Programming for Reliability and Safety in Robotics: The Role of Domain-Specific Languages

Domain Specific Programming for Safe and Reliable Robots

Rizwan, Momina

2024

Document Version:
Publisher's PDF, also known as Version of record

Programming for Reliability and Safety in Robotics: The Role of Domain-Specific Languages

Momina Rizwan

Licentiate Thesis, 2024

Department of Computer Science
Lund University

ISBN 978-91-8039-935-7 (electronic version)
ISBN 978-91-8039-934-0 (print version)
ISSN: 1652-4691
Licentiate Thesis 1, 2024

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: momina.rizwan@cs.lth.se

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2023

ABSTRACT

Autonomous robots must operate reliably and predictably in uncertain environments. Throughout the robot software development cycle, robot operators and developers must be able to specify their safety and functional requirements reliably and explicitly. To this end, we propose to use Domain-Specific Languages (DSLs) to address their needs. To show the applicability and effectiveness of this approach, we demonstrate two DSLs that are designed to enhance both the safety and reliability of how robot programmers write their code. Firstly, we extend the Declarative Robot Safety (DeROS) language to create ROSSMARie, which not only halts robot operations upon safety rule violations (such as proximity to humans) but also monitors and tries to recover from these violations autonomously. This extension allows robots to adapt to system failures and resume operations without human intervention, striking a balance between safety and task performance. We validate ROSSMARie on the ROS-based industrial platform SkiROS2, demonstrating its effectiveness in maintaining safety for two robot experiments: manipulation and navigation. Secondly, we explore the use of embedded DSLs for early bug detection in robot software development. Recognizing the challenges in predicting the full context of general-purpose robot components, our approach focuses on early error identification to avoid costly runtime failures and safety hazards. We introduce DSL design patterns tailored for robotics, implemented in Python, and apply these to SkiROS2. These patterns enable programmers to detect bugs early in the high-level contracts between robot capabilities and their world model and lower-level implementation code, such as behavior trees, performing consistency checks during the deployment phase rather than at runtime. This proactive approach significantly enhances safety by identifying potential skill execution issues before they affect robot behavior. An initial study with SkiROS2 developers confirms the utility of our DSL-based method in early bug detection and improving the maintainability of robot code. We provide a comprehensive approach to domain-specific robot programming, ensuring both the functional safety and operational efficiency of autonomous robots. By integrating DSL strategies, we provide a robust framework for developing reliable and safe robots capable of adapting to dynamic environments and complex tasks.

CONTRIBUTION STATEMENT

The following papers are included in this dissertation:

Paper I Momina Rizwan, Christoph Reichenbach, Volker Krueger. “ROSSMARie: A Domain-Specific Language To Express Dynamic Safety Rules and Recovery Strategies for Autonomous Robots”. In *Second Workshop on Quality and Reliability Assessment of Robotic Software Architectures and Components, June, 2nd, 2023, ICRA 2023, London, UK*. Conference website with selected contributions: qrarsac2023.

Paper II Momina Rizwan, Christoph Reichenbach, Volker Krueger. “Enhancing Robotic Autonomy: Strategies for Dynamic Safety and Immediate Recovery”. Technical Report 114, Lund Tekniska Högskola, 2024. **To be submitted.**

Paper III Momina Rizwan, Ricardo Caldas, Christoph Reichenbach, and Matthias Mayr. "EzSkiROS: A Case Study on Embedded Robotics DSLs to Catch Bugs Early." In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE), pp. 61-68. IEEE, 2023*.
DOI: 10.1109/RoSE59155.2023.00014.

Paper IV Momina Rizwan, Christoph Reichenbach, Ricardo Caldas, Matthias Mayr, and Volker Krueger. “EzSkiROS: Enhancing Robot Skill Composition with Embedded DSL for Early Error Detection”. **"Submitted for publication"** to the Special Issue “Robotics Software Engineering" as a Journal in Frontiers in Robotics and AI, section Computational Intelligence in Robotics.

The table below indicates the responsibilities Momina Rizwan had in writing each paper:

Paper	*Writing*	*Concepts*	*Implementation*	*Evaluation*	*Artifact*
I	●	◐	●	●	– –
II	●	◐	●	●	– –
III	◐	◐	◕	●	●
IV	◕	◐	◕	●	– –

The dark portion of the circle represents the amount of work and responsibilities assigned to Momina Rizwan for each individual step:

◔ Momina Rizwan was a minor contributor to the work

◐ Momina Rizwan was a contributor to the work

◕ Momina Rizwan led and did a majority of the work

● Momina Rizwan led and did almost all of the work

Sources: Artifact: `https://github.com/lu-cs-sde/EzSkiROS`, and A replication of the survey: `https://github.com/lu-cs-sde/EzSkiROS`.

ACKNOWLEDGEMENTS

I want to thank my supervisor, Christoph Reichenbach, whose expertise, understanding, and patience, added considerably to my experience. Coming from robotics, the software technology concepts were new to me. Your hard work and passion for teaching, helped me fit into this interdisciplinary area. I couldn't have asked for a better mentor.

I thank my co-supervisor, Volker Krueger for his support and advice throughout the process. His insights, feedback and, guidance played an important role in shaping my research.

I want to thank everyone in the SDE group and RSS group at Lund University for all the interesting talks we have had. I appreciate your willingness to share and explore research in software development during our reading group sessions. A special thanks to Görel Hedin, Elin Anna Topp, and Jacek Malec for all the support and advice.

I am immensely thankful to my friends, who have been more supportive than I could have ever hoped. I thank Idriss for all those fika breaks that helped me relax and stay entertained. I thank Noric for being an amazing listener during my rants and for those super helpful discussions on tackling tough situations. I thank Anton Risberg Alaküla for throwing those wild questions my way that really made me think hard about everything. And finally, huge thanks to Alexandru Dura, Matthias, and Alexander Dürr for your kindness and support all the way through.

Lastly, I want to express my deepest gratitude to my parents, my husband Faseeh and my daughter Rehana for their endless love and support throughout this journey.

I would also like to thank the Alice and Knut Wallenberg Foundation for funding my research through the Wallenberg AI, Autonomous Systems and Software Program (WASP).

To all of you, I am eternally grateful.

CONTENTS

INTRODUCTION

1 Motivation

Robots are becoming a big part of our lives, from building things in factories to helping out in our homes and hospitals. They are not just doing simple, repetitive tasks anymore. They are becoming more like smart assistants that can make decisions, understand their surroundings, and even learn from their experiences. This is called autonomy, and it is making robots more helpful but also more complex.

As robots become more autonomous, they need to handle more complicated tasks on their own, like navigating through a crowded room, or figuring out what to do if something unexpected happens. Even in factories, where humans and robots are usually kept separate, safety is still critically important. These industrial robots, some capable of handling a load up to 500 kg, operate in environments where even the slightest error can lead to significant damages to both the equipment and the surroundings.

Despite the physical barriers separating robots from humans, the need for high availability, reliability, and safety is paramount [Rob03]. In an industrial context, like a car production line, the malfunction or unreliability of one robot can halt the entire production process, leading to immense costs and productivity losses. A single robot stopping could mean a whole day's production is lost.

Moreover, as we push the boundaries of what robots can do, making them more autonomous and capable, the complexity of testing and ensuring their safety increases dramatically. When we extend these systems with new components or functionalities, the challenge is not just ensuring each part works; it is ensuring they all work together without causing unforeseen errors. What might seem like a small issue in one component can lead to a domino effect, causing delays, missed deadlines, and ultimately, failures in tasks that the robot was otherwise well-equipped to handle.

I aim to address these challenges from a software perspective using practices from software technology, in our case Domain-Specific Languages (DSLs). To

solve these problems, first we need to express these problems explicitly. Languages, whether natural or programming, let us express ideas, share knowledge, and solve problems. A DSL is a special kind of computer language made just for a particular type of task or field, like making websites or controlling robots. It uses terms and rules that are all about that specific area, making it easier for people working in that field to use. For example, the tool I am using to write this thesis LaTeX is a DSL used for document preparation and typesetting, especially for scientific documents. It uses specific commands and syntax to format text, create complex mathematical formulas, manage citations, and much more, making it particularly useful for people who need to create detailed and well-structured documents like research papers or theses.

Why do we use a Domain-Specific Languages (DSLs)?

Domain-Specific Languages (DSLs) allow for the explicit expression of domain concepts, rules, and constraints in a way that general-purpose programming languages often do not. According to the studies [Voe+19; Gra+08], DSLs can help developers to create more readable, maintainable, and safer code [Rév+00], especially in critical systems where safety is paramount.

In general-purpose languages like C++ or MATLAB, safety rules and constraints are often implemented implicitly through the code logic, as shown in the code example below. This can lead to misunderstandings or errors because the intent of the code or the constraints might not be clear, especially to someone who wasn't involved in writing the original code. Moreover, general-purpose languages are designed to be flexible and powerful, which means they don't inherently prevent you from doing unsafe operations.

Consider a scenario in a robotic control system where the controller calculates the speed of a motor. There's a safety requirement that the motor's speed should not exceed 100 radians per second to prevent damage to the mechanical system.

```
double calculateMotorSpeed(double input) {
    double speed = complex_computation(input);
    // Safety check
    if (speed > 100) {
        speed = 100; // capping the speed
    }
    return speed;
}
```

In this example, the safety rule (speed should not exceed $100 rad/sec$) is embedded in the code as a conditional statement. The first problem is that the safety rule is not explicitly defined as a domain rule (in a separate file) but is instead a part of the logic. A developer who wants to look at all the safety requirements cannot extract that information because it is hidden in a huge mess of operational code. The second drawback is that, if the safety rule changes, it needs to be updated everywhere it is used, which is error-prone.

Now, let's consider how a DSL might express the same safety rule. A DSL for motor control might allow you to define constraints directly as part of the language.

```
controller MotorController {
max_speed = 100 rad/s; // Explicitly defining the safety constraint
control_speed(input) {
    speed = complex_computation(input);
    description: "Ensure the maximum speed does not exceed safe
    operational limits."
    enforce max_speed; // Applying the safety constraint
}
}
```

This small DSL gives us the following benefits:

- the safety rule is an explicit part of the language syntax ($max\_speed = 100rad/s$) including the measurement unit. One could explicitly see/automatically detect errors like if we are adding rad/s to rad/min.
- the safety rule is defined once for a controller and enforced everywhere it's needed, making the system easier to maintain and less error-prone.
- the intent of the code is clear and any non-programmer safety expert reading the DSL script can understand that there's a maximum speed constraint on the motor.

Using a domain-specific language, one can express their specific concerns with precision and clarity. By focusing on creating more explicit, robust, and reliable ways of expressing safety concerns, we can improve how these robots operate, ensuring they do their jobs effectively without compromising safety. It is about making smarter, safer robots that can be trusted to work alongside us, enhancing productivity and safety in our industries.

To understand and explore the safety considerations in autonomous robots, in my thesis, I plan to tackle two approaches to enhance safety through Domain Specific Robot Programming. First, I will look at how we can make robots safer during runtime. To do this, I will use a DSL which allows us to express and enforce safety rules dynamically. We will also generate a 'runtime monitor' – a sort of a safety watchdog based on these rules. To save the world from yet another DSL, we take inspiration from an already existing DSL by Adam et al. [Ada+16]. Second, I aim to catch and fix some programming errors that could lead to unsafe robot behavior before the robots even start working. By spotting these errors early, we can prevent safety issues from happening, rather than dealing with them when they occur. This proactive approach complements the runtime safety measures, offering a more comprehensive safety strategy. As future work, I will discuss how these two DSLs could benefit from each other to increase their effectiveness.

1.1 Research Questions

In the rest of the thesis we will focus on the following research questions and their sub-research questions:

(I) What usually goes wrong with autonomous robots, and which of these issues can we address using Domain-Specific Languages (DSLs)?

(II) What stages in the robot's work cycle can DSLs be effectively utilized to enhance safety and reliability?

  (a) For each identified intervention point, what are the available data and possible actions that a DSL can take to make robots safe and more reliable?

(III) How effective are the DSLs in improving safety and reliability of autonomous robots?

  (a) How good are the DSLs at spotting these issues compared to regular programming languages when it comes to robots?

  (b) How DSLs improve the process and experience of creating and updating robot software, including how easy they are to learn, how they affect building time, mistake frequency, and keeping the software up-to-date?

1.2 Thesis Contribution

In order to answer the proposed research questions, in this thesis we will present the progress we made in making robots safer and more reliable through our design and implementation of Domain-Specific Languages (DSLs). Our contributions address the enumerated research questions as follows:

I In Papers I and II, we address environmental factors, including dynamic obstacles, uneven terrains, and interactions of humans and other robots, that could lead to unexpected safety hazards for autonomous robots. In Papers III and IV, we address the issue of incorrect representations within software components that do not match other components, leading to programming bugs or integration errors.

II To handle safety issues caused by environmental factors, Papers I and II describe an external DSL that generates a runtime monitor to intervene at runtime. Conversely, to deal with programming bugs, Papers III and IV present general DSL design patterns and two case studies using an internal DSL to detect these bugs at pre-launch time.

II (a) In Paper I and II, we use available sensor data to detect any unexpected environment changes and react to them by invoking alternative strategies or initiating fail-safe procedures. In Paper III and IV, we employ DSL design patterns to help developers catch these errors as soon as we have some context of other components (at pre-launch time) and reporting those errors to the developer before starting the robot.

III (a) In Papers I and II, the DSL can detect environmental hazards that can be sensed through sensor information provided by the software (ROS topics in our case). While in Papers III and IV, we can detect software bugs in the high-level contracts between the robot's capabilities and the robot's understanding of the world. We can also detect bugs in the lower-level implementation code, such as constructing behavior trees to control the robot's behavior based on its capabilities. We not only outperform general-purpose programming languages in identifying errors early, but also provide more explicit error messages.

III (b) In Papers III and IV, we perform initial user studies to gather feedback. The developer review of the DSL indicates a positive impact on code quality, notably enhancing correctness, readability, and clarity of skill dependencies in robot programming. The DSL streamlines the creation and debugging of behavior trees, leading to more concise and less error-prone code. Additionally, it improves error reporting, contributing to a better understanding of issues and overall code quality.

1.3 Thesis Outline

This thesis consists of six chapters, including this one. In Chapter II, we will introduce the essential tools and scientific concepts that underpin our research, providing a detailed background on Domain Specific Languages, Robot Operating System, and other related technologies. Chapter III maps the existing landscape of robotic programming and the role of DSLs, identifying the current advancements and the gaps our research addresses. Chapter IV synthesizes insights from the included papers, offering an integrated narrative of our findings and contributions. Chapter V details the contributions of each paper, emphasizing the advancements and innovations made in robot safety and reliability through DSLs. Finally, Chapter VI concludes the thesis, summarizing our work and discussing future directions for DSLs in robotics, highlighting potential interactions and benefits for safety and reliability in the field. Each chapter builds upon the last, guiding the reader through a comprehensive journey from foundational concepts to the implications and future of DSLs in robotics.

2 Background

To understand the contributions of this thesis, it is necessary to introduce tool and application domain that we used in our research. In this chapter, we delve into the foundational concepts and tools that form the base of our thesis, where we develop and explore two Domain-Specific Languages (DSLs) and their interactions. These DSLs are designed to enhance the safety and reliability of robots at crucial stages – runtime and ROS **(Robot Operating System) pre-launch time**. Our exploration is not just theoretical; we demonstrate the practical application of these DSLs by integrating them with **a skill-based control platform, SkiROS2**, showcasing their functionality in real-world scenarios.

The first of these DSLs, ROSSMARie, is designed to tackle unexpected safety hazards that might arise during a robot's operation. This DSL draws inspiration from an existing DSL, **DeROS** [Ada+16], and similarly generates a ROS node that functions as a Runtime Safety Monitor. **Runtime Monitoring** plays a critical role in actively managing potential risks during the robot's operation, ensuring immediate response to any emerging safety concerns.

In contrast, our second DSL, EzSkiROS, is oriented towards the early detection of programming errors. Operating at the skill composition level and within **Behavior Trees**, this DSL is instrumental in identifying and rectifying programming errors before the robot is even launched. This proactive approach aims to streamline operations, enhance safety, and reduce the debugging loop for developers.

2.1 ROS

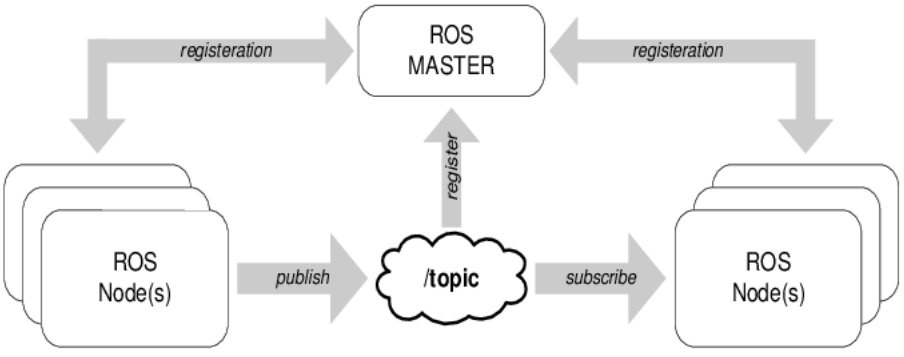


Figure 1: An image showing ROS communication layer. (source: [Ach+17])

The Robot Operating System (ROS) is an open-source framework for robotics software development. It provides a structured communications layer above the

host operating systems of a mixed compute cluster. ROS's architecture is modular, allowing for a distributed system of processes (nodes) that can communicate over a peer-to-peer network (Figure 1. It provides standard operating system services such as hardware abstraction, low-level device control, and commonly used functionality.

ROS has become a standard in robotics research and industry due to its flexibility, large user community, and extensive set of tools and libraries. It allows for rapid prototyping and testing, which is crucial in both academic research and industrial applications. Its modular architecture enables developers to build complex robotic systems by integrating various hardware and software components.

"pre-launch time" and "post-launch time" activities in ROS

In the context of robotics, particularly when discussing systems like the Robot Operating System (ROS), "pre-launch time" and "post-launch time" refer to the periods before and after the robot's software components or nodes are initiated.

1. **Pre-launch time** refers to the period before the robot's software or system has been initiated or activated. This is the phase where the system is not yet running, and no processes related to the robot's operation are active.

    - **Activities**
        - **Configuration:** Setting up parameters, configurations, and environment variables that will be used by the system.
        - **Setup:** Installing necessary software, loading drivers, and ensuring all hardware components are properly connected and functioning.
        - **Testing:** Running diagnostics or simulations to ensure that all components will work as expected once launched.
        - **Planning:** Defining goals, routes, or behaviors that the robot will execute once operational.

2. **Post-launch time** refers to the period after the robot's software or system has been initiated. This is when the robot is active, and its processes are running.

3. **Activities**

    - **Execution:** Running the actual tasks, processes, or behaviors that the robot is designed to perform. This includes movement, data processing, sensing the environment, and interacting with it.
    - **Monitoring:** Observing the robot's performance, checking for errors or unexpected behaviors, and ensuring it's operating within safe parameters.

- **Adaptation:** Adjusting the robot's behavior based on sensory input or changing conditions in the environment.
- **Communication:** Exchanging messages and data between different parts of the robot system or with external systems, often using ROS topics, services, and actions.

In ROS (Robot Operating System), these terms might not be formally defined, but they are conceptually used to describe the stages of a robot's operation cycle. ROS itself is a flexible framework for writing robot software and is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Understanding the distinction between pre-launch and post-launch is crucial for effectively developing, testing, and deploying robotic applications.

2.2 SkiROS2

SkiROS2 (Skill-based Robot Control Platform) [MRK23] is an advanced robot control system designed to enhance the development and execution of robot skills within the Robot Operating System (ROS) environment. Its primary purpose is to provide a structured and scalable way to define, manage, and execute complex robotic behaviors using a skill-based approach. Figure 2 shows the overall software architecture of SkiROS2. SkiROS2 focuses on modular skill definitions, allowing for the composition of high-level tasks from smaller, reusable skills. These skills are defined with preconditions, postconditions, and parameters, making them adaptable and capable of handling various scenarios.

In the context of ROS, SkiROS2 acts as a middleware layer that interfaces with different ROS components, such as sensors and actuators, to perform tasks. It allows developers to abstract away from low-level hardware interactions and focus on defining higher-level behaviors and skills. SkiROS2 is used in various applications, including industrial automation, service robotics, and research, where flexibility, reusability, and efficiency in robot programming are crucial [May+23; AMK23; May+22a].

SkiROS2 implements behavior trees as a core component of its skill execution framework. Behavior trees are a graphical modeling language used to control the flow of execution of tasks, typically in game development and increasingly in robotics. In SkiROS2, behavior trees organize the execution of skills into a tree structure, with each node representing a skill or a decision-making process.

The advantages of using behavior trees in SkiROS2 include:

- **Modularity:** Behavior trees allow for the creation of modular and reusable skills that can be easily composed and reconfigured for different tasks or robots.

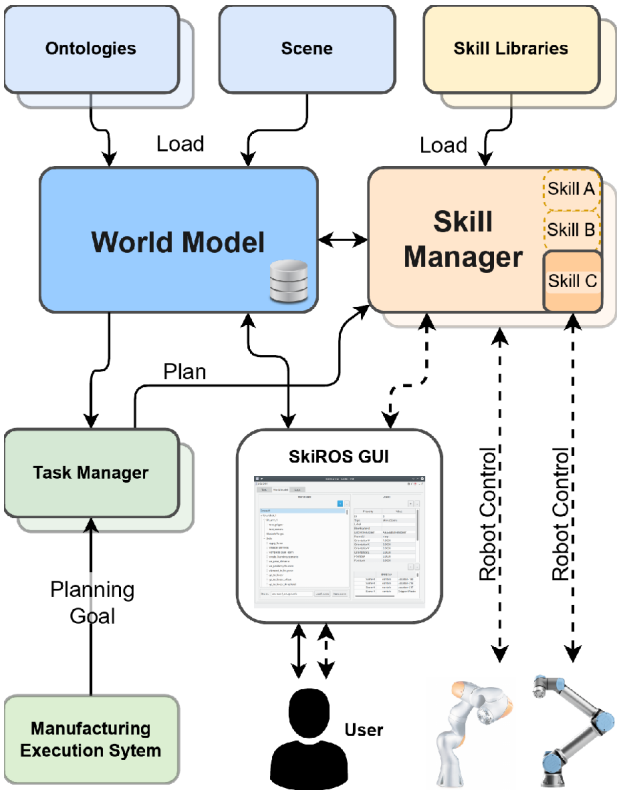


Figure 2: An outline of the SkiROS2 architecture. The world model stores the knowledge about the relations, environment, and the skills. The skill manager loads and executes the skills written as behavior trees. Dashed lines show control flows and solid lines information flows. Shaded blocks indicate possible multiple instances. (Source: [MRK23])

- **Flexibility:** They provide a flexible structure to handle dynamic changes and exceptions in the robot's environment, making it possible to adapt the execution flow on the fly.
- **Scalability:** As tasks become more complex, behavior trees in SkiROS2 can scale to manage these complexities without becoming unmanageable or overly complex themselves.
- **Intuitiveness:** The graphical nature and hierarchical structure of behavior trees make them relatively easy to understand and design, even for complex behaviors.
- **Debuggability:** The discrete and hierarchical nature of behavior trees allows for easier debugging and error handling compared to other control structures like finite state machines or sequential function charts.

In SkiROS2, behavior trees interact with the world model, skill descriptions, and the ROS ecosystem to perform tasks. Each node in the tree can represent a skill, condition, or control flow element (e.g., sequence, selector), allowing developers to define how and when each skill should be executed. This results in a powerful and intuitive way to manage robot behavior, ensuring that robots can perform a wide range of tasks reliably and efficiently.

2.3 Behavior Trees (BT)

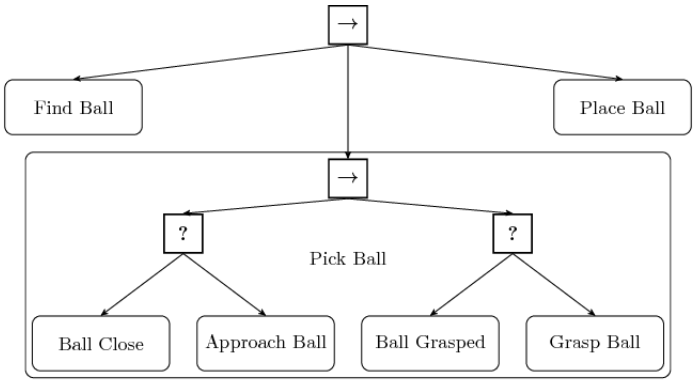


Figure 3: Behavior tree modelling a ball pick-place task. (source [CÖ18]

Behavior Trees can be seen as a graphical modeling language used for the specification of task execution in autonomous agents, particularly in robotics [CMÖ17] [CAÖ19]. A behavior tree is graphically represented as a directed tree (as shown

in Figure 3) in which the nodes are classified as root, control flow nodes, or execution nodes (tasks). For each pair of connected nodes the outgoing node is called parent and the incoming node is called child. The root has no parents and exactly one child, the control flow nodes have one parent and at least one child, and the execution nodes have one parent and no children. Graphically, the children of a control flow node are placed below it, ordered from left to right. They structure the control flow of a robotic application by organizing the tasks in a tree hierarchy, allowing for more modular, flexible, and reusable designs [CÖ18]. Behavior Trees offer several advantages in robotics, including clarity in representing complex behaviors, modularity for reusing and composing behaviors, and reactivity to changes in the environment. They facilitate the design and understanding of complex behaviors, making the development process more efficient and less error-prone.

2.4 Runtime Monitoring

Runtime monitoring is a system safety and reliability technique used to observe the behavior of a software system during its execution, ensuring that it operates correctly and safely. Here are key aspects of runtime monitoring:

- **Observation:** Runtime monitoring involves observing the states and outputs of a system as it operates. This is often done through sensors or probes that collect data about various aspects of the system's performance and behavior.
- **Specification of Correct Behavior:** It requires a predefined set of rules or specifications that describe the expected behavior of the system under various conditions. These specifications are usually derived from the system's requirements and can be expressed in various forms, including temporal logic, assertions, or contracts.
- **Detection of Deviations:** The core purpose of runtime monitoring is to detect deviations from the specified behavior. When the observed behavior differs from what is expected, the monitoring system can flag an error or anomaly. This detection is crucial for safety-critical systems where unexpected behavior could lead to severe consequences.
- **Response:** Upon detecting a deviation, the system can be designed to respond in various ways, depending on the severity and nature of the anomaly. Responses may include logging the error, alerting an operator, triggering a fail-safe mechanism, or attempting to correct the behavior automatically.
- **Dynamic and Continuous:** Unlike static analysis techniques that analyze the code without executing it, runtime monitoring is dynamic and occurs during the system's operation. It provides continuous oversight throughout the lifecycle of the system, offering protection against unexpected behaviors that might not have been anticipated during the design and testing phases.

- **Versatility:** Runtime monitoring can be used across various domains and types of systems, from simple software applications to complex, distributed, and embedded systems. It's particularly important in systems that operate in unpredictable environments or have high reliability and safety requirements.

In essence, runtime monitoring is an important technique to ensure that systems behave as expected during their operation, providing an additional layer of safety and reliability by detecting and responding to anomalies in real time.

2.5 DeROS

Declarative Robot Safety (DeRoS) is a domain-specific language (DSL) aimed at addressing safety concerns in robotics, particularly for mobile robots operating in unpredictable environments [Ada+16]. The language offers a simple and declarative syntax, making it easier for robotics experts with less software engineering knowledge to implement safety-related requirements. By separating the safety layer from the main functionality, DeRoS allows for the isolated specification of safety rules and facilitates the safety certification process.

One of the key advantages of DeRoS is its ability to drive the automatic generation of all safety-related code, reducing the risk of errors. This feature allows for implementation-independent reuse of the safety-related parts of a robot controller across different releases and products. As the DeRoS declaration does not need to change when the underlying software changes, it provides flexibility and scalability in developing safe robotic systems. The infrastructure can be reused across a range of products, while customization for safety is mainly achieved at a higher level using the DeRoS language.

The implementation of the low-level hardware interfacing and the code generation part of DeRoS is typically the responsibility of a skilled software development team. This division of roles results from the need for a more structured approach to robotics software development. To enhance the robustness of the safety layer, development of the code generator and execution supporting platform could be done by separate teams targeting different programming languages, depending on the specific robotics platform in use. The DeROS language supports C++ for ROS-based robots, considering it appropriate for safety-related functionality.

2.6 Later in the thesis

The next chapters will unpack the literature review of robotics software development and the crucial role played by DSLs and their interactions with other software components. We will illustrate how tools like SkiROS2can be significantly enhanced through these integrations, leading to safer robotic systems. Additionally, we will present case studies that not only demonstrate the practical applications of these DSLs to Reinforcement Learning module but also highlight our commitment to advancing the safety, reliability, and availability of robotic systems. These

discussions will set the stage for addressing the research questions raised in our study, underscoring the practical benefits and real-world implications of our work in the realm of robotics.

3 Related Work

In this Chapter, we set a map of the territory looking at the literature and positioning our research in that map.

3.1 Gaps in robotics software development

Robotics software is complex because it combines many parts like sensors, actuators, and decision-making programs. It is important that these robot components are safe, reliable, and always ready to work, especially because they are used in places where they need to be very precise and robust, like factories or faraway exploration areas. Unlike typical software, robotic software lacks standardized methods and processes that would make its development more systematic and easier [MNK22].

According to the study [Boz+19], the current state of robotics software development lacks a systematic development process. It relies on engineering craftsmanship instead of established engineering practices [Boz+19]. This situation poses challenges for reusability and reproducibility of available solutions. The study refers to a need for systematic approaches, methods, and tools to configure robots easily, specify robotic tasks in a user-friendly way, and enable robots to autonomously manage unpredictable situations. The study also identifies several challenges for the future of safety in mobile robotic systems.

Autonomous robots are increasingly being used in daily life tasks, often in unknown or partially unknown environments that might be shared with humans or other robots. This demands higher context awareness and adaptiveness capabilities. A significant number of current approaches lack support for adaptiveness and dealing with open systems where new actors can join dynamically [Boz+19; Boz+16]. It suggests that future research should focus more on adaptiveness and handling open systems. The study by [Boz+19] confirms that existing solutions aiming at managing safety for mobile robotic systems are not yet ready for everyday use. It highlights the need for turn-key solutions ready to address all challenges, indicating the complexity and criticality of safety and reliability in robotics software development.

In conclusion, developing reliable and safe robotics software presents numerous challenges. The **need for adaptability** is paramount, as robots must perform in dynamic, sometimes unpredictable environments [Par94] [Amb+16]. Robust error handling is crucial to ensure that robots can recover gracefully from unexpected situations [Ban+19].

3.2 Traditional ways to program robots

In this section, we're going to explore the usual methods and languages used to program robots, and why they sometimes fall short in making sure robots work

safely and correctly. These traditional methods have been the foundation for programming robots for a long time, but they often struggle with the advanced needs of today's robots, especially autonomous ones.

Software faults in ROS are common and varied, as noted by Dittrich, Yvonne, et al. in [Dit+17]. They include general programming errors like code exceptions and type errors, issues in building and deployment infrastructure, and run-time errors involving incorrect parameters or arguments. Other challenges include dealing with changes over time, clock mistimings, and discrepancies between the robot's model and the real world. [FN+20] also talks about dependency bugs in ROS and how they affect the software development process. Our research specifically focuses on these run-time errors and discrepancies between the robot's world model and the robot's capabilities written by developers.

Andezej Wasowski (at QRARSAC 2021) also pointed out why these faults happen. A major reason is the lack of a proper type system to connect different parts of the robot's software, leading to many runtime errors. Robots often misunderstand or incorrectly execute tasks because their programmed model of the world doesn't match reality. Additionally, there are weak systems in place for detecting and correcting errors automatically.

Moreover, studies like those by Gotlieb [GMS21] and Afzal [Afz+20] highlight the key challenges in automating robot software testing. These include unpredictable situations that the robot might not be programmed to handle, the sheer complexity of engineering such systems, the prevailing culture around testing these systems, coordination and documentation issues, the high costs involved, the complex environments the robots operate in, the lack of reliable check systems (oracle), challenges in integrating software and hardware, and a general distrust in simulations.

All these issues point to the need for improved programming methods that can handle the intricate nature of modern robots, ensuring they operate safely and reliably in all kinds of situations.

3.3 Other DSL-based approaches that address safety-critical systems

Domain Specific Languages (DSLs) have been increasingly recognized for their potential to improve the functional safety of robotic systems. Typically, DSLs execute models by generating code in a programming language, leading to increased productivity and higher quality. However, in safety-/mission-critical environments, the trustworthiness of generated code is often questioned due to uncertainties in the generation mechanisms. This skepticism makes it challenging to justify the use of language workbenches in such environments. Voelter et al. [Voe+19] argue that models created with domain-specific languages are easier to validate and that the additional risk resulting from the transformation to code can be mitigated by a suitably designed transformation and verification architecture. This

validation improves the trustworthiness of the generated code, making DSLs a viable option in safety-critical robot applications. The paper [Mar+21] presents a model-based approach to include safety considerations of the ArmAssist robotic system, a healthcare system designed for hand and arm rehabilitation. The integration of new functionalities in Papyrus for Robotics, including contracts, has been positively received by ArmAssist experts. This approach helps in making safety considerations explicit, which are often implicit or buried in low-level details like source code. Model-based approaches enhance design explicitness, facilitate communication among stakeholders, and support early-phase validation through formal assertions and contracts at component and system levels.

Miyazawa, Alvaro, et al. [Miy+19] introduce RoboChart, a domain-specific modeling language based on UML but with a restricted set of constructs. This restriction enables simplified semantics and automated reasoning, making it particularly suited for robotics. RoboChart emphasizes design patterns appropriate for robotics, modeling the physical robot explicitly in terms of its variables, events, and operations. This approach contributes to the expressiveness and reliability of safety-critical code in robotics.

Trezzy, Mickaël, et al. [Tre+21] discusses leveraging Model-Driven Engineering (MDE) to enhance the development process of Robot Operating System (ROS) applications. The authors advocate for the adoption of MDE methodologies to facilitate higher-level reasoning and improve accessibility for field engineers. This approach is particularly beneficial in enabling more sophisticated model analyses for validation and verification purposes. In this context, the paper [Tre+21] introduces MDE4ROS, a graphical Domain Specific Language (DSL) specifically designed to streamline the development of high-level robotics applications. MDE4ROS provides a comprehensive system view during the development phase through its graphical representation. It also promotes a higher level of abstraction, simplifying the development process by generating code for the ROS system. This graphical DSL is particularly tailored for robotics applications, incorporating specialized DSLs and code generation techniques that are well-suited for ROS environments. As such, MDE4ROS exemplifies a model-driven DSL approach, focusing on enhancing the efficiency and effectiveness of robotics application development within the ROS framework.

In conclusion, concepts from model-driven engineering can be applied at runtime for both functional and non-functional requirements in self-adaptive systems, potentially enhancing robot safety. While this is an interesting research direction, we are not looking at safety and reliability from a model-driven or formal verification perspective, but from the perspective of "what do today's robotics programmers actually do, and how can we help them".

3.4 DSLs for Static Error Detection of Behavior Tree Construction

The integration of behavior trees (BTs) with robotic systems often involves the use of DSLs and frameworks like the Robot Operating System (ROS). [Ghz+23] emphasize the growing use of BTs in open-source robotic applications supported by ROS, indicating their practicality in the real-world applications. However, verifying the safety and correctness of BTs remains a challenge.

[Hen+22] use SMTs to check safety properties specified in Linear Constraint Horn Clauses notation over Behavior Tree specifications. Moreover, [TT22] use Event-B for formal specification and verification of the BT instances, ensuring the maintenance of invariant properties.

From a static semantics perspective, BhTSL is an example where the compiler checks the source text for non-declared variables and variable redeclaration [Oli+20]. Despite the advancements in BT DSLs, there is a lack of DSLs performing static checks as rigorously as desired. According to the survey paper [Ghz+20], most used behavior tree DSLs, such as BehaviorTree.CPP [1], py_trees [2], and the Behavior Tree from UnrealEngine [3], primarily focus on runtime type safety and flexibility. For instance, MOOD2Be's[4] project from Horizon 2020, the BehaviorTree.CPP tool offers a C++ implementation of BTs with type safety [Fac19], but the type-checking capability is largely left to the developer and is subject to runtime checks. This indicates a gap in the domain of DSLs for BTs in ensuring consistency and preventing inconsistencies in implementation between skills or actions before runtime.

In general, behavior tree semantics have been studied thoroughly in the literature [CH10; CH11]. [ZFKA15] presents a simulator based validation of behavior trees by executing system requirements and captures system states as rules and facts in a database. It focuses on early validation of systems and identifies ways to optimize execution paths, thereby enhancing the efficiency and reliability of software development.

In conclusion, while there have been significant advancements in DSLs for robotics and BTs, there is a continuous need for the development of languages and tools that allow both static and early dynamic checking of behavior tree structure to ensure the safety, reliability, and efficiency of robotic systems.

3.5 Evaluation methods for DSLs

One of the main goals of DSLs is to ease the work of developers in different areas. However, to achieve this goal it is necessary to provide an evaluation of the

[1] https://github.com/BehaviorTree/BehaviorTree.CPP
[2] https://github.com/splintered-reality/py_trees
[3] https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees
[4] https://robmosys.eu/mood2be/

usability of such languages. Evaluating Domain Specific Languages (DSLs) is a subjective process, as evidenced by recent research in the field. Rodrigues et al. [PRBCZ17] conducted a systematic literature review to understand how usability has been assessed in DSLs, providing a comprehensive overview of the methods and criteria used in such evaluations. Barisic [Bar17] proposed a conceptual framework to support the iterative development process of DSLs, particularly focusing on usability evaluation, validated through multiple-case studies. Asici et al. [ATK21] introduced a comparative evaluation methodology based on the Analytical Hierarchy Process for multi-agent system DSLs, emphasizing criteria like appropriateness and development time reduction.

Furthermore, Bork et al. [BSK19] proposed an empirical evaluation technique for assessing the intuitiveness of DSL notations, applying it to a business continuity management modeling language. Rodrigues et al. [Pol+18] presented UsaDSL, a usability evaluation framework specifically for DSLs, utilizing a focus group method for validation. Lastly, Kahraman and Bilgen [KB15] developed a framework for the qualitative assessment of DSLs by integrating ISO/IEC 25010:2011 standard, CMMI maturity level evaluation approach, and the scaling approach used in DESMET into a perspective-based assessment.

Collectively, these studies underscore the importance of systematic, empirical, and integrated approaches in evaluating DSLs. They highlight the need for assessing various aspects such as usability, intuitiveness, and overall quality, ensuring that DSLs are effectively serving their intended purpose in various domains. These methodologies and frameworks provide valuable insights and tools for researchers and practitioners aiming to develop or assess DSLs in their work.

In our thesis, we employed various methods to assess our Domain Specific Languages (DSLs). Specifically, for ROSSMARie, we conducted a quantitative evaluation focusing on the transformation and generation of the safety monitor, although we have yet to perform a qualitative study for it. Conversely, for EzSkiROS, we undertook a comprehensive evaluation that includes both quantitative validation against errors and qualitative analysis through case studies.

4 Contribution Summary

In this Chapter, I will briefly review each paper included in the thesis and its contributions.

4.1 ROSSMARie: A Domain-Specific Language To Express Dynamic Safety Rules and Recovery Strategies for Autonomous Robots

This paper explores the idea of dynamic safety and recovery strategies in autonomous robotic systems. It addresses the issue of overly rigid safety strategies that prevent autonomous agents from completing their tasks. We propose "ROSSMARie", a DSL inspired by "DeROS" (cited as Adam et al., 2016 [Ada+16]), which is designed to specify safety constraints independent from the functionality of the robot. Unlike DeROS, which enforces immediate and aggressive stops for safety violations, ROSSMARie introduces capabilities for ongoing monitoring, recovery, and resumption of operations after hazardous conditions cease. This paper demonstrates ROSSMARie's potential to significantly enhance both the safety and functionality of robots, as evidenced by various simulated experiments.

Contributions of the paper

- Modify the semantics of re-implementation of DeROS (ROSSMARie) to keep monitoring and resume operation when the rule is no longer violated.
- Introduce strategies for autonomous robots to recover from safety hazards autonomously.
- Integration challenges of ROSSMARie with a skill-based robot platform SkiROS2.
- Verifies the effectiveness of ROSSMARie through its application in new scenarios, showcasing (in simulation) how it enhances both the safety and functionality of autonomous robots.

4.2 Enhancing Robotic Autonomy: Strategies for Dynamic Safety and Immediate Recovery

In this technical report, we expand upon the concept introduced in Paper I to assess how our proposed solution addresses the issue of aggressive safety stops that unduly limit robot autonomy. We argue that while safety is important for robots, it is often unnecessary to completely halt the robot in response to unexpected events. In this paper, we answer three research questions. Our initial solution seeks to answer the research question: How can we modify the DSL's semantics to continuously monitor and appropriately react to dynamic situations? This involves

adapting to changing conditions in real-time. Additionally, we propose alternative recovery strategies that are less extreme than complete shutdowns but still maintain the robot's safety. These strategies provide a more nuanced approach to maintaining safety while preserving operational functionality.

Contributions of the paper

- Refined semantics of DeROS to facilitate continuous monitoring of hazardous situations. By enabling continuous monitoring, robots are now equipped to promptly resume normal operations after hazardous conditions are mitigated, ensuring safety without undue interruptions.
- Expands the DeROS framework to allow more action templates for recovery strategies. This enhancement significantly broadens the framework's utility across different robotic platforms, particularly those equipped with robotic arms. The introduction of varied and adaptable recovery strategies allows for more tailored safety responses across diverse operational scenarios.
- Validates the effectiveness and applicability of ROSSMARie through a series of simulated and real robot experiments focusing on navigation and manipulation tasks. These experiments provide concrete evidence of the practical benefits and real-world impact of our enhancements, illustrating how they can significantly improve both safety and functionality in robotic systems.

4.3 EzSkiROS: A Case Study on Embedded Robotics DSLs to Catch Bugs Early

In this paper, we address the problem of developing general-purpose robot software, particularly the difficulty in detecting bugs early due to the varied execution contexts. In this work, we propose using embedded DSLs to enforce early checks, reducing debugging time and enhancing safety. The proposed solution uses an embedded DSL, EzSkiROS, which is integrated into SkiROS2, a skill-based robot control platform. Using DSL design patterns like Domain Language Mapping and Symbolic Tracing, EzSkiROS enables early dynamic checking of robot skill descriptions. The effectiveness of EzSkiROS is demonstrated through a user study, showing its utility in reporting errors at pre-launch and thus improving the development process. According to the user study, EzSkiROS improves code readability and maintainability, and allows early error detection for more effective and safer robot operations.

Contributions of the paper

- A survey of Python language features that enable DSL embedding;
- Two design patterns for *embedding DSLs in general-purpose programming languages* that address common challenges in robotics, with details on how to implement these patterns in Python;

- A case study of a robotics software SkiROS2 , in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs.

4.4 EzSkiROS: Enhancing Robot Skill Composition with Embedded DSL for Early Error Detection

In this paper, we further substantiate the use of DSL design patterns to enforce early checks at a lower-level skill implementation including the construction of consistent behavior trees. We present two analyses of different abstraction levels in SkiROS2 and show how we can use DSL design patterns to detect bugs at a pre-launch phase before runtime. Case Study I demonstrated the value of our design patterns by showing how they help detect bugs in the high-level contracts between a variety of robot capabilities and the robot's world model. Case Study II expands EzSkiROS by adapting the same techniques to detect bugs in lower-level implementation code, in our case that implementation uses a behavior tree to integrate different robot capabilities. Our work in this paper demonstrates how embedded DSLs can help robotics developers detect bugs early, even when the analysis depends on data that is not available until run-time. Our evaluation with EzSkiROS further suggests that embedded DSLs can achieve this goal while simultaneously increasing code maintainability.

Contributions of the paper

- Four design patterns for *embedding DSLs in general-purpose programming languages* that address common challenges in robotics, with details on how to implement these patterns in Python;
- A case study of a robotics software SkiROS2 , in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs, highlighting its effectiveness in identifying errors in both high-level skill descriptions and lower-level implementation details.
- A demonstration of how EzSkiROS detects various types of bugs in robot capabilities, world model contracts, and behavior trees, showcasing the DSL's comprehensive coverage and versatility in detecting bugs early.

5 Interaction between Domain Specific Languages for improved Reliability and Safety

In Section 1, I laid the foundation of the thesis with a research question "what are the factors affecting safety and reliability of autonomous robots and how we can deal with them using DSL?". Although this is a very broad question, in Section 4, we summarised two DSLs ROSSMARieand EzSkiROS. ROSSMARieis designed to handle runtime safety by actively monitoring the robot's environment and making real-time decisions to avoid hazards. EzSkiROS, on the other hand, is used to preemptively identify and correct programming errors.

In this Chapter, I will reflect on my work until now and how it all connects together to form deeper research questions. I will draw a picture emphasizing the use of the two DSLs ROSSMARie and EzSkiROS as a cohesive strategy to enhance the safety and reliability of autonomous robots. In the following sections, I will try to answer new research questions focusing on the integration between the two DSLs,

- How can we combine ROSSMARie and EzSkiROS to help make robots safer?
- Can EzSkiROS pass on skill level information to ROSSMARie to define more context-aware safety rules?
- How can ROSSMARie utilize static world model knowledge provided by EzSkiROS?
  - Can we predict the future using world model knowledge to have proactive measures?
  - Can we improve our recovery strategies based on the information about the world?
- What are the potential benefits and challenges of using EzSkiROS for behavior tree generation and modification in response to safety constraints defined by ROSSMARie?

5.1 Unified Vision of DSLs

In Section 4 I introduced two DSLs, each designed to enhance different aspects of robot safety and reliability. One focuses on dynamically adapting to environmental changes to prevent failures, while the other addresses an early detection of bugs that are caused by a lack of knowledge about other components. These DSLs might initially seem disjoint, but I believe their combined application could offer a more comprehensive solution to the challenging problems in robotics. This chapter explores how these DSLs can be viewed not as disjoint parts but as complementary strategies within a unified vision.

5.2 Opportunities:

Synergy of Both Languages can help make robots safer and more reliable

Both DSLs are designed with meaningful vocabulary to talk about and fix different parts of how robots work. Both DSLs have their own strengths. While ROSSMARie is good at monitoring and describing runtime safety constraints, EzSkiROS is adept at crafting meaningful low-level execution strategies using behavior trees (BTs). Uniquely, EzSkiROS can access the robot's static world knowledge and understand the context of high-level skills being executed, a feature that enriches its functionality beyond what is offered by ROSSMARie. By synergizing both DSLs, we can harness the advantages of each to create a more robust and efficient system. The following sections explore how these DSLs can complement and enhance each other:

Predicting safety hazards

One feedback that I got for ROSSMARie was "isn't it sometimes too late to detect unsafe scenarios based on sensor values if we are working with safety at runtime". Traditional runtime monitors resort to aggressive measures due to this latency in detection. I thought this was a valid point. To address this, I am considering to leverage abstract world model knowledge to get insight to "predict" the future. For instance, if the robot knows it is navigating through a school corridor, it could adopt proactive safety measures like reducing speed and being vigilant for unexpected obstacles, notably children. EzSkiROS could provide ROSSMARie with this anticipatory information, enhancing the system's predictive capabilities and allowing for more nuanced safety measures.

Skill-aware safety rules

Safety rules aren't one-size-fits-all; they vary depending on the task at hand. I discussed this problem in Paper I and II. For example, some tasks may necessitate close contact with objects, contradicting general safety rules that mandate keeping a distance. This discrepancy highlights the need for task-specific safety considerations, informed by high-level skill context. EzSkiROS, with its access to detailed skill information, can provide ROSSMARie with the necessary context to adapt safety rules accordingly, ensuring that safety measures are both effective and relevant to the task performed.

Generating recovery strategies as behavior trees

In practice, we've observed (Paper II) that defining recovery actions as singular behaviors in ROSSMARie might not account for task sequencing or the timely execution of subsequent recovery actions. Exploring behavior tree generation and

modification in response to safety constraints is a promising avenue. Behavior trees excel in managing task sequences and could effectively orchestrate a series of recovery actions, like rerouting upon detecting an obstacle. Integrating EzSkiROS's behavior management capabilities with ROSSMARie's safety monitoring could lead to more dynamic and responsive recovery strategies.

Combining these DSLs not only broadens the scope of problems we can address but also allows for handling more complex scenarios that might be challenging for a single DSL. This integration fosters a more comprehensive and flexible approach to robot programming.

5.3 Technical Challenges

One of the main challenges is how these two DSLs can exchange information effectively. As both DSLs interact with SkiROS2 in some way (EzSkiROS being embedded within the SkiROS2 API in Python and ROSSMARie connected through ROS), they have the potential to communicate through higher-level APIs or service layers provided by SkiROS2 and ROS respectively. This higher-level communication would enable the DSLs to share, generate, or consume information, creating a more integrated and coherent system.

Complex Integration Dynamics: Each DSL is designed with a specific purpose and operational context in mind. When two such languages are integrated, their interaction creates a complex dynamic. Understanding and evaluating how one DSL's output or behavior affects the other requires a deep understanding of both the individual DSLs and their combined operation.

Performance Metrics: Defining appropriate metrics to evaluate the interaction of DSLs is complex. These metrics must encompass not only individual DSL performance but also the effectiveness of their integration. For example, in the case of ROSSMARieand EzSkiROS, metrics might include the reduction in operational errors, improvement in safety protocol adherence, and efficiency in task execution.

I believe we need further research to explore these directions. My overall expectation is that by uniting ROSSMARie and EzSkiROS, we would be able to integrate the perspectives, capabilities, and insights of each DSL to enhance the safety and reliability of robotic systems.

6 Interaction between ROSSMARie and Reinforcement Learning

6.1 Constraining and Guiding Reinforcement Learning With Rule-Based Safety Monitoring

In Section 1, I outlined several research questions. One of the primary research questions is to identify the causes of unsafe robot behavior. I narrowed the focus of my research to use Domain-Specific Languages (DSLs) to address two key factors: the dynamic environmental factors causing unexpected failures and software faults due to incompatible representations within software components. To further answer RQ1 (mentioned in Chapter 1) in the future, I would like to address another source of unsafe behavior: autonomous robots using learning methods. While Chapter 5 focused more on how EzSkiROS and ROSSMARie can interact to unravel more opportunities to improve safety, in this chapter I will focus mainly on ROSSMARie and how can we use it to guide and constrain the exploration part of Reinforcement Learning (RL) using safety rules.

Note: This is a joint proposal created with Matthias Mayr.

6.2 Motivation

As robots get more advanced, making sure they are safe is more important than ever, especially now that they are starting to learn on their own and work closely with people. Reinforcement learning is a new way for robots to learn and get better by trying different things and seeing what works best. However, this introduces new complexities in ensuring safety.

Reinforcement learning algorithms, while powerful, start with an exploratory phase that can involve significant trial and error. This exploration is essential for learning but poses safety risks, especially when humans are in proximity. In industrial environments, understanding and predicting the robot's behavior is important for the safety of both workers and equipment.

Herein lies the motivation for integrating ROSSMARie with reinforcement learning. ROSSMARie, with its focus on expressing dynamic safety rules and recovery strategies, provides a structured approach to guide and restrict RL exploration. By embedding safety rules and acceptable operational parameters directly into the learning process, ROSSMARie can act as a safety monitor, steering the robot away from hazardous actions or states. Having a safety monitor is important to ensure the safety of equipment and workers, but also to allow humans working close to the assembly setup to predict the robot's behavior.

To better understand the problem, consider an assembly task where the robot needs to learn how to insert a piston into the engine. A mock-up of this task is shown in Fig. 1. The setting of this insertion requires to hold the object upright and to perform a spiral search motion. This task's objective is to increase the

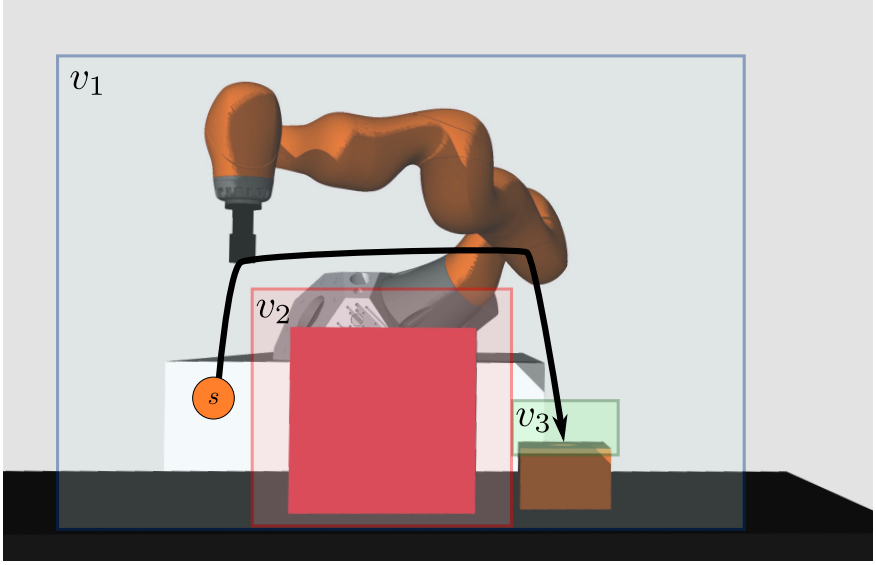


Figure 4: A robot in an expensive and fragile industrial environment must learn to avoid a fragile object and to perform a peg-in-hole insertion. The workspace is divided into different zones: allowed volume v_1, disallowed volume v_2 and allowed contact volume v_3. (source: [May+22c] [May+22b]

performance of the insertion. We consider here that there is a fragile assembly part (red box in Fig. 1). Even though our reward functions discourage closeness to the fragile object, there is the safety hazard of the robot hitting the fragile red object when learning a strategy to avoid it. Furthermore, a human worker is sharing the workspace and the insertion must not exceed certain forces. For the robot to be able to learn this piston insertion task safely, we propose an approach to incorporate a safety monitor to guide the exploration without needing any external intervention during episodes.

By using ROSSMARie as a safety overlay in this example, we can enable more robust learning processes that are sensitive to the dynamic safety requirements of shared human-robot workspaces. This approach allows robots to learn and adapt while ensuring that their exploratory behaviors are bounded within safe, predictable parameters. It represents a step towards more reliable and trustworthy robot operations in complex, unpredictable environments, enhancing both performance and safety in industrial robotics.

6.3 Formulating safety rules

ROSSMARie allows systems engineers to specify relevant risks in a formal language, integrating these directly into the robot's learning process. To show a possible application of ROSSMARie to define safety rules for the exploration part of Reinforcement Learning RL algorithms, an example is shown in Listing 1. Safety rules are categorized into robot operation, human-robot collaboration, and environmental conditions, covering a comprehensive range of safety considerations. A safety rule can consist of one or more safety conditions (entities shown in Listing 1) and one or more reactions that are executed when those conditions hold.

Each condition can either be used independently to obtain a safety rule (Line 34 in Listing 1) or combined with other conditions. This allows to implement detailed safety configurations such as on [Wan+21] with different reactions depending on the state of the robot system and the human. Safety rules can be divided into task-independent and task-dependent rules. The former set of rules follows from legal obligations, norms as well as robot operation limits and exist independent of the task at hand. Additionally, there are task-dependent rules that depend on the current setup of the workstation and the type of task and the expected robot behavior and interaction.

For each detected violation, one can define corrective actions in ROSSMARie to bring the robot to a safe state and provide feedback for the learning algorithm that can be utilized in reward functions. This feedback can help in refining the learning process and ensuring continuous, safe operation.

```
1  action decelerate;
2  action retraceToTheLastEndEffectorPose;
3  action turnOffActiveForce;
4  action logViolatedRule;
5
6  const maxInertia = 0.17 kgm.m
7  const volume_2 = [-0.8, 0.1, 0.65, -0.2, 0.5, 1.3] m
8  const maxDistanceAllowed = 0.08 m
9
10 ''' Task specific information coming from the skill
       description.'''
11 input contactRichTask = topic contact_allowed
12
13 ''' Inputs are mapped to the relevant ROS topic. '''
14 input distToOject = topic detectobject.dist
15 input eePose = topic endEffectorPose
16 input eeInertia = topic endEffectorInertia
17
18 ''' Entities that define bounded safety conditions '''
19 entity operation {
20     exceedsInertia :
21         eeInertia > maxInertia for 1.0 sec;
```

```
}

entity env {
    eeExceedsSpace:
        eePose not in volume_1;
    closeToObject:
        distToObject > maxDistanceAllowed;
    nonContact :
        not contactRichTask;
}

''' Task dependent safety rule '''
if env.eeExceedsSpace and env.nonContact
    then { decelerate;
            retraceToTheLastEndEffectorPose;
                logViolatedRule;};

''' Task independent safety rule '''
if env.closeToObject and
            operation.exceedsInertia
    then { turnOffActiveForce;
            decelerate;
                logViolatedRule;};
```

Listing 1: An example code excerpt to define safety rules for the scenario shown in Fig. 1. Actions definition and entities are defined to specify conditions and safety rules.

6.4 Conclusion

The idea of integrating ROSSMARie with RL is to ensure the safety of autonomous robots, particularly in dynamic and unpredictable environments. By guiding and constraining the exploration phase of RL with rule-based safety monitoring, robots can learn and adapt while adhering to strict safety protocols. This approach not only enhances the safety and reliability of robot operations but also contributes to the efficiency and robustness of the learning process itself.

In Figure 5, we conceptually show how we can integrate reinforcement learning with an isolated safety monitoring technique that detects violations of expert safety rules and augments or replaces actions with safe reactions. This design allows us to re-use safety rules intended for robot operations to also guide RL. Combining RL and rule-based safety monitoring allows expert safety knowledge to constrain the optimization strategy independent of the concrete RL algorithms being used and in arbitrary (industrial) environments. It also enables safer learning of black-box policies such as neural networks.

Although this is an initial sketch of the solution, we plan to focus on further developing and refining these ideas, exploring their potential and applications in ensuring the safe and effective operation of autonomous robots. The insights and methodologies presented here will guide my future work, aiming to contribute to the broader field of robotics and autonomous systems.

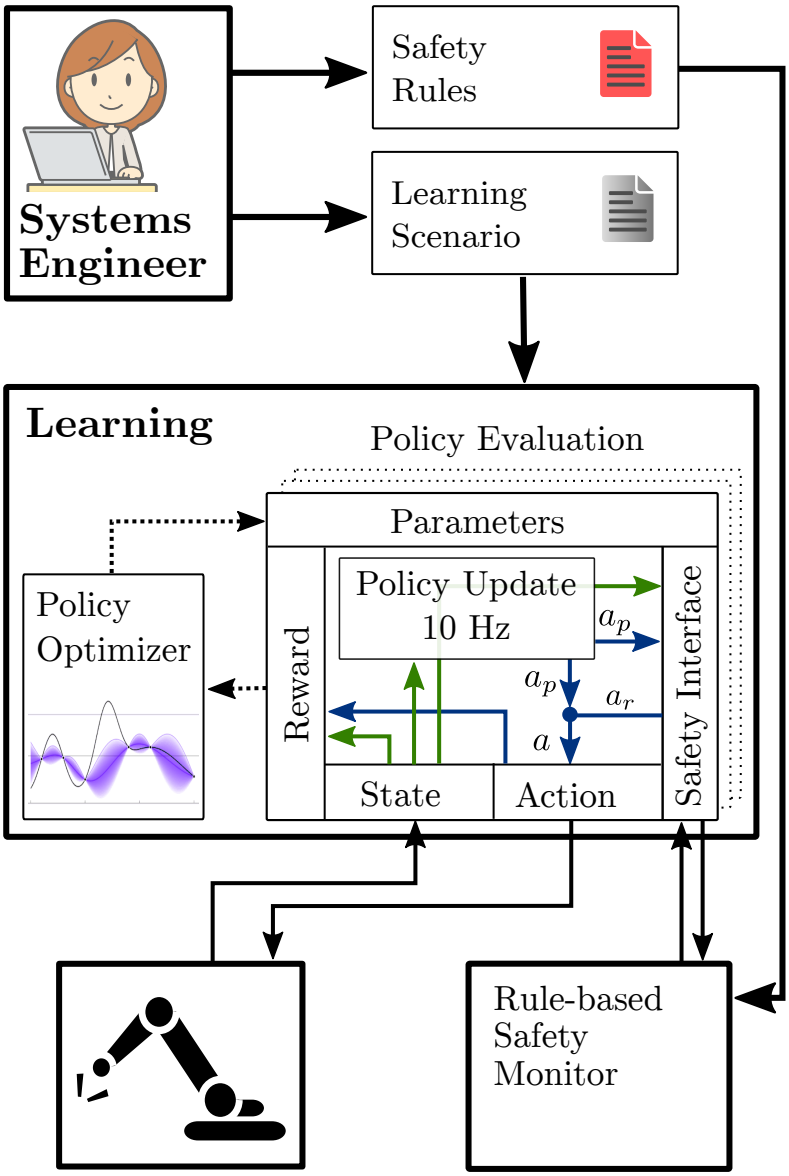


Figure 5: The proposed architecture of the safety implementation where the systems engineer defines both the safety rules and the learning scenario. When executing a policy, the state and action is communicated to the runtime monitoring system through the safety interface. The final action follows the reaction of the runtime monitor.

7 Conclusions and Future Work

This thesis primarily focuses on enhancing the safety and reliability of robotic systems through innovative software technology tools, with a special emphasis on Domain-Specific Languages (DSLs). Our exploration spans multiple levels of robotic operation, encompassing environmental interactions detectable via sensors, the learning processes during autonomous robot exploration, and addressing software bugs arising from limited context awareness at the design stage. The development and integration of DSLs, specifically ROSSMARieand EzSkiROS, form the cornerstone of our approach.

Key Insights and Findings: Through our research, we uncovered valuable insights into the application of these DSLs. Our evaluation of ROSSMARiehighlighted the necessity of task-specific recovery strategies. We found that effective runtime monitoring, particularly in human-interactive scenarios, must incorporate task context alongside sensor data to balance safety considerations accurately. While, EzSkiROSdemonstrated proficiency in identifying errors in high-level skill descriptions and was instrumental in detecting bugs in the pre- and post-conditions of existing SkiROS skills. Future testing on newly developed skills by various developers is planned, necessitating the public release of EzSkiROSfor broader usage and error collection.

My future endeavors, as outlined in Section 5, include a study of the interaction between DSLs where I aim to integrate skill-level information from EzSkiROS into ROSSMARie, enhancing recovery strategies and compensating for each DSL's limitations. Another future direction is to explore interactions of runtime monitoring and Reinforcement Learning(RL). As mentioned in Section 6, we have an initial plan to apply safety rules, derived from our DSLs, to the exploration phase of reinforcement learning.

We also plan to conduct further experiments to observe these interactions, and other components in the robotic system, particularly focusing on how these interactions influence the overall safety and reliability of the robot. Looking ahead, it would be immensely valuable to delve into the methodologies for evaluating DSLs and their interplay. Understanding and quantifying the impact of DSL interactions on robot performance, safety, and reliability would open new avenues in robotic software development. Additionally, exploring how runtime information gathered by ROSSMARiecould enhance the efficiency of EzSkiROSin subsequent runs presents an exciting research trajectory.

Final Thoughts: While several questions remain open for exploration, the groundwork laid in this thesis sets the stage for future advancements in the field of robotic safety and reliability. The integration of DSLs like ROSSMARieand ezSkiROS, coupled with their application in real-world scenarios, heralds a new era in the development of safer, more reliable autonomous robotic systems.

References

[Ach+17] MS Hendriyawan Achmad et al. “Tele-operated mobile robot for 3d visual inspection utilizing distributed operating system platform”. In: *International Journal of Vehicle Structures & Systems* 9.3 (2017), pp. 190–194.

[Ada+16] Sorin Adam et al. “Rule-based dynamic safety monitoring for mobile robots”. In: *Journal of Software Engineering for Robotics* 7.1 (2016), pp. 121–141.

[Afz+20] Afsoon Afzal et al. “A study on challenges of testing robotic systems”. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 96–107.

[AMK23] Faseeh Ahmad, Matthias Mayr, and Volker Krueger. “Learning to Adapt the Parameters of Behavior Trees and Motion Generators (BTMGs) to Task Variations”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2023, pp. 10133–10140.

[Amb+16] Stanislaw Ambroszkiewicz et al. “Fault Tolerant Automated Task Execution in a Multi-robot System”. In: *Intelligent Distributed Computing IX: Proceedings of the 9th International Symposium on Intelligent Distributed Computing–IDC’2015, Guimarães, Portugal, October 2015*. Springer. 2016, pp. 101–107.

[ATK21] Tansu Zafer Asici, Baris Tekin Tezel, and Geylani Kardas. “On the use of the analytic hierarchy process in the evaluation of domain-specific modeling languages for multi-agent systems”. In: *Journal of Computer Languages* 62 (2021), p. 101020.

[Ban+19] Siddhartha Banerjee et al. “Taking recoveries to task: Recovery-driven development for recipe-based robot tasks”. In: *The International Symposium of Robotics Research*. Springer. 2019, pp. 593–609.

[Bar17] Ankica Barišić. “Framework support for usability evaluation of domain-specific languages”. In: *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 2017, pp. 16–18.

[BSK19] Dominik Bork, Christine Schrüffer, and Dimitris Karagiannis. “Intuitive understanding of domain-specific modeling languages: proposition and application of an evaluation technique”. In: *Conceptual Modeling: 38th International Conference, ER 2019,*

Salvador, Brazil, November 4–7, 2019, Proceedings 38. Springer. 2019, pp. 311–319.

[Boz+16] Darko Bozhinoski et al. "Leveraging collective run-time adaptation for UAV-based systems". In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2016, pp. 214–221.

[Boz+19] Darko Bozhinoski et al. "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective". In: *Journal of Systems and Software* 151 (2019), pp. 150–179.

[CAÖ19] Michele Colledanchise, Diogo Almeida, and Petter Ögren. "Towards blended reactive planning and acting using behavior trees". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 8839–8845.

[CMÖ17] Michele Colledanchise, Richard M Murray, and Petter Ögren. "Synthesis of correct-by-construction behavior trees". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 6039–6046.

[CÖ18] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[CH11] Robert J Colvin and Ian J Hayes. "A semantics for Behavior Trees using CSP with specification commands". In: *Science of Computer Programming* 76.10 (2011), pp. 891–914.

[CH10] Robert Colvin and Ian J Hayes. "A semantics for Behavior Trees". In: (2010).

[Dit+17] Yvonne Dittrich et al. *Quality Assurance Process and Community Management in ROS*. Tech. rep. Technical report, rosin-project. eu, 2017.

[Fac19] Davide Faconti. "Mood2be: Models and tools to design robotic behaviors". In: *Eurecat Centre Tecnologic, Barcelona, Spain, Tech. Rep* 4 (2019).

[FN+20] Anders Fischer-Nielsen et al. "The forgotten case of the dependency bugs: on the example of the robot operating system". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 2020, pp. 21–30.

[Ghz+20] Razan Ghzouli et al. "Behavior trees in action: a study of robotics applications". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, pp. 196–209.

[Ghz+23] Razan Ghzouli et al. “Behavior Trees and State Machines in Robotics Applications”. In: *IEEE Transactions on Software Engineering* (2023).

[GMS21] Arnaud Gotlieb, Dusica Marijan, and Helge Spieker. “Testing Industrial Robotic Systems: A New Battlefield!” In: *Software Engineering for Robotics* (2021), pp. 109–137.

[Gra+08] Jeff Gray et al. “DSLs: the good, the bad, and the ugly”. In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. 2008, pp. 791–794.

[Hen+22] Thomas Henn et al. “Verification of Behavior Trees using Linear Constrained Horn Clauses”. In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2022, pp. 211–225.

[KB15] Gökhan Kahraman and Semih Bilgen. “A framework for qualitative assessment of domain-specific languages”. In: *Software & Systems Modeling* 14 (2015), pp. 1505–1526.

[Mar+21] Jabier Martinez et al. “Modelling the Component-based Architecture and Safety Contracts of ArmAssist in Papyrus for Robotics”. In: *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*. IEEE. 2021, pp. 13–18.

[MRK23] Matthias Mayr, Francesco Rovida, and Volker Krueger. “SkiROS2: A skill-based Robot Control Platform for ROS”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 6273–6280.

[May+22a] Matthias Mayr et al. “Combining Planning, Reasoning and Reinforcement Learning to solve Industrial Robot Tasks”. In: *arXiv preprint arXiv:2212.03570* (2022).

[May+22b] Matthias Mayr et al. “Learning Skill-based Industrial Robot Tasks with User Priors”. In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. 2022, pp. 1485–1492.

[May+22c] Matthias Mayr et al. “Skill-based Multi-objective Reinforcement Learning of Industrial Robot Tasks with Planning and Knowledge Integration”. In: *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2022.

[May+23] Matthias Mayr et al. “Using Knowledge Representation and Task Planning for Robot-Agnostic Skills on the Example of Contact-Rich Wiping Tasks”. In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2023, pp. 1–7.

[Miy+19] Alvaro Miyazawa et al. “RoboChart: modelling and verification of the functional behaviour of robotic applications”. In: *Software & Systems Modeling* 18 (2019), pp. 3097–3149.

[MNK22] Valery Marcial Monthe, Laurent Nana, and Georges Edouard Kouamou. “A Model-Based Approach for Common Representation and Description of Robotics Software Architectures”. In: *Applied Sciences* 12.6 (2022), p. 2982.

[Oli+20] Miguel Oliveira et al. “BhTSL, behavior trees specification and processing”. In: (2020).

[Par94] Lynne E Parker. “ALLIANCE: An architecture for fault tolerant, cooperative control of heterogeneous mobile robots”. In: *Proceedings of IEEE/RSJ international conference on intelligent robots and systems (IROS’94)*. Vol. 2. IEEE. 1994, pp. 776–783.

[PRBCZ17] Ildevana Poltronieri Rodrigues, Márcia de Borba Campos, and Avelino F Zorzo. “Usability evaluation of domain-specific languages: a systematic literature review”. In: *Human-Computer Interaction. User Interface Design, Development and Multimodality: 19th International Conference, HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part I 19*. Springer. 2017, pp. 522–534.

[Pol+18] Ildevana Poltronieri et al. “Usa-dsl: usability evaluation framework for domain-specific languages”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 2013–2021.

[Rév+00] Laurent Réveillere et al. “A DSL approach to improve productivity and safety in device drivers development”. In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE. 2000, pp. 101–109.

[Rob03] ABB Robotics. “A Dependable Real-Time Platform for Industrial Robotics”. In: *WADS 2003 Workshop on Software Architectures for Dependable Systems*. 2003, p. 19.

[TT22] Matteo Tadiello and Elena Troubitsyna. “Verifying Safety of Behaviour Trees in Event-B”. In: *arXiv preprint arXiv:2209.14045* (2022).

[Tre+21] Mickaël Trezzy et al. “Leveraging domain specific modeling to increase accessibility of robot programming”. In: *2021 IEEE International Workshop of Electronics, Control, Measurement, Signals and their application to Mechatronics (ECMSM)*. IEEE. 2021, pp. 1–9.

[Voe+19] Markus Voelter et al. “Using language workbenches and domain-specific languages for safety-critical software development”. In: *Software & Systems Modeling* 18.4 (2019), pp. 2507–2530.

[Wan+21] Lihui Wang et al. *Advanced Human-Robot Collaboration in Manufacturing*. Springer, 2021.

[ZFKA15] Saad Zafar, Naurin Farooq-Khan, and Musharif Ahmed. “Requirements simulation for early validation using Behavior Trees and Datalog”. In: *Information and Software Technology* 61 (2015), pp. 52–70.

INCLUDED PAPERS

ROSSMARIE: A DSL TO EXPRESS DYNAMIC SAFETY RULES AND RECOVERY STRATEGIES FOR AUTONOMOUS ROBOTS

1 Abstract

Ensuring functional safety is a critical challenge for autonomous robots, as they must operate reliably and predictably despite uncertainty. However, existing safety measures can over-constrain the system, limiting the robot's availability to perform its assigned task. To address this problem, we propose a more flexible strategy that equips robots with the ability to adapt to system failures and recover from those situations without human intervention. We extend a domain-specific language, Declarative Robot Safety (DeROS), whose runtime response is to stop a robot whenever it violates a safety rule (e.g., proximity to a human). Our extended language, ROSSMARie, adds the capability to monitor whether a rule is no longer violated and to recover and resume robot operation. We validate ROSSMARie on

Momina Rizwan, Christoph Reichenbach, Volker Krueger. "ROSSMARie: A Domain-Specific Language To Express Dynamic Safety Rules and Recovery Strategies for Autonomous Robots". In *Second Workshop on Quality and Reliability Assessment of Robotic Software Architectures and Components, June, 2nd, 2023, ICRA 2023, London, UK*. Conference website with selected contributions: qrarsac2023.

the ROS-based industrial platform SkiROS2 and verify its effectiveness in achieving safety and availability. Our experiments demonstrate that our DSL extension ensures functional safety while enabling robots to complete their tasks.

2 Introduction

Ensuring the safe behavior of robots in a dynamic and unpredictable environment where humans are present is a challenging task. Autonomous robots must be designed to operate effectively in uncertain environments and are able to adapt to system failures without external interference [Mül+21]. While deploying autonomous robots in real-life settings, the safety of the environment, human users, and the robot itself is of prior importance.

Runtime monitoring is one way of guaranteeing safety which has been explored in recent years [Mas+18]. A safety monitor is an independent component that can detect potential safety violations and intervene with recovery or corrective strategies. Adam et. al [Ada+16] trigger a stop action when the robot encounters an unexpected situation and waits for the operator to start the robot. While this strategy may be suitable for their experiments, it is a *conservative* strategy that for an autonomous robot For example, in the case of service robots operating in households, if the safety protocol requires the robot to stop whenever it encounters uneven terrain, it may not be able to complete its task, such as cleaning a room, as efficiently as it could if it were able to slow down until it has crossed the uneven area and then continue cleaning at its normal speed.

While stopping the robot is an important safety measure to prevent accidents, it can hinder the robot's ability to function effectively in certain contexts. Therefore, it is important to strike a balance between safety and functionality when designing safety strategies [Are+21]. Addressing these challenges requires innovative approaches to robot design and programming, as well as comprehensive safety protocols to ensure that robots can operate safely in a variety of contexts. By continuing to improve the safety behaviours of autonomous robots, we can unlock the full potential of these machines to make our lives easier and more efficient.

In this extended abstract, we focus on *functional safety* and assume that the sensors are reliable. A system is considered functionally safe if it operates correctly in response to its inputs and if it can't, then it should fail in a predictable manner [Ada+16]. We extend an existing domain-specific language designed in the spirit of DeROS [Ada+16] which generates a safety monitor (for ROS-based software) that enforces rules expressed in that language. The contributions include:

1. We modify the semantics of the language, enabling robots to resume after a hazardous/unsafe situation has passed.

2. We introduce strategies to recover from those safety hazards

3. We integrate the framework proposed by Adam et. al. [Ada+16] with SkiROS2, a skill-based platform for ROS, and we demonstrate the effectiveness of our recovery strategies by applying them to new scenarios.

To distinguish between the previous work and our implementation, we refer to our modified DSL as ROSSMARie.

3 Background: DeROS

DeROS [Ada+16] is a DSL to express dynamic safety rules for runtime monitoring based on informal safety specifications that provide information on components in Robot Operating System (ROS) (topics and nodes). The DeROS compiler then generates a runtime safety monitor to check the specified properties of the software system. This framework proposed by Adam et. al. [Ada+16] is aimed at isolating safety handling from the robot functionality and treating it as a cross-cutting concern.

4 ROSSMARie

ROSSMARie is an extension of DeROS with modified semantics. The runtime semantics of ROSSMARie enable continuous feedback from the sensors and allows the robot to resume its current operation as soon as the sensor values are in a safe range. We have implemented ROSSMARie in the JastAdd [HM03] meta-compiler.

4.1 Integration with SkiROS2

SkiROS2 [Rov+17] is a skill-based robot control platform that can execute multiple skills st the same time with the help of action server-based communication. Actions provide non-blocking background processing and are ideal for executing longer robot skills. While integrating ROSSMARie with SkiROS2, we encountered a problem where different modules send (publish) conflicting messages on the same ROS topic. This can result in jittering or unsafe behaviours from the robot. We realized that action servers need special handling. To address this, we added a *safety node* to choose which message to publish on the topic. Figure 1 shows an example of a safety node that filters out commands that set velocity. To introduce such a filter, we use the remap function in roslaunch. Remapping redirects a ROS node to publish on /unfil_cmd_vel instead of/cmd_vel.

To support ROS action servers in ROSSMARie code generation, we cancel the previous goal and send a new goal to the action server. DeROS, in comparison, can only support components that communicate using topic-based publish-subscribe communication.

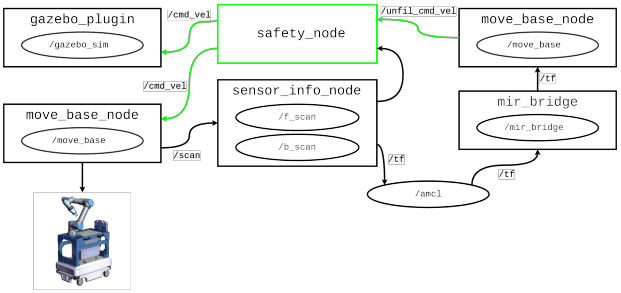


Figure 1: Safety monitor acting as a filter to avoid conflicting information published on a topic.

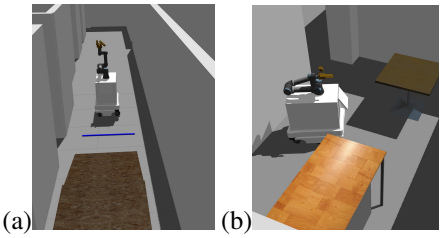


Figure 2: (a) Bump and a ramp in the corridor. (b) Placing a block on the unexpectedly high table with an unknown height.

5 Experiments

To showcase the recovery strategies, we performed experiments with a robot named Heron, as shown in Figure 2. Heron integrates a MIR 100 with a UR5e. The MIR 100 is an indoor autonomous mobile robot with a maximum payload capacity of 100 kg. It has two laser scanners and ultrasonic sensors providing 360° visual feedback. The maximum speed of the robot is 1.5 m/s forward and 0.3 m/s backward. The UR5e is a 20.6 kg robot arm with a maximum payload capacity of 5 kg. It has a six-axis force/torque sensor to detect collisions. We ran our experiments using SkiROS2 skills.

Case study I: We used ROSSMARie to define safety rules for Heron navigating in uneven terrain with three safety hazards i.e. bumps, slight and steep ramps shown in Figure 2(a). If the robot encounters a bump with one wheel (detected through an IMU sensor), our rules reduce the speed until the robot has fully crossed the bump. On the other hand, a slight ramp triggers an increase in speed to enter the operating area. If the ramp is steep, the robot recovery strategy is to go backward and replan. In all three scenarios, the robot tried to recover from the situation that could have led to damage to the robot.

Case Study II: Figure 2(b) shows a simulation setup where Heron's task is to move an object from one table and place it on another. An active compliance controller can produce vibrations in the robot arm when contact occurs while placing the object. To avoid any serious damage, we switch to position control whenever the force torque sensor detects those vibrations.

Case Study III: Proactive behaviours are required to avoid deadly/costly repercussions e.g. harming a human in a crash is more costly. For human-shared workspaces, we defined a safety rule to decrease arm speed whenever a human is detected in the room.

6 Limitations and Future Work

While conducting our experiments, we observed that recovery strategies can vary depending on the task at hand. While safety rules typically aim to maintain a safe distance from objects, some tasks require interacting with (e.g., pushing) an object. In such interaction scenarios, the robot may need to approach the object more closely than safety rules would normally allow. We plan to address such scenarios by allowing ROSSMARie's safety rules to be task-aware. During experiments, we also encountered conflicting recovery strategies for different safety rules. To identify and resolve such conflicts, we plan to statically check rules for overlap.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[Ada+16] Sorin Adam et al. "Rule-based dynamic safety monitoring for mobile robots". In: *Journal of Software Engineering for Robotics* 7.1 (2016), pp. 121–141.

[Are+21] Janis Arents et al. "Human–robot collaboration trends and safety aspects: A systematic review". In: *Journal of Sensor and Actuator Networks* 10.3 (2021), p. 48.

[HM03] Görel Hedin and Eva Magnusson. "JastAdd—an aspect-oriented compiler construction system". In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Mas+18] Lola Masson et al. “Tuning permissiveness of active safety monitors for autonomous systems”. In: *NASA Formal Methods Symposium*. Springer. 2018, pp. 333–348.

[Mül+21] Manuel Müller et al. “Industrial autonomous systems: a survey on definitions, characteristics and abilities”. In: *at-Automatisierungstechnik* 69.1 (2021), pp. 3–13.

[Rov+17] Francesco Rovida et al. “SkiROS— a skill-based robot control platform on top of ROS”. In: *Robot Operating System (ROS)*. Springer, 2017, pp. 121–160.

ENHANCING ROBOTIC AUTONOMY: STRATEGIES FOR DYNAMIC SAFETY AND IMMEDIATE RECOVERY

1 Introduction

Functional safety is a critical aspect during the operation of autonomous robotic systems. However, ensuring safe robot behavior has become a challenging task in the presence of dynamic and unpredictable environment where robots work alongside humans. Traditional definitions and approaches to functional safety, often guided by standards like ISO 12100 (Safety of Machinery) [Isoc] and ISO 26262 (Road Vehicles - Functional Safety) [Isob], can be overly conservative for the dynamic and complex nature of autonomous robots.

In traditional industrial settings, machinery is designed to perform a set of well-defined tasks in a controlled environment. Here, functional safety measures often involve deterministic responses to errors or unexpected situations, typically leading to an immediate shutdown or halt of the machine. For example, an industrial robotic arm might stop immediately if a safety perimeter is breached or if an internal fault is detected. While this approach effectively minimizes immediate risk, it doesn't align well with the concept of autonomy in robotics. Autonomous

Momina Rizwan, Christoph Reichenbach, Volker Krueger. "Enhancing Robotic Autonomy: Strategies for Dynamic Safety and Immediate Recovery". Technical Report 114, Lund Tekniska Högskola, 2024. **To be submitted.**

robots are expected to perceive their environment, make decisions, and continue operating despite uncertainties and changes, all while ensuring safety.

The conservative "stop and wait for the operator" approach negates the benefits of autonomy, as it requires human intervention to reset or address the issue, leading to downtime and reduced efficiency. Balancing safety and availability becomes a significant challenge, as overly aggressive safety measures can lead to frequent unnecessary stops, while lenient strategies might compromise safety. Recent ISO standards such as ISO-17757-2019 [Isoa] have introduced new rules for autonomous mobile robots (AMRs) to adapt to environmental changes. These include using systems to adjust speeds, disable certain operations, provide close-off areas, or make other necessary changes to ensure safe operation. However, autonomous robots are not limited to only mobile robots, but also include flying robots (drones), boats, and robotic arms.

Consider AMRs with an industrial manipulator that carries and assembles parts in a manufacturing environment. A human inspector works alongside the robot, checking the quality of these parts. As the robot assembles an electronic component, it senses the human inspector standing close to the assembly station. In such a scenario, the robot's standard operation might pose a risk to the human, especially if both reach for the same part or if the robot moves unexpectedly. Stopping the robot arm might seem like a safe response, but it could lead to the human being trapped between the robot and a wall or other obstacles, especially in a constrained space. Instead, the robot should adapt its behavior to the situation, perhaps by slowing down, entering a constrained motion mode, or switching to a gravity-free mode to ensure both safety and continued operation. This approach maintains safety while allowing the human inspector to continue their work without significant interruption.

Runtime safety monitoring has emerged as a promising approach to ensure safety requirements while maintaining operational efficiency [Mas+18; Mal+19]. By continuously observing the system's behavior and performance, runtime monitors can detect deviations or potential safety breaches and initiate appropriate responses. This approach allows for more nuanced safety strategies, such as reducing speed, rerouting, or other context-aware actions that maintain safety without completely stopping the system.

- **RQ1**: How can dynamic safety monitoring systems adapt to unpredictable environments while maintaining robust safety protocols?
- **RQ2**: What role do recovery strategies play in ensuring the continuous operation and safety of mobile robots in complex environments?
- **RQ3**: Are these recovery strategies task-dependant or they can be generalised?

In this paper, we aim to strike a balance between aggressive and lenient safety strategies in autonomous robotics, particularly focusing on domains such as robotic

arms. We introduce a Domain-Specific Language (DSL), ROSSMARie, inspired by DeROS [Ada+16], to articulate dynamic safety rules for autonomous robots and generate corresponding safety monitors. In the study [Ada+16], for most experiments a generic fault handling approach is used, meaning that a complete stop action was used as a reaction to any safety rule violation. While this approach fulfilled the specific safety requirements of the robots in those experiments, it may not be the optimal action for other robots. For example, in a situation where a mobile robot encounters an obstacle, instead of a generic stop or retreat response, the robot might choose to bypassing the obstacle if it's safe to do so. This keeps the robot operational while still adhering to safety protocols. The authors acknowledges the need for more advanced fault-handling strategies based on diagnosis and fault isolation, suggesting that DeRoS currently lacks a nuanced approach to handling different types of faults or safety issues. In this paper, we want to explore further recovery strategies and answer some research questions that we proposed earlier. The contributions of the research are as follows:

1. We refine the semantics of DeROS to enable continuous monitoring of hazardous situations. This enhancement enables a more dynamic and responsive safety monitoring system. By allowing continuous monitoring of hazardous situations, the system can quickly identify when a threat is no longer present and resume normal operations. This reduces unnecessary operational interruptions, improving efficiency while maintaining safety. It directly addresses RQ1.

2. We extend the framework with a broader range of action templates for recovery strategies. The introduction of more action templates for recovery strategies expands the framework's applicability to different robotic platforms, including those with robotic arms. This is done to validate RQ2 and RQ3.

3. We demonstrate the applicability and effectiveness of ROSSMARie through simulated and real robot experiments with navigation and manipulation tasks. This contribution provides empirical evidence addressing the research questions, showcasing the real-world impact of our proposed enhancements.

2 Background

Declarative Robot Safety (DeROS), a Domain-Specific Language (DSL) designed for expressing dynamic safety rules for mobile robots [Ada+16]. DeROS was specifically created to address the need for an isolated safety layer in robot software that is easy to understand and facilitates safety certification. It is intended to explicitly declare functional safety-critical concerns separately from the main program in terms of externally observable properties. According to the study presented in [Ada+16], developers create a DeROS program from an informal safety

specification, leveraging a system model for static consistency checks. The model details components like ROS topics and nodes. this model is required to automatically generate the necessary launch files for a ROS system. A DeROS compiler then generates a runtime safety component to monitor the software system's specified properties, ensuring dynamic adherence to safety rules. DeROS only support components that communicate using topic-based publish-subscribe.

In the DeRoS (Declarative Robot Safety) system, the following actions are mentioned as part of the recovery strategies:

- halting the robot completely as a response to safety rule violations.
- reduces the robot's speed, allowing it to continue operating but at a lower, safer speed. This is used when a less severe safety rule is violated.
- stopping the motor action is a more severe action than just stopping the robot, as it involves stopping the motor, and it remains active until manually reset by an operator.
- Controller OK Action action is used to reset the stop and low-speed actions. It signals that the robot's controller is functioning correctly and that it's safe to resume normal operation.

These actions, implemented in a dedicated robot-specific library, are a part of the functional safety features provided by DeRoS. The system's emphasis is on providing a straightforward and effective set of actions to respond to safety violations, primarily focusing on stopping or slowing down the robot to mitigate potential hazards. However, as noted in the paper, these actions, particularly the complete stop action, might not be the best fit for all types of robots, and an improved fault-handling based on diagnosis and fault isolation is required. In this paper, we are interested in how robots can use better fault-handling and recover automatically from safety hazards when faced with unforeseen events or failures.

3 Enhancing Functional Safety with Continuous Monitoring in ROSSMARie

Similar to DeROS, ROSSMARie compiler generates a runtime safety component designed to continuously monitors the robot's operational parameters and environmental factors against the predefined safety rules. The safety rules are evaluated based on real-time data received from various sensors and internal robot states. For instance, if a rule is set to monitor the robot's speed, ROSSMARie also checks the actual speed against the defined safety threshold.

Unlike DeROS, when a safety rule is violated (e.g., the robot exceeds a speed limit or detects an obstacle too close), ROSSMARie triggers predefined recovery actions while still updating the safety rules to monitor the current state and go

back to normal operational mode when the recovery actions are completed. For example, when encountering a human, the robot slows down and perform some cautionary behavior like having the arm go limp while human is around but as soon as the human leaves the robot goes back to finish the original task.

3.1 Implementation of ROSSMARie

In our pursuit to address the outlined research questions and to further validate the efforts of Adam et al. [Ada+16] in the field of functional safety in robotics, we have developed ROSSMARie, a DSL tailored to express dynamic safety rules and suitable recovery actions to adapt a robot's actions without aborting the given task. ROSSMARieis implemented using JastAdd [HM03], a meta-compilation system based on Java that supports Reference Attribute Grammars(RAGs). The language syntax of ROSSMARie closely follows the syntax of ROSSMARie to include features such as alias names for ROS topics, constants, measurement units, and intervals in logical expressions. This section discusses the implementation of ROSSMARie and how it contributes to addressing the research questions.

Unlike the examples in paper [Ada+16] which are aimed specifically to define safety rules for mobile robots, we apply ROSSMARie to describe rules for a robot arm as well. the safety rules for a robot with a robotic arm are shown in the Listing 1. To address RQ1, ROSSMARie implements a continuous monitoring mechanism, continuously updating the rules like "$oscillation\_not\_safe$" (as shown in Listing 1) with current force sensor information. This mechanism is crucial for ensuring that the robot can adapt its behavior in real time based on the current operational context. For instance, if a robot detects (through a force sensor attached to its end effector) that the arm is oscillating too fast, ROSSMARie let us express recovery actions like triggering a predefined safety response. The response of ROSSMARie in this case is defined as first decreasing stiffness and then retracting the arm until it reaches a safe state to continue the task as shown in Listing 1. This continuous monitoring ensures that safety is maintained dynamically, allowing the robot to resume normal operation as soon as the risk is mitigated.

The comprehensive action templates from the paper [Ada+16] are discussed in Section 2. To address RQ2, we have implemented more action templates than the authors mentioned in [Ada+16] to better suit the safety of robotic arms, enabling more sophisticated and domain-specific recovery strategies. These templates include actions like decrease_stiffness and go to torque_control_mode (shown in line 1-3 in Listing 1)to manage torques for arm motion, ensuring that the robot arm moves smoothly and predictably, thus avoiding those sudden or jerky movements that could pose risks. This is introduced as an additional proactive measure to the safety hazard encountered above. Considering the risk assessment of the robot arm, we also introduced a "gravity-free mode" action template which is aimed at the scenarios where human is involved. Gravity-free mode, or zero-gravity mode, is a state where the robot arm moves as if there is no gravity acting

```
1   action decrease_stiffness;
2   action retract;
3   action torque_control_mode;
4
5   const max_force = -15 N/sec
6   const max_torque = 60 Nm
7
8   input force = topic Wrench.force
9   input torque = topic Twist.angular
10  entity force_monitor{
11  oscillation_not_safe :
12     force.z > max_force for 3.0 sec;
13  }
14  entity torque_monitor{
15  torque_too_high:
16     torque > max_torque;
17  }
18  if force_monitor.oscillation_not_safe then { decrease_stiffness;
     retract;};
19  if torque_monitor.torque_too_high then { torque_control_mode;};
```

Listing 1: An example ROSSMARie code to specify safety rules for a robotic arm.

on it. This mode is typically achieved by counterbalancing the gravitational forces on the arm, making it feel weightless and easy to move manually. In the context of safety, entering gravity-free mode can be an effective strategy when close human interaction is detected, as it significantly reduces the risk of injury from sudden or forceful movements. The robot can switch to this mode as a precautionary measure, ensuring safety while still allowing some degree of operation or manual repositioning by a human operator.

3.2 Integration with SkiROS2

In order to integrate these action templates as robot skills, we use a robot control platform SkiROS2. Integrating our domain-specific language, ROSSMARie, with the SkiROS2 [Rov+17] robot control platform presented unique challenges, particularly in handling action server-based ROS communication. SkiROS2 [Rov+17], a skill-based robot control platform built atop the ROS framework, executes multiple skills concurrently using action server-based communication. This method is ideal for longer robot skills due to its non-blocking background processing capabilities.

Challenges faced To be able to integrate with SkiROS2, we had to generate recovery actions to cancel SkiROS2 skills which uses action server-based communication. However, our initial implementation of ROSSMARie(similar to DeROS was limited to components communicating via topic-based publish-subscribe, ne-

cessitating significant enhancements to support action servers.

In ROS, topic-based communication is the standard method for nodes to exchange messages. It's based on a publish-subscribe model where publishers send messages on a topic and subscribers receive them. This model is simple and effective for many scenarios but lacks the ability to handle long-running, stateful tasks that require feedback and potential preemption. Actions, on the other hand, are designed for these more complex tasks. They allow for continuous feedback during task execution and can be preempted or canceled if necessary. This makes actions particularly suitable for robot skills that involve prolonged activities or require the ability to stop and start in response to changing conditions. Each action maintains a state for the duration of a goal, and multiple goals can be processed in parallel, each with its unique state.

Another issue that we faced during the integration of ROSSMARie with SkiROS2 is that different modules were publishing conflicting messages on the same ROS topic, leading to erratic or unpredictable robot behavior. We realized that action servers need special handling. To resolve this, we introduced a *safety filter node* in our architecture. It is important to note that because DeROS didn't involve action-server based communication, they never needed a safety filter node. We will explain the safety filter node in the next section because it is important to show how that works.

3.3 Safety Filter Node

In this section, we will explain the safety filter node for ROSSMARie implementation. Although DeRoS system did include a concept referred to as the "DeRoS safety node" but it does not act as a filter to all the operational commands. In our case, the safety filter (as shown in Figure1 node acts as a mediator, ensuring that only the appropriate commands are executed by the robot, thereby preventing conflicts and ensuring smooth operation. It filters out potentially harmful commands based on the current safety context and rules defined in ROSSMARie. This node uses the remap function in roslaunch to redirect messages from the standard command topic to a filtered command topic, where the safety node has the opportunity to vet and modify the commands as needed.

For example, if a command is issued to set the robot's velocity that exceeds a safe threshold, the safety filter node can intercept this command and adjust the velocity to a safer level or stop the robot altogether, depending on the defined safety rules.

Implementing Action Server Support To fully support ROS action servers in ROSSMARie, we implemented functionality to cancel any previous goals and send new goals to the action server. This capability is crucial for maintaining the flexibility and responsiveness of the robot, allowing it to adapt to new instructions and safety constraints dynamically.

By enhancing ROSSMARie to handle action server-based communication, we've

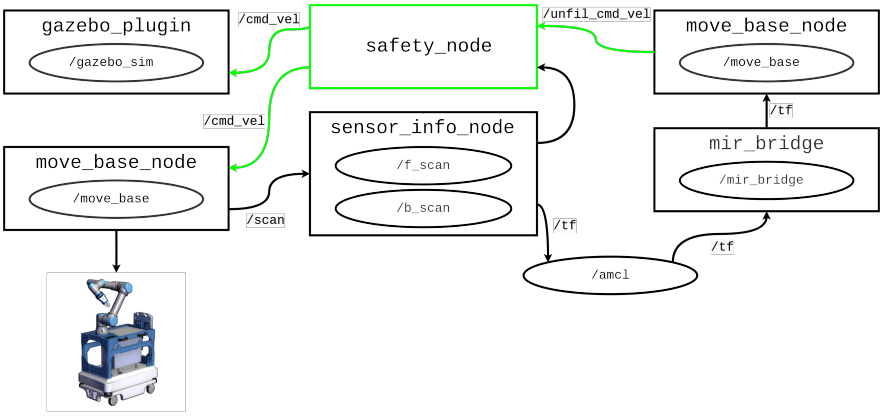


Figure 1: Safety node acting as a filter to avoid conflicting information published on a topic.

significantly broadened the range of robot skills and scenarios our DSL can support. This enhancement directly addresses the need for more sophisticated and adaptable safety mechanisms in complex robotic systems.

4 Experiments

For the experiments on real robot, we used a robot platform Heron (as shown in Figure 1), which is an integration of MIR 100 and UR5e robot by ARHO. Heron integrates the mobility and flexibility of the MIR base with the precision and dexterity of the UR5 arm, creating a comprehensive solution for automation and robotics challenges. MIR 100 is an indoor autonomous mobile robot with a maximum payload capacity of 100 kg. For navigation, it uses two laser scanners and ultrasonic sensors providing 360° visual feedback. It is also equipped with wheel encoders, and an IMU sensor, allowing for precise and safe maneuvering in diverse environments. The maximum speed of the robot is 1.5 m/s forward and 0.3 m/s backward. Mounted on the MIR base, the UR5 robotic arm is known for its lightweight design and exceptional accuracy. UR5e is a 20.6 kg robot arm with a maximum payload capacity of 5 kg. It provides six degrees of freedom and has a six-axis force/torque sensor to detect collisions.

The Heron platform is powered by a robust electrical system, ensuring consistent and reliable performance during extended operations. The high-level controller software, built on the ROS framework, runs on a powerful computing unit, ensuring real-time processing and decision-making. This software architecture allows for easy integration of additional sensors like a camera for visual feedback, making Heron a highly customizable and scalable solution.

In terms of safety, the Heron platform incorporates several features to ensure the well-being of human operators and the integrity of the surrounding environment. The system includes bumper sensors, emergency stop buttons, and advanced algorithms for detecting and avoiding obstacles. A dedicated safety PLC works in tandem with the high-level controller to monitor the system's status and execute safety protocols when necessary. Visual and auditory indicators, such as stack lights and buzzers, provide clear and immediate feedback about the robot's state and actions, further enhancing the safety measures.

The risk assessment of the robot indicates that while MIR base is well equipped with bumpers and safety features to maintain a certain distance from obstacles, it is not very good at adapting to the uneven terrain causing a risk of damaging itself. On the other hand, UR5 is an industrial robot with a metal body intended to be used in separate safety cells and is not designed for safe interaction.

4.1 Case Studies

(a) *Experimental Setup I: Heron Mobile Robot Navigating Unstructured Terrain*

Objective: This experiment aims to assess the navigation and recovery strategies of a MiR100 mobile robot when encountering unexpected terrain features like bumps and slopes in a corridor while it is on a mission to pick up a block from a specific room. The task demonstrates the robot's ability to adapt to unstructured environments typical in factory floors.

```
action moveBack;
action lowSpeed;
action increaseTorque;
const maxSpeed = 0.5 m/s
const reasonabletilt =  10 deg
const maxtilt = 20 deg
input orientation = topic imuInformation
entity imuSensorSystem {
    gentleSlope :
        orientation.pitch() not in reasonabletilt
        for 4.0 sec;

    steepRamp :
        orientation.pitch() not in maxtilt
        for 4.0 sec;

}

entity driveSystem {
    maxspeedExceeded :
        linearSpeed > maxSpeed for 2.0 sec;
    moving :
        linearSpeed > minSpeed for 4.0 sec;
}
```

```
25
26 if imuSensorsystem.gentleSlope and driveSystem.moving
27     then { increaseTorque; };
28
29 if imuSensorsystem.steepRamp and driveSystem.moving
30     then { lowSpeed; moveBack; };
```

Listing 2: An example ROSSMARie code to specify safety rules for

Setup Details: MiR100, a compact and robust mobile robot equipped with an Inertial Measurement Unit (IMU) sensor to detect changes in movement and orientation. The corridor leading to the target room features an unexpected bump and a gentle slope, simulating an unstructured factory floor. The robot's objective is to navigate the corridor, overcome the terrain challenges, and enter a room with a sloped entrance to pick up a designated block.

The unstructured terrain poses a risk of damage to the robot's body due to the unexpected bump and gentle slope. The robot's standard response to such sudden changes in terrain might be to stop entirely upon detecting a sharp jerk from the bump or the onset of a slope, which, while safe, could lead to task abandonment or significant delays. To address these challenges, we used ROSSMARie to express safety rules as shown in Listing 2. We also implemented adaptive recovery strategies based on the severity and nature of the terrain encountered:

- Upon detecting a bump through the IMU sensor indicating a jerky motion, the safety rule ensures to reduce the speed, allowing for safer passage over the bumpy terrain. This cautious approach minimizes the risk of damage while maintaining task progress.
- When a gentle slope is detected, instead of stopping, the robot increases its torque output temporarily to maintain momentum and stability until the slope levels off. This adjustment is crucial for ensuring that the robot can successfully navigate up or down slopes without halting its mission.

Observation: As the MiR100 approached the room with the sloped entrance, the IMU sensor detected the incline, triggering the robot to increase torque and adjust its speed accordingly. This adaptive response allowed the robot to navigate into the room smoothly. Once inside and on level ground, the robot returned to its normal torque operation and proceeded to successfully complete the task of picking up the block. The implemented recovery strategies demonstrated the robot's capability to adapt to minor structural changes in the terrain without stopping or aborting the task. This resilience is critical in dynamic and unstructured environments, ensuring that robots can carry out their duties efficiently and effectively, even when faced with unexpected obstacles.

The experiment with the MiR100 mobile robot underscores the significance of intelligent and adaptive recovery strategies in robotic navigation. By employing

advanced sensors and responsive control systems, the robot can adapt to various terrain challenges, ensuring continued operation and task completion. These capabilities are especially valuable in industrial settings, where environments are often unpredictable and dynamic. The successful navigation and task completion in this experiment highlight the potential of such adaptive strategies in enhancing the autonomy and efficiency of mobile robots in real-world applications.

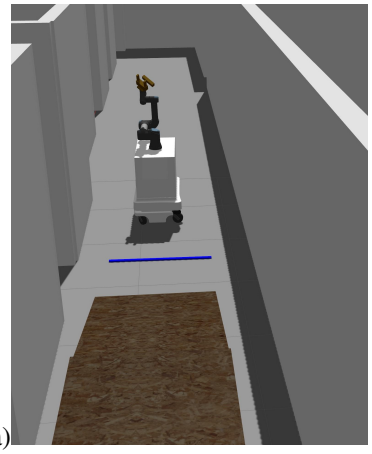
(a)  (b) 

Figure 2: (a) Simulation of the robot Heron crossing a corridor with a bump and ramp. (b) Real-robot experiment with Heron in the corridor with a speed bump.

(a) *Experimental Setup II: UR5 Arm in a Pick and Place Task*

Objective: The experiment is designed to demonstrate the execution of a pick and place task using a UR5 robotic arm. The task involves picking up a blue block and placing it on a specified goal location. The robot's behavior is managed using a Behavior Tree (BT) controlled through the SkiROS framework.

Setup Details: The UR5 arm is equipped with a gripper to handle the blue block. The UR5 arm is fitted with force sensors to detect contact forces and torques. The task involves the robot arm picking up a blue block from a specified location and placing it on a table whose height may vary between trials.

While placing the block, there was uncertainty about the commanded goal location for the block, including the varying heights of tables. This led to instances where the robot miscalculated the height and, upon contact with the table, the arm started to oscillate due to the impedance controller. These oscillations produced alarming noises and posed a risk to both the robot arm and the table. To express this safety constraint, we use ROSSMARie to define safety rules that concerns the functionality of a robot with arms (as shown in Listing 1). To address this issue, we

implemented a recovery action triggered by the detection of oscillations through the arm's force sensor. Upon detecting such an event:

1. **Retraction:** The arm first retracts from the table to avoid further contact and potential damage. This retraction is a safety measure to ensure that no additional stress is exerted on the table or the arm.

2. **Controller Adjustment:** After retracting, the controller's parameters are adjusted to account for the detected discrepancy in height. This may involve recalibrating the height estimation or adjusting the impedance control parameters to better handle variations in table height.

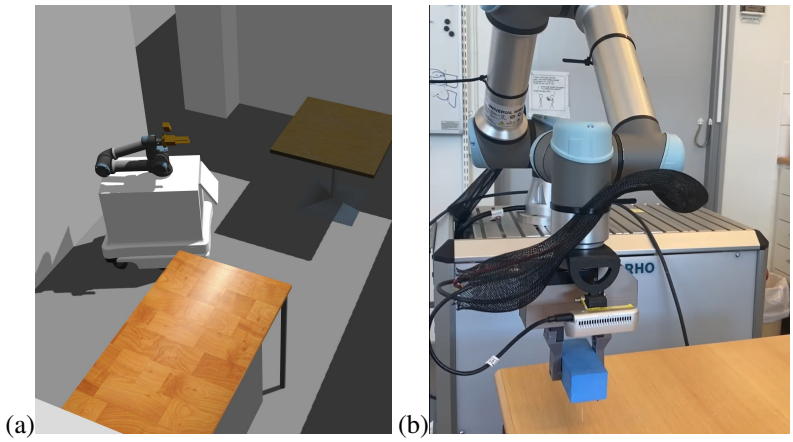
Because of the refined semantics of the language, when the robot senses that the oscillations are now controlled then it attempts the placement task again, approaching the table more cautiously and with better-informed height estimation.

Observation: While performing the task, it was observed that the torques exerted by the robot arm were excessively high at times. This was attributed to the motion planner sending goal locations that were too ambitious, prompting the robot to make abrupt movements to reach these points quickly. Such sudden movements led to spikes in torque, affecting the overall stability and safety of the operation. To mitigate this, we integrated a system-level torque control mechanism that continuously monitors torque values. This control system acts as a safeguard, ensuring that the robot operates within safe torque thresholds and preventing abrupt or unsafe movements. The implementation of this safety measure addresses a critical oversight in the initial code logic, where no checks for such torque spikes were in place. In Figure 3, we test our safety monitor in simulation and on the real robot.

The experiment with the UR5 arm performing a pick-and-place task underlines the importance of adaptive and responsive safety measures in robotic operations. By integrating force sensing, recovery actions, and system-level torque control, the setup ensures that the robot can adapt to unexpected scenarios and perform tasks safely and effectively. This approach not only protects the hardware but also contributes to a safer interaction environment, crucial for real-world applications where robots and humans coexist.

4.2 Discussion

The demonstrations conducted with the UR5 arm and the MiR100 mobile robot provide practical answers to our research questions. For RQ1, both robots showcased real-time behavioral adaptation to maintain safety without unnecessary halts; the UR5 arm adjusted its actions upon detecting oscillations, and the MiR100 navigated uneven terrain by modulating speed and torque. The continuous monitoring and immediate adjustment strategies seen in the experiments highlight how the semantics of safety rules can be modified for better responsiveness. Addressing RQ2, the varied nature of these robots – an arm and a mobile unit – demonstrates



(a) (b)

Figure 3: (a) Simulation of Heron placing a block. (b) Real-robot experiment of Heron placing a blue box on an unexpectedly high table with unknown height.

the strategies' applicability across different robotic platforms. Together, these experiments show a promising direction for making autonomous robots safer and more adaptive to their environments.

5 Related Work

Weber, Jörg, and Franz Wotawa. [WW10] presented a comprehensive approach for autonomous robots to autonomously handle software failures, focusing on runtime diagnosis, reconfiguration, planning with degraded capabilities, and ensuring the monitorability of plans. The study shows that the robot can switch to different, less efficient actions or new goals when original ones become unattainable due to the loss of certain capabilities. This approach allows the robot to continue its mission, albeit at reduced performance, leading to graceful degradation. while their approach also aims to enhance the safety and reliability of autonomous systems, the paper focuses on recovery and continued operation in the face of software failures, employing a combination of runtime diagnosis and AI planning for graceful degradation. In contrast, our work is geared towards defining and enforcing specific safety conditions to prevent hazards, potentially with some recovery mechanisms, but with a primary focus on rule compliance and hazard avoidance.

There is a rich body of work on the design of runtime assurance components for safety-critical systems [Fer+20; SVCH22; RCD21; Mas+18; Ada+16]. Some of these works present language-based approaches that instrument an implementation of a system to assure that certain executions are enforced to satisfy certain requirements, other approaches combine design time techniques with runtime ver-

ification techniques to assure that environment assumptions made at design time also hold at runtime.

Work on active monitors [Mas+18] synthesizes safety strategies for monitors and defines two properties *safety* and *permissiveness*. These properties ensure that any safety strategy should not remove the reachability of states that are intended, thus not defeating the very purpose of the robot. They bind the functionality and safety models together to get an orthogonal state space. Although the aim of this work closely matches our aim, they present a model-based approach while our runtime monitor deals with unexpected safety hazards which are hard to model.

6 Conclusion

In this paper, we present a DSL, ROSSMARie which follows a nuanced approach compared to the inspiration language DeROS to ensure the functional safety of robots without unnecessarily hindering their operational tasks. We appreciate the efforts of the authors of the study [Ada+16] and further verify the language with recovery strategies rather than generic fault-handling. We further substantiate the study by applying the safety rules to a robot arm which shows the generic applicability of dynamic safety rules designed in the spirit of DeROS. We perform simulated and physical robot experiments to demonstrate the viability of the proposed recovery strategies that maintain safety while making logical sense in the context of the task at hand.

Our observation is that in an ideal world, we want to have generalized safety rules that fit all the safety hazards a robot encounters but in reality some recovery strategies are task-dependant and vary from time to time. To understand this observation, we further explain the limitations of our work in the next section.

7 Limitations and Future Work

Our domain-specific language-based runtime approach, however, has limitations. We identify and discuss unsafe behaviors that are not yet handled by our language. We delve into the reasons behind these scenarios and explore potential solutions or mitigations. Understanding these limitations is crucial for future enhancements and for setting realistic expectations for the current system's capabilities. Some limitations are as follows:

- Our framework currently lacks a sophisticated mechanism for prioritizing rules, which can lead to conflicts when multiple safety constraints are applicable simultaneously. We discuss the implications of this limitation and propose potential strategies for introducing rule prioritization, ensuring that the most critical safety measures are always given precedence.

- We consider scenarios where the robot's task might inherently involve actions typically deemed unsafe, such as pushing an object or erasing a whiteboard. We discuss how our system can differentiate between genuinely hazardous situations and those that are part of a normal, intended operation. This discussion includes potential strategies for the robot to intelligently discern and adapt to such contexts, ensuring safety without aborting essential tasks.

By addressing these limitations and discussing potential pathways for future research, we aim to contribute to the ongoing development of safer, more reliable, and more effective autonomous robotic systems. Our work underscores the importance of continuous improvement and adaptation in the field of robotic safety, particularly as robots become increasingly integrated into diverse and dynamic human environments.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[Ada+16] Sorin Adam et al. "Rule-based dynamic safety monitoring for mobile robots". In: *Journal of Software Engineering for Robotics* 7.1 (2016), pp. 121–141.

[Isoa] *Earth-moving machinery and mining Autonomous and semi-autonomous machine system safety*. Standard. International Organization for Standardization, 2019.

[Fer+20] Angelo Ferrando et al. "ROSMonitoring: a runtime verification framework for ROS". In: *Towards Autonomous Robotic Systems: 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings 21*. Springer. 2020, pp. 387–399.

[HM03] Görel Hedin and Eva Magnusson. "JastAdd—an aspect-oriented compiler construction system". In: *Science of Computer Programming* 47.1 (2003), pp. 37–58.

[Mal+19] Piergiuseppe Mallozzi et al. "A runtime monitoring framework to enforce invariants on reinforcement learning agents exploring complex environments". In: *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*. IEEE. 2019, pp. 5–12.

[Mas+18] Lola Masson et al. “Tuning permissiveness of active safety monitors for autonomous systems”. In: *NASA Formal Methods Symposium*. Springer. 2018, pp. 333–348.

[RCD21] Quazi Marufur Rahman, Peter Corke, and Feras Dayoub. “Run-Time Monitoring of Machine Learning for Robotic Perception: A Survey of Emerging Trends”. In: *IEEE Access* 9 (2021), pp. 20067–20075.

[Isob] *Road vehicles Functional safety Part 1: Vocabulary*. Standard. International Organization for Standardization, Dec. 2018.

[Rov+17] Francesco Rovida et al. “SkiROS— a skill-based robot control platform on top of ROS”. In: *Robot Operating System (ROS)*. Springer, 2017, pp. 121–160.

[Isoc] *Safety of machinery General principles for design Risk assessment and risk reduction*. Standard. International Organization for Standardization, Nov. 2010.

[SVCH22] Marco Stadler, Michael Vierhauser, and Jane Cleland-Huang. “Towards flexible runtime monitoring support for ROS-based applications”. In: *Proceedings of the 4th International Workshop on Robotics Software Engineering*. 2022, pp. 43–46.

[WW10] Jörg Weber and Franz Wotawa. “Combining runtime diagnosis and ai-planning in a mobile autonomous robot to achieve a graceful degradation after software failures”. In: *International Conference on Agents and Artificial Intelligence*. Vol. 2. SCITEPRESS. 2010, pp. 127–134.

EZSKIROS: A CASE STUDY ON EMBEDDED ROBOTICS DSLS TO CATCH BUGS EARLY

1 Abstract

When we develop general-purpose robot software components, we rarely know the full context that they will execute in. This limits our ability to make predictions, including our ability to detect program bugs early. Since running a robot is an expensive task, finding errors at runtime can prolong the debugging loop or even cause safety hazards. In this paper, we propose an approach to help developers find bugs early with minimal additional effort by using embedded Domain-Specific Languages (DSLs) that enforce early checks. We describe DSL design patterns suitable for the robotics domain and demonstrate our approach for DSL embedding in Python, using a case study on an industrial tool SkiROS2, designed for the composition of robot skills. We demonstrate our patterns on the embedded DSL EzSkiROS and show that our approach is effective in performing safety checks while deploying code on the robot, much earlier than at runtime. An initial study with SkiROS2 developers show that our DSL-based approach is useful for early

Momina Rizwan, Ricardo Caldas, Christoph Reichenbach, and Matthias Mayr. "EzSkiROS: A Case Study on Embedded Robotics DSLs to Catch Bugs Early." In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE), pp. 61-68. IEEE, 2023*.
DOI: 10.1109/RoSE59155.2023.00014.

bug detection and improving the maintainability of robot code.

2 Introduction

The design and coding of robotic systems to perform socio-technical missions has never been more relevant or challenging. To ensure that robot developers can meet market demands with confidence in the correctness of their systems, a range of development tools and techniques is required. Specifically, robot development tools should provide expressive programming languages and frameworks that allow human developers to describe correct robot behavior [Bru+07].

For example, *SkiROS2*[3] [RGK17] is a skill-based knowledge integration tool for autonomous mission execution. It allows roboticists to write robot skills such as "pick" or "drive" skill. Skills are defined in a modular way to allow interoperability between different tasks and robot systems. Each skill description is based on pre-conditions that are checked before a skill execution, and post-conditions that are checked after the skill execution. In SkiROS2, these conditions are based on the robot's knowledge, organised into an ontology. An ontology represents the concepts and relations in the domain to check whether conditions necessary for the execution are met.

As a concrete example, the parameters for a "pick" skill shown in Figure 1, have ontology relations such as "gripper is part of the robot arm", which are used to infer other parameters such as "which arm to move" or "what is the location of the object". An object should not be part of the robot arm, as this would imply that the object always moves with the arm. The developer must be careful when writing such relationships, as bugs introduced at this stage tend to remain silent and can be difficult to debug.

To avoid such errors, we propose to use a DSL to allow us to analyse the code for possible errors after build time, while deploying it on the actual robot. The benefits of using DSLs to aid debugging, visualization, and static checking are well-known. DSLs have been used for mission specification [Dra+21], and robot knowledge modeling [Ceh+11]. Nordmann et al. [Nor+16] collect and categorise over 100 such DSLs for robotics in their *Robotics DSL Zoo*[3].

In this paper, we propose to help robot developers, who write control logic in Python, to catch bugs early by embedding DSLs directly in Python. We support our case through:

- A survey of Python language features that enable DSL embedding;
- Two design patterns for *embedding DSLs in general-purpose programming languages* that address common challenges in robotics, with details on how to implement these patterns in Python;

[3] `https://github.com/RVMI/skiros2`
[3] https://corlab.github.io/dslzoo

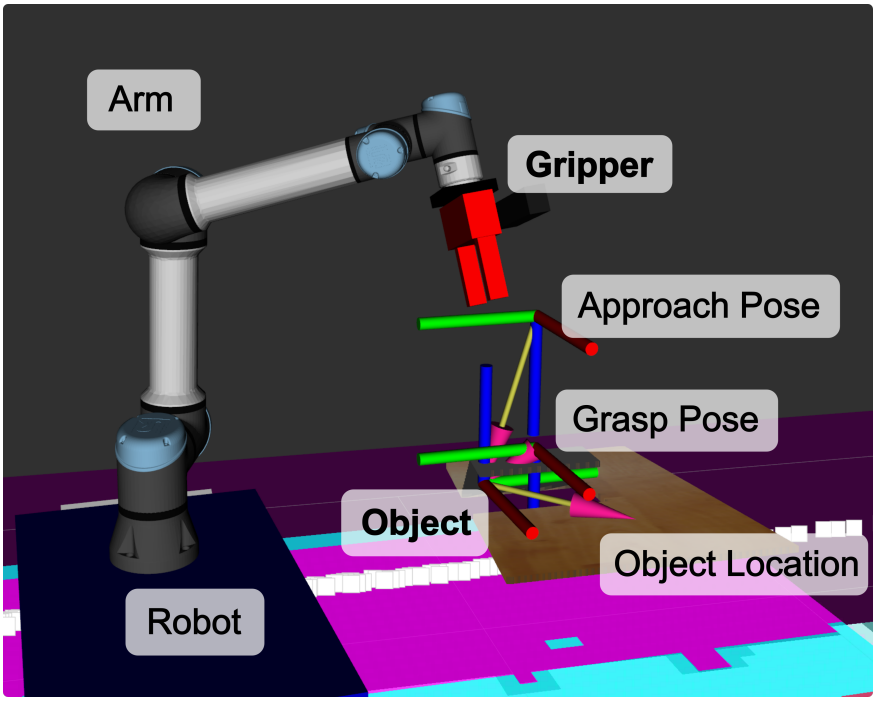


Figure 1: The robot using a pick skill with a visualization of the necessary parameters. To run this skill, we only need the Gripper and the Object parameters. SkiROS2 can deduce all other necessary parameters through a set of rules in the skill description shown in Listings 3 and 5.

- A case study of a robotics software SkiROS2 , in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs.

To this end, we place our contributions in perspective by analysing the state-the-art in embedded DSLs for robotics Section 3. Then, in Section 4, we offer a set of four robotic DSL design patterns to embed DSL constructs in Python. In short, our design patterns aim at lessening cognitive complexity (i.e., readability and modifiability) and shortening the feedback loop by materializing ontology constructs in a type system. Moreover, in Section 6, we demonstrate the efficacy of applying the four patterns in a skill-based robot development framework in a case study with SkiROS2, originating EzSkiROS. We conduct a user study with experts in robotics and SkiROS2 to evaluate to what extent our EzSkiROS promotes readability and modifiability, Section 7. EzSkiROS and the material used for the case study are available in an online replication package. Finally, we conclude the paper and outline possible directions to follow up with the study.

3 Related Work

Several studies have explored the use of model-driven approaches for programming robots. Buch et al. [Buc+14] describe an internal Domain-Specific Language (DSL) over C++ to sequence robotic skills using pre- and post-conditions. Their DSL uses a model-driven approach to instantiate a textual representation of the assembly sequence, which is interpreted to execute the assembling behavior. However, it is unclear whether the authors use early checking techniques to prevent erroneous sequences. The authors argue in favor of a loop between simulation and active learning to overcome uncertainties in the environment. Kunze et al. [KRB11] propose the Semantic Robotic Description Language (SRDL), a different model-based approach that matches robot descriptions and actions via static analysis of robot capability dependencies. SRDL uses Web Ontology Language (OWL) notation to model knowledge about robots, capabilities, and actions.

Coste-Manière and Turro [CMT97] propose MAESTRO, an external DSL for specifying reactive behavior and checking in the robotics domain that handles complex and hierarchical missions prone to concurrency and requiring portable solutions. MAESTRO allows specification of user-defined typed events that may or must occur before (pre-conditions), during (hold-conditions), or after (post-conditions) task execution. MAESTRO offers type-checking of user-defined types and stop condition checks.

4 Embedding Robotics DSLs in Python

DSLs can help developers by simplifying notation and improving performance or error detection. However, developing and maintaining DSLs requires effort. For *external DSLs* (e.g., MAESTRO, SRDL), much of this effort comes from building a language frontend. *Internal* or *embedded DSLs* (as in Buch et al. [Buc+14]) avoid this overhead, and instead re-use an existing "host" language, possibly adjusting the language's behaviour to accommodate the needs of the problem domain.

We look at Python as one of the three main supported languages of the popular robotics platform ROS [Qui+09]. The other two languages, C++ and LISP, also support internal DSLs, but with different trade-offs.

4.1 Python Language Features for DSLs

While Python's syntax is fixed, it offers several language constructs that DSL designers can repurpose to reflect their domain, such as freely overloadable infix operators (excluding the type-restricted boolean operators), type annotations (since Python 3.0), and decorator mechanisms [Mog+13].

Listing 1 illustrates some of these techniques. Class `A` represents a *deferred* operation `op` with parameters `args`. `A.eval` (Line 18) forces recursive evaluation. The `@staticmethod` decorator tells Python that this method takes no implicit `self`

```
1  class M(type):
2    def __getattribute__(self, name):
3      if name[0] == '_' and name[1:].isdecimal():
4        return self(lambda x: x, int(name[1:]))
5      return type.__getattribute__(self, name)
6
7  class A(metaclass=M):
8    def __init__(self, op, *args):
9      self.op = op
10     self.args = args
11   def __add__(self, other):
12     return A(int.__add__, self, other)
13   def __call__(self, arg : int):
14     return A(int.__mul__, self, arg)
15   @staticmethod
16   def eval(e):
17     if isinstance(e, A):
18       return e.op(*(A.eval(y) for y in e.args))
19     return e
20
21 a = A._2 + 1  # a = A.__add__(A(λ x:x, 2), 1)
22 b = a(4)      # b = a.__call__(4)
23 print(A.eval(b))  # evaluates (2 + 1) * 4
```

Listing 1: DSL-friendly Python features: decorator (line 15), metaclass (lines 1–7), overloading (lines 11,13), type annotations (line 13)

parameter. DSL designers can define other decorators to transform the semantics of functions, methods, or classes.

The *metaclass* M in line 1 allows class A (line 7) to handle references to unknown class attributes. The code in lines 2–5 checks for attributes that start with an underscore and continue in decimal digits. Line 21 shows how we can write A._2 to construct an instance of A (via line 4).

Class A overloads infix addition in line 11 and function call notation in line 13, which allows instances of A to participate in addition and to behave like callable functions (lines 21 and 22). While the code is a toy example, it illustrates how a DSL designer can construct in-memory representations of complex computations for *staging*, which could (e.g., in A.eval) perform optimisations or translate the code representation into a more efficient format (e.g., for a GPU).

Line 13 illustrates Python's type annotations, annotating parameter arg with type int. By default, such annotations have no runtime effect, but DSL designers can access and repurpose them to collect DSL-specific information without interference from Python. Since Python 3.5 (with extensions in 3.9), these annotations also allow type parameters (e.g., x : list [int]).

Finally, Python permits dynamic construction of classes (and metaclasses),

which we have found particularly valuable for the robotics domain: since the system configuration and world model used in robotics are often specified outside of Python (e.g., in configuration files or ontologies) but are critical to program logic, we can map them to suitable type hierarchies at robot launch time (just after build time).

4.2 Robotics DSL Design Patterns

Domain Language Mapping Our first pattern's *purpose* is *to make domain notation visible in Python, to decrease notational overhead.* It is a direct application of the "Piggyback" DSL implementation pattern documented by Spinellis [Spi01].

As an example, the ontology specification language OWL allows us to express the relationships and attributes of the objects in the world, the robot hardware and the robot's available capabilities (skills and primitives). Existing libraries like *owlready2* [Lam17] already expose these specifications as Python objects, so if the ontology contains a class pkg:Robot, we can create a new "Robot" object by writing

```
r = pkg.Robot("MyRobotName")
```

and iterate over all known robots by writing

```
for robot in pkg.Robot.instances(): ...
```

The owlready2 library creates these classes at runtime, based on the contents of the ontology specification files. Thus, changes in the ontology are immediately reflected in Python: if we rename pkg:Robot in the ontology, the code above will trigger an error when executed.

While Moghadam et al. expressed concern about "syntactic noise" for DSL embedding in earlier versions of Python [Mog+13] compared to external DSLs, found such noise to be modest in modern Python, and instead emphasise the advantages of embedding in a language that is already integrated into the ROS environment and that developers are familiar with.

In tools like SkiROS2, combining Python code, ontologies and configuration files at runtime introduces points of failure. To detect such failures early, we propose a second pattern:

Early Dynamic Checking The *purpose* of this pattern is *to detect type and configuration errors in a critical piece of code early, such as during robot launch time, with no or minimal extra effort for developers*. The *conditions* for this pattern are:

- We can collect all critical pieces of code at a suitably early point during execution
- The critical code does not depend on return values of operations that we cannot predict at robot deployment

The *behaviour* of this pattern is as follows:

```
1 def expand(self, skill):
2   skill.setProcessor(Sequential())
3   skill(
4     self.skill("Navigate", ""),
5     self.skill("WmSetRelation", "wm_set_relation",
6       remap={'Dst': 'TargetLocation'},
7         specify={'Src': self.params["Robot"].value,
8         'Relation': 'skiros:at', 'RelationState': True}),)
```

Listing 2: Constructing the behavior tree of a drive skill in SkiROS2. It is a sequential execution of the compound drive skill "Navigate" and a primitive skill to update the world model ("WmSetRelation").

- We execute all critical pieces of code early, while redefining the semantics of the predetermined set of operations (e.g. ontology relations from our previous example) to immediately return or to only perform checking

In Python, configuration and type errors only trigger software faults once we run code that depends on faulty data. In robotics, we might find such code in operations that (a) run comparatively late (e.g., several minutes after the start of the robot) and (b) are difficult to unit-test (e.g., due to their coupling to specific ROS functionality and/or robotics hardware). For robotics developers, both challenges increase the cost of verification and validation [Rei21]: a fault might trigger only after a lengthy robot program and require substantial manual effort to reproduce. For example, a software module for controlling an arm might take a configuration parameter that describes the target arm pose. If arm control is triggered late (e.g., because the arm is part of a mobile platform that must first reach its goal position), any typos in the arm pose will also trigger the fault late. If the pose description comes from a configuration file or ontology, traditional static checkers will also be ineffective. We can only check for such bugs after we have loaded all relevant configuration.

Through careful software design, developers can work around this problem, e.g., by checking that code and configuration are well-formed as soon as possible, before they run the control logic. If the critical code itself is free of external side effects, the check can be as simple as running the critical code twice. For example, SkiROS2 composes *BTs* [CÖ18] within such critical Python code (Listing 2): composing (as opposed to running) these objects has no side effects, so we can safely construct them early to detect simple errors (e.g., typos in parameter names). This is a typical example that eludes static checking but is amenable to Early Dynamic Checking: line 7 depends on `self.params["Robot"].value`, which is a configuration parameter that we cannot access until the robot is ready to launch. Not all robotics code is similarly declarative. Consider the following example, in a hypothetical robotics framework in which all operations are subclasses of `RobotOp`

and must provide a method run() that takes no extra parameters:

```
class MyRobotOp(RobotOp):
  def __init__(self, config): # Configure
    self.config = config
  def check(self):  # Check configuration
    assert self.config.mode in ["A", "B"]
    assert isinstance(self.config.v, int)
  def run(self):    # Run with configuration
    if self.config.mode == "A":
      self.runA();
    elif self.config.mode == "B":
      self.runB(self.config.v + 10);
    else:
      fail()
```

Here, the developers introduced a separate method check() that can perform early checking during robot initialisation or launch. However, check() and run() both have to be maintained to make the same assumptions.

The Early Dynamic Checking pattern instead uses internal DSL techniques to allow developers to use the same code in two different ways: (a) for checking, and (b) for logic.

In our example, calling run() "normally" captures case (b). For case (a), we can also call run(), but instead of passing an instance of MyRobotOp, we pass a *mock* instance of the same class, in which operations like runA() immediately return:

```
class MyRobotOpMock:
  def __init__(self, parent):
    self.parent = parent
  @property
  def config(self):
    # self.config  = self.parent.config
    return self.parent.config
  def runA(self):
    pass # mock operation: do nothing
  def runB(self, arg):
    pass # mock operaiton: do nothing
```

If we execute MyRobotOpMock.run() with the same configuration as MyRobotOp , run() will execute almost as for MyRobotOp but immediately return from any call to runA or runB. If the configuration is invalid, e.g., if config.mode == "C" or config.v == false, running MyRobotOpMock.run() will trigger the error early.

Since Python can reflect on a class or an object to identify all fields and methods, we can construct classes like MyRobotOpMock at run-time: instead of writing them by hand, we can implement a general-purpose mock class generator that constructs methods like runA and accessors like config automatically. If the configuration objects may themselves trigger side effects, we can apply the same technique to them.

However, the above implementation strategy is only effective if we know that the critical code will only call methods on self and other Python objects that we

know about ahead of time. We can relax this requirement by controlling how Python resolves nonlocal names:[1]

```
FunctionType(MyRobotOp.run.__code__, globals() | { 'print' : g})(obj)
```

This code will execute `obj.run()` via the equivalent `MyRobotOp.run(obj)`, but replace all calls to `print` by calls to some function `g`. The same technique can use a custom map-like object to detect at runtime which operations the body of the method wants to call and handle them suitably.

However, the more general-purpose we want to allow the critical code to be, the more challenging it becomes to apply this pattern. For instance, if the critical code can get stuck in an infinite loop, so may the check; if this is a concern, the check runner may need to use a heuristic timeout mechanism. A more significant limitation is that we may not in general know what our mocked operations like `runA()` should return, if anything. If the critical code depends on a return value (e.g., if it reads ROS messages), the mocked code must be able to provide suitable answers. The same limitation arises when the critical code is in a method that takes parameters. If we know the type of the parameter or return value, e.g. through a type annotation, we can exploit this information to repeatedly check (i.e., *fuzz-test*) the critical code with different values; however, without further cooperation from developers, this method can quickly become computationally prohibitive.

If we know that the code in question has simple control flow, we may be able to apply the next pattern, Symbolic Tracing.

Symbolic Tracing The *purpose* of this pattern is *to detect bugs in a critical piece of code early, if that code depends on parameters or operation return values, with minimal extra effort for developers.* The *conditions* for this pattern are that

- We can access and execute the critical code
- We have access to sufficient information (via type annotations, properties, ...) to simulate parameter values and operation return values *symbolically* (see below)
- The number of control flow paths through the critical code is small (see below)

The *behaviour* of this pattern is as follows:

1. We execute the critical code while passing symbolic values as parameters and/or returning symbolic values from operations of relevance
2. We collect any constraints imposed by operations on the symbolic values
3. After executing the critical code, we verify the constraints against the problem domain

[1] Python's `eval` function offers similar capabilities, but as of Python 3.10 does not appear to allow passing parameters to code objects.

Here, a *symbolic* parameter is a special kind of mock parameter that we use to record information [Kin76].

Consider the following RobotOp subclass:

```
class SetArmSpeedOp(RobotOp):
  def run(self, speedup):
    self.setArmSpeed(speedup)
    self.setArmSafety(speedup)
```

This class only calls two operations, but its run operation depends on a parameter speedup about which we know nothing a priori — thus, we cannot directly apply the Early Dynamic Checking pattern.

In cases where we lack prior knowledge about an operation, it may still be possible to obtain useful insights about it. For example, if we are aware that setArmSpeed accepts only numeric parameters and setArmSafety only accepts boolean parameters, we can flag this code as having a type error. To avoid blindly testing various parameters, we can pass a symbolic parameter to the run function and employ a modified version of the mock-execution strategy used in Early Dynamic Checking. The mock objects can be adapted as follows:

```
TYPE_CONSTRAINTS = []

class SetArmSpeedOpMock:
  def setArmSpeed(self, obj):
    TYPE_CONSTRAINTS.append((obj, float))
  def setArmSafety(self, obj):
    TYPE_CONSTRAINTS.append((obj, bool))
```

We can now (1) create a fresh object obj and an SetArmSpeedOpMock instance that we call mock, (2) call SetArmSpeedOp.run(mock, obj), and (3) read out all constraints that we collected during this call from TYPE_CONSTRAINTS, and check them for consistency, which makes it easy to spot the bug. If the constraints come from accesses to obj (e.g., method calls like obj.__add__(1) that result from code like obj + 1), obj itself can collect the resultant constraints.

Depending on the problem domain, constraint solving can be arbitrarily complex, from simple type equality checks to automated satisfiability checking [BR20]. It can involve dependencies across different pieces of critical code (e.g., to check if all components agree on the types of messages sent across ROS channels, or to ensure that every message that is sent has at least one reader). However, this approach requires information about specific operations like setArmSpeed and setArmSafety, which can be providedto Python in a variety of ways, e.g., via type annotations.

As an example, consider an operation that picks up a coffee from the table with a gripper, where we annotate all parameters to run with Web Ontology Language (OWL) ontology types:

```
1 class PickCoffeeTableOp(RobotOp):
2   def run(self, robot : rob.Robot,
3           gripper : rob.Gripper,
4           coffee_table : world.Furniture):
5     // bug:
6     assert coffee_table.robotPartOf(robot);
7     ...
```

This example is derived from the SkiROS2 ontology, with minor simplifications. In the above SkiROS2 code, the developer intended to write a precondition that to be able to pick a coffee cup, the robot should be close to the table. Instead, the developer mistakenly wrote that a robot should be a part of the coffee table.

The ontology requires that robotPartOf is a relation between a technical Device and a Robot. However, Furniture is not a subtype of Device, so the assertion in line 6 is unsatisfiable.

We can again detect this bug through symbolic tracing. This time we must construct symbolic variables for robot, gripper, and coffee_table that expose methods for all applicable relations, as described by their types. For instance, gripper will contain a method robotPartOf(gripper, obj) that records on each call that gripper and obj should be in a robotPartOf relation. Meanwhile, coffee_table will not have such an operation. When we execute run(), we can then defer to Python's own type analysis, which will abort execution and notify us that coffee_table lacks the requisite method.

Key to this symbolic tracing is our use of mock objects as symbolic variables. Symbolic variables reify Python variables to objects that can trace the operations that they interact with, in execution order, and translate them into constraints.

The main *limitation* of this technique stems from its interaction with Python's boolean values and control flow, e.g. conditionals and loops. Python does not allow the boolean operators to return symbolic values, but instead forces them (at the language level) to be bool values; similarly, conditionals and loops rely on access to boolean outcomes. Thus, when we execute code of the form `if x: ...`, we must decide right there and then if we should collapse the symbolic variable that x is bound to True or False. While we can re-run the critical code multiple times with different decisions per branch, the number of runs will in general be exponential over the number of times that a symbolic variable collapses to a bool.

4.3 Alternative Techniques for Checking

Internal DSLs are not the only way to implement the kind of early checking that we describe. The mypy tool[2] is a stand-alone program for type-checking Python code. Mypy supports plugins that can describe custom typing rules, which we could use e.g. to check for ontology types. Similarly, we could use the Python `ast` module to implement our own analysis over Python source code. However,

[2] `https://mypy-lang.org/`

both approaches require separate passes and would first have to be integrated into the ROS launch process. Moreover, they are effectively static, in that they cannot communicate with the program under analysis; thus, we cannot guarantee that the checker tool will see the same configuration (e.g., ontology, world model).

Another alternative would be to implement static analysis over the bytecode returned by the Python disassembler `dis`, which can operate on the running program. However, this API is not stable across Python revisions[3].

An external DSL such as MAESTRO [CMT97] would similarly require a separate analysis pass. However, it would be able to offer arbitrary, domain-specific syntax and avoid any trade-offs induced by the embedding in Python (e.g., boolean coercions). The main downside of this technique is that it requires a completely separate DSL implementation, including maintenance and integration.

5 Case Study: An open source software for skill-based robot execution

As a case study, we implement our patterns on the skill-based robot control platform SkiROS2 [RGK17]. SkiROS2 is used by several research institutions in the context of industrial robot tasks [May+22c; May+22b; Wut+21; May+22a]. It is implemented in Python, on top of the ROS [Qui+09] middleware. SkiROS2 uses behavior trees (BTs) [CÖ18] formalism to represent procedures. We refer the reader to [CÖ18] for a general introduction to BTs, to [Rov+17] for a thorough introduction to *SkiROS1* and to [RGK17] for BTs in SkiROS2.

SkiROS2 implements a layered, hybrid control architecture (Fig. 2) to define and execute parametric *skills* for robots [Bøg+12; Kru+16]. As the figure shows, SkiROS2 represents knowledge about the skills, the robot and the environment in a *World Model* (WM) with the *Ontologies* specified in OWL format. This explicit representation, built upon the World Wide Web Consortium's Resource Description Framework standard(RDF), allows the use of existing ontologies.

Skills in SkiROS2 are parametric procedures that modify the world state from an initial state to a final state according to pre- and post-conditions [Ped+16]. Skills can be either primitive or compound skills. Primitive skills are atomic actions that implement functions that change the real world, such as moving a robot arm. Whereas, compound skills allow to use primitive skills and other compound skills in a BT to build more complex behaviors. An example for such a connection is shown in Listing 2. All of these skills are loaded by the *Skill Manager* at robot launch time (shown in Fig. 2.

Every skill implements a *Skill Description* and a *Skill Implementation* as shown in Fig. 2. *Skill Description* consists of four elements:

1. *Parameters* define input and output of a skill

[3] `https://docs.python.org/3/library/dis.html`

```
class Pick(SkillDescription):
  def createDescription(self):
    self.addParam("Robot", Element("cora:Robot"), ParamTypes.Inferred)
    self.addParam("Arm", Element("rparts:ArmDevice"), ParamTypes.Inferred)
    self.addParam("StartPose", Element("skiros:TransformationPose"),
     ParamTypes.Inferred)
    self.addParam("GraspPose", Element("skiros:GraspingPose"), ParamTypes.
     Inferred)
    self.addParam("ApproachPose", Element("skiros:ApproachPose"),
     ParamTypes.Inferred)
    self.addParam("Workstation", Element("scalable:Workstation"),
     ParamTypes.Inferred)
    self.addParam("ObjectLocation", Element("skiros:Location"), ParamTypes.
     Inferred)
    self.addParam("Object", Element("skiros:Product"), ParamTypes.Required)
    self.addParam("Gripper", Element("rparts:GripperEffector"), ParamTypes.
     Required)

    self.addPreCondition(self.getRelationCond("ObjectLocationContainObject"
     , "skiros:contain", "ObjectLocation", "Object", True))
    self.addPreCondition(self.getRelationCond("GripperAtStartPose", "skiros
     :at", "Gripper", "StartPose", True))
    self.addPreCondition(self.getRelationCond("NotGripperContainObject", "
     skiros:contain", "Gripper", "Object", False))
    self.addPreCondition(self.getRelationCond("ObjectHasAApproachPose","
     skiros:hasA", "Object", "ApproachPose", True))
    self.addPreCondition(self.getRelationCond("ObjectHasAGraspPose", "
     skiros:hasA", "Object", "GraspPose", True))
    self.addPreCondition(self.getRelationCond("RobotAtWorkstation", "skiros
     :at", "Robot", "Workstation", True))
    self.addPreCondition(self.getRelationCond("
     WorkstationContainObjectLocation", "skiros:contain", "Workstation", "
     ObjectLocation", True))
    self.addPostCondition(self.getRelationCond("NotGripperAtStartPose", "
     skiros:at", "Gripper", "StartPose", False))
    self.addPostCondition(self.getRelationCond("GripperAtGraspPose", "
     skiros:at", "Gripper", "GraspPose", True))
    self.addPostCondition(self.getRelationCond("
     NotObjectContainedObjectLocation", "skiros:contain", "ObjectLocation",
      "Object", False))
    self.addPostCondition(self.getRelationCond("GripperContainObject", "
     skiros:contain", "Gripper", "Object", True))
```

Listing 3: An excerpt of the parameters, pre- and post-conditions of a pick skill in SkiROS2 without EzSkiROS. It depends heavily on the usage of string to refer to parameters or classes in the ontology.

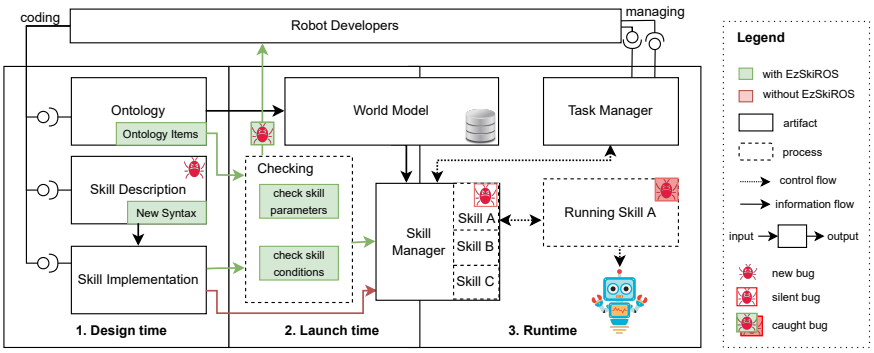


Figure 2: A diagram with the different components of SkiROS2, their relations and the additions by EzSkiROS. Previously, a bug that has been introduced in a skill description by a developer will often only trigger at runtime. EzSkiROS addresses these costs and risks by finding a wide range of bugs at launch time when the skills are loaded at launch time.

2. *Pre-conditions* must hold before the skill is executed
3. *Hold-conditions* must be fulfilled during the execution
4. *Post-conditions* are checked once the execution finished

Listing 3 shows how developers define these skills in SkiROS2 by calling the Python method addParam to set parameters and similarly to define pre- and post-conditions. Parameters are typed, using a primitive (e.g., str) and WM element types (e.g., Element("concept")), and can be *optional* or *inferred* from the world model.

Pre-conditions allow SkiROS2 to check requirements for skill execution, and to automatically infer skill parameters. For example, in the pick skill shown in Listing 3, the parameter "Object" in line 10 is Required, i.e., it must be set before skill execution. At execution time, SkiROS2 infers the parameter "container" by reasoning about the pre-condition rule "ObjectLocationContainObject" (line 13). If "Object" is semantically not at a location in the WM, the pre-conditions are not satisfiable: the skill cannot be executed.

6 Concise and Verifiable Robot Skill Interface EzSkiROS

We have validated our design patterns in an internal DSL EzSkiROS, which adds **Early Dynamic Checking** (Section 4.2) to Skill Descriptions. Following a user-centered design methodology, we developed EzSkiROS by first identifying needs

```
24 class Pick(SkillDescription):
25 def description(self,
26       Robot: INFERRED[cora.Robot],
27       Arm: INFERRED[rparts.ArmDevice],
28       StartPose: INFERRED[skiros.TransformationPose],
29       GraspPose: INFERRED[skiros.GraspingPose],
30       ApproachPose: INFERRED[skiros.ApproachPose],
31       Workstation: INFERRED[scalable.Workstation],
32       ObjectLocation: INFERRED[skiros.Location],
33       Object: skiros.Product,
34       Gripper: rparts.GripperEffector):
35
36   self.pre_conditions += ObjectLocation.contain(Object)
37   self.pre_conditions += Gripper.at(StartPose)
38   self.pre_conditions += ~ Gripper.contain(Object)
39   self.pre_conditions += Object.hasA(ApproachPose)
40   self.pre_conditions += Object.hasA(GraspPose)
41   self.pre_conditions += Robot.at(Workstation)
42   self.pre_conditions += Workstation.contain(ObjectLocation)
43   self.post_conditions += ~ Gripper.at(StartPose)
44   self.post_conditions += Gripper.at(GraspPose)
45   self.post_conditions += ~ ObjectLocation.contain(Object)
46   self.post_conditions += Gripper.contain(Object)
47
```

Listing 4: The skill description of the pick skill shown in Listing 3 with EzSkiROS. We respresent OWL classes in Python as identifiers in type declarations.

for early bug checking via semi-structured interviews with skilled roboticists who use SkiROS2, reviewed documentation, and manual code inspection. We found that even expert skill developers made errors in writing Skill Descriptions, and that Python's dynamic typing only identified bugs when they triggered faults during robot execution.

We designed EzSkiROS to simplify how Skill Descriptions are specified, with the intent to increase their readability, maintainability, and writability. We map ontology objects and relations into Python's type system. Skill Descriptions can then directly include ontology information in type annotations. Listing 5 illustrates the EzSkiROS syntax on the example of the pick skill from Listing 3. The EzSkiROS variant avoids several redundant syntactic elements and specifies type information through type annotations instead of string encodings.

6.1 EzSkiROS implementation

We follow *owlready2*'s approach to **Domain Language Mapping** in exposing the ontology as Python types and objects. For instance in Listing 5, line 3 describes a parameter Robot with the type annotation INFERRED[cora.Robot]. Here, cora .Robot is a Python class that we dynamically generate to mirror an OWL class 'Robot' in the OWL namespace 'cora'. INFERRED is a parametric type that tags

```
<owl:ObjectProperty rdf:about="http://rvmi.aau.dk/ontologies/skiros.#hasA">
    <rdfs:subPropertyOf rdf:resource="http://rvmi.aau.dk/ontologies/skiros.#
     spatiallyRelated"/>
    <rdfs:range rdf:resource="#TransformationPose"/>
    <rdfs:domain rdf:resource="#Product"/>
</owl:ObjectProperty>
```

Listing 5: The definition of the object property 'hasA' in the SkiROS ontology.

inferred parameters. We mark *optional* parameters analogously as OPTIONAL; all other parameters are *required*. At robot launch time, we use Python's reflection facilities to extract and check this parameter information, both to link with SkiROS2' skill manager and for part of our **Early Dynamic Checking**.

For additional checking, we utilise **Symbolic Tracing** as described in Section 4.2, deferring Python's own language semantics to identify any mistyped names in the skill conditions. This step collects all pre-, post-, and hold conditions via the overloaded Python operator '+=' (lines 13–23). We then check for ontology type errors among these conditions.

6.2 Validation

We validate our DSL implementation by integrating it with SkiROS2 to see how it behaves with a real skill running on a robot[4]. To demonstrate the effectiveness of EzSkiROS, we use a 'pick' skill written in EzSkiROS (Listing 5) and load it while launching a simulation of a robot shown in Figure 1.

Listing 6 shows that the *ObjectProperty* 'hasA' is a relation allowed only between a 'Product' and a 'TransformationPose'. If we introduce a nonsensical relation like Object.hasA(Gripper), then the early dynamic check in EzSkiROS over ontology types returns a type error:

```
TypeError: Gripper: <class 'ezskiros.param_type_system.rparts.GripperEffector
    '> is not a (skiros.TransformationPose | skiros.TransformationPose)
```

7 Evaluation

To evaluate the effectiveness and usability of the Domain Specific Language (DSL) in detecting bugs at launch time, we conducted a user study with robotics experts. Seven robotic skill developers participated in our user study, including one member of the SkiROS2 development team. The user study consisted of three phases: an initial demonstration, a follow-up discussion, and a feedback survey[5]. Due to

[4] Available online in https://github.com/lu-cs-sde/EzSkiROS

[5] A replication of the surveyhttps://github.com/lu-cs-sde/EzSkiROS

time limitations, we defer a detailed study, with exercises for users to write new skills in EzSkiROS, to the future.

To showcase the embedded DSL and the early bug checking capabilities of EzSkiROS, we presented a video showing (1) a contrast between the old and new skill description written in EzSkiROS and (2) demonstrating how errors in the skill description are detected early at launch time by intentionally introducing an error in the skill conditions.

During the follow-up discussion, we encouraged participants to ask any questions or clarify any confusion they had about the EzSkiROS demonstration video.

After the discussion, we invited the participants to complete a survey to evaluate the readability and effectiveness of the early ontology type checks implemented in EzSkiROS. The survey included Likert-scale questions about *readability*, *modifiability*, and *writability*. Six participants answered 'strongly agree' that EzSkiROS improved readability, and one answered 'somewhat disagree'. For modifiability, four of them 'strongly agree' but three participants answered 'somewhat agree' and 'neutral'. All the participants answered 'strongly agree' or 'somewhat agree' that EzSkiROS improved writability.

To gain more in-depth insights, the survey also included open-ended questions, e.g.: (a) "Would EzSkiROS have been beneficial to you, and why or why not?", (b) "What potential benefits or concerns do you see in adopting EzSkiROS in your work?", and (c) "What potential benefits or concerns do you see in beginners, such as new employees or M.Sc. students doing project work, adopting EzSkiROS?".

For question (a), all participants agreed that EzSkiROS would have helped them. Participants liked the syntax of EzSkiROS, they thought that it takes less time to read and understand the ontology relations than before. One of them claimed that "pre- and post- conditions are easy to make sense". They also found that mapping the ontology to Python types would have helped reduce the number of lookups required in the ontology. One of the participants said, "in my experience, SkiROS2 error messages are terrible, and half the time they are not even the correct error messages (i.e. they do not point me to the correct cause), so I think the improved error reporting would have been extremely useful.".

For question (b), the majority of participants reported that EzSkiROS's concise syntax is a potential benefit, which they believe would save coding time and effort. One participant found EzSkiROS's specific error messages useful, responding that "the extra checks allow to know some errors before the robot is started" while one participant answered that EzSkiROS does not benefit their current work but it might be useful for writing a new skill from scratch. None of the participants expressed any concerns about adopting EzSkiROS in their work.

For question (c), one developer acknowledges the benefits of EzSkiROS by saying "In addition to the error reporting, it seems much easier for a beginner to learn this syntax, particularly because it looks more like "standard" object oriented programming (OOP)". One person claimed that EzSkiROS would help beginners, describing SkiROS2 as "it is quite a learning curve and needs some courage to

start learning SkiROS2 from the beginning autonomously".

In summary, the results of the user evaluation survey indicate a positive perception of EzSkiROS in terms of readability and writability. Most respondents found EzSkiROS to be easy to read and understand, with only one exception. In addition, respondents found EzSkiROS's early error checking to be particularly useful in detecting and resolving errors in a timely manner. This suggests that EzSkiROS is an effective tool for improving code quality and productivity.

8 Conclusion

Our work demonstrates how embedded DSLs can help robotics developers detect bugs early, even when the analysis depends on data that is not available until runtime. Our evaluation with EzSkiROS further suggests that embedded DSLs can achieve this goal while simultaneously increasing code maintainability. In the future, we plan to do a detailed user study where the users write the skill descriptions in EzSkiROSthemselves. We also plan to apply the DSL patterns explained in this paper to enable early bug checking in other areas of robot software development, such as compound skill construction with behaviour trees or safety monitoring, without requiring developers to move from their main development language to an external specification language.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[BR20] Hampus Balldin and Christoph Reichenbach. "A domain-specific language for filtering in application-level gateways". In: *GPCE 2020*. 2020, pp. 111–123.

[Bøg+12] Simon Bøgh et al. "Does your robot have skills?" In: *Proceedings of the 43rd international symposium on robotics*. VDE Verlag GMBH. 2012.

[Bru+07] Davide Brugali et al. "Trends in robot software domain engineering". In: *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 3–8.

[Buc+14] Jacob Pørksen Buch et al. “Applying Simulation and a Domain-Specific Language for an Adaptive Action Library”. In: *Simulation, Modeling, and Programming for Autonomous Robots*. 2014, pp. 86–97.

[Ceh+11] Ines Ceh et al. “Ontology driven development of domain-specific languages”. In: *Computer Science and Information Systems* 8.2 (2011), pp. 317–342.

[CÖ18] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[CMT97] Eve Coste-Maniere and Nicolas Turro. “The maestro language and its environment: Specification, validation and control of robotic missions”. In: *RSJ International Conf. on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS*. Vol. 2. IEEE. 1997.

[Dra+21] Swaib Dragule et al. “Languages for specifying missions of robotic applications”. In: *Software Engineering for Robotics*. Springer, 2021, pp. 377–411.

[Kin76] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[Kru+16] Volker Krueger et al. “A Vertical and Cyber–Physical Integration of Cognitive Robots in Manufacturing”. In: *Proceedings of the IEEE* 104.5 (2016), pp. 1114–1127.

[KRB11] Lars Kunze, Tobias Roehm, and Michael Beetz. “Towards semantic robot description languages”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011.

[Lam17] Jean-Baptiste Lamy. “Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies”. In: *Artificial intelligence in medicine* 80 (2017), pp. 11–28.

[May+22a] Matthias Mayr et al. “Combining Planning, Reasoning and Reinforcement Learning to solve Industrial Robot Tasks”. In: *arXiv preprint arXiv:2212.03570* (2022).

[May+22b] Matthias Mayr et al. “Learning Skill-based Industrial Robot Tasks with User Priors”. In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. 2022, pp. 1485–1492.

[May+22c] Matthias Mayr et al. “Skill-based Multi-objective Reinforcement Learning of Industrial Robot Tasks with Planning and Knowledge Integration”. In: *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2022.

[Mog+13] Mikael Moghadam et al. “Towards python-based domain-specific languages for self-reconfigurable modular robotics research”. In: *arXiv preprint arXiv:1302.5521* (2013).

[Nor+16] Arne Nordmann et al. “A Survey on Domain-Specific Modeling and Languages in Robotics”. In: *Journal of Software Engineering in Robotics (JOSER)* 7.1 (2016), pp. 75–99.

[Ped+16] Mikkel Rath Pedersen et al. “Robot skills for manufacturing: From concept to industrial deployment”. In: *Robotics and Computer-Integrated Manufacturing* 37 (2016), pp. 282–291.

[Qui+09] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[Rei21] Christoph Reichenbach. “Software ticks need no specifications”. In: *ICSE-NIER 2021*. IEEE. 2021, pp. 61–65.

[RGK17] Francesco Rovida, Bjarne Grossmann, and Volker Krüger. “Extended behavior trees for quick definition of flexible robotic tasks”. In: *RSJ International Conf. on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 6793–6800.

[Rov+17] Francesco Rovida et al. “SkiROS— a skill-based robot control platform on top of ROS”. In: *Robot Operating System (ROS)*. Springer, 2017, pp. 121–160.

[Spi01] Diomidis Spinellis. “Notable design patterns for domain-specific languages”. In: *Journal of systems and software* 56.1 (2001), pp. 91–99.

[Wut+21] D. Wuthier et al. “Productive Multitasking for Industrial Robots”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 12654–12661.

EZSKIROS: ENHANCING ROBOT SKILL COMPOSITION WITH EMBEDDED DSL FOR EARLY ERROR DETECTION

1 Abstract

When developing general-purpose robot software components, we often lack complete knowledge of the specific contexts in which they will be executed. This limits our ability to make predictions, including our ability to detect program bugs statically. Since running a robot is an expensive task, finding errors at runtime can prolong the debugging loop or even cause safety hazards. In this paper, we propose an approach to help developers catch these errors as soon as we have some context (typically at pre-launch time) with minimal additional effort. We use embedded DSL techniques to enforce early checks. We describe design patterns suitable for robot programming and show how to use these design patterns for DSL embedding in Python, using two case studies on an open-source robot skill platform SkiROS2, designed for the composition of robot skills. These two case studies help us understand how to use DSL embedding on two abstraction levels: the high-level skill description that focuses on what the robot can do and

Momina Rizwan, Christoph Reichenbach, Ricardo Caldas, Matthias Mayr, and Volker Krueger. "EzSkiROS: Enhancing Robot Skill Composition with Embedded DSL for Early Error Detection". **"Submitted for publication"** to the Special Issue "Robotics Software Engineering" as a Journal in Frontiers in Robotics and AI, section Computational Intelligence in Robotics.

under what circumstances, and the lower-level decision making and execution flow of tasks. Using our DSL EzSkiROS, we show how our design patterns enable robotics software platforms to detect bugs in the high-level contracts between the robot's capabilities and the robot's understanding of the world. We also apply the same techniques to detect bugs in the lower-level implementation code, such as writing behavior trees to control the robot's behavior based on its capabilities. We perform consistency checks during the code deployment phase, significantly earlier than the typical runtime checks. This enhances overall safety by identifying potential issues with the skill execution before they can impact robot behavior. An initial study with SkiROS2 developers shows that our DSL-based approach is useful for finding bugs early and thus improving the maintainability of code.

2 Introduction

The design and implementation of robotic systems to perform socio-technical missions has never been more relevant or challenging. To ensure that robot developers can meet market demands with confidence in the correctness of their systems, a range of development tools and techniques is required. Specifically, robot development tools should provide expressive programming languages and frameworks that allow human developers to describe correct robot behavior [Bru+07]. One such robot development platform is *SkiROS2*[1], a skill-based robot control platform with knowledge integration. *SkiROS2* [MRK23] allows developers to define modular skills for autonomous mission execution.

These skills, ranging from “pick” to “drive”, are modularly defined with pre- and post-conditions. In SkiROS2, the assessment and validation of these conditions rely on the robot's knowledge, systematically organized into an ontology. These ontologies are a rich, interlinked representation of concepts and relationships within a specific domain. It serves as a foundation for verifying that all necessary conditions for skill execution are satisfied. For instance, in an automated assembly line or robotic healthcare surgery, the ontology would encompass all relevant entities and their relationships, providing a comprehensive context for skill execution.

Consider the "pick" skill as an example. The pre-conditions might include ontology-based relationships like *"gripper is part of the robot arm"*. This relationship assists in deducing additional parameters such as *"which arm to move"* by employing subtle semantic differences of entities and their relationships in the ontology. For example, if we say the gripper is part of the arm then we know which arm to move if we want to pick an object with the gripper. The distinction between relationships like *"is part of"* and *is holding* is critical in ensuring the correct application of parameters and actions during skill execution.

[1] `https://github.com/RVMI/skiros2`

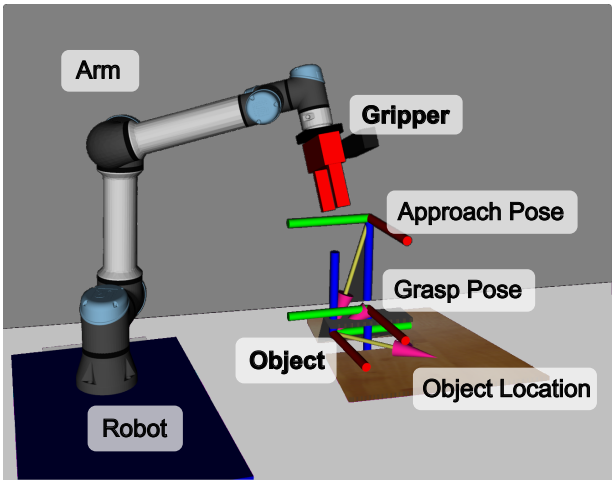


Figure 1: The robot using a pick skill with a visualization of the necessary parameters. To run this skill, we only need the Gripper and the Object parameters. SkiROS2 can deduce all other necessary parameters through a set of rules in the skill description shown in Listings 3 and 5.

The developer must be careful when declaring such relationships, as bugs introduced at this stage can lead to silent errors, disrupting the skill's behavior and potentially leading to incorrect or inefficient task execution. The reason is that some of these errors in skill description are logical errors that would not manifest as explicit runtime errors. Certain errors may only become evident when a particular skill is executed, which could be weeks later when demonstrating the robot under specific circumstances that are not immediately predictable. Such delayed detection makes troubleshooting and rectifying these errors more challenging. Therefore, properly defining relationships and conditions within the ontology and skill descriptions is crucial for technical correctness and ensuring operational reliability robotic skills in real-world applications.

In SkiROS2, each high-level skill description acts as a behavioral contract, setting parameters and conditions that the corresponding implementations must satisfy. These descriptions guide the development of concrete skill implementations. Many implementations use extended BTs that reuse other existing skills, relying on their pre-conditions and post-conditions for a structured execution. Extended BTs in SkiROS2 merge task-level planning and execution, allowing for modularity and reactivity [RGK17]. The reactivity comes inherently from BTs in the way with which tasks are organized in a BT defines their priority order, where more important tasks interrupt less important ones [Iov+22]. However, constructing consistent and correct BTs is crucial, as inconsistencies can lead to unexpected failures and outcomes.

To avoid such errors, we propose using a Domain-Specific Language (DSL) to allow for analyzing the code for potential errors before deploying it on the actual robot. Our proposed approach ensures that the high-level abstract skill descriptions align with the lower-level BTs, providing a comprehensive framework for skill execution. DSLs offer specific constructs for defining and connecting nodes, conditions, and actions, enforcing correct patterns and practices, thus reducing the likelihood of logical or structural errors. The benefits of using DSLs to aid debugging, visualization, and static checking are well-recognized, making them a valuable tool in robot software development. DSLs have been used for mission specification [Dra+21], and robot knowledge modeling [Ceh+11]. Nordmann et al. [Nor+16] collect and categorise over 100 such DSLs for robotics in their *Robotics DSL Zoo*[2].

In this paper, we aim to support robot developers, particularly those who write control logic in Python, to catch bugs early by embedding DSLs directly in Python. We support our case through:

- Four design patterns for *embedding DSLs in general-purpose programming languages* that address common challenges in robotics, with details on how to implement these patterns in Python;
- A case study of a robotics software SkiROS2 , in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs, highlighting its effectiveness in identifying errors in both high-level skill descriptions and lower-level implementation details.
- A demonstration of how EzSkiROS detects various types of bugs in robot capabilities, world model contracts, and behavior trees, showcasing the DSL's comprehensive coverage and versatility in detecting bugs early.

Lastly, we discuss the advancements and distinctions of our approach compared to the initial insights presented in the paper [Riz+23], providing an overview of the evolution and impact of our design patterns.

3 Related Work

Several studies have explored the use of model-driven approaches for programming robots, focusing on the development of DSLs to enhance the reliability of robotic systems. [Buc+14] describe an internal Domain-Specific Language (DSL) over C++, that incorporates structuring of complex actions, where actions are modeled through sets of parameters, and each action contains a pre-condition specifying the state of relevant parts. This structure implies the use of pre- and post-conditions in sequencing robotic skills.. Unlike our DSL, their DSL uses a model-driven approach. It instantiates a textual representation of the assembly sequence,

[2] https://corlab.github.io/dslzoo

which is interpreted to execute the assembling behavior. However, it is unclear if they use early checking techniques to prevent erroneous sequences. While it discusses error handling and the probabilistic approach to tackle uncertainties, specific methods like early checking techniques are not clearly outlined.

[KRB11] propose the Semantic Robotic Description Language (SRDL), a model-based approach that utilizes Web Ontology Language (OWL) notation to match robot descriptions and actions through static analysis of robot capability dependencies. SRDL models knowledge about robots, capabilities, and actions, contributing to the understanding and specification of robotic behaviors. However, the extent to which SRDL supports early dynamic checking in general-purpose languages remains unclear, highlighting the need for further exploration in this area.

[CMT97] propose MAESTRO, an external DSL for specifying reactive behavior and checking in the robotics domain. MAESTRO focuses on complex and hierarchical missions, accommodating concurrency and portability requirements. It allows the specification of user-defined typed events and conditions, offering type-checking of user-defined types and stop condition checks to ensure the correctness and safety of specified behaviors.

Behavior trees BTs have emerged as an effective method for modeling and executing autonomous behaviors of robots, particularly in dynamic environments. Unlike the traditional Finite State Machines (FSMs), BTs represent action selection decisions in a hierarchical tree structure enhancing flexibility in planning and replanning robot behavior. As [DP22] highlight, BTs offer a more maintainable approach to decision-making than FSMs, which is crucial in the rapidly evolving field of robotics. Originally developed for the video game industry, BTs have been widely adopted in robotics due to their modularity and scalability. [Iov+22] presents a detailed survey of BTs in robotics and AI, discussing their application, evolution, and the benefits. BTs are composed of various types of nodes, including control nodes (e.g., sequences, selectors), leaf nodes (e.g., tasks, conditions), and decorator nodes (modifying the behavior or output of other nodes), organized in a tree structure from a root node and branching out.

The integration of BTs with robotic systems often involves the use of DSLs and frameworks like the Robot Operating System (ROS). [Ghz+23] emphasize the growing use of BTs in open-source robotic applications supported by ROS, indicating their practicality in the real-world applications. However, verifying the safety and correctness of BTs remains a challenge.

[Hen+22] use SMTs to check safety properties specified in Linear Constraint Horn Clauses notation over Behavior Tree specifications. Moreover, [TT22] use Event-B for formal specification and verification of the BT instances, ensuring the maintenance of invariant properties.

From a static semantics perspective, BhTSL is an example where the compiler checks the source text for non-declared variables and variable redeclaration [Oli+20]. Despite the advancements in BT DSLs, there is a lack of DSLs performing static checks as rigorously as desired. According to the survey pa-

per [Ghz+20], most used behavior tree DSLs, such as BehaviorTree.CPP [3], py_trees [4], and the Behavior Tree from UnrealEngine [5], primarily focus on runtime type safety and flexibility. For instance, MOOD2Be's[6] project from Horizon 2020, the BehaviorTree.CPP tool offers a C++ implementation of BTs with type safety [Fac19], but the type-checking capability is largely left to the developer and is subject to runtime checks. This indicates a gap in the domain of DSLs for BTs in ensuring consistency and preventing inconsistencies in implementation between skills or actions before runtime.

In conclusion, while there have been significant advancements in DSLs for robotics and BTs, there is a continuous need for the development of languages and tools that allow both static and early dynamic checking to ensure the safety, reliability, and efficiency of robotic systems. Future research should focus on enhancing the capabilities of DSLs to perform comprehensive checks and verifications, both at design time and runtime, to address the increasing complexity and demands of modern robotic applications.

4 Embedding Robotics DSLs in Python

Domain-Specific Languages (DSLs) can help developers by simplifying notation, improving performance or error detection. However, developing and maintaining DSLs requires effort. For *external DSLs* (e.g., MAESTRO, SRDL), much of this effort comes from building a language frontend. *Internal* or *embedded DSLs* (as in [Buc+14]) avoid this overhead, and instead re-use an existing "host" language, possibly adjusting the language's behavior to accommodate the needs of the problem domain.

We look at Python as one of the three mainly supported languages of the popular robotics platform ROS (cf. [Qui+09]). The other two languages, C++ and LISP, also support internal DSLs, but with different trade-offs.

4.1 Python Language Features for DSLs

While Python's syntax is fixed, it offers several language constructs that DSL designers can repurpose to reflect their domain, such as freely overloadable infix operators (excluding the type-restricted boolean operators), type annotations (since Python 3.0), and decorator mechanisms (cf. [Mog+13]).

Listing 1 illustrates some of these techniques. Class `A` represents a *deferred* operation `op` with parameters `args`. `A.eval` (Line 18) forces recursive evaluation. The `@staticmethod` decorator tells Python that this method takes no implicit `self`

[3] https://github.com/BehaviorTree/BehaviorTree.CPP
[4] https://github.com/splintered-reality/py_trees
[5] https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees
[6] https://robmosys.eu/mood2be/

```
1  class M(type):
2    def __getattribute__(self, name):
3      if name[0] == '_' and name[1:].isdecimal():
4        return self(lambda x: x, int(name[1:]))
5      return type.__getattribute__(self, name)
6
7  class A(metaclass=M):
8    def __init__(self, op, *args):
9      self.op = op
10     self.args = args
11   def __add__(self, other):
12     return A(int.__add__, self, other)
13   def __call__(self, arg : int):
14     return A(int.__mul__, self, arg)
15   @staticmethod
16   def eval(e):
17     if isinstance(e, A):
18       return e.op(*(A.eval(y) for y in e.args))
19     return e
20
21 a = A._2 + 1  # a = A.__add__(A(λ x:x, 2), 1)
22 b = a(4)      # b = a.__call__(4)
23 print(A.eval(b))  # evaluates (2 + 1) * 4
```

Listing 1: An example with DSL-friendly Python features: Line 15 shows a decorator, metaclass is shown in lines 1–5, overloading in lines 11 and 13 and finally type annotations in line 13.

parameter. DSL designers can define other decorators to transform the semantics of functions, methods, or classes.

The *metaclass* M in line 1 allows class A (line 7) to handle references to unknown class attributes. The code in lines 2–5 checks for attributes that start with an underscore and continue in decimal digits. Line 21 shows how we can write A._2 to construct an instance of A (via line 4).

Class A overloads infix addition in line 11 and function call notation in line 13, which allows instances of A to participate in addition and to behave like callable functions (lines 21 and 22). While the code is a toy example, it illustrates how a DSL designer can construct in-memory representations of complex computations for *staging*, which could (e.g., in A.eval) perform optimisations or translate the code representation into a more efficient format (e.g., for a GPU).

Line 13 illustrates Python's type annotations, annotating parameter arg with type int. By default, such annotations have no runtime effect, but DSL designers can access and repurpose them to collect DSL-specific information without interference from Python. Since Python 3.5 (with extensions in 3.9), these annotations also allow type parameters (e.g., x : list [int]).

Finally, Python permits dynamic construction of classes (and metaclasses), which we have found particularly valuable for the robotics domain: since the system configuration and world model used in robotics are often specified outside of Python (e.g., in configuration files or ontologies) but are critical to program logic, we can map them to suitable type hierarchies at robot pre-launch time (just after build time).

4.2 Robotics DSL Design Patterns

In the following, we list our DSL design patterns with a brief summary block that highlights each pattern's purpose and key implementation concepts. For each, we show examples and discuss trade-offs, as appropriate.

Domain Language Mapping

Pattern:	**Domain Language Mapping**
Purpose:	Make domain notation visible in host language, reduce notational overhead
Implementation:	See the "Piggyback" DSL implementation pattern documented by [Spi01]

Domain Language Mapping identifies language concepts in the host language that correspond to the domain language in some sense, and then uses the techniques described in [Spi01] to implement them. This mapping can be manual, or be the result of reflection.

As an example, the Web Ontology Language (OWL) allows us to express the relationships and attributes of the objects in the world, the robot hardware and the robot's available capabilities (skills and primitives). Existing libraries like *owlready2* (cf. [Lam17]) already expose these specifications as Python objects, so if the ontology contains a class pkg:Robot, we can create a new "Robot" object by writing

```
r = pkg.Robot("MyRobotName")
```

and iterate over all known robots by writing

```
for robot in pkg.Robot.instances(): ...
```

While [Mog+13] expressed concerns about "syntactic noise" for DSL embedding in earlier versions of Python, when compared to external DSLs, we found such noise to be modest in modern Python, and instead emphasise the advantages of embedding in a language that is already integrated into the ROS environment and that developers are familiar with.

Maintenance and Integration Considerations When domain knowledge is available in machine-readable form, much or all of the mapping process may be automable. For example, the *owlready2* library creates these classes at runtime, based on the contents of the ontology specification files. Thus, changes in the

ontology are immediately reflected in Python: if we rename pkg:Robot in the ontology, our earlier code example will trigger an error when it encounters pkg.Robot in the Python source code.

Another strategy for automating the mapping process is to generate code in the host language. In our example, this code would take the form of Python modules, such as pkg.py, that contain classes and methods to reflect the mapping (e.g., a class Robot). This strategy mirrors DSL implementation strategies for host languages that lack advanced reflection facilities, such as C (cf. [LMB92]).

Code generation has two potential disadvantages over reflection. First, code generation persists a *snapshot* of the domain language mapping. The build and development process must thus ensure that this snapshot is kept fresh, and prevent developers from accidentally modifying the generated code. Second, code generation requires the domain language mapping to take place *before* build time. When domain knowledge is only available at pre-launch time, the generated code will necessarily be stale, which may render this implementation strategy useless.

In our discussions with practitioners, we did however observe a key advantage that code generation offers: Since the mapping becomes visible as Python source code, it is also available to language servers and integrated development environments, and may help developers find bugs in their code even earlier.

Early Dynamic Checking

Pattern:	**Early Dynamic Checking**
Purpose:	Detect type and configuration errors in a critical piece of code early, such as during robot pre-launch time, with no or minimal extra effort for developers.
Implementation:	Execute all critical pieces of code early, while redefining the semantics of the predetermined set of operations (e.g. ontology relations from our previous example) to immediately return or to only perform checking

In tools like SkiROS2, combining Python code, ontologies and configuration files at runtime introduces points of failure. To detect such failures early, we propose a second pattern. The *conditions* for this pattern are:

- We can collect all critical pieces of code at a suitably early point during execution
- The critical code does not depend on return values of operations that we cannot predict at pre-launch time

In Python, configuration and type errors only trigger software faults once we run code that depends on faulty data. In robotics, we might find such code in operations that (a) run comparatively late (e.g., several minutes after the start of the robot) and (b) are difficult to unit-test (e.g., due to their coupling to specific ROS

```
1 def expand(self, skill):
2   skill.setProcessor(SerialStar())
3   skill(
4     self.skill("Navigate", ""),
5     self.skill("WmSetRelation", "wm_set_relation",
6       remap={'Dst': 'TargetLocation'},
7         specify={'Src': self.params["Robot"].value,
8         'Relation': 'skiros:at', 'RelationState': True}),)
```

Listing 2: Constructing the behavior tree (BT) of a drive skill in SkiROS2. It is a sequential execution of a compound skill (a skill with its own BT of smaller, executable skills) "Navigate" and a primitive skill (an atomic skill that can not be broken down into smaller parts) to update the world model "WmSetRelation".

functionality and/or robotics hardware). For robotics developers, both challenges increase the cost of verification and validation [Rei21]: a fault might trigger only after a lengthy robot program and require substantial manual effort to reproduce. For example, a software module for controlling an arm might take a configuration parameter that describes the target arm pose. If the arm controller is triggered late (e.g., because the arm is part of a mobile platform that must first reach its goal position), any typos in the arm pose will also trigger the fault late. If the pose description comes from a configuration file or ontology, traditional static checkers will also be ineffective. We can only check for such bugs after we have loaded all relevant configuration.

Through careful software design, developers can work around this problem, e.g., by checking that code and configuration are well-formed as soon as possible, before they run the control logic. If the critical code itself is free of external side effects, the check can be as simple as running the critical code twice. For example, SkiROS2 composes *BTs* [CÖ18] within such critical Python code (Listing 2): composing (as opposed to running) these objects has no side effects, so we can safely construct them early to detect simple errors (e.g., typos in parameter names). This is a typical example that eludes static checking but is amenable to Early Dynamic Checking: line 7 depends on self.params["Robot"].value, which is a configuration parameter that we cannot access until the robot is ready to launch. Not all robotics code is similarly declarative. Consider the following example, in a hypothetical robotics framework in which all operations are subclasses of RobotOp and must provide a method run() that takes no extra parameters:

```
1 class MyRobotOp(RobotOp):
2   def __init__(self, config): # Configure
3     self.config = config
4   def check(self):  # Check configuration
5     assert self.config.mode in ["A", "B"]
6     assert isinstance(self.config.v, int)
```

```
7   def run(self):    # Run with configuration
8     if self.config.mode == "A":
9       self.runA();
10    elif self.config.mode == "B":
11      self.runB(self.config.v + 10);
12    else:
13      fail()
```

Here, the developers introduced a separate method check() that can perform early checking during robot initialisation or pre-launch. However, check() and run() both have to be maintained to make the same assumptions.

The Early Dynamic Checking pattern instead uses internal DSL techniques to allow developers to use the same code in two different ways: (a) for checking, and (b) for logic.

In our example, calling run() "normally" captures case (b). For case (a), we can also call run(), but instead of passing an instance of MyRobotOp, we pass a *mock* instance of the same class, in which operations like runA() immediately return:

```
1 class MyRobotOpMock:
2   def __init__(self, parent):
3     self.parent = parent
4   @property
5   def config(self):
6     # self.config  = self.parent.config
7     return self.parent.config
8   def runA(self):
9     pass # mock operation: do nothing
10  def runB(self, arg):
11    pass # mock operaiton: do nothing
```

If we execute MyRobotOpMock.run() with the same configuration as MyRobotOp , run() will execute almost as for MyRobotOp but immediately return from any call to runA or runB. If the configuration is invalid, e.g., if config.mode == "C" or config.v == false, running MyRobotOpMock.run() will trigger the error early.

Since Python can reflect on a class or an object to identify all fields and methods, we can construct classes like MyRobotOpMock at run-time: instead of writing them by hand, we can implement a general-purpose mock class generator that constructs methods like runA and accessors like config automatically. If the configuration objects may themselves trigger side effects, we can apply the same technique to them.

However, the above implementation strategy is only effective if we know that the critical code will only call methods on self and other Python objects that we know about ahead of time. We can relax this requirement by controlling how Python resolves nonlocal names:[7]

```
FunctionType(MyRobotOp.run.__code__, globals() | { 'print' : g})(obj)
```

[7]Python's *eval* function offers similar capabilities, but as of Python 3.10 does not seem to allow passing parameters to code objects.

This code will execute `obj.run()` via the equivalent `MyRobotOp.run(obj)`, but replace all calls to `print` by calls to some function `g`. The same technique can use a custom map-like object to detect at runtime which operations the body of the method wants to call and handle them suitably.

However, the more general-purpose we want to allow the critical code to be, the more challenging it becomes to apply this pattern. For instance, if the critical code can get stuck in an infinite loop, so may the check; if this is a concern, the check runner may need to use a heuristic timeout mechanism. A more significant limitation is that we may not in general know what our mocked operations like `runA()` should return, if anything. If the critical code depends on a return value (e.g., if it reads ROS messages), the mocked code must be able to provide suitable answers. The same limitation arises when the critical code is in a method that takes parameters. If we know the type of the parameter or return value, e.g. through a type annotation, we can exploit this information to repeatedly check (i.e., *fuzz-test*) the critical code with different values; however, without further cooperation from developers, this method can quickly become computationally prohibitive.

If we know that the code in question has a simple control flow, we may be able to apply the next pattern, Symbolic Tracing.

Symbolic Tracing

Pattern:	**Symbolic Tracing**
Purpose:	Detect Bugs in a critical piece of code early, if that code depends on parameters or operation return values, with minimal extra effort for developers.

Implementation:

1. Execute the critical code while passing symbolic values as parameters and/or returning symbolic values from operations of relevance
2. Collect any constraints imposed by operations on the symbolic values
3. After executing the critical code, verify the constraints against the problem domain

Here, a *symbolic* value is a special kind of mock value that we use to record information (cf. [Kin76]).

The *conditions* for this pattern are that

- We can access and execute the critical code
- We have access to sufficient information (via type annotations, properties, . . .) to simulate parameter values and operation return values *symbolically* (see below)
- The number of control flow paths through the critical code is small (see below)

Consider the following RobotOp subclass:

```
class SetArmSpeedOp(RobotOp):
  def run(self, speedup):
    self.setArmSpeed(speedup)
    self.setArmSafety(speedup)
```

This class only calls two operations, but its run operation depends on a parameter speedup about which we know nothing a priori — thus, we cannot directly apply the Early Dynamic Checking pattern.

In cases where we lack prior knowledge about an operation, it may still be possible to obtain useful insights about it. For example, if we are aware that setArmSpeed accepts only numeric parameters and setArmSafety only accepts boolean parameters, we can flag this code as having a type error. To avoid blindly testing various parameters, we can pass a symbolic parameter to the run function and employ a modified version of the mock-execution strategy used in Early Dynamic Checking. The mock objects can be adapted as follows:

```
TYPE_CONSTRAINTS = []

class SetArmSpeedOpMock:
  def setArmSpeed(self, obj):
    TYPE_CONSTRAINTS.append((obj, float))
  def setArmSafety(self, obj):
    TYPE_CONSTRAINTS.append((obj, bool))
```

We can now (1) create a fresh object obj and an SetArmSpeedOpMock instance that we call mock, (2) call SetArmSpeedOp.run(mock, obj), and (3) read out all constraints that we collected during this call from TYPE_CONSTRAINTS, and check them for consistency, which makes it easy to spot the bug. If the constraints come from accesses to obj (e.g., method calls like obj.__add__(1) that result from code like obj + 1), obj itself can collect the resultant constraints.

Depending on the problem domain, constraint solving can be arbitrarily complex, from simple type equality checks to automated satisfiability checking [BR20]. It can involve dependencies across different pieces of critical code (e.g., to check if all components agree on the types of messages sent across ROS channels, or to ensure that every message that is sent has at least one reader). However, this approach requires information about specific operations like setArmSpeed and setArmSafety, which can be provided to Python in a variety of ways, e.g., via type annotations.

As an example, consider an operation that picks up a coffee from the table with a gripper, where we annotate all parameters to run with OWL ontology types:

```
1 class PickCoffeeTableOp(RobotOp):
2   def run(self, robot : rob.Robot,
3           gripper : rob.Gripper,
4           coffee_table : world.Furniture):
5     // bug:
6     assert coffee_table.robotPartOf(robot);
7     ...
```

This example is derived from the SkiROS2 ontologies, with minor simplifications. In the above SkiROS2 code, the developer intended to write a pre-condition that to be able to pick a coffee cup, the robot should be close to the table. Instead, the developer mistakenly wrote that a robot should be a part of the coffee table.

The ontology requires that robotPartOf is a relation between a technical Device and a Robot. However, Furniture is not a subtype of Device, so the assertion in line 6 is unsatisfiable.

We can again detect this bug through symbolic tracing. This time we must construct symbolic variables for robot, gripper, and coffee_table that expose methods for all applicable relations, as described by their types. For instance, gripper will contain a method robotPartOf(gripper, obj) that records on each call that gripper and obj should be in a robotPartOf relation. Meanwhile, coffee_table will not have such an operation. When we execute run(), we can then defer to Python's own type analysis, which will abort execution and notify us that coffee_table lacks the requisite method.

Key to this symbolic tracing is our use of mock objects as symbolic variables. Symbolic variables reify Python variables to objects that can trace the operations that they interact with, in execution order, and translate them into constraints.

The main *limitation* of this technique stems from its interaction with Python's boolean values and control flow, e.g. conditionals and loops. Python does not allow the boolean operators to return symbolic values, but instead forces them (at the language level) to be bool values; similarly, conditionals and loops rely on access to boolean outcomes. Thus, when we execute code of the form `if x: ...`, we must decide right there and then if we should collapse the symbolic variable that x is bound to True or False. While we can re-run the critical code multiple times with different decisions per branch, the number of runs will in general be exponential over the number of times that a symbolic variable collapses to a bool.

Source Provenance Tracking

Pattern:	**Source Provenance Tracking**
Purpose:	Make early dynamic error reports more actionable by reporting relevant source locations
Implementation:	Dynamic stack inspection

The intent in early error detection in (embedded) DSLs is generally to prevent undesirable behavior. When this undesirable behavior is due to a problematic user

specification, it is– in our experience– valuable to point the user to the problematic specification. In practice, "blaming" the right part of the program can be nontrivial, since the disagreement may be across multiple user specifications ([Ahm+11] discuss this challenge in more detail).

Handling multiple conflicting constraints can be particularly challenging for embedded DSLs. Let's say that we are using a technique like *Symbolic Tracing* in two user-defined functions, declaration() and implementation(), such that implementation() must *ensure* the constraints that are *required* declaration():

```
def declaration(x : int):
  require(x > 0)
  require(x < 10)

def implementation(x : int):
  ensure(x > 3)
  ensure(x <= 10)

...
# more code follows
...
register_implementation(declaration, implementation)
```

In the above example, we might find a bug: implementation allows x = 10, but this is not allowed according to declaration(). A typical but naïve implementation of such a consistency check might simply inform the user that declaration and implementation disagree about what x is allowed to do, and raise an exception.

The programmer must now identify the line of code that is the culprit by hand. In practical scenarios such as our case studies, there may be multiple declaration and implementation functions in the same file (usually as methods), which further complicates the task.

Reflection can help us here: for example, given a function object in Python, we can use reflection to access `implementation.__code__.co_firstlineno` and `implementation.__code__.co_filename` to obtain the location at which the function was defined in the form of the first line of code and the source file name. For larger definitions even this information may be insufficiently precise.

Some languages offer facilities that allow us to obtain even the exact lines of code that were responsible for the error (lines 3 and 7, in our example). While some languages support this inspection through macro- or preprocessor facilities (e.g., `__LINE__` and `__FILE__` in C), Python 3.1 and later offer direct read access to the call stack via inspect.stack(). The symbolic tracing code for reqire() and ensure() can then "walk" this stack down until it finds the first stack frame that belongs to the code under analysis, and extract file name and line number from there. The symbolic tracer can then attach this *provenance* information to the constraint and expose it to the user if the constraint is contributing to some error report.

4.3 Alternative Techniques for Checking

Internal DSLs are not the only way to implement the kind of early checking that we describe. The *mypy* tool[8] is a stand-alone program for type-checking Python code. Mypy supports plugins that can describe custom typing rules, which we could use e.g. to check for ontology types. Similarly, we could use the Python `ast` module to implement our own analysis over Python source code. However, both approaches require separate passes and would first have to be integrated into the ROS launch process. Moreover, they are effectively static, in that they cannot communicate with the program under analysis; thus, we cannot guarantee that the checker tool will see the same configuration (e.g., ontology, world model).

Another alternative would be to implement static analysis over the bytecode returned by the Python disassembler `dis`, which can operate on the running program. However, this API is not stable across Python revisions[9].

An external DSL such as MAESTRO [CMT97] would similarly require a separate analysis pass. However, it would be able to offer arbitrary, domain-specific syntax and avoid any trade-offs induced by the embedding in Python (e.g., boolean coercions). The main downside of this technique is that it requires a completely separate DSL implementation, including maintenance and integration.

5 SkiROS2: An open source software for skill based robot execution

As a case study, we implement our patterns on an open-source software for skill-based robot execution SkiROS2 [MRK23]. SkiROS2 is used by several research institutions in the context of industrial robot tasks as demonstrated in [May+23; May+22a; May+22c; May+22b; AMK23; Wut+21]. It is a complete re-implementation of the predecessor SkiROS1by [Rov+17], and is implemented in Python on top of the ROS [Qui+09] middleware. SkiROS2 uses behavior trees (BTs) [CÖ18] formalism to represent procedures. The platform implements a layered, hybrid control architecture to define and execute parametric *skills* for robots [Bøg+12; Kru+16]. SkiROS2 system architecture is shown in (Fig. 2) which illustrates how different components interact with each other at different phases. It uses ontologies to represent the comprehensive knowledge about the world. As the figure shows, SkiROS2 represents knowledge about the skills, the robot and the environment in a WM with the *Ontologies* specified in OWL format. This explicit representation, built upon the World Wide Web Consortium's Resource Description Framework (RDF) [HKR09] standard, allows the use of existing ontologies. This approach to knowledge management is important for complex decision-making and reasoning in autonomous systems [CA22].

[8] `https://mypy-lang.org/`
[9] `https://docs.python.org/3/library/dis.html`

Central to SkiROS2's architecture is its world model, which serves as a dynamic repository of the robot's environment and state. This model continuously updates and maintains a semantic representation of the surroundings, objects, and the robot's own status. The integration of the world model with the ontology (as shown in Fig. 2 ensures that the robot has a thorough understanding of its operational context, enhancing its interaction capabilities with the environment.

Skills in SkiROS2 are parametric procedures that modify the world state from an initial state to a final state according to pre- and post-conditions [Ped+16]. Every skill has a *Skill Description* and one or more *Skill Implementation* as shown in Fig. 2. The *Skill Description* consists of four elements:

1. *Parameters* define input and output of a skill. The types of these parameters can vary from certain primitive data types to a world model element in the ontologies.
2. *Pre-conditions* must hold before the skill is executed
3. *Hold-conditions* must be fulfilled during the execution
4. *Post-conditions* indicates that a skill has successfully executed

These conditions are checked by the *Skill Manager* as shown in the Fig. 2. When a skill is invoked, the system first checks the pre-conditions to decide if it is safe or viable to start the skill. During execution, hold-conditions are continuously monitored to ensure ongoing criteria are met. Finally, once the skill reports its completion, post-conditions are checked to confirm successful execution. These checks are essential to maintain the robustness, safety, and reliability of robotic operations, ensuring that skills are only performed when appropriate and achieve the intended results.

The Listing 3 shows how developers define a 'pick' skill in SkiROS2 by calling the Python method addParam to set the parameters of the skill and similarly to define its pre- and post-conditions. The parameters are typed, using basic datatypes (e.g., str) or a WM element defined in ontology, and can be *required*, *optional* or *inferred* from the world model. Pre-conditions allow SkiROS2 to check requirements for skill execution, and to automatically infer skill parameters from the world model. For example, in the pick skill shown in Listing 3, the parameter "Object" in line 10 is Required, i.e., it must be set before the execution of the skill. At execution time, SkiROS2 infers the parameter "ObjectLocation" (line 9) by reasoning about the pre-condition rule "ObjectLocationContainObject" (line 13). If "Object" is semantically not at a location in the WM, the pre-conditions are not satisfiable and the skill cannot be executed.

A *Skill Implementation*, on the other hand acts as a class that implements the interface *Skill Description* and refers to the actual coding and logic that enables a robot to perform a task. Skills can be either primitive or compound skills. Depending on the type of skill, primitive skills implement atomic functions that change

```
class Pick(SkillDescription):
  def createDescription(self):
    self.addParam("Robot", Element("cora:Robot"), ParamTypes.
    Inferred)
    self.addParam("Arm", Element("rparts:ArmDevice"), ParamTypes.
    Inferred)
    self.addParam("StartPose", Element("skiros:TransformationPose
    "), ParamTypes.Inferred)
    self.addParam("GraspPose", Element("skiros:GraspingPose"),
    ParamTypes.Inferred)
    self.addParam("ApproachPose", Element("skiros:ApproachPose"),
     ParamTypes.Inferred)
    self.addParam("Workstation", Element("scalable:Workstation"),
     ParamTypes.Inferred)
    self.addParam("ObjectLocation", Element("skiros:Location"),
    ParamTypes.Inferred)
    self.addParam("Object", Element("skiros:Product"), ParamTypes
    .Required)
    self.addParam("Gripper", Element("rparts:GripperEffector"),
    ParamTypes.Required)

    self.addPreCondition(self.getRelationCond("
    ObjectLocationContainObject", "skiros:contain", "
    ObjectLocation", "Object", True))
    self.addPreCondition(self.getRelationCond("GripperAtStartPose
    ", "skiros:at", "Gripper", "StartPose", True))
    self.addPreCondition(self.getRelationCond("
    NotGripperContainObject", "skiros:contain", "Gripper", "
    Object", False))
    self.addPreCondition(self.getRelationCond("
    ObjectHasAApproachPose","skiros:hasA", "Object", "
    ApproachPose", True))
    self.addPreCondition(self.getRelationCond("
    ObjectHasAGraspPose", "skiros:hasA", "Object", "GraspPose",
    True))
    self.addPreCondition(self.getRelationCond("RobotAtWorkstation
    ", "skiros:at", "Robot", "Workstation", True))
    self.addPreCondition(self.getRelationCond("
    WorkstationContainObjectLocation", "skiros:contain", "
    Workstation", "ObjectLocation", True))
    self.addPostCondition(self.getRelationCond("
    NotGripperAtStartPose", "skiros:at", "Gripper", "StartPose",
    False))
    self.addPostCondition(self.getRelationCond("
    GripperAtApproachPose", "skiros:at", "Gripper", "ApproachPose
    ", True))
    self.addPostCondition(self.getRelationCond("
    NotObjectContainedObjectLocation", "skiros:contain", "
    ObjectLocation", "Object", False))
    self.addPostCondition(self.getRelationCond("
    GripperContainObject", "skiros:contain", "Gripper", "Object",
     True))
```

Listing 3: An excerpt of the parameters, pre- and post-conditions of a pick skill in SkiROS2 without EzSkiROS. It depends heavily on the usage of strings to refer to parameters or classes in the ontology.

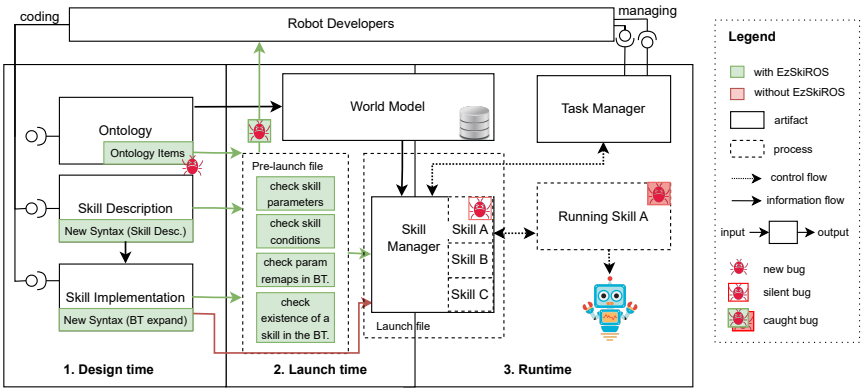


Figure 2: A diagram with the different components of SkiROS2, their interaction during different time phases, and the advancements by EzSkiROS(shown as green blocks). In SkiROS2, a bug that has been introduced in a skill description by a developer will often only trigger at runtime. EzSkiROS addresses these costs and risks by adding checks to find a wide range of bugs by running a pre-launch file where the skills are loaded before runtime.

the real world, such as moving a robot arm, while compound skills build complex behaviors in a BT. An example of a pick *Skill Implementation* is shown in Listing 4.

The *createDescription* method (line 2,Listing 4) sets the description (interface) to an implementation. The *expand* method (line 5,Listing 4) within the skill implementation uses behavior trees to structure the execution of skills. Each node in the tree could represent a specific skill (action node) or a decision-making process (commonly known as a control flow node) that determines which skill to execute next, as illustrated in Figure 3. The control flow node sets the processor and specifies how the compound skill decomposed into a behavior tree (line 6). In SkiROS2, control flow nodes or processors dictate how a compound skill invokes its child skills. Before delving into specific processors, it is essential to understand the common states a node might return during execution:

- SUCCESS indicates that the skill or all skills (in case of compound skills) have been completed successfully.
- FAILURE indicates that the skill has failed to complete successfully or conditions for success are not met.
- RUNNING indicates that the skill is still in progress and has not yet reached a conclusion of success or failure.

These states are not exclusive to compound skills but are also applicable to leaf nodes. Following is the list of processors and how they operate in these states:

```
48 class pick(SkillBase):
49     def createDescription(self):
50         self.setDescription(Pick(), self.__class__.__name__)
51 
52     def expand(self, skill):
53         skill.setProcessor(SerialStar())
54         skill(
55             self.skill("SwitchController", "", specify={'
    Controller': 'joint_config'}),
56             self.skill("MoveitCartesianSpaceMotion", "", remap={'
    GoalPose': 'ApproachPose'}),
57             self.skill("WmSetRelation", "wm_set_relation",
58                        remap={'Src': 'Gripper', 'Dst': '
    ApproachPose'},
59                        specify={'Relation': 'skiros:at', '
    RelationState': True}),
60             self.skill("HSVDetection", ""),
61             self.skill("SwitchController", "", specify={'
    Controller':'compliant'}),
62             self.skill("ApproachMovement", "go_to_linear", remap
    ={'Target':'GraspPose'}),
63             self.skill("WmSetRelation", "wm_set_relation",
64                        remap={'Src': 'Gripper', 'Dst': '
    GraspPose'},
65                        specify={'Relation': 'skiros:at', '
    RelationState': True}),
66             self.skill("Wait", "", specify={"Duration": 2.0}),
67             self.skill("ActuateGripper", "", specify={'Open':
    False}),
68             self.skill("WmMoveObject", "wm_move_object",
69                        remap={ "TargetLocation": "Gripper"}),
70             self.skill("ApproachMovement", "go_to_linear", remap
    ={'Target':'ApproachPose'}),
71             self.skill("Wait", "", specify={"Duration": 2.0})
72         )
```

Listing 4: The skill implementation of the pick *Skill Description* shown in Listing 3.

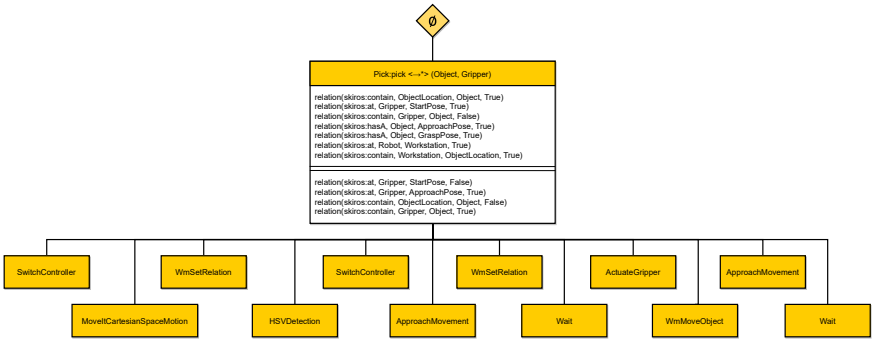


Figure 3: The BT of the pick skill in the *eBT* format [RGK17]. It has a *SerialStar* operator and will execute all children in sequence. The pre-conditions and post-conditions are shown.

- **Serial** processes the children one by one in order until all succeed. It will continuously loop through the children until one returns RUNNING or FAILURE or until all children succeed. The Serial processor returns SUCCESS only if all children succeed, FAILURE if any child fails, and RUNNING if any child is still in the process of completing its task. **SerialStar** is a variation of Serial processor, which remembers which skills have already succeeded so it does not repeat them. The returned states are the same as Serial but with the added efficiency of not repeating successful children.

- **Selector** runs its children one after the other until one succeeds (returning SUCCESS), or all fail (returning FAILURE). If a child is in progress (RUNNING), the processor will also return RUNNING. The key feature here is that it prioritizes success among its children, effectively ignoring failures unless all fail. **SelectorStar** is a variation of *Selector* but remembers which skills have already been tried and succeeded, so it does not repeat them.

- **ParallelFf** (Parallel First Fail) invokes all the children at the same time. It returns SUCCESS only if all children succeed. If any child fails, it immediately returns FAILURE and halts the other children.

- **ParallelFs** (Parallel First Stop) also runs all the children simultaneously. However, it stops all processes and returns SUCCESS as soon as any child succeeds or FAILURE if any child fails, regardless of the others' states.

When we mention that a node "returns" something, we are referring to the result of an operation or computation performed by that node. This result dictates the next action in the behavior tree, like whether to continue, stop, or try a different approach.

In Listing 4, the skill () operator allows to set the skill children, which can be an action node or another control node (to make a nested control structure). To add several children at once it is possible to use the syntax as shown in the Listing 4 (line 9-22). Each child has a template `self.skill(skilltype, label="", remap=, specify=)` which is a placeholder for a *skilltype* replaced at run-time with an available implementation specified as *label*. While *remap* and *specify* are used to manage how parameters and variables are passed and used within the tree. "remap" refers to the redirection or reassignment of input or output parameters from one part of the tree to another and "specify" is used to set a static value of a parameter.

The relationship between *Skill Descriptions* and BTs is evident in how the *expand* function uses the behavior tree structure to implement the skill's logic. The parameters, pre-conditions, hold-conditions and post-conditions defined in the *Skill Description* guide the construction and execution of BTs. For instance, the pre-conditions in a skill description determine when a particular branch of the behavior tree is activated, and the post-conditions signal when a skill or sequence of skills has been successfully completed.

All of these skills are loaded by the *Skill Manager* at robot launch time (shown in Fig. 2).

6 Case Study I: Concise and Verifiable Robot Skill Interface

We have validated our design patterns in an internal DSL EzSkiROS, which adds **Early Dynamic Checking** (Section 4.2) to Skill Descriptions. Following a user-centered design methodology, we developed EzSkiROS by first identifying needs for early bug checking via semi-structured interviews with skilled roboticists who use SkiROS2, reviewed documentation, and manual code inspection. We found that even expert skill developers made errors in writing Skill Descriptions, and that Python's dynamic typing only identified bugs when they triggered faults during robot execution.

We designed EzSkiROS to simplify how Skill Descriptions are specified, with the intent to increase their readability, maintainability, and writability. We map ontology objects and relations into Python's type system. Skill Descriptions can then directly include ontology information in type annotations. This approach streamlines the syntax by avoiding redundant syntactic elements and specifying type information through annotations rather than string encodings, as illustrated with the example of the pick skill in Listing 5. Listing 5 illustrates the EzSkiROS syntax on the example of the pick skill from Listing 3. The Skill Description in Listing 5 is more concise and intuitive, with type annotations providing a clear and direct way to specify the types of parameters and their ontology information.

In EzSkiROS, we employed *owlready2*'s approach to **Domain Language Map-**

ping in exposing the world model elements in the ontology as Python types and objects. For instance in Listing 5, line 3 describes a parameter Robot with the type annotation INFERRED[cora.Robot]. Here, cora.Robot is a Python class that we dynamically generate to mirror an OWL class 'Robot' in the OWL namespace 'cora'. INFERRED is a parametric type that tags *inferred* parameters. We mark *optional* parameters analogously as OPTIONAL; all other parameters are *required*. At robot pre-launch time, we use Python's reflection facilities to extract and check this parameter information, both to link with SkiROS2' skill manager and for part of our **Early Dynamic Checking**. In addition to our ontology types, we also allowed basic data types (str, float, int, bool) in EzSkiROS, enforcing that each must specify a default value. Originally, SkiROS2 also allowed the parameters of data types *list* and *dict*. However, in EzSkiROS, we restricted the use of *lists* and *dicts* as it was not clear if we would need this in practice. One of the developers claimed that *dicts* are considered 'hacks' in the system's context. While *lists* are valid for representing e.g. joint configurations, it might be better served by a specialized joint-config type to encapsulate their complexities and intended use more accurately. We allowed *enums* to handle such parameters, acknowledging that enums cannot encode lists or dicts but it can provide a more controlled and predictable set of values, enhancing the system's integrity and reliability.

```
1  class Pick(SkillDescription):
2      def description(self,
3              Robot: INFERRED[cora.Robot],
4              Arm: INFERRED[rparts.ArmDevice],
5              StartPose: INFERRED[skiros.TransformationPose],
6              GraspPose: INFERRED[skiros.GraspingPose],
7              ApproachPose: INFERRED[skiros.ApproachPose],
8              Workstation: INFERRED[scalable.Workstation],
9              ObjectLocation: INFERRED[skiros.Location],
10             Object: skiros.Product,
11             Gripper: rparts.GripperEffector):
12
13         self.pre_conditions += ObjectLocation.contain(Object)
14         self.pre_conditions += Gripper.at(StartPose)
15         self.pre_conditions += ~Gripper.contain(Object)
16         self.pre_conditions += Object.hasA(ApproachPose)
17         self.pre_conditions += Object.hasA(GraspPose)
18         self.pre_conditions += Robot.at(Workstation)
19         self.pre_conditions += Workstation.contain(ObjectLocation)
20         self.post_conditions += ~Gripper.at(StartPose)
21         self.post_conditions += Gripper.at(GraspPose)
22         self.post_conditions += ~ObjectLocation.contain(Object)
23         self.post_conditions += Gripper.contain(Object)
```

Listing 5: The skill description of the pick skill shown in Listing 3 with EzSkiROS. We respresent OWL classes in Python as identifiers in type declarations.

In addition to skill parameters, we also want to make sure that skill conditions

satisfy contracts in our ontology. These pre-, post-, and hold-conditions can be expressed in different ways depending on what aspects of the robot's environment and state we want to assess.

According to SkiROS2 documentation, one can define a skill with the help of four kinds of skill conditions:

1. *ConditionHasProperty* are unary relations to check whether a certain element or entity has a specific property. It's useful when the skill needs to verify certain attributes or characteristics of objects or elements before proceeding. When a condition checks for a property, it's essentially querying the ontology to see if the entity conforms to certain criteria or states defined within it. For instance, if an ontology defines that a "door" entity can have a "state" property with values "open" or "closed," the *ConditionHasProperty* might check if the door's state is "open."

2. *ConditionProperty* are binary relations which rely on the ontology to understand and evaluate properties or attributes of entities. However, it might be used to assess the value or state of a property rather than just its presence. For example, it could check whether the temperature (property) of a machine is within a certain range.

3. *ConditionRelation* is used to evaluate the relationships between different elements or entities. It's crucial for tasks that require understanding spatial or hierarchical relationships, such as "is next to," "is on top of," or "is part of." This condition utilizes the relational information in the ontology to assess how entities are related to each other. Ontologies define not just entities but also the possible relationships between them. For example, it might check if "object A is on top of object B" by referring to the ontology's definitions of "object A," "object B," and the "on top of" relationship.

4. *AbstractConditionRelation* is a more generalized or template form of *ConditionRelation*, which can be specified or extended for various specific relational conditions.

Since all types of skill conditions rely heavily on the ontology for their evaluation regardless of , it is important to add **Early Dynamic checking** to detect mistyped conditions. We utilise **Symbolic Tracing** as described in Section 4.2. This step collects all pre-, post-, and hold conditions via the overloaded Python operator '+=' (lines 13–23). We then check for wrong ontology relations and ontology type errors among these conditions. Since we use **Domain Language Mapping** to expose the world model entities as classes and relations as Python methods, Python's own name analysis will catch such mistyped ontology relation or entity names, and the symbolic values that we pass into the description method capture all type information that we need for type-checking.

We test our DSL implementation by integrating it with SkiROS2 to see how it behaves with a real skill running on a robot[10]. To demonstrate the effectiveness of our type check in EzSkiROS, we use a 'pick' skill written in EzSkiROS (Listing 5) and load it while launching a simulation of a robot shown in Figure 1.

```
<owl:ObjectProperty rdf:about="http://rvmi.aau.dk/ontologies/skiros
    .#hasA">
    <rdfs:subPropertyOf rdf:resource="http://rvmi.aau.dk/ontologies/
    skiros.#spatiallyRelated"/>
    <rdfs:range rdf:resource="#TransformationPose"/>
    <rdfs:domain rdf:resource="#Product"/>
</owl:ObjectProperty>
```

Listing 6: The definition of the object property 'hasA' in the SkiROS ontology.

Listing 6 shows that the *ObjectProperty* 'hasA' is a relation allowed only between a 'Product' and a 'TransformationPose'. If we introduce a nonsensical relation like Object.hasA(Gripper), then the early dynamic check in EzSkiROS over ontology types returns a type error:

```
TypeError: Gripper: <class 'ezskiros.param_type_system.rparts.
    GripperEffector'> is not a skiros.TransformationPose
```

In addition to the error message, we also provide the source of the error highlighting the line containing the error.

6.1 Evaluation

To evaluate the effectiveness and usability of EzSkiROS in detecting bugs at pre-launch time, we conducted a user study with robotics experts. Seven robotic skill developers participated in our user study, including one member of the SkiROS2 development team. The user study consisted of three phases: an initial demonstration, a follow-up discussion, and a feedback survey[11]. Due to time limitations, we defer a detailed study, with exercises for users to write new skills in EzSkiROS, to the future.

To showcase the embedded DSL and the early bug checking capabilities of EzSkiROS, we presented a video showing (1) a contrast between the old and new skill description written in EzSkiROS and (2) demonstrating how errors in the skill description are detected early at pre-launch time by intentionally introducing an error in the skill conditions.

During the follow-up discussion, we encouraged participants to ask any questions or clarify any confusion they had about the EzSkiROS demonstration video.

After the discussion, we invited the participants to complete a survey to evaluate the readability and effectiveness of the early ontology type checks imple-

[10] Available online in `https://github.com/lu-cs-sde/EzSkiROS`

[11] A replication of the survey`https://github.com/lu-cs-sde/EzSkiROS`

mented in EzSkiROS. The survey included Likert-scale questions about *readability*, *modifiability*, and *writability*. Six participants answered 'strongly agree' that EzSkiROS improved readability, and one answered 'somewhat disagree'. For modifiability, four of them 'strongly agree' but three participants answered 'somewhat agree' and 'neutral'. All the participants answered 'strongly agree' or 'somewhat agree' that EzSkiROS improved writability.

To gain more in-depth insights, the survey also included open-ended questions, e.g.: (a) "Would EzSkiROS have been beneficial to you, and why or why not?", (b) "What potential benefits or concerns do you see in adopting EzSkiROS in your work?", and (c) "What potential benefits or concerns do you see in beginners, such as new employees or M.Sc. students doing project work, adopting EzSkiROS?".

For question (a), all participants agreed that EzSkiROS would have helped them. Participants liked the syntax of EzSkiROS, they thought that it takes less time to read and understand the ontology relations than before. One of them claimed that "pre- and post- conditions are easy to make sense". They also found that mapping the ontology to Python types would have helped reduce the number of lookups required in the ontology. One of the participants said, "in my experience, SkiROS2 error messages are terrible, and half the time they are not even the correct error messages (i.e. they do not point me to the correct cause), so I think the improved error reporting would have been extremely useful.".

For question (b), the majority of participants reported that EzSkiROS's concise syntax is a potential benefit, which they believe would save coding time and effort. One participant found EzSkiROS's specific error messages useful, responding that "the extra checks allow to know some errors before the robot is started" while one participant answered that EzSkiROS does not benefit their current work but it might be useful for writing a new skill from scratch. None of the participants expressed any concerns about adopting EzSkiROS in their work.

For question (c), one developer acknowledges the benefits of EzSkiROS by saying "In addition to the error reporting, it seems much easier for a beginner to learn this syntax, particularly because it looks more like "standard" object oriented programming (OOP)". One person claimed that EzSkiROS would help beginners, describing SkiROS2 as "it is quite a learning curve and needs some courage to start learning SkiROS2 from the beginning autonomously".

In summary, the results of the user evaluation survey indicate a positive perception of EzSkiROS in terms of readability and writability. Most respondents found EzSkiROS to be easy to read and understand, with only one exception. In addition, respondents found EzSkiROS's early error checking to be particularly useful in detecting and resolving errors in a timely manner. This suggests that the users perceived EzSkiROS as an effective tool.

7 Case Study II: Verifiable construction of a behavior tree in Skill Implementation

We further substantiate our design patterns by extending EzSkiROS to add **Early Dynamic Checking** to the implementation of compound skills through the verifiable construction of behavior trees. In this process, our design methodology involved identifying the requirements for the construction of BTs by examining the BT specifications, analyzing GitHub issues encountered by developers when writing BTs in SkiROS2 by performing a systematic search for specific keywords including "Behavior Tree", "Remaps" and "Skill Implementation". Subsequently, we engaged in a verification process with the developers to ensure the validity of the identified issues.

First modification in the skill implementation is to make the link between *SkillDescription* and *SkillBase* as indicated by Listing 3 and Listing 4 respectively. We use **Dynamic Language Mapping** to dynamically link a skill implementation (*SkillBase*) to a *SkillDescription*. Instead of having each skill implementation manually create its description as shown in line 2 of Listing 4, this design pattern sets up the description internally within the *SkillDescription* interface. This means when we implement the "Pick" skill, we inherit from Pick.SkillBase (as shown in Listing 5 line 1), the necessary description and parameter handling is already in place.

Recall from the discussion from Section **??**, on how behavior trees are constructed in the *Skill Implementation* phase. Originally, skills and their implementations were passed as strings, for example, the previous version of the BT specification for "pick" consists of a skill $ApproachMovement.go\_to\_linear$ as `self . skill ("ApproachMovement", "go_to_linear", remap = 'Target':'GraspPose')`. This approach could be error-prone if we pass a string which does not match any skill description or its implementation. To address this issue, we use **Domain Language Mapping** to map the available *SkillDescription* classes as $BehaviorNodes()$. Behavior Nodes class represent the nodes of the behavior tree (BT) we define in `expand` method. In this way, we enable more explicit and clear references to skills within the behavior tree. Instead of vaguely referring to a skill with a string, programmers can directly call the skill with its specific implementation, such as $ApproachMovement.go\_to\_linear$ (as shown in Listing 7 line 10). This not only makes the code clearer and easier to understand but also reduces the risk of errors related to incorrect skill references. With Domain Language Mapping, the SkillDescription class (shown in Listing 5) serves a dual purpose: it helps construct BehaviorNodes and assists in defining parameters, post-conditions, preconditions, and hold-conditions for the skills. This dual usage streamlines the process of defining and implementing skills, ensuring that all necessary information and functionality are encapsulated within a single, coherent structure.

As shown previously in Listing 4, behavior trees were specified in the *expand* method where a list of skills are passed to a `skill ()` wrapper after initializing a

processor (line 7-24). Each child has a template `self.skill(skilltype, label="", remap=, specify=)` where *remap* and *specify* are used to pass parameters to the skills. From our analysis of the BT specifications, we found the concept of "remapping" to be critical. Remapping allows parameters from one skill to be redirected or reassigned to the parameters of another, ensuring that the right data is available where and when it is needed. However, if not managed correctly, remapping can lead to issues, particularly when non-existent parameters are referenced. Consider the case of a compound skill named $SkillB$ with parameters $paramA$ and $paramB$. If an incorrect remapping occurs, such as `self.skill("SkillA", "skill_a", remap='param':'paramC')`, where $paramC$ doesn't exist in $SkillB$ this would typically lead to errors or unintended behavior. It is crucial that such issues are detected and reported early in the development process to prevent complications later in the skill execution. The system or the developers should ensure that remappings reference existing and appropriate parameters. In EzSkiROS (example shown in Listing 7), we use **Dynamic Language Mapping** to address this issue. This method exposes skill parameters to their respective *Skill Descriptions* as objects, which can accessed and manipulated more intuitively. For instance, in Listing 7, we pass the skill parameters of the Pick skill as an input to the $expandBT$ method and are accessed directly as `params.ApproachPose`, simplifying the process of defining and checking parameter remappings. This approach not only makes skill specifications more natural and easier to understand but also links parameters to the skill being implemented (Pick), ensuring that the entire system works cohesively.

Additionally, we improve error messages through **Source Provenance Tracking** to make the debugging of behavior trees easier than before. Its true utility lies in pinpointing the exact location of issues, providing file and line numbers for where things went wrong. This capability is invaluable when dealing with complex behavior trees, as it allows developers to quickly find and address issues without manually sifting through the code.

Need for static Pre-/Post-Condition Matching in SkiROS

As mentioned in Sections **??** and **??**, pre- and post-conditions in SkiROS2 serve as essential components for ensuring the correct execution of skills to complete a robot's task. These conditions are checked by the Skill Manager before starting and parameterizing the skill. These conditions are also important if you use a task planner and they play an important role in scaling SkiROS2 to meet more complex challenges. The key motivations to include pre-conditions are:

- for planning, pre- and post-conditions are vital to understand and verify whether a skill can be initiated (pre-condition) and confirming if the skill's goal has been achieved (post-condition) after execution,
- for dynamic sanity checks (without using a planner), these checks are crucial for maintaining the integrity of skill execution, especially in complex scenarios where multiple skills interact. While they might seem less critical

```
class pick(Pick.SkillBase):

    def expandBT(self, params):
        self.serialstar(
            SwitchController(Controller='joint_config'),
            MoveitCartesianSpaceMotion(GoalPose=params.
    ApproachPose),
            WmSetRelation.wm_set_relation(Src=params.Gripper, Dst
    =params.ApproachPose, Relation='skiros.at', RelationState=
    True),
            HSVDetection(),
            SwitchController(Controller='compliant'),
            ApproachMovement.go_to_linear(Target=params.GraspPose
    ),
            WmSetRelation.wm_set_relation(Src=params.Gripper, Dst
    =params.GraspPose, Relation='skiros.at', RelationState=True),
            Wait(Duration=2.0),
            ActuateGripper(Open=False),
            WmMoveObject(TargetLocation=params.Gripper),
            ApproachMovement.go_to_linear(Target=params.
    ApproachPose),
            Wait(Duration=2.0)
        )
```

Listing 7: The EzSkiROS representation of the skill implementation shown in Listing 4. Here the inheritance from $Pick.SkillBase$ links the Pick skill description shown in Listing 5 to its implementation.

in controlled or smaller settings, their importance escalates as the complexity and scale of tasks grow. Poor quality or incorrectly defined conditions can significantly limit the ability of SkiROS2 to scale and handle complex, dynamic tasks efficiently.

If we don't use a planner, then manually creating parent skills or adjusting existing ones without thorough checks can lead to mismatches between expected and actual skill behaviors. Static checking of pre-/post-conditions becomes essential to identify and correct these errors early in the development cycle, preventing potential failures during execution. To verify this requirement, we randomly selected five SkiROS2 skills written by developers to understand the prevalence of errors. Among those five skills, four of them failed the following basic checks:

- Sequence Skills: For a sequence skill s = sequence(A, B, C), the pre-condition of 's' must entail the pre-condition of 'A', and the aggregate post-conditions of 'A' must entail the pre-condition of 'B', and so on.
- Selector Skills: For a selector skill s = selector(A, B, C), the pre-condition of 's' must entail the conjunction of the pre-conditions of 'A', 'B', and 'C'.

Post-conditions of 's' can be conservatively checked as any of the children can lead to success without a predetermined order.

- Parallel Skills: For parallel skills, all children must succeed, with specific differences in handling the completion and order. This requires that no post-condition of one skill may invalidate the pre-condition of another due to the simultaneous nature of execution.

This evidence points to a common oversight in defining these conditions carefully and makes it important to have robust tooling to ensure that pre- and post-conditions are correctly matched and implemented.

To address these challenges, we plan to create a comprehensive mapping and verification system in the future. This system would track all pre- and post-conditions, manage dependencies and changes, handle remapping accurately, and ensure that all conditions are consistent and verifiable at each step of the skill execution. It would likely involve a combination of static analysis tools, careful structuring of skill descriptions, and possibly enhancements to the SkiROS2 framework to support more robust condition checking and error reporting.

8 Overall Evaluation of the Extended EzSkiROS

Our evaluation of the extension of EzSkiROS (as mentioned in Case Study II) is primarily based on an in-depth review provided by an experienced SkiROS2 developer and maintainer who has used the tool for transforming old SkiROS2 code into EzSkiROS. We requested developer feedback on various aspects of EzSkiROS, including its strengths and weaknesses, the impact on code readability and writability, the ease of code translation, the comprehensibility of errors encountered, and any general observations or suggestions they might have. The user's experience offers valuable insights into the strengths, weaknesses, and overall impact of EzSkiROS on skill description development in robotics.

Strengths and Weaknesses The developer highlighted several key strengths of EzSkiROS:

- **Early Detection of Misuse:** EzSkiROS enables the detection of misuse in the world model before the skills are utilized, enhancing the correctness of the code.

- **Validation of Naming in Conditions:** The tool validates naming in pre-conditions and post-conditions, ensuring consistency and correctness in element types and names.

- **Improved Error Messaging:** Compared to traditional SkiROS, EzSkiROS provides clearer and more concise error messages.

- **Readability:** There is a significant improvement in the readability of skill descriptions, and skill implementations of both compound and primitive skills.

However, the developer also noted a primary weakness:

- **Developer Productivity:** Despite the aforementioned strengths, the developer expects that EzSkiROS will not provide substantial productivity benefits. The developer attributes this to the dynamic nature of most checks and the fact that world model errors abort Python execution, leading to one error being reported at a time.

Impact on Code Quality The developer review suggests that EzSkiROS positively impacts the code quality in several ways:

- **Correctness:** By enforcing element types on parameters and consistent naming, the correctness of the code is improved.
- **Readability and Intuitiveness:** The conciseness and clarity in pre- and post-conditions make the code easier to read and understand.
- **Clarity in Skill Dependencies:** The dependencies between Skill Description and SkillBase (Skill Implementation) of a skill are more apparent in the code.
- **Conciseness in Writing behavior trees:** Writing behavior trees for compound skills has become more concise and less cluttered.

Translation Process The developer reported the translation of existing skill descriptions to EzSkiROS to be straightforward. The time required for translation depends on the number of skill descriptions to be converted but it can be automated.

Error Reporting and Understanding The user affirmed that the errors identified by EzSkiROS were sensible and contributed to a better understanding of the issues in the skill descriptions.

General Feedback The developer acknowledged EzSkiROS as a significant step forward, particularly in moving from string-based descriptions to more natural and correct Python code. The reduction in common errors due to the validation of parameter names and world element relations was especially noted. For future work, the developer suggested:

- **Static Analysis Integration:** Implementing static analysis to run checks on modules and skills independently, possibly integrated with a linter, to further reduce bugs at an early stage.
- **Code Generation for Enhanced Development Experience:** Utilizing code generation to enable features like autocompletion and static checks during

coding, particularly for the world model, to improve the development experience.

The user review provides an insightful evaluation of EzSkiROS, highlighting its strengths in improving code readability, correctness, and error messaging. The contribution of EzSkiROS to reducing common errors and improving the overall quality of skill descriptions is evident. According to the reviewer, it falls short in significantly enhancing developer productivity due to the fact that we do dynamic checks at pre-launch and the user suggests static analysis. It is important to note here that static check requires certain information (ontology, robot configuration) to be available at development time, which is not guaranteed. Modulo this caveat, we see no fundamental barrier towards using the techniques that we describe here for both pre-launch and static checks in practice, using language server or development environment plugins.

9 Conclusion

In this paper, we present two analyses of different abstraction levels of a robotic software and how can we use DSL design patterns to detect bugs at a pre-launch stage before runtime. Case Study I demonstrated the value of our design patterns by showing how they help detect bugs in the high-level contracts between a variety of robot capabilities and the robot's world model. Case Study II expands EzSkiROS by adapting the same techniques to detecting bugs in lower-level implementation code, in our case that implementation uses a behavior tree to integrate different robot capabilities.

In exploring the relationship between the two analyses, it's important to ask: Do they work separately, depend on each other, or are they independent yet work better together, creating a stronger combined effect than each would alone? The study shows that analysis of behavior trees (Case Study II) require information about the skill parameters from the higher level descriptions to check correct information being passed on between skills. Behavior trees also need to access the pre-, post- and hold-conditions from the skill descriptions of the skill being implemented. On the other hand, the higher level analysis (Case Study I) is standalone but can benefit from the BT sequencing information to suggest pre- and post-conditions to the developer. Our work demonstrates how embedded DSLs can help robotics developers detect bugs early, even when the analysis depends on data that is not available until run-time. Our evaluation with EzSkiROS further suggests that embedded DSLs can achieve this goal while simultaneously increasing code maintainability.

In our future work, we plan to collect some objective results to further substantiate our efforts. We plan to make EzSkiROS publicly available to SkiROS2 users so people can write skills and transform their old skills into EzSkiROS, and we can

get some error reports and if people find the error reports helpful. We aim to conduct an in-depth user study to explore how EzSkiROS assists users in writing skill descriptions and detecting bugs in Behavior trees through pre-and post-condition matching. This study will mainly focus on understanding the user experience with-EzSkiROS, particularly in terms of its usability and effectiveness in early bug detection. A significant aspect of this study will be to extend the possibility of the integration of the two analyses at different abstraction levels and how their combination influences the bug detection process. We are particularly interested in whether this integration simplifies the process of writing error-free skill descriptions and how it impacts the overall development workflow. By analyzing the data collected from this study, we expect to gain valuable insight into the practical applications and limitations of EzSkiROS. This will not only help us in refining the tool but also contribute to the broader understanding of skill programming in robotics.

Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[AMK23] Faseeh Ahmad, Matthias Mayr, and Volker Krueger. “Learning to Adapt the Parameters of Behavior Trees and Motion Generators (BTMGs) to Task Variations”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2023, pp. 10133–10140.

[Ahm+11] Amal Ahmed et al. “Blame for all”. In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2011, pp. 201–214.

[BR20] Hampus Balldin and Christoph Reichenbach. “A domain-specific language for filtering in application-level gateways”. In: *GPCE 2020*. 2020, pp. 111–123.

[Bøg+12] Simon Bøgh et al. “Does your robot have skills?” In: *Proceedings of the 43rd international symposium on robotics*. VDE Verlag GMBH. 2012.

[Bru+07] Davide Brugali et al. “Trends in robot software domain engineering”. In: *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 3–8.

[Buc+14] Jacob Pørksen Buch et al. "Applying Simulation and a Domain-Specific Language for an Adaptive Action Library". In: *Simulation, Modeling, and Programming for Autonomous Robots*. 2014, pp. 86–97.

[CA22] Angelo Cangelosi and Minoru Asada. *Cognitive Robotics*. MIT Press, 2022.

[Ceh+11] Ines Ceh et al. "Ontology driven development of domain-specific languages". In: *Computer Science and Information Systems* 8.2 (2011), pp. 317–342.

[CÖ18] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[CMT97] Eve Coste-Maniere and Nicolas Turro. "The maestro language and its environment: Specification, validation and control of robotic missions". In: *RSJ International Conf. on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS*. Vol. 2. IEEE. 1997.

[DP22] Eric Dortmans and Teade Punter. "Behavior Trees for Smart Robots Practical Guidelines for Robot Software Development." In: *Journal of Robotics* (2022).

[Dra+21] Swaib Dragule et al. "Languages for specifying missions of robotic applications". In: *Software Engineering for Robotics*. Springer, 2021, pp. 377–411.

[Fac19] Davide Faconti. "Mood2be: Models and tools to design robotic behaviors". In: *Eurecat Centre Tecnologic, Barcelona, Spain, Tech. Rep* 4 (2019).

[Ghz+20] Razan Ghzouli et al. "Behavior trees in action: a study of robotics applications". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 2020, pp. 196–209.

[Ghz+23] Razan Ghzouli et al. "Behavior Trees and State Machines in Robotics Applications". In: *IEEE Transactions on Software Engineering* (2023).

[Hen+22] Thomas Henn et al. "Verification of Behavior Trees using Linear Constrained Horn Clauses". In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2022, pp. 211–225.

[HKR09] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[Iov+22] Matteo Iovino et al. “A survey of behavior trees in robotics and ai”. In: *Robotics and Autonomous Systems* 154 (2022), p. 104096.

[Kin76] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[Kru+16] Volker Krueger et al. “A Vertical and Cyber–Physical Integration of Cognitive Robots in Manufacturing”. In: *Proceedings of the IEEE* 104.5 (2016), pp. 1114–1127.

[KRB11] Lars Kunze, Tobias Roehm, and Michael Beetz. “Towards semantic robot description languages”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011.

[Lam17] Jean-Baptiste Lamy. “Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies”. In: *Artificial intelligence in medicine* 80 (2017), pp. 11–28.

[LMB92] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc*. " O’Reilly Media, Inc.", 1992.

[MRK23] Matthias Mayr, Francesco Rovida, and Volker Krueger. “SkiROS2: A skill-based Robot Control Platform for ROS”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 6273–6280.

[May+22a] Matthias Mayr et al. “Combining Planning, Reasoning and Reinforcement Learning to solve Industrial Robot Tasks”. In: *arXiv preprint arXiv:2212.03570* (2022).

[May+22b] Matthias Mayr et al. “Learning Skill-based Industrial Robot Tasks with User Priors”. In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. 2022, pp. 1485–1492.

[May+22c] Matthias Mayr et al. “Skill-based Multi-objective Reinforcement Learning of Industrial Robot Tasks with Planning and Knowledge Integration”. In: *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2022.

[May+23] Matthias Mayr et al. “Using Knowledge Representation and Task Planning for Robot-Agnostic Skills on the Example of Contact-Rich Wiping Tasks”. In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2023, pp. 1–7.

[Mog+13] Mikael Moghadam et al. “Towards python-based domain-specific languages for self-reconfigurable modular robotics research”. In: *arXiv preprint arXiv:1302.5521* (2013).

[Nor+16] Arne Nordmann et al. “A Survey on Domain-Specific Modeling and Languages in Robotics”. In: *Journal of Software Engineering in Robotics (JOSER)* 7.1 (2016), pp. 75–99.

[Oli+20] Miguel Oliveira et al. “BhTSL, behavior trees specification and processing”. In: (2020).

[Ped+16] Mikkel Rath Pedersen et al. “Robot skills for manufacturing: From concept to industrial deployment”. In: *Robotics and Computer-Integrated Manufacturing* 37 (2016), pp. 282–291.

[Qui+09] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[Rei21] Christoph Reichenbach. “Software ticks need no specifications”. In: *ICSE-NIER 2021*. IEEE. 2021, pp. 61–65.

[Riz+23] Momina Rizwan et al. “Ezskiros: A case study on embedded robotics dsls to catch bugs early”. In: *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*. IEEE. 2023, pp. 61–68.

[RGK17] Francesco Rovida, Bjarne Grossmann, and Volker Krüger. “Extended behavior trees for quick definition of flexible robotic tasks”. In: *RSJ International Conf. on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 6793–6800.

[Rov+17] Francesco Rovida et al. “SkiROS— a skill-based robot control platform on top of ROS”. In: *Robot Operating System (ROS)*. Springer, 2017, pp. 121–160.

[Spi01] Diomidis Spinellis. “Notable design patterns for domain-specific languages”. In: *Journal of systems and software* 56.1 (2001), pp. 91–99.

[TT22] Matteo Tadiello and Elena Troubitsyna. “Verifying Safety of Behaviour Trees in Event-B”. In: *arXiv preprint arXiv:2209.14045* (2022).

[Wut+21] D. Wuthier et al. “Productive Multitasking for Industrial Robots”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 12654–12661.