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

Momina Rizwan[1], Ricardo Caldas[2], Christoph Reichenbach[1] and Matthias Mayr[1]

Abstract—**In robotics, we do not have all the information available at all times. This limits our ability to make predictions, including our ability to detect program bugs early. However, running a robot is an expensive task and finding errors only during runtime might prolong the debugging loop or even cause safety hazards. In this paper, we propose to help developers find bugs early with minimal extra 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 robotic skill composition. We demonstrate our patterns on the embedded DSL EzSkiROS and show that our approach is effective at performing safety checks during the robot launch time, much earlier than at run time. In interviews with robotics developers familiar with the SkiROS2 software stack, they report that they find our DSL-based approach useful not only for finding bugs early, but also to increase robotics code maintainability.**

I. Introduction

Coding robotic systems that carry out socio-technical missions was never more relevant, nor harder. Software engineering for robotics should thus promote a set of development tools and techniques to assist robot developers to meet time-to-market expectations with confidence of correctness. Specifically, robot development tools should provide expressive programming languages and frameworks that allow human developers to describe correct robot behavior [1].

One such tool is *SkiROS2*[3] [2], a robot control platform in Python that allows roboticists to write robot skills such as "pick" or "drive" with pre- and post-conditions. These conditions can be used together with ontologies, which represent concepts and relations in the domain, e.g., automated assembly line, or robotic healthcare surgery, to infer additional information and to check whether all necessary conditions for the execution are met. Concretely, Figure 1 depicts a robotic arm that should be used to pick an object. The 'pick' skill in this example has conditions such as "gripper is part of the robot arm" that are used to automatically determine other parameters such as "which arm to move" or "what is the object's location". In other words, to configure this 'pick' skill it is sufficient to state which gripper to use and object to pick and the remaining parameters can be resolved by automatic reasoning.

Importantly, writing and maintaining skills such as 'pick' (Fig. 1) requires precise knowledge of the relations in the

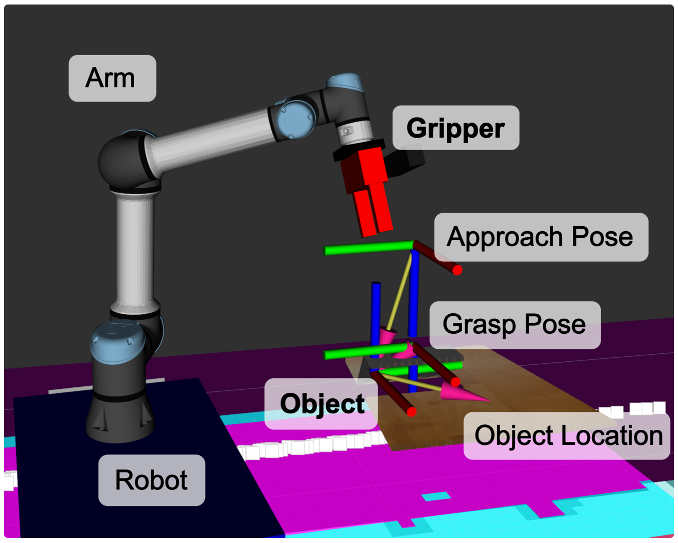

Fig. 1. The robot using a pick skill with a visualization of the necessary parameters. To run this skill, we only need the Gripper and the Object parameters. SkiROS2 can deduce all other necessary parameters through a set of rules in the skill description shown in Listing 5 and 6.

ontologies. However, manually keeping the code and the ontology in sync is error-prone and bugs introduced in this phase will often remain silent until runtime.

This problem can be addressed with Domain-Specific Languages (DSLs). They have demonstrated their ability to fill this role in a variety of tasks in robotics, including high-level mission specification [3], kinematics and dynamics representation [4], robot knowledge modeling [5]. Nordmann et al. [6] collect and categorise over 100 such DSLs for robotics in their *Robotics DSL Zoo*[3]. The benefits of DSLs for aiding optimization, debugging, visualization, and static checking are well-established.

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

- A survey of key language features in Python that support DSL embedding;
- Two design patterns for *embedding robotics DSLs in general purpose programming languages*, with details on how to implement these patterns in Python;
- A case study with a tool for robot skill design and execution in which we introduce the DSL EzSkiROS to detect type errors and other bugs early.

[1]Department of Computer Science, Faculty of Engineering (LTH), Lund University, SE 221 00 Lund, Sweden. E-mail: <firstname>.<lastname>@cs.lth.se.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, SE 417 56 Gothenburg, Sweden. E-mail: <firstname>.<lastname>@chalmers.se.

[3]https://github.com/RVMI/skiros2

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

II. Related Work

Extending robotics programming languages to prevent erroneous sequences was studied in the assembly domain [7], [8]. Buch J. et al. [7] define an internal DSL over C++ to ease the sequencing of robotic skills (i.e., a set of actions enveloped in pre- and post-conditions). Using a model-driven approach, their DSL instantiates a textual representation of the assembling sequence, which is interpreted for performing the assemble behavior. It is unclear from the paper whether and how 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. This approach, however, does not consider that some bugs (e.g., due to wrongly specified ontology relations) may be caught before simulation.

Coste-Manière È. and Turro N. [9] propose MAESTRO, an external DSL targeting the specification of reactive behavior and checking in the robotics domain. MAESTRO aims at handling complex and hierarchical missions prone to concurrency and requiring portable solutions. Importantly, the DSL enables the specification of user-defined typed events that may occur before (pre-conditions), during (hold-conditions) or after (post-conditions) a task execution. The mission specification language, *MAESTRO*, features checking the user-defined types with respect to names, number of parameters, and stop conditions. In opposition to internal DSLs, MAESTRO can only benefit developers that are willing to adopt the language as a whole or adapt the front-end to their needs.

Kunze L. et al. [10] propose the Semantic Robotic Description Language (SRDL) bridging the gap between high-level description of robots and low-level primitives. To this end, the authors use the Web Ontology Language (OWL) notation for modeling knowledge about robots, capabilities and actions. SRDL matches robot descriptions and actions using static analysis of the robot capability dependencies, i.e. they analyse the specification before it is executed. The static analysis may catch bugs early in the OWL ontology specification and prevent bugs being carried to runtime. Although, the authors demonstrate the applicability of their approach in a controlled scenario, it is unclear whether SRDL supports early dynamic checking in general purpose languages.

III. 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*, much of this effort comes from building a language frontend. *Internal* or *embedded DSLs* 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 here look at Python as one of the three main supported languages of the popular robotics platform ROS [11]. The other two languages, C++ and LISP, also support internal DSLs, but with different trade-offs.

A. *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 language. DSL designers can freely overload most of Python's infix operators (mainly excluding the boolean operators, which must always return a boolean) and repurpose Python's type annotation and decorator mechanisms. Moghadam et al. previously also identified an earlier version of Python as an attractive target for embedded DSLs [12] due to it being a "modern multi-paradigm dynamic and high-level language," and exploit its decorator mechanism.

Listing 1 illustrates some of these techniques. Class A represents a *deferred* arithmetic operation op with parameters args. To evaluate this deferred operation, we can call A.eval (Line 15) on it. The @staticmethod decorator tells Python that this method does not take an implicit self parameter. DSL designers can define other decorators to transform the semantics of individual functions, methods, or classes.

Python also allows us to introduce special handlers for properties (fields or methods) of unknown name. For class attributes, this requires a *metaclass*, for which we define M in Line 1. The definition there checks for accesses to attributes that start with an underscore and continue in decimal digits, and converts them into "deferred integers", so that A._7 is an A object that will evaluate to 7.

Finally, class A overloads infix addition in line 11 and function call notation in line 13, so that we can use this notation with a custom meaning in lines 21 and 22. Such "deferred" computation has little impact in a toy example like ours but allows us to constructing in-memory representations of complex computations for *staging*, where we could then (e.g., in A.eval) perform optimisations or translate the code representation into a more efficient format (e.g., for evaluation on a GPU).

Listing 1. A collection of DSL-friendly Python features.

```
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
```

In Listing 1, line 13 also illustrates Python's type annotation mechanism, annotating parameter `arg` with type `int`. This annotation normally has no effect in Python, but DSL designers can use the language's reflection operations to read it out. Thus, developers can use annotations to pass information to the DSL, without interference from Python. These annotations (since Python 3.5) may also include type parameters, so that we can e.g. annotate `x : list[int]` (type parameters for Python standard library types are only available as of Python 3.9).

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 catkin configuration files or ontologies) but critical to program logic, we can map them to suitable type hierarchies at robot launch time.

B. *Robotics DSL Design Patterns*

Domain Language Mapping Our first pattern's *purpose* is *to make domain notation visible in Python, in order to decrease notational overhead*. It directly applies Spinellis' "Piggyback" DSL implementation pattern [13], to which we defer for implementation details.

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* [14] 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 [12] compared to external DSLs, we have so far 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 practice, combining Python code, ontology specifications, and other configuration 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 during early execution

The *behaviour* of this pattern is:

- We execute all critical pieces of code early, while redefining the semantics of the predetermined set of operations to immediately return or to only perform checking

In Python, configuration and type errors only trigger software faults once the software executes code that depends on the faulty data. In robotics, we commonly 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 ROS and/or robotics hardware). For robotics developers, both challenges increase the cost of verification and validation [15]: an error might trigger only at the end of 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 desired 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 error late. If the pose description comes from a configuration file or from the ontology, traditional static checkers will also be ineffective: we can only check after we have loaded all relevant configuration.

Through careful software design, developers can work around this problem, e.g., by verifying that the code and the configuration are well-formed as soon as possible, before they run the control logic — though this means that they must manually ensure that the checks and the control logic make the same assumptions. If the critical code itself is free of external side effects, the check can be as simple as running the critical code twice. Early Dynamic Checking is particularly easy to use with functions and methods that have no side effects. For example, SkiROS2 composes *behavior trees (BTs)* [16] within such critical Python code (Listing 2): composing (as opposed to running) these objects has no further side effects, so that we can safely construct them early to detect simple errors (e.g., typos in the parameter name). Note that we again cannot check before robot launch time (e.g., statically): 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

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").

```
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}),)
```

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:

Listing 3. Early dynamic checking without DSL support: Developers have manually constructed a `check()` method.

```
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). To capture case (a), we would also call run(), but instead of passing in an instance of MyRobotOp, we would pass in a *mock* instance of the same class, in which operations like runA() immediately return:

Listing 4. Early dynamic checking with mock class: with MyRobotOpMock we can check MyRobotOp.run() directly and need no separate MyRobotOp.check() method.

```
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() would 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)
```

[1]Python's eval function offers similar capabilities, but as of Python 3.10 does not appear 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 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 information (type annotations etc.) that is sufficient for simulating parameter values and operation return values *symbolically* (see below)
- The critical code does not use complex control flow

The *behaviour* of this pattern is that

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

Here, a *symbolic* parameter is a special kind of mock parameter that we use to record information [17].

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 a priori nothing — thus, we cannot directly apply the Early Dynamic Checking pattern.

If we know that we are only dealing with operations that we have a priori knowledge about, we can still find out useful information about this operation. Assume that we know that setArmSpeed may only take numeric parameters, and that setArmSafety may only take a boolean parameter: we may want to flag this code as having a type error. To detect this error without blindly trying out different parameters, we can pass a symbolic parameter to run instead, and again use the mock-execution strategy that we used for Early Dynamic Checking, but adapted so that the mock objects record information:

```
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 itself (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 [18], and 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, the approach requires us to have information about operations — setArmSpeed and setArmSafety, in the example above. We can pass this information to Python in a variety of ways, e.g., via type annotations.

As an example, consider an operation that picks up a coffee 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 ontology, with minor simplifications. Intuitively, it seems nonsensical that a coffee table should be a robotPartOf a robot, and indeed the ontology requires that robotPartOf is a relation between a Device and a Robot. Since Furniture is not a subtype of Device, we should never be able to satisfy the assertion in line 6.

We can again detect this bug through symbolic reasoning. 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 that takes one additional parameter and records that the gripper and the parameter object 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, or handle the error ourselves if we prefer to provide more information to the programmer.

Key to this symbolic reasoning 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.

C. 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, 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 [9] 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.

IV. Case Study: Concise and Verifiable Robot Skill Interfaces with EzSkiROS

As a first case study, we selected the skill-based robot control platform SkiROS2 [2]. With its components depicted in Fig. 2, SkiROS2 implements a layered, hybrid control architecture to define and execute parametric *skills* for robots [19], [20]. These parametric skills such as "pick" or "drive" are general skills that can be used for a variety of tasks and on different robot systems. SkiROS2 represents knowledge about the skills, the robot and the environment in a world model (WM) with the ontologies specified being in OWL

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

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

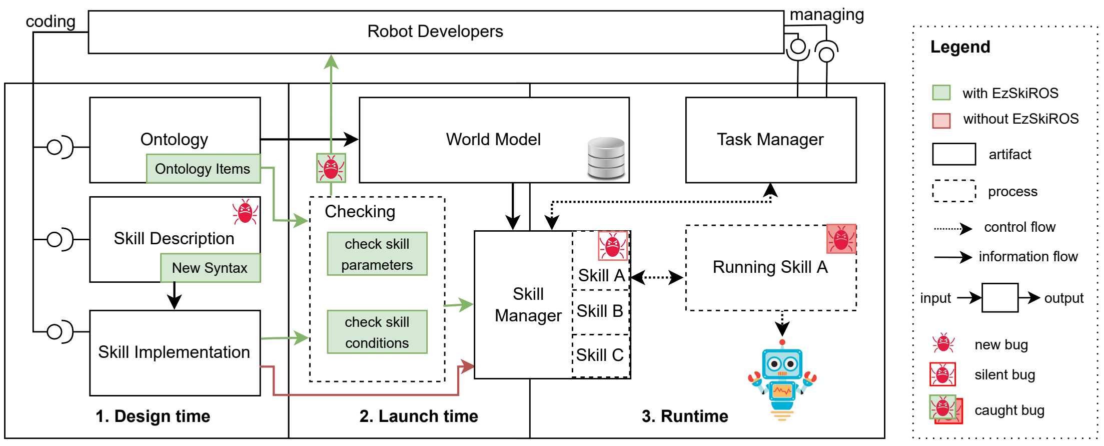

Fig. 2. A diagram with the different components of SkiROS2, their relations and the additions by EzSkiROS. Previously, a bug that has been introduced in a skill description by a developer will often only trigger at runtime. EzSkiROS addresses these costs and risks by finding a wide range of bugs at launch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

format. This explicit representation, built upon the World Wide Web Consortium's Resource Description Framework standard(RDF), allows for the usage of existing ontologies.

SkiROS2 utilizes behavior trees (BTs) [16] as a procedure representation formalism. We refer the reader to [16] for a general introduction to BTs, to [21] for a thorough introduction to *SkiROS1* and to [2] for the implementation in SkiROS2.

SkiROS2 is used by several research institutions in the context of industrial robot tasks [22], [23], [24], [25]. It is implemented in Python and utilizes the Robot Operating System (ROS) [11] as a middleware for programming paradigms and communication.

A. Robot Skill Model in SkiROS

Skills in SkiROS2 are parametric procedures that modify the world state from an initial state to a final state according to the pre- and post-conditions [26]. These conditions are formulated in the skill description together with hold conditions and the skill parameters.

Skills can either be primitive or compound skills. Primitive skills are the atomic actions and the typically implement functions that change the real world such as moving a robot arm. The second type of skills are compound skills. They allow to connect primitive skills and other compound skills with a BT to build more complex behaviors. And example for such a connection is shown in Listing 2. It defines a sequence of the "Navigate" skill and a skill to update the location of the robot in the WM if the navigation has succeeded.

The idea behind this skill model is that skill can be used in a modular and adaptive fashion with different hardware platforms by providing a semantic hardware description in the WM, having a device abstraction layer and knowing the kinematic robot description. The skills are loaded by the skill manager at robot launch time.

B. Skill Description

Every skill implements a skill description that defines the action performed by the skill on a semantic level. The description consist of four elements:

1) *Parameters* define input and output of a skill
2) *Pre-conditions* must hold before the skill is executed
3) *Hold-conditions* must be fulfilled during the execution
4) *Post-conditions* state the effects of a skill and are checked once the execution finished

Listing 5 shows how developers define these skills in SkiROS2 by calling a Python method addParam to introduce parameters and similarly to define pre- and post-conditions.

Skill parameters have three categories: 1) required, 2) optional and 3) inferred. All have in common that they can be a fundamental Python datatype such as "string" or a WM element in the form Element("concept").

We can use pre-conditions to check relations and properties that need to hold to allow the execution, or for automatically inferring skill parameters. For example in the pick skill shown in Listing 5, the parameter "object" in line 10 is required and needs to be set before the execution. The known value can be used to infer the parameter "container" by reasoning about the pre-condition rule "ObjectLocationContainObject" stated in line 13. If the selected object is semantically not at a location in the WM, the rule does not hold and the skill will not be executed.

C. EzSkiROS: Language and Semantics

To ensure a *user-centered design* [27] of our DSL and to validate if early bug checking is desired, we conducted semi-structured interviews with roboticists developing skills in SkiROS2. In addition, we employed code inspection techniques and reviewed SkiROS2 documentation to verify the cross-cutting concerns of the domain. We observed that even expert skill developers were prone to making mistakes while writing skill descriptions. Since Python is dynamically typed, bugs are only detected during robot execution.

Considering the above domain analysis, we develop EzSkiROS language which aims at catching bugs early using static (early dynamic) checking. As a by-product, we get

Listing 5. 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.

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

```python
class Pick(SkillDescription):
def description(self, Robot: INFERRED[cora.Robot],
      Arm: INFERRED[rparts.ArmDevice],
      StartPose: INFERRED[skiros.TransformationPose],
      GraspPose: INFERRED[skiros.GraspingPose],
      ApproachPose: INFERRED[skiros.ApproachPose],
      Workstation: INFERRED[scalable.Workstation],
      ObjectLocation: INFERRED[skiros.Location],
      Object: skiros.Product,
      Gripper: rparts.GripperEffector):

  self.pre_conditions += ObjectLocation.contain(Object)
  self.pre_conditions += Gripper.at(StartPose)
  self.pre_conditions += ~ Gripper.contain(Object)
  self.pre_conditions += Object.hasA(ApproachPose)
  self.pre_conditions += Object.hasA(GraspPose)
  self.pre_conditions += Robot.at(Workstation)
  self.pre_conditions += Workstation.contain(ObjectLocation)
  self.post_conditions += ~ Gripper.at(StartPose)
  self.post_conditions += Gripper.at(GraspPose)
  self.post_conditions += ~ ObjectLocation.contain(Object)
  self.post_conditions += Gripper.contain(Object)
```

a simplified syntax of the skill description which increase writability and readability of the skill. The main idea is to use ontology items to define a type system over skill parameters and check it against skill parameters and skill conditions to detect errors at launch time (EzSkiROS components shown in Figure 2).

In the code example in Listing 6, the pick skill takes *skill parameters* as input arguments to the description method and append the ontology relations in skill conditions.

D. Implementing EzSkiROS

For *Domain Language mapping*, we use Python's ability to dynamically generate mock classes for all the from the ontology types in the WM. For instance, in Listing 6 cora.Robot is a class that we generated dynamically from 'Robot' class in the ontology specification file 'cora.owl'. Then we used Python's type annotation mechanism to annotate the skill parameters with these *ontology types*. To represent the *Inferred*, *Optional* and *Required* skill parameters in a concise format, we use Generics to be able to express INFERRED[rparts.ArmDevice] as syntactic sugar. Python ignores this tagging but our call can read information about the parameter types and utilise them for checking and maintaining the logic from the old SkiROS2 API. By using skill parameters as input arguments, we can use Python's own language semantics to check any mistyped names in the skill conditions.

We overload the operator += to append relations in the list of pre-/post-conditions. We also check if these relations are correctly defined by using *symbolic tracing*. According to the ontology, a relation 'contain' can be between ObjectLocation/-Gripper and an object, which defines a fact that object can either be at a location or in the robot's gripper and cannot be suspended in air. If we define a nonsensical relation, then by symbolic reasoning we can detect this bug.

E. 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 6) and load it while launching a simulation of a robot shown in Figure 1.

Listing 7. The definition of the object property 'hasA' in the SkiROS ontology.

```xml
<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 7 shows that the *ObjectProperty* 'hasA' is a relation allowed only between a 'product' and a 'TransformationPose'.

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

If we introduce a nonsensical relation like `Object.hasA(Gripper)`, then the early dynamic check in EzSkiROS over ontology types returns a type error, as shown in Listing 8.

Listing 8. The error message generated while loading the pick skill.

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

F. Evaluation

To evaluate the effectiveness and usability of the Domain Specific Language (DSL), we conducted a user study with robotics experts. In the study we assessed what benefits robotics practitioners from academia and industry saw in EzSkiROS, for reading and writing skill descriptions in SkiROS2. Seven robotic skill developers participated in our user study, including one member of the SkiROS2 development team. We categorise the participants on the basis of their experience with SkiROS2 skill descriptions: *experts* who have used SkiROS2 for more than a year, and beginners, who have not. The user study consisted of three phases: an initial demonstration, a follow-up discussion, and a feedback survey. We have included a replication package documenting the artifacts used in this evaluation[5].

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 gather more in-depth insights, the survey also incorporated open-ended questions, including: (a) "Would EzSkiROS have been beneficial to you, and why or why not?", (b) "What potential benefits or concerns do you see about adopting EzSkiROS in your work?", and (c) "What potential benefits or concerns do you see about 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. The beginners liked the way one can write ontology relations in the skill conditions with 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". Experienced programmers found that the mapping of 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 of 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."

[5]https://github.com/lu-cs-sde/EzSkiROS

For question (b), the majority of the participants reported that EzSkiROS' concise syntax is one potential benefit, which they believe would save coding time and effort. One participant found EzSkiROS' specific error messages useful and answered that "The extra checks allow to know some mistakes 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. No participants expressed concerns about adopting EzSkiROS in their work.

For question (c), one developer acknowledges the benefits of EzSkiROS by answering "In addition to the error reporting, it seems significantly 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 and described SkiROS2 as "It is quite a learning curve and need 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 both readability and writability. The majority of respondents found EzSkiROS to be easy to read and understand, with only one exception. Additionally, respondents found the early bug checking feature of EzSkiROS to be particularly useful in detecting and resolving errors in a timely manner. This suggests that EzSkiROS is an effective tool for increasing code quality and productivity.

V. Conclusion and Future Work

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 demonstrates that embedded DSLs can achieve this goal while simultaneously increasing code maintainability. We expect that our patterns can enable early bug checking in other domains in robotics, such as the construction of compound skills with behaviour trees in SkiROS2 or safety monitoring, without requiring developers to move from their main development language to an external specification language.

References

[1] D. Brugali, A. Agah, B. MacDonald, I. A. Nesnas, and W. D. Smart, "Trends in robot software domain engineering," in *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 3–8.

[2] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 6793–6800.

[3] S. Dragule, S. G. Gonzalo, T. Berger, and P. Pelliccione, “Languages for specifying missions of robotic applications,” in *Software Engineering for Robotics*. Springer, 2021, pp. 377–411.

[4] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, “Robcogen: a code generator for efficient kinematics and dynamics of articulated robots, based on domain specific languages,” *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 36–54, 2016.

[5] I. Ceh, M. Crepinšek, T. Kosar, and M. Mernik, “Ontology driven development of domain-specific languages,” *Computer Science and Information Systems*, vol. 8, no. 2, pp. 317–342, 2011.

[6] A. Nordmann, S. Wrede, and J. Steil, “Modeling of movement control architectures based on motion primitives using domain-specific languages,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2015.

[7] J. P. Buch, J. S. Laursen, L. C. Sørensen, L.-P. Ellekilde, D. Kraft, U. P. Schultz, and H. G. Petersen, “Applying simulation and a domain-specific language for an adaptive action library,” in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer International Publishing, 2014, pp. 86–97.

[8] J. S. Laursen, J. P. Buch, L. C. Sørensen, D. Kraft, H. G. Ellekilde, and U. P. Schultz, “Towards error handling in a dsl for robot assembly tasks,” *arXiv preprint arXiv:1412.4538*, 2014.

[9] E. Coste-Maniere and N. Turro, “The maestro language and its environment: Specification, validation and control of robotic missions,” in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS’97*, vol. 2. IEEE, 1997, pp. 836–841.

[10] L. Kunze, T. Roehm, and M. Beetz, “Towards semantic robot description languages,” in *2011 IEEE International Conference on Robotics and Automation*. IEEE, may 2011.

[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “"ros: an open-source robot operating system",” in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[12] M. Moghadam, D. J. Christensen, D. Brandt, and U. P. Schultz, “Towards python-based domain-specific languages for self-reconfigurable modular robotics research,” *arXiv preprint arXiv:1302.5521*, 2013.

[13] D. Spinellis, “Notable design patterns for domain-specific languages,” *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[14] J.-B. Lamy, “Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies,” *Artificial intelligence in medicine*, vol. 80, pp. 11–28, 2017.

[15] C. Reichenbach, “Software Ticks Need No Specifications,” in *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results Track*. IEEE - Institute of Electrical and Electronics Engineers Inc., 2021, 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE 2021 ; Conference date: 23-05-2021 Through 29-05-2021.

[16] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[17] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[18] H. Balldin and C. Reichenbach, “A Domain-Specific Language for Filtering in Application-Level Gateways,” in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE ’20. New York, NY, USA: ACM, 2020. [Online]. Available: https://doi.org/10.1145/3425898.3426955

[19] S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger, and O. Madsen, “Does your robot have skills?” in *Proceedings of the 43rd international symposium on robotics*. VDE Verlag GMBH, 2012.

[20] V. Krueger, A. Chazoule, M. Crosby, A. Lasnier, M. R. Pedersen, F. Rovida, L. Nalpantidis, R. Petrick, C. Toscano, and G. Veiga, “A vertical and cyber–physical integration of cognitive robots in manufacturing,” *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1114–1127, 2016.

[21] F. Rovida, M. Crosby, D. Holz, A. S. Polydoros, B. Großmann, R. P. Petrick, and V. Krüger, “Skiros—a skill-based robot control platform on top of ros,” in *Robot Operating System (ROS)*. Springer, 2017, pp. 121–160.

[22] M. Mayr, F. Ahmad, K. Chatzilygeroudis, L. Nardi, and V. Krueger, “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.

[23] M. Mayr, C. Hvarfner, K. Chatzilygeroudis, L. Nardi, and V. Krueger, “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.

[24] D. Wuthier, F. Rovida, M. Fumagalli, and V. Krüger, “Productive multitasking for industrial robots,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 12 654–12 661.

[25] M. Mayr, F. Ahmad, K. Chatzilygeroudis, L. Nardi, and V. Krueger, “Combining planning, reasoning and reinforcement learning to solve industrial robot tasks,” *arXiv preprint arXiv:2212.03570*, 2022.

[26] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen, “Robot skills for manufacturing: From concept to industrial deployment,” *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.

[27] A. Barisic, V. Amaral, M. Goulao, and B. Barroca, “How to reach a usable dsl? moving toward a systematic evaluation,” *Electronic Communications of the EASST*, vol. 50, 2012.