even if it is intended for personal care robots, its safety requirements combined with the ones from standards like ISO 13849, ISO 25119, etc. could be used during the design and the development of other types of mobile robots until specific safety standards are prepared or released.

Although the ISO 13482 standard provides safety requirements covering all aspects of robot development, e.g. mechanics, electrical, hardware, software, etc., our interest is in the software part, and therefore only the software related requirements from this standard are considered in this paper. The ISO 13482 standard provides an overview of typical safety-related functions and the required minimum performance level (PL) for their implementation. The mentioned functions are: emergency stop, protective stop, limits to workspace, safety-related speed control, safety-related force control, hazardous collision avoidance and stability control.

2.2 Commercial applications and legal regulations

We are interested in commercial applications of mobile robots. In Europe, from the regulatory point of view, mobile robots are *currently* treated like any other machine and thus have to comply with European Directives like 2006/95/EC, 2001/95/EC, and 2006/42/EC. One group of the stated requirements concern

safety, and the usual way of dealing with it, is to perform a risk assessment and a risk reduction, following a standard like ISO 12100. The result is a list of risks and corresponding actions to be taken in order to keep them at an acceptable level. The safety risk assessment is the primary source for the functional safety requirements of the robot, and implementing them demands certain safety performance levels (e.g., ISO 13849); however, the safety performance levels refer only to qualitative aspects of the software development without any reference to quantitative aspects like latency, performance or reaction time. Addressing, at the robot system level, the non-functional properties like responsiveness or deterministic behaviour are considered to be crucial [19]. A key challenge when developing software complying with higher safety performance levels is to keep the project cost low and the time to market short. Certification of a new safety-critical system for the aviation sector is estimated to use more than 50% of the project resources, and similar numbers are applicable in other domains such as the medical, automotive or energy sector [20]. The higher the required safety performance level, the higher the required software code quality, the level of testing and the detail of the documentation. Extensive testing and documentation together with frequent code review adds up to the project cost and defers the release date. Moreover, the effort required to release a new version of the same software can become comparable with the one needed for a new product, due to the safety certification requirements.

2.3 Fault analysis

Software remains free of faults if it was fault-free after design and implementation, but unfortunately erroneous design and implementation can occur [21]. Mission fulfilment or catastrophic failures avoidance in the presence of faults is achievable by a combination of fault prevention, fault tolerance, fault removal and fault forecasting [15]. Fault prevention aims to avoid introducing faults into a system and to avoid their occurrence, fault tolerance aims to preserve the service in the presence of faults, fault removal deals with the reduction of the number and the severity of the faults, and fault forecasting addresses the prediction of the quantitative aspect of the faults while also estimating their impact on the system. All of them are of interest to safety in general but also to functional safety. Although systematic methods of analysing and predicting the faults of a system exists, such as fault mode effect analysis (FMEA) or fault tree analysis (FTA), this is not demanded by most of the existing safety standards [22] (only a risk assessment is required). An extension to FMEA is the failure mode, effects and criticality analysis (FMECA) where the FMEA is followed by a criticality analysis (CA). FMECA has been used for safety-related software for autonomous mobile robots [23], and Hazard Operability together with code generation from Unified Modeling Language (HAZOP/UML) has been proposed for elicitation of safety rules for critical autonomous systems [24].

Fault detection and fault recovery are important topics when discussing safety. The typical mechanisms for implementing fault detection are [25]: timing checks (implemented as, e.g., watchdogs), reasonableness checks (e.g., testing that the data are in an a priori known interval) and monitoring for diagnostics (the system's behaviour is compared to a model and the resulted difference, called residual, is used as a signature for fault detection). Fault recovery implementation in robotics follows two main approaches [23]: execution control and replanning, involving the mission planning layer of the robot control. In case of the execution control, when a malfunction is detected, the execution control is in charge of recovery. In the case of a component-based system, this could be achieved by resetting the troublesome component, repeating the command or reconfiguring the system and switching the execution to use redundant components, if possible.

Often, OTS (off-the-shelf) software components are used in a robotic system, and the presence of faults in these components is an issue [15]. In such cases, bug correction might be impossible. Moreover the quality of such software is unknown, making safety certification very troublesome.

2.4 Software safety and Model Driven Engineering

Safety-critical software can be implemented in a general-purpose language and then verified automatically [26], and fault-tolerance can be improved using traditional techniques for software reliability, such as n-version programming [27]. Alternatively, using model-driven software development, the software can be specified in a high-level formalism from which an implementation satisfying the required properties can be automatically derived [28], [29]. Formal modelling enables analysis of more abstract properties, such as the safety of a robot, to be formally verified [30], [31], [32]. Automatic generation and generative programming [33], [34] has the added benefit of accelerating software development [13], [35], [36], [37]. Automatically derived code from models has been used for dealing with software and hardware errors in robots [38], for generating code for safety-critical embedded systems [39], and for formal verification [40]. Several system verification techniques have been proposed, like using model checking in the case of programmable logic controllers (PLCs) [41] or using runtime physical characteristics of models by specifying them with a DSL [42].

Concepts from model driven engineering can also be applied at runtime, both for functional and non-functional requirements for self-adaptive systems [43], and could be applied for robot safety as well. While this is an interesting research direction, however, it remains an open question how to properly address safety in this context [43], although we speculate that our proposed runtime monitoring approach could be useful in this regard.

In this paper we use a simple metamodel to describe existing ROS software, enabling both static analysis of the integrity of

the software [44], as well as elimination of programming errors by using a high-level domain-specific language.

2.5 Safety in robot control architectures

Although the field of robotics safety is broad, the focus of this paper is on functional safety seen from the software point of view. Concretely we are concerned with how to produce code that can be used by the safety-related part of the robot's controller and how to reduce the chances of introducing software faults (bugs) in this part of the software. Faults in control architectures for mobile robots can be categorised as [45]: 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 [46], 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. 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).

The robot control architecture plays an important role in achieving the safety goals. Gribov and Voos [47] present a model-based, multilayer software architecture addressing robot safety. Its aim is to ease the mapping between safety cases, the associated hazards and the parts of the software addressing them, providing in this way the chain of evidence always required during certification. The proposed architecture introduces a proxy component, acting as a safety monitor on the chain of the components involved in the safety-related functionality of the robot. The resulting distributed architecture contains several safety-related ROS proxy nodes in charge of different functional safety requirements. Although this approach has the advantage of not demanding changes in the already existing components, when used with a complex robot it spreads the safety-related functionality, which is a drawback [23]. The problem of writing code for the proxy nodes is not addressed, but a need of implementing that software on separate, higher reliability, hardware is identified. In our approach, in contrast to using proxy nodes, we propose a single, central node in charge of the functional safety. The code for such a node is automatically generated from a high-level specification understandable by other robotics experts who are not themselves software specialists.

Safety monitors appear in the literature under different names [24], like safety bag [48], guardian agent [49], safety manager [50], checker [51], autonomous safety system [52], or diverse monitor [53], and are integrated in the safety-aware architecture of projects like the robotic satellite servicer [52], the LAAS architecture for autonomous systems [51] and the electronic railway interlocking system Elektra [48].

Crestani et al [23] propose a fault tolerant robot control architecture that addresses, in a structured and global way, the fault tolerance principles: fault detection, fault isolation, fault identification and fault recovery. An issue highlighted in this work is the fault tolerance requirement for the detection modules themselves, but because of their simplicity they are considered to be error free. In programming, even simple code could contain bugs, and we argue that automatically generating such detection software instead of manually writing it will reduce the risk of introducing errors in this safety-related code. Moreover, using a high-level specification for the code generation is a means to address the cases of complex failure detection, not only the simple ones.

2.6 DSL languages and expressiveness

Unlike more general-purpose runtime monitoring systems based on temporal logic, such as Eagle [54], we focus on providing a simple specification language easily accessible to non-experts. We support the notion that *"a successful DSL is above all one that is being used. To achieve this goal, the designer may need to downgrade, simplify and customize a language. In doing so, DSL development contrasts with programming language research where generality, expressivity and power should characterize any new language."* [55]. Our DSL, which we show fulfils the identified use cases, is intended to be accessible to non-programming experts, and therefore should be simple. Nevertheless we do want our DSL also to be useful in other scenarios. Taking inspiration from different domains, such as Complex Event Processing, could be useful for enhancing the language expressiveness and features. For example, the SASE language [56], designed for processing an RFID stream of data, queries `EVENT`s organized in `SEQ`uences and occurring `WITHIN` a time period, by using a `WHERE` clause applied to boolean combinations of predicates. Despite of its syntactic simplicity we argue that a program written in this language is not appropriate for the safety experts: the syntax is modelled after SQL which has many features not easily understood by non-programmers [57], [58].

Model-driven software development (MDSD) is increasingly used in robotics. Toolchains like BRIDE [59] or SmartSoft [60] provide strong support for MDSD, and rely on visual DSLs for development. However, a problem with these toolchains is the difficulty of integrating legacy code, making them best suited for developing new applications. Lotz et al [61] present two textual DSLs (SmartTCL [62] and VML [63]) meant to extend the visual component support

offered by SmartSoft. The VML language is designed to express constrained optimisation problems at runtime using a solver, so unlike our much simpler approach to safety supervision, VML programs cannot in general be mapped to a simple runtime monitor appropriate for execution as safety-critical code. Moreover, VML does not support triggering actions based on conditions that hold for a period of time.

A multitude of languages are available for modelling component-based systems outside the MDSD toolchains. One such language is SysML [64], which provides *Block* and *Port* constructs to model components and their connections. However, its weak semantics makes it unsuitable for modelling a ROS-based system, unlike the system description language that we use together with our safety-oriented textual DSL. Another such language is AADL [65], which can model both software and hardware components, and is an established standard modelling language already used within the embedded domain for, e.g., aerospace [66], robotics [67] and automotive [68]. We believe that AADL is suitable to model a ROS-based system, but investigating this direction is future work.

3 Analysis: Software Safety

3.1 Safety in component-based software

A commonly attributed reason for the popularity of ROS is the large amount of freely available software for the platform [69], 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 [13], [37]. 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 [12]. 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.

However, ROS does not fulfil hard real-time requirements, which is an issue for safety. Although using a real time (RT) flavour of Linux like RTAI [70] or Xenomai [71] could address the scheduling aspect of RT, the lack of real-time component intercommunication remains. A solution is to combine two middlewares [72]: one (ROS) used for non-RT tasks, and another one, like OROCOS [10], used for the RT parts. Another approach [73] is to run a modified version of ROS (RT-ROS), on a dedicated core of a dual-core processor hosting an RT operating system (Nuttx [74]). These approaches could be applied to the safety-related ROS node generated by our DSL in the case when it requires RT capabilities, by either deploying it on the RT-ROS part of the system or by enhancing our DSL to generate code for an RT framework like OROCOS. Both directions are considered future work.

3.2 Separation of roles and of concerns

Writing software for a robot requires both programming skills and robotics knowledge. Often a roboticist is not a programming expert. Writing software for safety-related parts of the robot places further demands on the developer. Moreover, a safety engineer/safety specialist is unlikely to be a programming expert, as the functional safety is a domain not directly linked to programming. Writing software for safety-critical systems requires expert knowledge about the programming language used as well as following strict coding rules, e.g., compliance with coding rules like MISRA C [75]. The usual way to develop such software is to add functional safety experts to the safety-specialised programmers team. This approach has the communication barrier problem: people with different backgrounds could have a different understanding of the same problem, which often is the cause of misunderstandings. Also the productivity of such a team is affected by the time required to be spent to clarify/align the views between the different team members. On the other hand, the team diversity and the need of such discussions could be good and help identifying potential problems. Achieving the right balance between alignment and code productivity will strongly depend on the communication skills of the team members, making the project output team-dependent. Another issue associated with writing safety-critical software is programmer productivity. The safety standards typically require the V-type model of software development to be used implying specification, coding, testing at several levels, documentation, etc. A central requirement for the safety-related software development is that the code development should follow or be compatible with a quality standard such as ISO 9001/9003. All those requirements decrease the speed of code development. A way to address the productivity issue while preserving the code quality is to use automatic code generation [76]. The generated code would automatically follow the coding rules making it (almost) impossible for programmers to break them. The generated code could be the output of an MDE tool chain or the output of a DSL. In this paper, we propose to use a safety-related DSL to alleviate the conflicting domain expertise requirements while also addressing the productivity and code quality concerns. By using our DSL, a safety expert could directly define the safety rules required without being a software expert. A safety software development expert will

be in charge of implementing the library functions needed to customise/adapt/glue the generated code with a certain middleware/framework. In this way, the adaptation is performed once for a specific robotics platform, and can be reused for different versions or releases of the same robot. The only part requiring a significant change would be the safety rules themselves.

3.3 Safety certification

Another problem faced when developing safety-critical software is the need of the product certification. However, this subject is poorly covered in literature [77]. The safety standards, depending on the performance levels required to be achieved, allow self-certification in particular cases, but the usual practice is to use an external assessor for all certification activities. With all the effort placed in making the standards clear and unambiguous, the reality is that often this is not achieved, making the certification a process influenced by the assessor's subjectivity in interpreting the standard [78]. Therefore the quantity and quality of the proofs of compliance could play an important role in speeding up the overall certification process. Distributing the implementation of the functional safety functionality in the whole robot's software will require that all the code should be written following the same requirements for writing safety-related software, and produce the documentation to prove that. Although this is possible, it is not practical due to the high cost of the project. In the case of robotics, often OTS components and middleware are used, meaning that a critical part of the software is written by non-team members, making it hard or even impossible to document the compliance with safety standards for this part of the software. To address this issue, we propose that the functional safety-related software is encapsulated in a dedicated node, and is in charge of monitoring, from the functional safety point of view, the software written by non-team members, if that software could affect the robot's safety. Following this proposal, the current implementation of our DSL produces the complete ROS node software code. The only role of this node is to implement the required functional safety resulting from the risk assessment, following in this way the separation of concerns concept, even if, redundantly, parts of safety-related functionality could already be present in other system's components, e.g., by being implemented in the framework or in the externally-sourced components. We hypothesise that the software safety certification can be limited only to this specially developed safety component.

4 Rule-based Dynamic Robot Safety

We propose a software architecture for implementing the safety-related functionality of the robot separated from the main functionality, driven by a domain-specific language (DSL) for declaratively specifying the safety requirements. In more detail, safety rules, typically identified when performing the risk assessment, are described at a high level using the

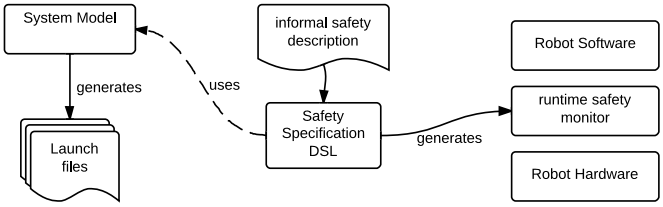


Fig. 1: DeRoS usage workflow

Declarative Robot Safety (DeRoS) DSL. DeRoS 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 DeRoS 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 DeRoS declaration does not need to change when the underlying software changes (except that names shared between DeRoS 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 DeRoS language. In the longer term, we believe that software testing can benefit from our approach: safety-critical parts of test cases could be automatically generated from the safety specification; however, this is left as future work.

The implementation of the low-level hardware interfacing and the code generation part of DeRoS is 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 [37]. 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 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, we support C++ as we consider it as being a more appropriate programming language for safety-related functionality.

4.1 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,

```
use "RobotSafety.system" import RobotSafety

action stop;
const max_tilt = toRad(10 deg)
const max_speed = 0.5 m/s
input orientation = topic /fmInformation/imu.orientation
input linear_speed = topic /fmKnowledge/encoder_odom.twist.twist.linear.x

entity imu_sensor_system
{
  tilt_not_ok:
    orientation.roll() not in [-max_tilt, max_tilt] or
    orientation.pitch() not in [-max_tilt, max_tilt];
}

entity drive_system
{
  max_speed_exceeded: linear_speed > max_speed for 2 sec;
}

if imu_sensor_system.tilt_not_ok then { stop; };

if drive_system.max_speed_exceeded for 0.4 sec then {stop; };
```

Fig. 2: Declarative Robot Safety (DeRoS) DSL example: enforcing stop on erratic behaviour

for example using off-the-shelf virtualisation techniques, or on different hardware. Fig. 1 shows the overall workflow of using DeRoS. The developer derives a DeRoS program from an informal safety specification that can access information from a system model which provides information on components (topics and nodes), thereby providing static consistency checks of the specification. The DeRoS 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 [12].

4.2 The language by example

A simple example of the DeRoS language is presented in Fig. 2 (a more complex one is described in Section 6). The listing demonstrates the main features of the language.

A DeRoS program starts by specifying the file where the information about the robot is defined (referred as system):

```
use "RobotSafety.system" import RobotSafety
```

The system file captures the robot model, since DeRoS is built on top of a ROS-node-modelling DSL (presented in Section 5.3). It specifies e.g., the datatype for messages, topics names and message fields (see Section 5.3.2 for details).

The functional-safety-related actions possible for the robot are declared using the `action` keyword:

```
action stop;
```

The actions are implemented as functions in a robot-specific library. In the example, the only action declared is to stop the robot movement.

A DeRoS program can use named constants with an optional measurement unit:

```
const max_tilt = toRad(10 deg)
const max_speed = 0.5 m/s
```

A set of predefined measurement units is built into the language, and contains usual measurement units like m/s, sec (seconds), deg (degrees), rad (radians), etc. Support for user-defined measurement units is left as future work. The constant value can be a decimal number or the result of a conversion function, as illustrated in the example: the `max_tilt` value is the result of converting 10 degrees into radians.

In the input definition section, names are assigned to ROS topics used in the program:

```
input orientation =
  topic /fmInformation/imu.orientation
```

The name `orientation` is now an alias for accessing the value published on this topic when writing safety rules.

A key part of a DeRoS code is the entities section:

```
entity imu_sensor_system {
  tilt_not_ok:
    orientation.roll()
    not in [-max_tilt ,max_tilt] or ...

entity drive_system {
  max_speed_exceeded :
    linear_speed > max_speed for 2 sec;
}
```

Here the functional safety rules are defined in terms of logical entities. The rules are updated when new data arrive on the `input` topics. In the example, the entity section first describes the `imu_sensor_system` containing one rule: `tilt_not_ok`. This rule combines several conditions by using logical operators. The rule `tilt_not_ok` is fulfilled when the robot tilt angle, calculated from the `orientation` using the `pitch()` respectively `roll()` functions, is outside of a predefined interval. For the rule expression, the language provides support for intervals of all kinds (closed, open, semi-open).

DeRoS also accepts temporal conditions: e.g., a logical expression is assessed and has to remain true for a period of time. In the code example, the `max_speed_exceeded` rule requires that the `linear_speed` of the robot, defined in the inputs section, should not exceed the `max_speed` value for two seconds. A timer-based mechanism present in the generated code triggers the rules containing temporal conditions even in the case when no new messages are arriving on the `input` topics for periods of time exceeding the specified time value.

The behaviour section implements the functional safety requirements by associating robot actions with selected event occurrences:

```
if imu_sensor_system.tilt_not_ok
  for 0.4 sec then { stop;};
if drive_system.max_speed_exceeded
  then {stop;};
```

In this example, the primitive action `stop` is invoked by the rules in two different scenarios (when tilt is not ok and when maximum speed is exceeded), causing the robot to stop moving. The primitive actions, such as `stop`, are implemented in the underlying robot firmware. If the `tilt_not_ok` rule from the `imu_sensor_system` entity or the `max_speed_exceeded` rule from the `drive_system` entity is triggered, the robot movement is stopped through the `stop` action. Since this condition can momentarily occur during normal operation, e.g., due to IMU sensor noise, a temporal condition is added specifying that the tilt must be exceeded for at least 0.4 seconds, meaning that all the discrete values received in an interval of minimum 0.4 seconds should exceed the maximum allowed tilt. In this way, the temporal conditions are used to filter out false positive triggering of the safety rules. For enhancing the language expressiveness, and for providing the programmer with appropriate syntax options to express in the best possible way the safety rules, we decided to allow temporal conditions for both entity and behaviour rules.

SafetyProg	S	::=	$M\ I\ A^+\ C^*\ I^*\ E^+\ B^+$
Import	M	::=	$\texttt{'use'}\ N_M\ \texttt{'import'}\ N_M$
Action	A	::=	$\texttt{'action'}\ N_A\ \texttt{';'}$
Constant	C	::=	$\texttt{'const'}\ N_C\ \texttt{'='}$ $(\ c\ U?\mid N_F\ \texttt{'('}\ c\ U?\ \texttt{')'})$
Input	I	::=	$\texttt{'input'}\ N_I\ \texttt{'topic'}\ N_T$
Entity	E	::=	$\texttt{'entity'}\ N_E\ \texttt{'\{'}\ R^+\ \texttt{'\}'}$
Rule	R	::=	$N_R\ \texttt{':'}\ (\ X\mid(\ X\ \texttt{'or'}\ X\ )\mid$ $(\ X\ \texttt{'and'}\ X\ )^+\ F?\ \texttt{';'}$
Expression	X	::=	$L\ (\ \texttt{'='}\mid\texttt{'<'}\mid\texttt{'<='}\mid\ldots)$ $(\ v\mid N_C\mid L\ )$
LeftTerm	L	::=	$\texttt{'not'}?\ N_T\mid N_S\mid N_I$ $(\texttt{'.'}\ N_F\ \texttt{'()'})?$
Behaviour	B	::=	$\texttt{'if'}\ (\ (\ T\mid(\ T\ \texttt{'or'}\ T\ )\mid$ $(\ T\texttt{'and'}\ T\ )\ )\ F?\ )^+$ $\texttt{'then'}\ \texttt{'\{'}\ N_A^*\ B^*\ \texttt{'\};'}$
Term	T	::=	$\texttt{'not'}?\ N_E\ \texttt{'.'}\ N_R$
For	F	::=	$\texttt{'for'}\ t\ \texttt{'sec'}$
Unit	U	::=	$\texttt{'sec'}\mid\texttt{'rad'}\mid\texttt{'deg'}\mid\ldots$

$N_A, N_C, N_E, N_F, N_I, N_M, N_R, N_S, N_T \in$ name space of corresponding construct
$c, t, v \in \mathbb{R}$ (real numbers)

Fig. 3: The Extended Backus-Naur Form (EBNF) grammar of high-level DeRoS programs

In summary, the entity rules define concrete safety-related events, the behavioural part of the specification concerns what action to take and when based on combinations of these events. In general, DeRoS supports multiple entities and multiple compound rules defined inside every entity. The rules can be constructed around nodes or topics. 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.

4.3 Syntax

The Extended Backus-Naur Form (EBNF) of DeRoS is shown in Fig. 3. A DeRoS program is divided into several parts: an import (M) section, an actions (A) section, two optional sections constants (C) and inputs (I), an entities (E) section and a behaviours (B) section.

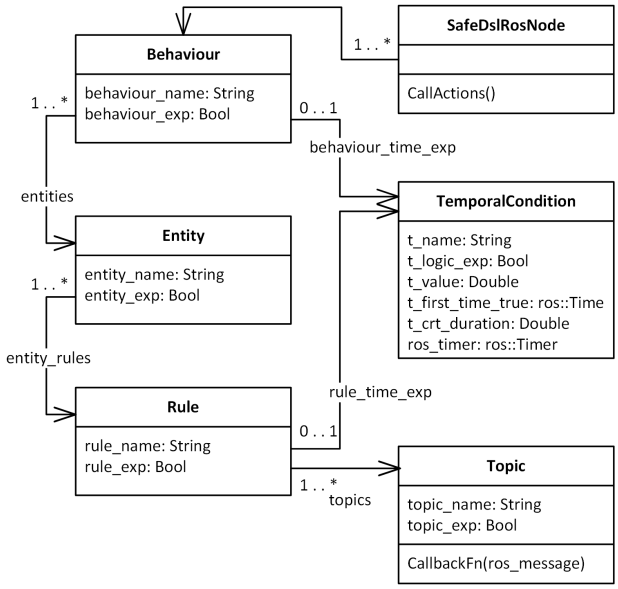


Fig. 4: DeRoS abstract runtime model

The actions section declares one or more actions A each identified by their N_A names. The actions are library functions that implement the functional safety robot behaviour.

Instead of repeating the same numerical value several times in the program, a constant can be defined by assigning the wanted value to a constant name N_C. The syntax for defining constants allows to both associate a numerical value c to a constant and to specify a measurement unit for it. This is particularly useful for safety-related programs as it opens the way for measurement unit consistency check inside expressions. The DeRoS language provides support for converting numerical values through the usage of function names N_F which take the numerical values (with or without measurement units) as parameters. The current implementation of the language supports only predefined conversion functions, so the N_F function name is an element of the built-in set of functions (`toRad` that converts a value expressed in degrees to radians and `toDeg` that converts radians into degrees).

Sometimes the left terms used in the expressions evaluated inside the DeRoS program could be long, like in the case of a ROS topic name including message fields. Even more, the same long string could be used several times in the program impeding in the end its readability. To avoid such problems, the language permits a short name N_I association for topics N_T in the inputs I group.

The entities E groups together rules R under a common name N_E using a deliberately kept simple syntax, to enforce a consistent style of declarations that compares values from the controller to constants. Every rule R has a name N_R associated with it and a number of boolean-evaluated expressions X combined using logical operators (and and or). An expression X consists of a left term L compared to a numerical value v or a constant N_C defined in the constants sections. Both a boolean value ($true$ or $false$) and a real number are legal values for v. The left terms L of the rule expressions are directly specifying a topic or a reference to an input N_I. In case of a boolean type of left terms, a negation operator (not) can be used. A function N_F from a pre-defined set of functions can be used in the expression following a Java-like syntax, e.g., `orientation.roll()`. A temporal condition could be expressed for the rule by specifying a period of time t for which the rule expression has to remain true as shown by the $for\ F$ syntax.

In the behaviour section, boolean expressions T represented by the expressions defined in the entity section are assessed and actions N_A are triggered. Nested behavioural expressions are allowed. The terms T are composed from two parts: an entity name N_E and a rule name N_R. As for the rules, it is possible to use temporal conditions when assessing a term T logic value. Moreover it is possible to use the negation operator (not). Complex behaviour expressions could be constructed by combining terms T using logical operators (and and or).

4.4 Runtime behaviour

This section explains the semantics in terms of an abstract runtime model, the actions that occur at runtime, and an algorithm describing the way we handle temporal conditions. While the implementation uses a different, more compact representation, the semantics of this representation can be understood in terms of the abstract runtime model presented in Fig. 4.

A ROS node (*DeRoS safety node*) is launched and data-sets are created for every *Entity*, *Rule* and *Behaviour* reference in the safety specification file. The node creates a ROS subscriber and a corresponding callback function (*CallbackFn*) for every *Topic*, part of each *Rule*, referenced in the safety specification. Also, for each ROS topic expression *topic_exp* part of the *Topic* and referenced in *Rule*, a unique variable is used and its value is updated when the corresponding callback function (*CallbackFn*) is called (each time a new message is published on the topic).

If a temporal condition is present in the rule expression, a boolean variable (*t_logic_exp*) is used. For every *Rule* from every *Entity*, a variable *rule_exp* is associated with the expression of the respective rule. If the rule contains a temporal condition expression, the *rule_exp* is additionally checked if the *t_logic_exp* is *true*.

For the *Behaviour* part, a boolean variable *behaviour_exp* is used for every behaviour logic expression. Similar to the rule entity, if temporal conditions are used, boolean time variables further condition the logical expression.

An action diagram showing how the DSL executes two specific actions that triggers a safety behaviour of the robot is shown in Fig. 5. The chain of variables is updated every time a new *ros_message* is received: the *CallbackFn* is called triggering a chain-update of the variables, starting with *topic_exp* and

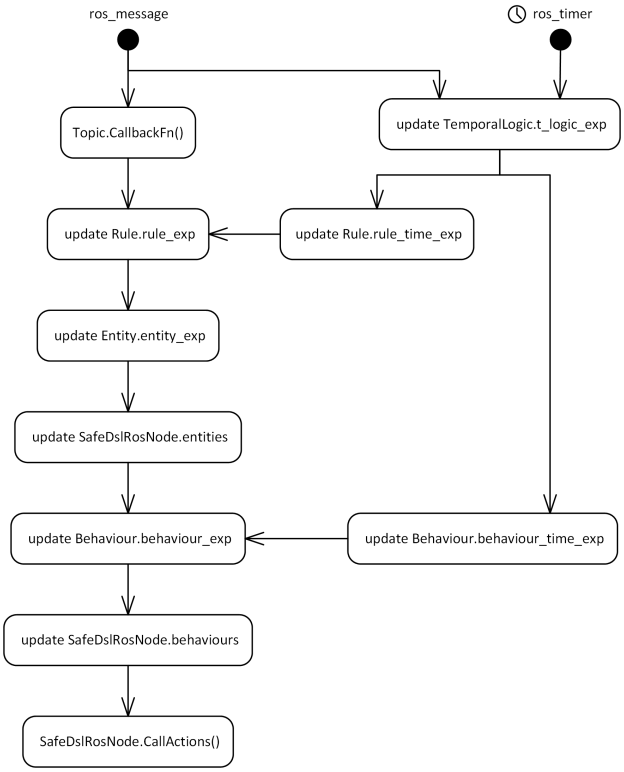


Fig. 5: Actions diagram for DeRoS program

ending with all *behaviours* of the *DeRoS safety node*. If the value of the end variable *behaviour_exp* attached to *Behaviour* switches from *false* to *true*, the actions associated with the respective *Behaviour* rule are called through *CallActions*. This architecture makes the safety node behave in an event-driven manner, ensuring a minimum time delay between the moment when a rule condition is fulfilled and the reactive rule action is triggered.

The *temporal conditions handling* triggers additional chain-update of variables. The algorithm used is presented in Fig. 6. While both the entity rules and the behaviour use the same mechanism, the algorithm depicts the case of a temporal condition for *Behaviour*: every time a *ros_message* is received or a dedicated *ros_timer* expires, if the condition in *behaviour_exp* is satisfied for the first time, the current time value is stored in *t_first_time_true* variable. If the following messages continue to fulfil the *behaviour_exp* logical condition, the current time *crt_time* is compared with the one stored in *first_time_true* when the condition has been fulfilled for the first time, and the result is stored in *t_crt_duration*. If *t_crt_duration* exceeds or equals the time value *t_value* for the temporal condition, the combined behaviour-expression *and* time-expression becomes *true*. In the case when the new received message invalidates the condition in the logic expression, the *t_first_time_true* variable is cleared, resetting in this way the temporal condition mechanism.

```
tc = new TemporalCondition
b = new Behaviour
tc.t_first_time_true = 0
tc.t_logic_exp = false
tc.t_value = time_condition_constant
when (ros_message or ros_timer) received
  update(b.behaviour_exp)
  if b.behaviour_exp = true
       if tc.t_first_time_true = 0
         tc.t_first_time_true=current_time()
       else
         t_crt_duration =
           current_time()-tc.t_first_time_true

         if t_crt_duration >= tc.t_value
           tc.t_logic_exp = true

         tc.t_first_time_true = 0
  else
    tc.t_first_time_true = 0
    tc.t_logic_exp = false
```

Fig. 6: Temporal condition update algorithm (pseudo code)

A periodic *ros_timer* with a frequency of 30 Hz is in the current implementation used for minimising the risk that the temporal condition expressions will be delayed in being evaluated by the frequency of the receiving *ros_message*. The same algorithm is used for *Rule* with the only difference that the *behaviour_exp* is replaced by *rule_exp*.

Internal variables are used by DeRoS to latch the needed values from every external message. A new message arrival triggers an update of the internal variable, therefore making it possible to assess the rules even if the external messages are coming asynchronously, and at different frequencies.

5 Implementation

5.1 Architecture

The architecture of the software components of a robot using the DeRoS-generated safety node is presented in Fig. 7. The *Robot sensors* component includes all the sensors of the robot and provides *sensor data* to the drivers. Following the example from Section 4.1, the figure depicts the drivers for the *IMU* and *Odometry*, but other driver-components could exist. Their role is to process the *sensor data* and to provide specific data messages (like *orientation* and *odometry*) to the components implementing the robot control - grouped in the figure under the *Control* block - as well as to the functional-safety responsible component *DeRoS safety node*. The *DeRoS safety node* component code is generated from the safety specification using the DeRoS language and its compiler. The *Actions* component takes as inputs the *functional-safety actions* and the *functionality actions* generated by the *DeRoS safety node* and by the *Control* components respectively, and resolves the conflicts between the two different sets of actions

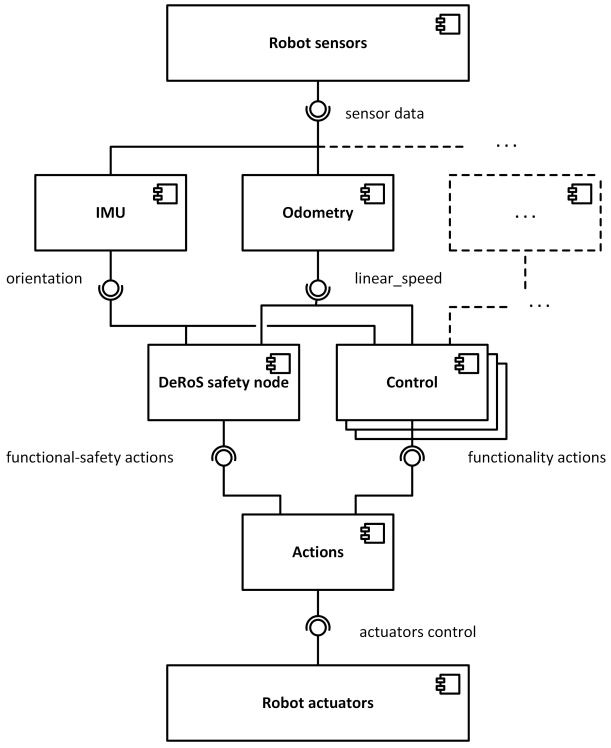


Fig. 7: DeRoS-based robot: component architecture

by up-prioritising the *functional-safety actions*. It provides the *actuators control* interface controlling the *Robot actuators*.

5.2 Code generation

We now describe the implementation of the *DeRoS safety node* component. The class diagram, presented in Fig. 8, consists of two classes: the generic ROS node boiler-plate type of class (*safety_nodeImpl*) and the subclass implementing the specific node functionality (*safetyClass*). The base class has several attributes: *node* storing the ROS node handle, *Parameters* where ROS node parameters values are kept in a structure, as well as handles for publishers (in the example *stop_robot_pub*) and the corresponding ROS messages (in the example *stop_robot_msg*). A part of the methods of *safety_nodeImpl* class configures a generic ROS node (*configureFromParameters*, *configurePublishers* and *configureSubscribers*) while the rest handles the node specific functionality (*on_fmInformation_imuMsg()* and *on_fmKnowledge_encoder_odomMsg()*) and are virtual methods required to be implemented in the subclass. In addition to the attributes inherited from the base class, the *safetyClass* has two ROS timers: one (*t*) used by the temporal conditions implementation and the other (*pub_state_t*) controls the frequency of the message used to publish the value of the *stop_robot_msg* implementing the *stop* action (see section 4.1). The *setup()* method initialises the node by calling the configuration methods of the base class and also calls an

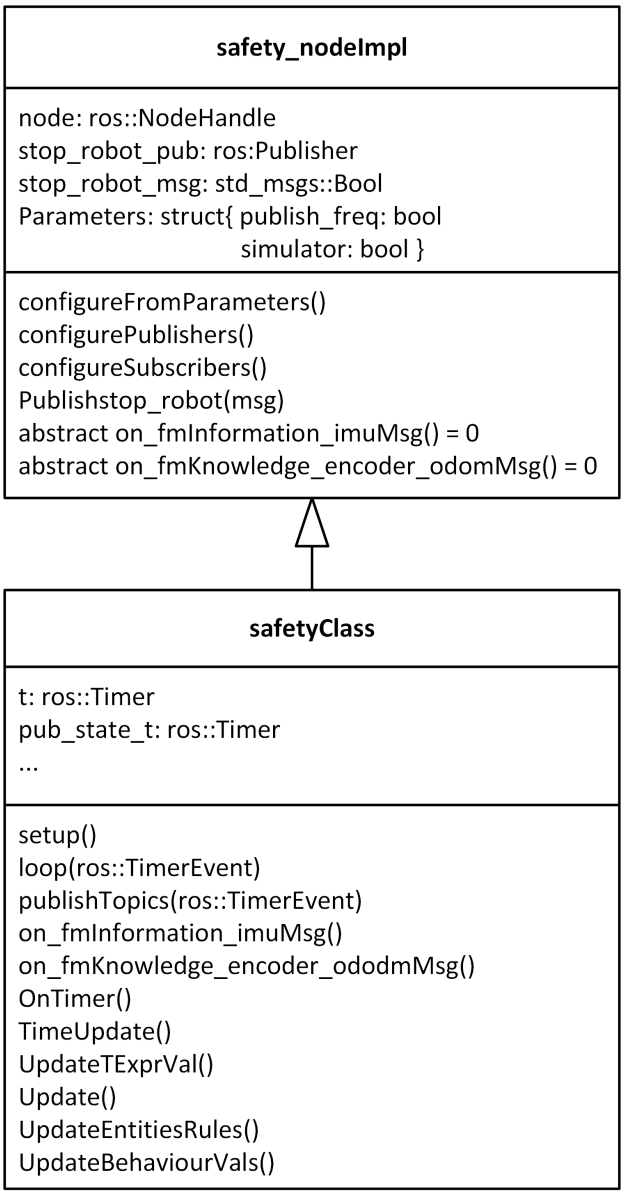


Fig. 8: Class diagram of the generated DeRoS safety node

external function (*CustomInit()*) used to specify additional robot-specific parameters, topics or timers.

The DeRoS compiler generates several files: the ROS node description file used by the ROS node code generator to produce the base class (*safety_nodeImpl*), and the files implementing the specific node functionality (*safetyClass*). Code generation for the former class relies on two library files: the first implementing the DeRoS built-in functions, e.g., *toRad*, *roll*, and the second implementing the functional safety robot-specific *action* functions and the *CustomInit()* function.

An excerpt from a DeRoS program and the generated C++ source code is shown in Fig. 9. The code generator of the DeRoS compiler creates corresponding callback functions

```
const max_speed = 0.5 m/s
...
input linear_speed =topic
  /fmKnowledge/encoder_odom.
  twist.twist.linear.x
...




...
entity drive_system {
  max_speed_exceeded :
  linear_speed > max_speed
  for 2 sec;
...




...
if drive_system.max_speed_exceeded
  then
  {
    stop;
  };
...




...
```

```

...
double n_9;
...
void ci_safetyClass::on_fmKnowledge_encoder_odomMsg(
        const nav_msgs::Odometry::ConstPtr& msg )
{
  n_9 = msg->twist.twist.linear.x ;
  ...
  TimeUpdate();
  Update();
}
...
bool t_0 = false;
bool t_0_expr  = false;
double t_0_val = 2.0;
ros::Time t_0_first_true = (ros::Time)0;
...
bool drive_system_max_speed_exceeded = false;
...
void ci_safetyClass::UpdateEntitiesRules()
{
  drive_system_max_speed_exceeded = ((n_9>0.5)&&t_0);
  ...
}
...
bool if_1 = false;
bool if_1_prev = false;
...
void ci_safetyClass::UpdateBehaviourVals()
{
  UpdateEntitiesRules();
    ...
  if_1=(drive_system_max_speed_exceeded==true);
}
...
void ci_safetyClass::Update()
{
  UpdateBehaviourVals();
    ...
  if(if_1 != if_1_prev)
  {
    if_1_prev = if_1 ;

    if(if_1 == true)
    {
        stop(this) ;
    }
  }
}
...
```

Fig. 9: DeRoS code generation. Left: DeRoS code excerpt, right: the generated C++ code

```
void ci_safetyClass::TimeUpdate()
{
  UpdateBehaviourVals();
  UpdateTExprVal();

  ros::Time crt_time = ros::Time::now();
  if(t_0_expr)
  {
    if(t_0_first_true == (ros::Time)0)
      t_0_first_true = crt_time;
    else
    {
      double crt_duration = crt_time.toSec() -
                  t_0_first_true.toSec() ;
      if(crt_duration >= t_0_val)
      {
        t_0 = true;
        Update();
      }
      t_0_first_true = (ros::Time)0;
    }
  }
  else
  {
    t_0_first_true = (ros::Time)0;
    t_0 = false;
  }
  ...
}
```

Fig. 10: DeRoS time condition generated code

($CallbackFn$) for every $Topic$, part of each $Rule$ from the $Entity$ section as well as unique variables for each ROS topic expression $topic\_exp$. For example, for lines 3-5 in the DeRoS program, the compiler generates the lines 3-12.

If a temporal condition expression is present in the rule, either in the $Entity$ or $Behaviour$ section, a set of boolean variables are generated. Also, for each $Rule$ expression, a boolean variable is created. In the example, for lines 13-16 in the DeRoS program, lines 14-19 are generated.

The $Rule$ expressions in each $Entity$ are evaluated inside the *UpdateEntitiesRules()* function, shown in lines 21-25. The code replaces the *max_speed* with its numerical value using the information from line 1 in the DeRoS code excerpt.

The code for the $Behaviour$ part is generated in a similar way. For example, for the lines 27-31 in the DeRoS program, variables are generated (lines 27-28 in the generated code) and the $Behaviour$ rule expression associated variable is updated by the *UpdateBehaviourVals()* function (lines 30-35 in the generated code). The code generation for the $Behaviour$ ends with the *Update()* function from where the safety actions are called (lines 37-50 in the generated C++ code).

The code for updating the variables for the time conditional expressions is part of the *TimeUpdate()* function implementation. The code generation for the *t_0* part of the function shown in Fig. 10 follows the algorithm presented in Fig. 6.

The code generation for functions updating variables when a new message is received or when a periodic timer expires follows the action diagram presented in Fig. 5: in our example, the *CallbackFn* is *on_fmKnowledge_encoder_odomMsg()* which calls the *TimeUpdate()* and the *Update()* functions. The same *TimeUpdate()* function responsible for updating the temporal conditions (together with the *UpdateTExprVal()* function) is called by the code generated for the callback function bound with the timer event:

```
void ci_safetyClass::OnTimer()
{
  TimeUpdate();
}
```

The $Rule$ and $Entity$ expressions are updated inside the *UpdateEntitiesRules()* function, while the $Behaviour$ updates are implemented in the *UpdateBehaviourVals()* and the actions are triggered inside the *Update()* function implementation.

5.3 ROS integration

A ROS-based robot is composed of ROS nodes intercommunicating by the means of topics using a publish/subscribe mechanism. Writing a ROS node in C++ requires considerable boiler-plate code, and in order to speed up the process of writing the code for such a node, a ROS node code generation DSL has been created.

Usually, ROS-based robot code is launched by means of a launch file where parameters, subscribing and publishing topics interconnecting the nodes are specified. For complicated robots the launch files could be quite large and often contain errors [44]. To address this, we have created a DSL tool for specifying and automatically generating the launch file for a ROS system. From a DeRoS point of view, the actual language used for modelling the system is not that important as long as the model contains the required information (the publishing topics and the corresponding message types).

Although the BRIDE toolchain [59], like our ROS modelling DSL, offers similar capabilities to generate the skeleton source code for ROS nodes and to produce the corresponding launch files, it lacks support for integrating the legacy code. BRIDE promotes a visually-based design, in contrast with our approach which uses a human-readable file-based textual DSL. Moreover, the DeRoS-based complete generation of the ROS safety node for the behavioural part of the node represents a step forward compared to BRIDE, since its code generation capabilities are limited to the instantiation of the components forming the system, and to the application-specific logic.

5.3.1 ROS node description file

A dedicated DSL is used to model each component (ROS node) which is available for the system designer to use. The DSL serves two purposes, the first is to provide the component developer with a language which can quickly generate the boiler-plate code for the target implementation

```
node ci_safety_node : ci_safety

parameters
  name time_update_freq : double = 10.0
  name publish_freq : double = 10.0
  name upside_down : bool = false

publishes
  topic stop_robot : std_msgs.Bool

subscribes
  topic /fmInformation/imu : sensor_msgs.Imu
  topic /fmKnowledge/encoder_odom :
              nav_msgs.Odometry
```

Fig. 11: ROS node description file example

language (C++/Python), the second purpose is to capture the interface of the component in a model which can be easily accessed by other tools.

The DSL allows the component developer to model the parameters, publishers, subscribers and their respective datatype. Furthermore, there is language support for specifying units on the communicated messages. Fig. 11 shows an example of the language in use.

The ROS node can be generated in either C++ or Python. The generated code consists of a class which has a public publisher method for each declared publisher in the model. For all subscribers declared an unimplemented method is generated which must be implemented (by subclassing) by the component developer. Finally, all parameters are provided as members of a struct, which can be accessed and used by the subclass. An example of the class diagram of a generated ROS node is shown in in Fig. 8.

5.3.2 ROS system description file

The ROS system description DSL is a language for declaring systems of components. An excerpt from an example file is presented in Fig. 12. The top-level part of this language is a system, which contains a set of topics to which publishers and subscribers can be connected. Furthermore, the system contains a number of subsystems declared by the system designer which can be used to group components. A component is an instantiation of a ROS node DSL component, and the system designer can assign values to declared parameters, and can connect publishers and subscribers to topics. The ROS system DSL can validate that both the publishers and the subscribers in a connection use the same datatype.

5.4 DeRoS development environment

The DeRoS language, developed following an MDSD approach, relies on a text editor developed using the Xtext [79] framework. The editor provides static validations such as syntax checking, model correctness validation for, e.g., topics

```
system RobotSafety

imports
  import "vectornav_vn100_node.rnd"
  import "differential_odometry_node.rnd"
  ...
topics
  topic /fmInformation/imu
  topic /fmKnowledge/encoder_odom
  ...
subsystem  Safety
{
  subsystem fmSensors
  {
    node vectornav_vn100:vectornav_vn100_node
    parameters
      param publish_topic_id =
        "/fmInformation/imu"
      param frame_id = "imu_link"
      ...
    wirings
      connect publish_topic_id to
        /fmInformation/imu
      ...
  }

  subsystem fmProcessors
  {
    ...
    node differential_odometry : odometry_node
    wirings
      connect odom_pub to
        /fmKnowledge/encoder_odom
  }
...
```

Fig. 12: Excerpt from ROS system description file

being defined in the system file, but also supports the programmer with assistance for completion or keyword and error highlighting. All those features are important when developing safety-related software: the tools used for development should help to avoid introducing bugs.

6 Case Study

The process of developing a commercial robot means fulfiling two closely interconnected aspects: completing its mission and ensuring that the robot remains safe. The latter part is a mandatory requirement from the legal point of view. Often, certification against safety standards is the preferred way to demonstrate the compliance to the safety requirements. This certification confirms that the robot's implementation fulfils the requirements of the claimed standards, and that it will react safely in the contexts identified during the risk assessment. This could imply that in specific cases safety will prevail mission fulfilment.

This section exemplifies the process of writing a DeRoS program, starting from the risk assessment, identifying the

Fig. 13: The *FIX-Robo* robot

functional safety features, deciding the safety-related actions and ending with the DeRoS program and tests in a realistic scenario.

6.1 The robot used: *FIX-Robo*

FIX-Robo [80] (see Fig. 13), is a relatively large mink-feeding robot: its weight is several hundreds of kilograms, and it is powered by a diesel engine. For navigation it uses RTK GPS outdoors and indoors it relies on a laser scanner (LIDAR), wheel encoders and an IMU sensor. In addition, an RFID reader helps the indoors position estimation. The high-reliability part of the functional safety of the robot is handled by a safety PLC in charge of shutting off the diesel engine when any of the bumper sensors, feeding arm displacement sensor or stop button are activated. The PLC exchanges status information with the high-level controller, but also monitors it. The high-level controller software is built around the FroboMind ROS-based framework for field robots [81], and runs on an Intel i5-2210M platform with 4GB RAM and 16GB SSD disk drive. The operating system is Ubuntu 12.04. The robot is hydraulically powered for both the Ackerman steering and the driving system. A CAN-based controller steers the robot via electromagnetic valves. The moving speed and direction are controlled by the position of a single pedal, which stops the movement when it is in its neutral position. The pedal, when the robot is in automatic mode, is moved by a linear actuator. A stack light shows the state of the robot, and a buzzer generates warning sounds when instructed by the main controller. The large size and the significant engine power increase the safety risks of the robot, therefore the functional safety ROS node, generated using the DeRoS DSL, together with the PLC based system, play a central role in the safety-related part of the robot's commercial certification.

6.2 Risk assessment and the safety states

A risk assessment followed by a risk reduction was conducted as described in ISO 12100. As a result, an Excel file containing the list of the highest risks associated with the robot functionality, together with ways of reducing them, has been produced. For the *FIX-Robo*, the highest risks identified were: the robot moving too fast, roll-over, hitting an obstacle, humans being unaware of the robot state, carbon monoxide accumulation in case of the robot being stuck for a prolonged time while the engine is still running, collision of the feeding arm with obstacles, and main robot controller software malfunctioning. Based on the identified list of hazards required to be lowered, the functional safety requirements have been identified: (1) an average nominal speed of 0.5m/s with an absolute maximum of 1m/s when driving forward and a nominal speed of 0.25 m/s with an absolute maximum of 0.4 m/s when driving backwards; (2) nominal maximal robot tilt of 8 degrees with a momentary maximum of 10 degrees; (3) three safety zones supervised by LIDAR: a warning zone, a stop zone and a collision zone; (4) bumpers and stop button triggering the engine stop; (5) a time limit of 10 minutes for the engine running while the robot is in automatic mode and is not moving; (6) a keep-alive signal to be generated by the main robot controller; (7) feeding arm collision sensor triggering an engine stop. The identified risks match the list of safety functions mentioned in ISO 13842. The exceptions are: limits to workspace (not needed because the farms where the robot is supposed to operate are fenced) and force control (softness of the chosen bumpers combined with the limited pushing power of the robot resulted in assessing dedicated force control functionality as being unnecessary to implement).

Safety-related standards, such as ISO 13849, specify that the robot should be placed in a safe state in case of a safety action being triggered. For the *FIX-Robo*, the identified safety states are low speed and robot movement stop, with or without engine being turned off. The safety actions triggered for entering the safety states are: (1) lower the speed in case of exceeding the average speed or when an obstacle has been detected inside the warning zone, (2) stop the robot in case the maximum speed has been exceeded, an obstacle has been detected inside the stop zone, the tilt angle has been exceeded, stop button, bumper or arm sensor has been triggered, keep-alive signal from the main controller is missing or the 10 minutes time limit has been exceeded for the robot not moving while being in automatic mode.

A problem that needed to be addressed when considering the safety states and the safety actions is their possible interference with the actions initiated by the mission controller. Currently, DeRoS provides no direct solution for this issue, but relies on the mission controller being resilient towards the safety actions. The safety rules and their actions are known when the functionality of the mission controller is implemented, and are supposed to be considered together with the correct robot state handling in case of the safety actions being triggered. The results of the risk assessment are also implemented by the robot's main controller, and not only by the DeRoS-generated node which acts like the safety bag in Elektra [48].

6.3 The DeRoS program for *FIX-Robo*

The functional safety requirements identified during the risk assessment are used to write the DeRoS program. First the safety actions and the constants used are declared:

```
action low_speed;
action stop;
action motor_stop;
action controllerOk;

const max_tilt_neg = toRad(-8 deg)
const max_tilt_pos = toRad(8 deg)
const danger_tilt_neg = toRad(-10.0 deg)
const danger_tilt_pos = toRad(10.0 deg)
```

The actions are implemented in a dedicated robot-specific library. The `low_speed` and `stop` actions use the main robot's controller functionality through a dedicated topic. The `motor_stop` action is implemented in the PLC code and is triggered by the safety ROS node generated by the DeRoS code. Between the PLC and the main robot's controller, a keep-alive signal is used to both signal that the controller works fine and to initiate an engine stop by purposely interrupting the keep-alive signal. The `stop` action results in the movement of the speed-controlling pedal in the neutral position. Both `stop` and `low_speed` actions are reset when the `controllerOk` action is called. In contrast, the `motor_stop` action once triggered, remains active until an operator manually resets it.

```
entity imu_sensor_system {
  tilt_not_ok:
    ((orientation.pitch() not in
    [max_tilt_neg , max_tilt_pos])
    or (orientation.roll() not in
      [max_tilt_neg , max_tilt_pos])) for 4sec;

  rollover_imminent:
    ((orientation.pitch() not in
    [danger_tilt_neg , danger_tilt_pos])
    or (orientation.roll() not in
    [danger_tilt_neg , danger_tilt_pos]));
}
```

Similarly, the entity representing the collision detection system defines states for no obstacle, far obstacle and close obstacle, and collision danger. The entity representing the PLC system defines the stop button, bumpers or arm being activated, as well as the manual, learning and auto modes. Last, the drive system defines the moving states: stopped, moving, maximum speed being exceeded or dangerous speed.

The behavioural rules, where the actions to be taken when the safety rules are triggered, have been defined in terms of these entities. The following code exemplifies safety rules for the tilt limit being exceeded and the stop button being activated:

```
// tilt limit exceeded
if((imu_sensor_system.tilt_not_ok
  and drive_system.moving
  and PLC_system.auto_mode)
  or imu_sensor_system.rollover_imminent) then
{ motor_stop; stop;
  log_state("motor stop: max tilt exceeded");
};

// stop button activated
if(PLC_system.stop_button_activated
  and (PLC_system.auto_mode
  or PLC_system.learn_mode)) then
{ motor_stop; stop;
  log_state("motor stop: stop button activated");
};
```

A total of 14 safety behavioural rules have been defined covering the safety cases identified during the risk assessment. One of the safety behavioural rules is used to reset the safety system when the operator resolves the safety issue by placing the robot in manual mode using a dedicated switch:

```
if( PLC_system.manual_mode) then
{ controllerOk;
  log_state("safety: manual mode");
};
```

6.4 DeRoS in action: *FIX-Robo* safety test

The functional safety features of the *FIX-Robo* have been tested in a mink farm as illustrated in Fig. 14. Different test scenarios have been used to test the fulfilment of the safety requirements identified during the risk assessment (annotated on the map) as follows: (1) Warning zone triggered; (2) Stop zone triggered; (3) Collision zone triggered; (4) Front bumper triggered; (5) Rear bumper triggered; (6) Arm collision triggered; (7) Stop button activated; (8) Forward maximum speed exceeded for less than 4 seconds; (9) Forward maximum speed exceeded for more than 4 seconds; (10) Forward dangerous speed exceeded momentarily; (11) Backward maximum speed exceeded for less than 2 seconds; (12) Backward maximum speed exceeded for more than 2 seconds; (13) Backward dangerous speed exceeded momentarily; (14) Software crash; (15) Stationary in auto mode for more than 10 minutes.

Test cases 1-7 were performed by manually triggering the safety events while the robot was in autonomous mode. Test cases 8-13 were performed when the robot was in a semi-autonomous mode by manually overriding the actuator controlling the speed pedal. Test case 14 was executed by simulating a software crash, by manually killing the DeRoS node process in Linux, and for test case 15 the robot was kept in idle state (not moving) for more than 10 minutes.

The robot responded in every test case as expected: in test case 1 the robot reduced the speed; in test case 2 and 3 it stopped; in test cases 4-7, 9-10, 12-15 the robot stopped and also the engine was turned off; in test cases 8 and 11 the robot had no safety reaction as expected (those test cases are negative tests cases for 9 respectively 12).

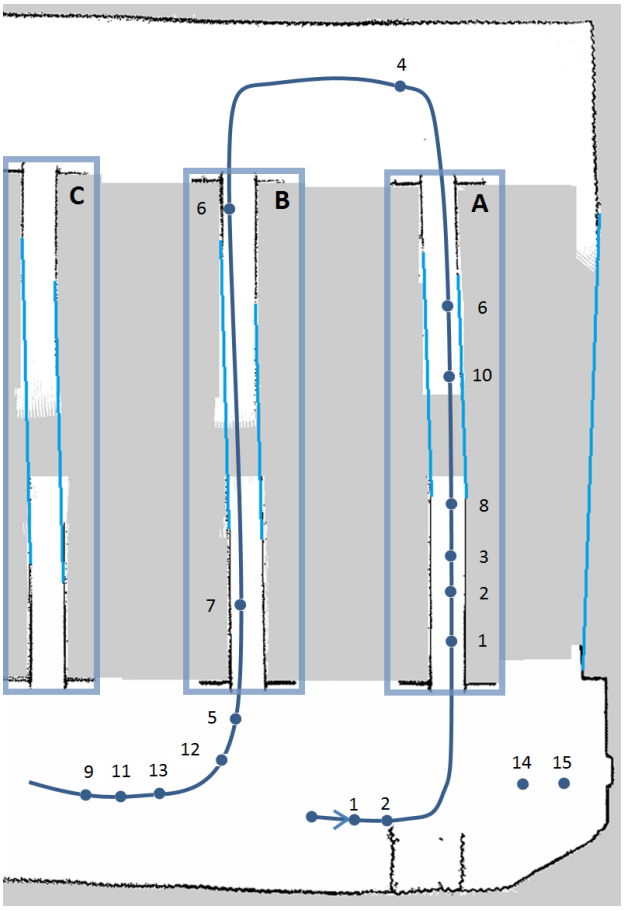


Fig. 14: The *FIX-Robo* safety test in a mink farm. A-C are mink stalls, 1-15 are the places where the test cases have been triggered

Fig. 15: The *Intelligent One* robot

7 Experiments

7.1 Experimental setup

Experiments have been conducted using two different robots developed at Conpleks, and have been performed in two scenarios: using the physical robot and simulating it in Gazebo [82].

The first robot used is the mink-feeding robot (*FIX-Robo*) presented in Section 6. The second robot, the *Intelligent One* [83] (see Fig. 15), is designed for marking sport fields. It is differentially steered and the low-level interface resembles that of many tracked and wheeled field robots. For navigation it uses an RTK GPS together with wheel encoders and an IMU sensor, while for propulsion it uses two electrical motors with built-in controllers steered through a CAN interface. Although the controller from a hardware point of view is the same as the one used by *FIX-Robo*, software-wise it is significantly simpler, reflecting the lower complexity of the robot missions. It runs on the same operating system (Ubuntu 12.04) as *FIX-Robo*.

Both robots contain safety-related nodes implemented using the DeRoS DSL. For *Intelligent One*, two safety rules have been used (maximum speed and tilt being exceeded), and for *FIX-Robo* a total of 4 entities summing up 14 rules and 13 safety actions triggering behaviour rules have been implemented after being identified by the risk assessment performed for the commercial robot. For comparison reasons, we wanted to use similar safety rules for both robots, but the maximum tilt exceeded experiment for *FIX-Robo* is both complicated and dangerous to perform, especially if repeated several times, therefore only the ”maximum speed being exceeded safety rule” performance has been assessed for it.

In all cases, for both robots, the experiments have been repeated at least ten times. In the experiments, we have measured the time between when a certain safety rule condition has been fulfilled and the assigned action has been triggered.

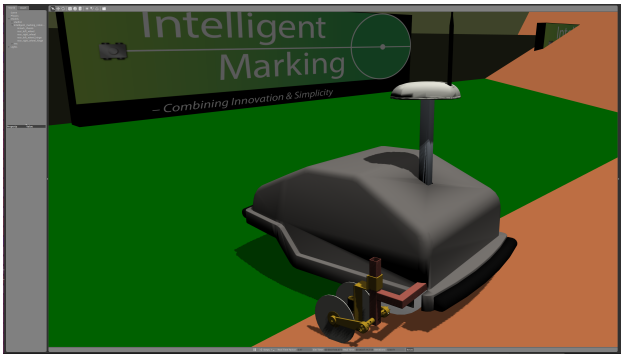
7.2 DeRoS performance tests

7.2.1 Experiments on physical robots

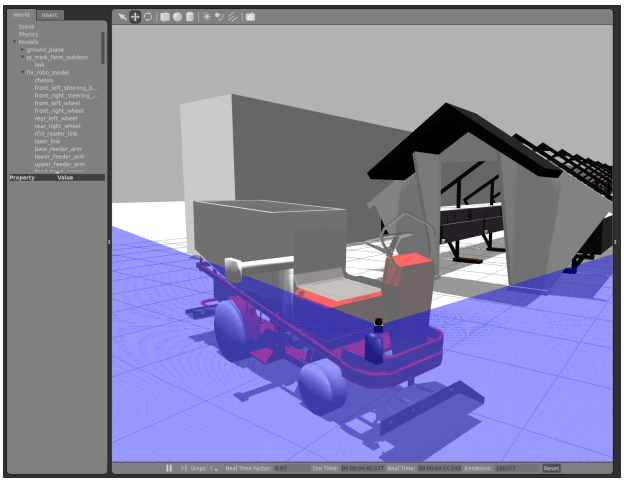
In all tests performed, the *Intelligent One* robot has been manually remote-controlled using an Android tablet running a specially developed control app. For the *FIX-Robo*, the maximum speed exceeded tests have been performed by manually driving the robot. All the other safety rules in the safety specification file have been verified for correctness while the robot has been placed in autonomous mode. For both robots, the results have been recorded during the experiments in ROS bags, and later processed and analysed.

7.2.2 Simulated experiments

The Gazebo simulator (version 1.9) has been used to perform the simulated experiments, for both *Intelligent One* and *FIX-Robo* (see Fig. 16). The robots have been manually controlled



(a) The *Intelligent One* robot



(b) The *FIX-Robo* robot

Fig. 16: The (a) *Intelligent One* and (b) *FIX-Robo* robots in Gazebo

from the keyboard via a specially developed ROS steering node. During the experiments, the results have been recorded in ROS bags, and later processed and analysed in the same way as for the physical robot type of experiments.

7.2.3 Relation to previous experiments

In previous work [14], we have performed experiments on two other robots running different software: a physical *Frobit* robot and the standard, simulated iClebo *Kobuki* robot from the ROS distribution. The Frobit [84] is a small, low-cost robot designed for rapid prototyping; it is running the standard deployment of the FroboMind ROS-based software framework for field robots [8] 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.

The experiments were conducted using a specially written ROS test node, as illustrated in Fig. 17. The node consists of a common part implementing the launch of the test cases and another part containing the test case-specific code for the test environment (physical or simulated). For the Frobit, the test node was able to physically interrupt control lines of the wheel encoders and motors through an Arduino-type board communicating with the test program via a serial connection. For the Kobuki, the velocity commands coming from the robot's controller are altered by the test program to simulate misbehaviour. An earlier version of the DeRoS language, generating Python code, was used. In all cases we measured the time between the introduced fault and the sending of the stop command by the safety node.

Experiment	Ideal	Avg	SD	Min	Max
Intelligent One					
R_S	2.0	2.00271	0.00461	2.00006	2.01374
S_S	2.0	2.01150	0.00741	2.00200	2.02500
R_T	0.4	0.40635	0.00916	0.40002	0.42600
S_T	0.4	0.40825	0.00526	0.40000	0.42300
FIX-Robo					
R_S	4.0	4.00509	0.00400	4.00127	4.01705
S_S	4.0	4.00462	0.00747	4.00000	4.02000

TABLE 1: Experimental results. All times are in seconds. R indicates real robot, S indicates simulator performed experiments, in both cases maximum speed $_S$ or maximum tilt $_T$ exceeded

For the Frobit experiments, our DSL was used to implement rules supervising the wheel encoders, and limiting the linear speed of the robot (in total 3 entities and 9 rules). For the Kobuki experiments, the rules enforced 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).

The results of these experiments performed with the earlier DeRoS version generating Python code proved the feasibility of the concept: our language and generator were capable of enforcing safety rules relating to hardware failure and could be applied to standard robots such as the Frobit and Kobuki. However, there was a poor response time in the safety node (for details please consult [14]). The average reaction time to a safety rule violation exceeded the ideal minimum expected reaction time between 50% and 240% in case of the Frobit experiments and between -15% and 23% in the case of Kobuki experiments (with censoring of a few spurious values). The standard deviation in case of the proof-of-concept experiments was between 0.02 and 0.40. The main reason for the observed low performance is the polling mechanism used for the time conditional rule expressions.

7.3 Results

Table 1 presents the data obtained for our experiments conducted with the *Intelligent One* and the *FIX-Robo*. The columns in the table refer to the ideal minimum expected time according to the safety rules (Ideal), average time over the repeated experiments (Avg), standard deviation (SD), minimum (Min) and maximum (Max) reaction time of the safety-related node measured during the experiments.

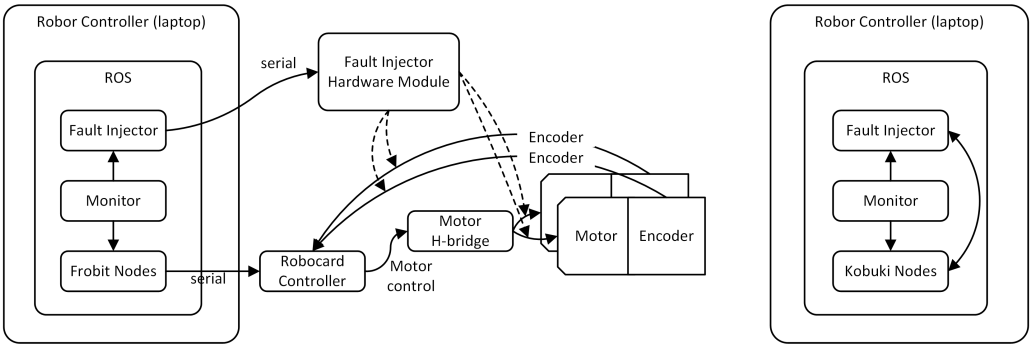


Fig. 17: Robot setup: Frobit hardware setup (left) and Kobuki setup (right), earlier experiments [14]

During the experiments, the top 10 maximum processor-intensive OS processes have been monitored with once-per-second sampling frequency. The ROS node running the safety specification was always using under 2% of the processor power and less than 0.1% of the total available memory.

7.4 Discussion

All the tests performed were successful: the robot stopped when expected to stop. The differences in reaction time between the experiments performed using the real robot and the simulator are very small, showing similar reaction time in both types of experiments. The results are in line with our expectations from such an event-driven system: reaction time spreads in the range of 2 ms to 11 ms.

The results obtained demonstrate that the new event-driven approach is a better choice for a safety-responsible node. The average response time in case of a safety rule violation has been in the range of 0.12% to 2% from the ideal one, but always delayed no more than 11 ms. Also the standard deviation is low (between 0.004 and 0.009). In contrast, the earlier Python-based implementation showed a delay in the response time between 7 ms and 700 ms, most of the time exceeding 20 ms. Also the standard deviation was high, between 0.02 and 0.40.

8 Conclusion

We have shown that it is possible to use DeRoS to generate the implementation of the safety rules identified during a safety risk assessment. DeRoS has a simple syntax, making it easy to express the safety rules, and enables the generation of runtime safety monitoring code, as our experiments demonstrate. Moreover, based on our experience for using the DSL for safety certification of a commercial robot, we find that addressing the functional safety of robots with DeRoS is efficient and easily customisable, even with partial access to source code.

In most experiments a complete stop action has been used as the reaction to any safety rule violation. That fulfilled our robots safety requirements, but is obviously not the best action for other robots. An improved fault-handling, based on diagnosis and fault isolation, will be addressed in a future work together with enhancing the DeRoS language for static detection of overlapping safety rules or of potential contradictions. We note that for the implementation of safety-related rules, when using our DSL, we are dependent on the interfaces of the robot (e.g., the exposed interfaces between the different ROS nodes). If the needed information for implementing the safety rules is not published, the appropriate ROS nodes of the robot would have to be modified to publish it.

The DeRoS language has been used mainly by the authors, but the language syntax has been informally discussed with other people. An empirical investigation of the usability of the DeRoS language is considered future work. The DSL modelling ROS nodes has been used, with positive feedback, by all the software engineers working at Conpleks. However, the scalability of this DSL is unknown as it is custom to our work, and therefore not being used or validated by others. The reason for the development of the language was the need to capture the system structure (nodes, topics) in a model which could be queried from DeRoS. Alternatives to replace our custom DSL for modelling the ROS system structure could be the AADL language [65]. We also expect that a more comprehensive model of the system itself can extend the safety monitoring coverage.

The DeRoS language syntax has been extended compared to the previous work, e.g., with support for alias names for ROS topics, constants can be defined, measurement units can be specified, intervals are legal in logical expressions. All those additions together with a general syntax clean-up have greatly improved the readability of programs written in DeRoS. Our proposed safety architecture proved helpful during the safety certification process for CE marking a commercial robot (*FIX-Robo*). Further, the experiments with our DSL on other robots, demonstrates the validity of the architecture for a wide range of robot complexity and size.

Acknowledgements

The work for this paper has been partially supported by the SAFE project. The authors would like to thank the anonymous

reviewers of the earlier version of this paper, presented at SIMPAR 2014, as well as the participants providing feedback.

REFERENCES

[1] D. Kohanbash, M. Bergerman, K. M. Lewis, and S. J. Moorehead, "A safety architecture for autonomous agricultural vehicles," in *American Society of Agricultural and Biological Engineers Annual Meeting*, July 2012. 1

[2] H. Griepentrog, N. Andersen, J. Andersen, M. Blanke, O. Heinemann, T. Madsen, J. Nielsen, S. Pedersen, O. Ravn, and D.-L. Wulfsohn, "Safe and reliable: further development of a field robot," in *Precision agriculture '09*, E. Henten, D. Goense, and C. Lokhorst, Eds. Wageningen Academic Publishers, 2009, pp. 857–866. 1

[3] S. Bouraine, T. Fraichard, and H. Salhi, "Provably safe navigation for mobile robots with limited field-of-views in unknown dynamic environments," in *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA)*, May 2012, pp. 174–179. 1

[4] H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter, "Guaranteeing functional safety: Design for provability and computer-aided verification," *Auton. Robots*, vol. 32, no. 3, pp. 303–331, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10514-011-9271-y 1

[5] H. Griepentrog, C. Jæger-Hansen, O. Ravn, N. Andersen, J. Andersen, and T. Nakanishi, "Multilayer controller for field robots - high portability and modularity to ease implementation," Hannover, Germany, p. 6, 2012, paper presented at LAND.TECHNIK - AgEng 2011. 1

[6] A. Beck, N. Andersen, J. Andersen, and O. Ravn, "MobotWare - a plug-in based framework for mobile robots," in *IAV 2010*. Int. Fed. of Automatic Control, 2010. 1

[7] A. Reske-Nielsen, A. Mejnertsen, N. A. Andersen, O. Ravn, M. Nørremark, and H. W. Griepentrog, "Multilayer controller for outdoor vehicle," *Zonn: Automation Technology for Off-Road Equipment Bonn, Germany*, vol. 1, pp. 41–49, 2006. 1

[8] K. Jensen, A. Bøgild, S. Nielsen, M. Christiansen, and R. Jørgensen, "FroboMind, proposing a conceptual architecture for agricultural field robot navigation," Spain, 2012, paper presented at CIGR 2012. 1, 7.2.3

[9] P. Nebot, J. Torres-Sospedra, and R. J. Martnez, "A new HLA-based distributed control architecture for agricultural teams of robots in hybrid applications with real and simulated devices or environments," *Sensors*, vol. 11, no. 4, pp. 4385–4400, 2011. [Online]. Available: http://www.mdpi.com/1424-8220/11/4/4385 1

[10] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the IEEE ICRA 2001*, vol. 3, 2001, pp. 2523–2528 vol.3. 1, 3.1

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5. 1

[12] A. Lotz, A. Steck, and C. Schlegel, "Runtime monitoring of robotics software components: Increasing robustness of service robotic systems," in *Proceedings of the 15th International Conference on Advanced Robotics*. IEEE, 2011, pp. 285–290. 1, 3.1, 4.1

[13] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: the HyperFlex toolchain," in *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 6414–6420. 1, 2.4, 3.1

[14] S. Adam, M. Larsen, K. Jensen, and U. P. Schultz, "Towards rule-based dynamic safety monitoring for mobile robots," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 207–218. 1.1, 7.2.3, 17

[15] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. 2.1, 2.3

[16] É. Baudin, J.-P. Blanquart, J. Guiochet, and D. Powell, "Independent safety systems for autonomy," Citeseer, LAAS-CNRS, Tech. Rep. 07710, September 2007. 2.1

[17] ISO, "Robots and robotic devices safety requirements for industrial robots part 2: Robot systems and integration," International Organization for Standardization, Geneva, Switzerland, ISO 10218-2, 2011. 2.1

[18] ——, "Robots and robotic devices–Safety requirements–Non-medical personal care robot," International Organization for Standardization, Geneva, Switzerland, ISO 13482, 2014. 2.1

[19] A. Lotz, A. Hamann, I. Lütkebohle, D. Stampfer, M. Lutz, and C. Schlegel, "Modeling non-functional application domain constraints for component-based robotics software systems," in *International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-15)*, 2015. 2.2

[20] R. Baheti and H. Gill, "Cyber-physical systems," *The impact of control technology*, vol. 12, pp. 161–166, 2011. 2.2

[21] B. A. Gran and J. E. Simensen, "Applying a reliability metric for assessing computer-based logical diagrams," in *Safety and Reliability of Complex Engineered Systems: ESREL 2015*. CRC Press, 2015, pp. 4045–4052. 2.3

[22] N. Yakymets, S. Dhouib, H. Jaber, and A. Lanusse, "Model-driven safety assessment of robotic systems," in *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013, pp. 1137–1142. 2.3

[23] D. Crestani, K. Godary-Dejean, and L. Lapierre, "Enhancing fault tolerance of autonomous mobile robots," *Robotics and Autonomous Systems*, vol. 68, pp. 140–155, 2015. 2.3, 2.5

[24] A. M. Mokhtar, J. Guiochet, D. Powell, J.-P. Blanquart, and M. Roy, "Elicitation of executable safety rules for critical autonomous systems," *Embedded Real Time Software and Systems (ERTS2), Toulouse, France*, 2012. 2.3, 2.5

[25] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell, "Fault tolerance in autonomous systems: How and how much," in *Proceedings of the 4th IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments, Nagoya, Japan*, 2005. 2.3

[26] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2004. 2.4

[27] D. Powell, J. Arlat, Y. Deswarte, and K. Kanoun, "Tolerance of design faults," in *Festschrift Randell*, C. Jones and J. Lloyd, Eds., 2011, pp. 428–452. 2.4

[28] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, ser. LNAI, no. 6472. Darmstadt: Springer, 2010, pp. 324–335. 2.4

[29] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006. 2.4

[30] N. Yakymets, S. Dhouib, H. Jaber, and A. Lanusse, "Model-driven safety assessment of robotic systems," in *Proceedings of the 2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2013. 2.4

[31] J. Corrales, F. Candelas, and F. Torres, "Safe humanrobot interaction based on dynamic sphere-swept line bounding volumes," *Journal of Robotics and Computer-Integrated Manufacturing*, vol. 27, pp. 177–185, Feb. 2011. 2.4

[32] A. Paraschos, N. Spanoudakis, and M. Lagoudakis, "Model-driven behavior specification for robotic teams," in *Proceedings of the 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Valencia, Spain, Jun. 2012. 2.4

[33] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 2.4

[34] R. Oftadeh, M. M. Aref, R. Ghabcheloo, and J. Mattila, "Unified framework for rapid prototyping of Linux based real-time controllers with Matlab and Simulink," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2012. 2.4

[35] M. Bordignon, K. Stoy, and U. Schultz, "Generalized programming of modular robots through kinematic configurations," in *Proceedings of IROS 2011: the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 3659–3666. 2.4

[36] U. Schultz, M. Bordignon, and K. Stoy, "Robust and reversible execution of self-reconfiguration sequences," *Robotica*, vol. 29, no. 1, pp. 35–57, 2011. 2.4

[37] A. Steck, A. Lotz, and C. Schlegel, "Model-driven engineering and runtime model-usage in service robotics," in *Proceedings of Generative Programming and Component-Based Engineering (GPCE)*. ACM, 2011. 2.4, 3.1, 4

[38] J. Inglés-Romero, C. Vicente-Chicote, and O. B. B. Morin, "Using Models@Runtime for designing adaptive robotics software: an experience report," in *Proceedings of the 1st International workshop on Model Based Engineering for Robotics (RoSym10)*, 2010. 2.4

[39] D. Pilaud, "Efficient automatic code generation for embedded systems," *Microprocessors and Microsystems*, pp. 501–504, 1997. 2.4

[40] S. Chaa and J. Yoob, "A safety-focused verification using software fault trees," *Future Generation Computer Systems*, vol. 28, pp. 1272–1282, Oct. 2012. 2.4

[41] O. Ljungkrantz, K. Åkesson, C. Yuan, and M. Fabian, "Towards industrial formal specification of programmable safety systems," *IEEE Transactions on Control Systems Technology*, vol. 20, pp. 1567–1574, 2012. 2.4

[42] A. de Roo, H. Sözer, and M. Aksit, "Verification and analysis of domain-specific models of physical characteristics in embedded control software," *Journal of Information and Software Technology*, vol. 54, pp. 1432–1453, Dec. 2012. 2.4

[43] B. H. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe *et al.*, "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.time*. Springer, 2014, pp. 101–136. 2.4

[44] M. Larsen, S. Adam, U. Schultz, and R. N. Jørgensen, "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, pp. 409–418. 2.4, 5.3

[45] D. Crestani and K. Godary-Dejean, "Fault tolerance in control architectures for mobile robots: Fantasy or reality?" in *7th National Conference on Control Architectures of Robots*, Nancy, France, 2012. 2.5

[46] M. Blanke, M. R. Blas, S. Hansen, J. C. Andersen, and F. Caponetti, "Autonomous robot supervision using fault diagnosis and semantic mapping in an orchard," in *Fault Diagnosis in Robotic and Industrial Systems*. iConcept Press Ltd, 2012, pp. 1–22. 2.5

[47] V. Gribov and H. Voos, "A multilayer software architecture for safe autonomous robots," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE, 2014, pp. 1–8. 2.5

[48] P. Klein, "The safety-bag expert system in the electronic railway interlocking system Elektra," *Expert Systems with Applications*, vol. 3, no. 4, pp. 499–506, 1991. 2.5, 6.2

[49] J. Fox, "Designing safety into medical decisions and clinical processes," in *Computer Safety, Reliability and Security*. Springer, 2001, pp. 1–13. 2.5

[50] C. Pace and D. Seward, "A safety integrated architecture for an autonomous safety excavator," in *International Symposium on Automation and Robotics in Construction*, 2000. 2.5

[51] F. Py and F. Ingrand, "Online execution control checking for autonomous systems," *Intelligent Autonomous Systems 7*, p. 273, 2002. 2.5

[52] S. Roderick, B. Roberts, E. Atkins, and D. Akin, "The Ranger robotic satellite servicer and its autonomous software-based safety system," *Intelligent Systems, IEEE*, vol. 19, no. 5, pp. 12–19, 2004. 2.5

[53] IEC, "Functional safety of electrical/electronic/programmable electronic safety related systems," International Electrotechnical Commission, IEC 61508, 2000. 2.5

[54] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, pp. 44–57. 2.6

[55] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, and J.-P. Tolvanen, "DSLs: the good, the bad, and the ugly," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2008, pp. 791–794. 2.6

[56] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 407–418. 2.6

[57] M. Zviedris and R. Liepins, "Readability of a diagrammatic query language," in *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2014, pp. 227–228. 2.6

[58] E. Yannakoudakis and C. Cheng, "A domain-oriented approach to improve the user-friendliness of SQL," *Computer Standards & Interfaces*, vol. 9, no. 2, pp. 127 – 141, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0920548989900056 2.6

[59] A. Bubeck, F. Weisshardt, and A. Verl, "BRIDE-a toolchain for framework-independent development of industrial service robot applications," in *Proceedings of the 41st International Symposium on Robotics ISR/Robotik 2014*. VDE, 2014, pp. 1–6. 2.6, 5.3

[60] C. Schlegel, A. Lotz, and A. Steck, *Robotic software systems: From code-driven to model-driven software development*. INTECH Open Access Publisher, 2012. 2.6

[61] A. Lotz, J. F. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, and C. Schlegel, "Towards a stepwise variability management process for complex systems: A robotics perspective," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 5, no. 3, pp. 55–74, 2014. 2.6

[62] A. Steck and C. Schlegel, "Managing execution variants in task coordination by exploiting design-time models at run-time," in *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2011, pp. 2064–2069. 2.6

[63] J. F. Inglés-Romero, A. Lotz, C. V. Chicote, and C. Schlegel, "Dealing with run-time variability in service robotics: towards a DSL for non-functional properties," *arXiv preprint arXiv:1303.4296*, 2013. 2.6

[64] OMG, "Systems Modeling Language (SysML), version 1.4," 2015, http://www.omg.org/spec/SysML/1.4/PDF/. 2.6

[65] SAE, "Architecture Analysis & Design Language (AADL)," Internet Requests for Comments, SAE, SAE AS5506B, September 2012. [Online]. Available: http://standards.sae.org/as5506b/ 2.6, 8

[66] T. Noll, "Safety, dependability and performance analysis of aerospace systems," in *Formal Techniques for Safety-Critical Systems*. Springer, 2014, pp. 17–31. 2.6

[67] G. Biggs, K. Fujiwara, and K. Anada, "Modelling and analysis of a redundant mobile robot architecture using AADL," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 146–157. 2.6

[68] H. Yu and Y. Yang, "Latency analysis of automobile ABS based on AADL," in *Proc. of the 2012 Int. Conf. on Industrial Control and Electronics Engineering (ICICEE)*. IEEE, 2012, pp. 1835–1838. 2.6

[69] S. Cousins, "Exponential growth of ROS," *IEEE Rob. Aut.*, vol. 18, no. 1, pp. 19–20, 2011. 3.1

[70] P. Mantegazza, "DIAPM RTAI for Linux: Whys, whats and hows," in *Real Time Linux Workshop, Vienna University of Technology*, 1999. 3.1

[71] Xenomai, "Xenomai home page," 2015, http://www.Xenomai.org. 3.1

[72] W. Fetter Lages, D. Ioris, and D. C. Santini, "An architecture for controlling the Barrett WAM robot using ROS and OROCOS," in *Proceedings of the 41st International Symposium on Robotics. ISR/Robotik 2014*. VDE, 2014, pp. 1–8. 3.1

[73] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao, "RT-ROS: A real-time ROS architecture on multi-core processors," *Future Generation Computer Systems*, vol. 56, pp. 171–178, 2016. 3.1

[74] G. Nutt, "NuttX home page," 2015, http://www.nuttx.org/. 3.1

[75] MISRA, "MISRA C:2012: Guidelines for the use of the C Language in Critical Systems," Motor Industry Software Reliability Association, MISRA-C, 2012. 3.2

[76] D. Pilaud, "Efficient automatic code generation for embedded systems," *Microproc. and Microsystems*, vol. 20, no. 8, pp. 501–504, 1997. 3.2

[77] J. T. M. Ingibergsson, U. P. Schultz, and M. Kuhrmann, "On the use of safety certification practices in autonomous field robot software development: A systematic mapping study," in *Product-Focused Software Process Improvement*. Springer, 2015, pp. 335–352. 3.3

[78] J. L. de la Vara and R. K. Panesar-Walawege, "SafetyMet: A metamodel for safety standards," in *Model-Driven Engineering Languages and Systems*. Springer, 2013, pp. 69–86. 3.3

[79] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2010, pp. 307–309. 5.4

[80] Conpleks ApS, Minkpapir AS, "FIX-Robo," https://www.youtube.com/watch?v=q8h63rYoNQ0. 6.1

[81] K. Jensen, M. Larsen, S. H. Nielsen, L. B. Larsen, K. S. Olsen, and R. N. Jørgensen, "Towards an open software platform for field robots in precision agriculture," *Robotics*, vol. 3, no. 2, pp. 207–234, 2014. 6.1

[82] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.(IROS 2004).*, vol. 3. IEEE, 2004, pp. 2149–2154. 7.1

[83] Conpleks ApS, Intelligent Marking ApS, "Intelligent One," https://www.youtube.com/watch?v=47loZddm0GY. 7.1

[84] L. B. Larsen, K. S. Olsen, L. Ahrenkiel, and K. Jensen, "Extracurricular activities targeted towards increasing the number of engineers working in the field of precision agriculture." XXXV CIOSTA & CIGR V Conference, Billund, Denmark, July 2013. 7.2.3

Sorin Adam received his B. Sc. and M. Sc. degrees in electronics and computer science from the Polytechnic University of Bucharest, in 1995 and 1998, respectively. Currently, he is a Ph. D. student at the University of Southern Denmark. His research interest covers robotics safety, software engineering, domain specific languages, and embedded systems.

Morten Larsen received his B. Eng. in Electrotechnics and M. Sc. degrees in Robot systems engineering from the University of Southern Denmark, in 2010 and 2012, respectively. Currently he is a Ph. D. student at Aarhus University, Department of Engineering, which he started in 2013. His Research interest covers software engineering and model driven engineering applied to agricultural robotics domain.

Kjeld Jensen received his B. Sc. in Physics and M. Sc. degrees in Applied Mathematics in 2001 and 2005, respectively, and the Ph. D. degree in robotics from the University of Southern Denmark, in 2014. Currently, he is an Assistant Professor at the University of Southern Denmark. His research interest covers mobile robotics, unmanned aerial systems and cyber-physical systems.

Ulrik Pagh Schultz received his B.Sc. and M. Sc. degrees in computer science from the University of Aarhus, in 1995 and 1997, respectively, and the Ph.D. degree in computing from University of Rennes, in 2000. From 2000 until 2005, he was a faculty member at University of Aarhus. Currently, he is an Associate Professor at the University of Southern Denmark. His research interest covers robotics, domain-specific languages, and software engineering. Currently he is Chair of the IFIP Working Group 2.11 on Program Generation (since 2013). He is a member of ACM and IEEE.