CHALMERS
UNIVERSITY OF TECHNOLOGY

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

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]

2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE) | 979-8-3503-0184-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/RoSE59155.2023.00014

***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 bug detection and improving the maintainability of robot code.**

I. 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 [1].

For example, *SkiROS2*[3] [2] 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.

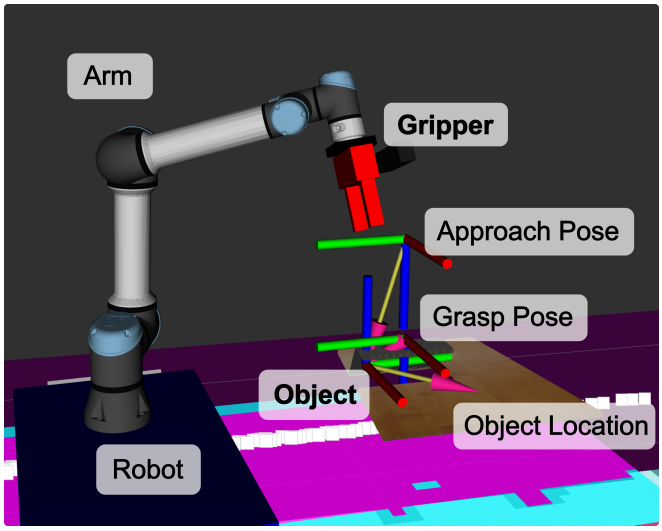

Fig. 1. The robot using a pick skill with a visualization of the necessary parameters. To run this skill, we only need the Gripper and the Object parameters. SkiROS2 can deduce all other necessary parameters through a set of rules in the skill description shown in Listings 3 and 4.

To avoid such errors, we propose to use a Domain-Specific Language (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 [3], and robot knowledge modeling [4]. Nordmann et al. [5] 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;
- A case study of a robotics software SkiROS2 , in which we introduce our DSL EzSkiROS for early detection of type errors and other bugs.

[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

979-8-3503-0184-7/23/$31.00 ©2023 IEEE

II. Related Work

Several studies have explored the use of model-driven approaches for programming robots. Buch et al. [6] 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. [7] 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 [8] 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.

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* (e.g., MAESTRO, SRDL), much of this effort comes from building a language frontend. *Internal* or *embedded DSLs* (as in Buch et al. [6]) 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 [9]. 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, such as freely overloadable infix operators (excluding the type-restricted boolean operators), type annotations (since Python 3.0), and decorator mechanisms [10].

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 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).

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

```
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
```

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).

B. 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 [11].

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* [12] 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 [10] 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:

- 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 [13]: 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 *behavior trees (BTs)* [14] 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:

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}),)
```

```
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 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:[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)

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

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

Here, a *symbolic* parameter is a special kind of mock parameter that we use to record information [15].

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 [16]. 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.

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).

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.

```
1  class Pick(SkillDescription):
2    def createDescription(self):
3      self.addParam("Robot", Element("cora:Robot"),
           ParamTypes.Inferred)
4      self.addParam("Arm", Element("rparts:ArmDevice"),
           ParamTypes.Inferred)
5      self.addParam("StartPose", Element("skiros:
           TransformationPose"), ParamTypes.Inferred)
6      self.addParam("GraspPose", Element("skiros:
           GraspingPose"), ParamTypes.Inferred)
7      self.addParam("ApproachPose", Element("skiros:
           ApproachPose"), ParamTypes.Inferred)
8      self.addParam("Workstation", Element("scalable:
           Workstation"), ParamTypes.Inferred)
9      self.addParam("ObjectLocation", Element("skiros:
           Location"), ParamTypes.Inferred)
10     self.addParam("Object", Element("skiros:Product"),
           ParamTypes.Required)
11     self.addParam("Gripper", Element("rparts:
           GripperEffector"), ParamTypes.Required)
12
13     self.addPreCondition(self.getRelationCond("
           ObjectLocationContainObject", "skiros:contain", "
           ObjectLocation", "Object", True))
14     self.addPreCondition(self.getRelationCond("
           GripperAtStartPose", "skiros:at", "Gripper", "
           StartPose", True))
15     self.addPreCondition(self.getRelationCond("
           NotGripperContainObject", "skiros:contain", "
           Gripper", "Object", False))
16     self.addPreCondition(self.getRelationCond("
           ObjectHasAApproachPose","skiros:hasA", "Object",
           "ApproachPose", True))
17     self.addPreCondition(self.getRelationCond("
           ObjectHasAGraspPose", "skiros:hasA", "Object", "
           GraspPose", True))
18     self.addPreCondition(self.getRelationCond("
           RobotAtWorkstation", "skiros:at", "Robot", "
           Workstation", True))
19     self.addPreCondition(self.getRelationCond("
           WorkstationContainObjectLocation", "skiros:
           contain", "Workstation", "ObjectLocation", True))
20     self.addPostCondition(self.getRelationCond("
           NotGripperAtStartPose", "skiros:at", "Gripper", "
           StartPose", False))
21     self.addPostCondition(self.getRelationCond("
           GripperAtGraspPose", "skiros:at", "Gripper", "
           GraspPose", True))
22     self.addPostCondition(self.getRelationCond("
           NotObjectContainedObjectLocation", "skiros:
           contain", "ObjectLocation", "Object", False))
23     self.addPostCondition(self.getRelationCond("
           GripperContainObject", "skiros:contain", "Gripper
           ", "Object", True))
```

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 [8] 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.

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

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

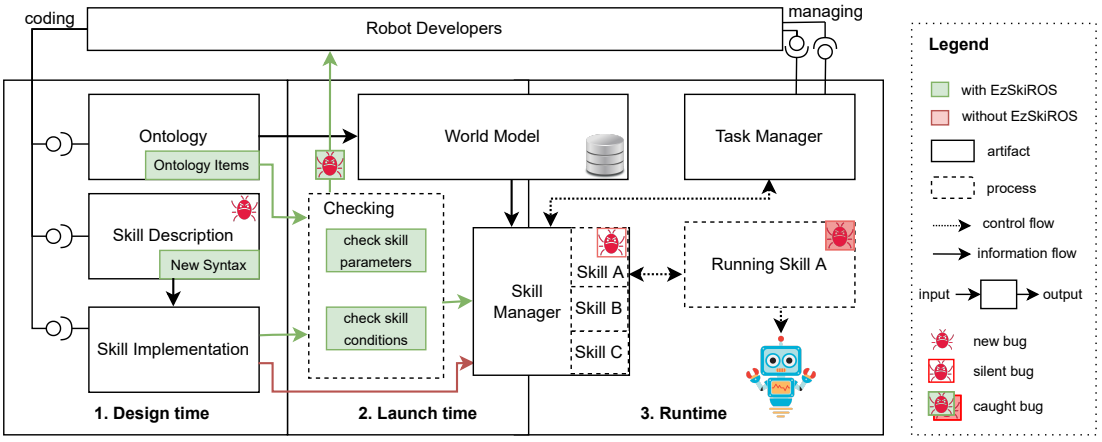

Fig. 2. A diagram with the different components of SkiROS2, their relations and the additions by EzSkiROS. Previously, a bug that has been introduced in a skill description by a developer will often only trigger at runtime. EzSkiROS addresses these costs and risks by finding a wide range of bugs at launch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

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.

```
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)
```

IV. 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 [2]. SkiROS2 is used by several research institutions in the context of industrial robot tasks [17], [18], [19], [20]. It is implemented in Python, on top of the Robot Operating System (ROS) [9] middleware. SkiROS2 uses behavior trees (BTs) [14] formalism to represent procedures. We refer the reader to [14] for a general introduction to BTs, to [21] for a thorough introduction to *SkiROS1* and to [2] for BTs in SkiROS2.

SkiROS2 implements a layered, hybrid control architecture (Fig. 2) to define and execute parametric *skills* for robots [22], [23]. 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 [24]. 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
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.

V. Concise and Verifiable Robot Skill Interface EzSkiROS

We have validated our design patterns in an internal DSL EzSkiROS, which adds **Early Dynamic Checking** (Section III-B) 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. Listing 4 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.

A. EzSkiROS implementation

We follow *owlready2*'s approach to **Domain Language Mapping** in exposing the ontology as Python types and objects. For instance in Listing 4, 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 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 III-B, 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.

B. 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 4) and load it while launching a simulation of a robot shown in Figure 1.

Listing 5 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:

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

```
<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>
```

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

C. 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 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

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

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

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.

VI. 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 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 Knut and Alice Wallenberg Foundation.

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 *RSJ International Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 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] 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.

[5] A. Nordmann, N. Hochgeschwender, D. L. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics," *Journal of Software Engineering in Robotics (JOSER)*, vol. 7, no. 1, pp. 75–99, 2016.

[6] 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*, 2014, pp. 86–97.

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

[8] E. Coste-Maniere and N. 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.

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

[10] 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.

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

[12] 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.

[13] C. Reichenbach, "Software ticks need no specifications," in *ICSE-NIER 2021*. IEEE, 2021, pp. 61–65.

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

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

[16] H. Balldin and C. Reichenbach, "A domain-specific language for filtering in application-level gateways," in *GPCE 2020*, 2020, pp. 111–123.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[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] 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.

[23] 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.

[24] 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.