Controlling Diversity - A Guide To DSLs For Programming Robots

Kai Biermeier

Paderborn University, Warburger Str. 100, 33098 Paderborn, Germany
kaibm@mail.upb.de

Abstract. Robot programming challenges many programmers due to diversity of robotic platforms, tools, programming languages, concepts and application contexts. Moreover, especially portability and suitability to domain experts expectations enforce additional overhead when developing robotic systems. To cope with these kinds of complexity in traditional software engineering, model driven engineering (MDE) has been carried out as pretty useful. Therefore, it nowadays gains also more and more acceptance in robot programming. The core idea is to create a layer of abstraction over the differences encoded in programming languages, platforms etc. This will hides diversity and sequentially improve usability and the work in multidisciplinary fields. In recent years, many researchers which cope with such problems introduce Domain Specific Languages (DSLs) that more or less formally found on the principles of MDE. This work should give an overview of existing approaches and evaluate their strengths and pitfalls. Especially, portability to multiple platforms and the suitability to Domain Experts will be evaluated.

Keywords: Domain Specific Languages · Model-Driven Engineering · Robot Programming · survey

1 Introduction

Diversity within and regarding robots is a major challenge when developing software to deploy on robotic platforms [6] [9] [10]. Globalisation opens the world market to many companies; also those that facilitate robots. Among others, to this end, many components, libraries, platforms and complete robots are available that could be used for developing robotic systems. It is necessary for competitive companies to be aware of most of those items and their possible combinations to create efficient, functional and modern products. Therefore, the developers of a robotic system have to be aware of changes in used components and most often related software. Since it is often not trivial to ship a program written for one robotic system or one component to another, it could be useful to create a language to transform to different platforms. Additionally to advantages regarding platform Independence, DSLs often also provide commonly used functions easily accessible. To this end, commonly used behavior has not to be implemented by hand considering sensor data, state, errors etc.. This improves readability, enables quickly prototyping and preserves the user most often from errors. With enough abstractions, it is even possible that end users can program robots by their own. As shown later when considering the real examples, DSLs can be used for different purposes and on different abstraction levels.

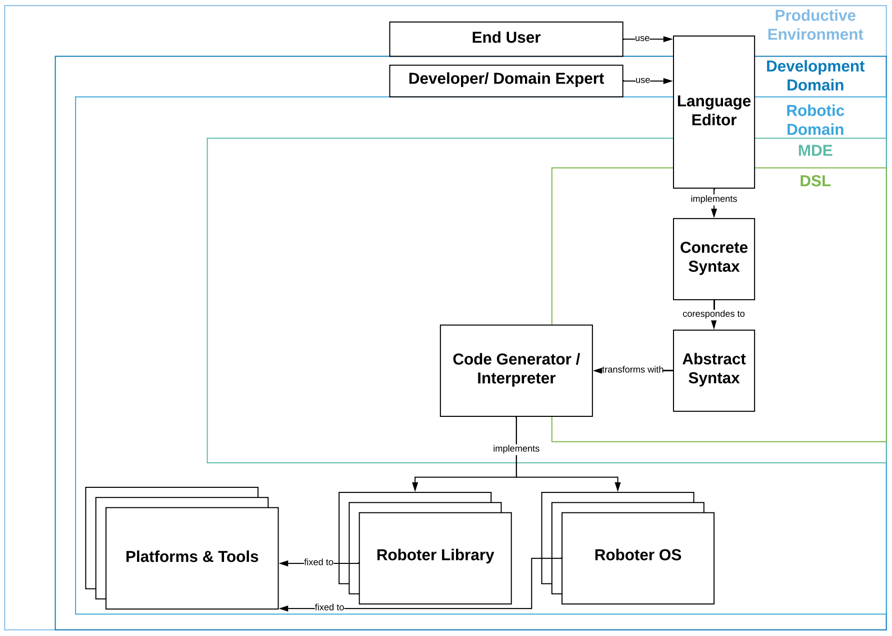


Fig. 1. MDE Architecture with DSL at core

Nevertheless, DSLs are only the surface for a model-driven architecture as visualized in Fig. 1 but at core level. A model can be understood as a discreet representation of a real world object, process or other phenomena. DSLs are mainly composed of the *Concrete Syntax*, *Abstract Syntax*, Semantics and Pragmatics. While the first two are in robotic settings typically implemented directly, Semantics are encoded via a *Code Generator* or *Interpreter*. Pragmatics are often mined after the usage of a DSL. Therefore to our best knowledge, they are not considered properly by robot programming research. For short, the Concrete Syntax is the shape of the language as it can be expressed by the user [13]. The Abstract Syntax is the computable representation of the language. To every instance of a model in the *Concrete Syntax* there corresponds an instance in the *Abstract Syntax*. The specific parts of a DSL are explained in detail in section 3. To embed the DSL in our robotic domain, we need to consider several additional components and references. These are the respective libraries, operating systems and tools and their respective links to the code generator. Hence hard-

ware changes can arise frequently, it is often necessary to use other libraries and maybe also another operating system than the priory planed. To that demand, a robot programming DSL has to be flexible regarding the usage of components, platforms, software and users.
The editor which relies on the Concrete Syntax of the language has to connect all layers of the robotic software system. While it is conceivable that a DSL can aid at the development layers of a robotic system to program it, it is also possible to support the end-user to adapt the system to his/her needs. The following sections are structured in this way: Section 2 will introduce a notion for the model-based engineering paradigm. Section 3 will discuss the definition of a DSL together with its characteristics. Section 4 will describe challenges regarding robot programming. Section 5 will discuss previous work on the development of DSLs for robot programming. In the last section, we will conclude our findings on DSLs for robot programming and present necessary future work for develop this research field.

2 Model Driven Engineering & Domain Specific Languages

When talking about DSLs, one will most probably come across the term of Model Driven Engineering (MDE) as a core principal to realize a DSL. While it is feasible to develop a DSL without applying MDE techniques, one will save much time when utilizing the previously mentioned. In the MDE paradigm, nearly everything is recognized as a model. A model is an abstraction of something in the real world. For example the weather forecast can be understood as a model of the weather. It is precise enough to give a good prediction while it cannot predict the weather completely. If we would build a model with any property of the weather addressed, we would end up with a model that would be infinite large. While on the one hand this is impossible because we do not know any property of the weather, we further have most probably not enough computational power and resources to analyze or construct such a model. Therefore, it is the aim of every modeler to describe a real world phenomena as detailed as possible but also as complex as necessary [3]. To trade wisely is the core to efficient and useful models. To make model engineering more safe regarding requirements to fulfil, a model often corresponds to a so called metamodel. A metamodel is a model which describes the structure of models for a specific domain and their relation. A domain is an abstract concept of the application context. For our weather forecasting model the related Domain could be for example the concept weather in general or weather forecasting. Any degree of detail is possible but the model has to fit the requirements it was build for in a minimalist way.
An additional feature of MDE is its support for modifying and synchronization operations on models. These operations are called model transformations. Model transformations can be categorised on different dimensions [3]. The first dimension is the source and target kind of a model transformation. A model transformation can be model-to-model or model-to-text. While a text can be also con-

sidered as a model the second kind is a sub case of the first. Nevertheless, text is typically harder to transform to another model. Therefore, model-to-model transformations typically describe a transformation to a model formalism which can be again transformed. The second dimension targets the count of models involved and the permitted transformation direction. Unidirectional transformations describe operations that convert a model from one domain to another. In this case, the conversion in the reverse direction is in general not allowed. Bidirectional transformations are the counterpart to unidirectional transformations. They describe transformations which are allowed to execute from the source to the target domain and reverse. Typically, they could also describe transformations between more than two models. One can think of the difference of these modes like the difference between the mathematical implication and equivalence. The next dimension to mention is the direction of abstraction. Vertical model transformation describe operations that change the amount of details in a model. Horizontal model transformations describe operations that change the application domain of a model. Another dimension to recognize is the transformation target place. An in-place model transformation modifies the source model and replaces it afterwards. An out-place model transformation modifies a copy of the source model. The last dimension to mention when talking about model transformations is the source-target domain correspondence. If we are talking about an endogenous model transformation the abstract target model domain is the same as of the source model domain. The reverse holds when talking about an exogenous model transformation.

To clarify the difference between the dimensions of abstraction direction and source-target domain correspondence, it is worth mentioning that a vertical transformation can also make changes on sub domain while an exogenous transformation will always change the higher level domain concept. For example a transformation between multiple model formats of weather forecasting models is vertical but not exogenous. The subdomain as a specific format change while the higher level concept of weather forecasting stays the same. Typically, a system designed under MDE principles will follow the so called Model Driven Architecture [3]. It consists of 4 layers. The first layer is the computational independent model (CIM). This model will make no assumptions regarding the used programming languages or platform. The second layer is the so called platform independent model. This model is formally defined on one language but makes no assumptions regarding the platform it will be executed or used on. The third layer is the layer of platform specific models. Models on this layer are formalised for a specific language and platform. Nevertheless, they are described in a modelling language and not a programming language. Therefore, they are not executable directly but interpretatable. Consequentially, the last layer of the MDA is the layer of Code.

To describe models in a formal and usable way many MDE projects implement a DSL. As described before, a DSL consists of four components: Abstract Syntax, Concrete Syntax, Semantics and Pragmatics [13]. The Abstract Syntax is the computable instance of a model. A model in the Abstract Syntax has typically

to be conform to a metamodel defined for it. A metamodel is often defined with the Eclipse Modelling Framework (EMF) [1]. The meta model restricts a model in its structure. It is similar to an XML schema. The Concrete Syntax is the shape of a language. Therefore, a Concrete Syntax can be graphically or textual. A tool often used to create a textual Concrete Syntax is Xtext [2]. With Xtext one can define a grammar that parse to an Abstract Syntax tree by means of an EMF model. An example for a tool to create a graphical Concrete Syntax is Sirius [3]. The graphlike notation of UML class diagrams is an example of an often used Concrete Syntax of a modelling language (in this case UML class diagrams). As explained before, Semantics as another important part is often encoded as a Code Generator or Interpreter. This is not the only possible way. In general, it is possible to define Semantics in three different ways: as denotational Semantics, as translational Semantics and as operational Semantics. Translational Semantics is the generalization of Code Generation. It means to map the DSL to another language that already has a Semantic defined. Operational Semantics means nothing else than interpreting the model and "executing it". The last kind of Semantics is also the one that is very rarely used. Denotational Semantics means to map the DSL directly to mathematical objects. Mathematical objects have a clear behavior therefore this also defines a Semantic but in a very abstract and most often complex way. Therefore, this method is only used if one needs some very specific guarantees that he can enforce as mathematical equations. The last and often forgotten part of a DSL are the Pragmatics. Pragmatics define a kind of common way to describe something in a DSL. While it is possible to solve a programming task in many ways, there is sometimes a way that is better than others. The same holds for instances of a DSL and its parts. For example in Java you don't have to put brackets around the body of an if-statement if it only contains one command. Nevertheless, it is easier to read and understand if you put those brackets at these positions every time.
A general dimension that differ in used DSLs is the degree of independence [13]. An internal DSL is a language that can be parsed with the language parser of the language it is used in. It can be considered a library that adds domain specific aspects. An external DSL is a language that will be parsed separately to any programming language and therefore do not rely on any specific general purpose language. Some languages also cannot clearly be classified as internal or external because they use the language syntax they are used in, but in a quiet other way than normal programs in the context language would use it. An example for that is the Object Querying language LINQ [4]. These DSLs are called embedded.

[1] https://www.eclipse.org/modeling/emf/
[2] https://www.eclipse.org/Xtext/
[3] https://www.eclipse.org/sirius/overview.html
[4] https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/

3 Robot Programming and related challenges

Robot Programming is a branch of software engineering with complex challenges regarding controlling and managing diversity. Directly related are problems with ease of use and organising multidisciplinary work. An overview of those problems is given in Fig 2.

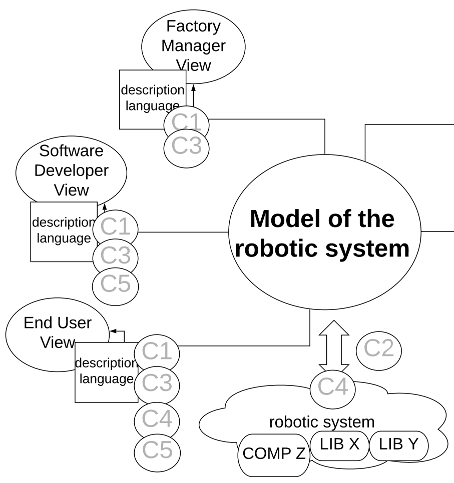


Fig. 2. challenges for robot programming

Typically when inventing a robotic system, multiple stakeholders are involved. These are for example the developers of specific components and libraries, the developers of a complete robot, developers of a robotic swarm and factory managers who want to use robots to produce their products. For example a developer of a component should be able to define which information parameters his component needs and produces. Therefore, a common challenge is:

C1) Every domain expert should be able to define his view on the robotic system without having to consider expert knowledge out of his domain.

When developing a suitable DSL for a robotic component, the developer of the DSL should consider the workflow of the persons who will use the DSLs. The DSL users should be familiar with the syntax or the syntax is at least very close to the syntax domain experts for that component typically use. Moreover, the DSL should be efficient such that it do not produce additional overhead for the users. If a user of the DSL is not faster than a person programming the code directly in C, companies will likely not accept such an approach. If the DSLs are not closer to the formalism robot component experts typically use or think with, they will typically not accept it.

C1a) The description languages should be defined as precise as possible and as complete as necessary.

A formalism suitable to express functionality of a robotic component is typically not enough when consider it to use in industry. Therefore, a trusted Interpreter or generator has to be implemented to make the functionality usable on a real system in a reliable way. Therefore, another challenge arise:

C1b) The description languages for each respective domain (or view) needs a clear Semantic to be usable on a real system.

Since many companies already use robots it is neither recommended nor accepted to implement the languages completely from scratch. They should rely, implement or be compatible to previously used tools to be money and time efficient. It is also possible to implement a transformation from the old formalism to the new. Therefore, we recognize another challenge:

C2) Migration strategies should be considered to simplify the transition from traditionally programmed code to suitable languages domain experts understand.

In a future where everyone owns a robot we can not consider every person a robot expert or rather a programmer for the same. Therefore, it is necessary that people of any intellectual level should be at least able to use basic functionality of the robot. Moreover, the DSL should allow for more complex operations when the knowledge of the user grows.

C3) Domain specific task programming/description should scale on ease and therewith expressiveness

A major problem when writing code for robotic systems is the often platform dependent code involved. Typically, code to control specific robotic components like actuators or sensors are bundled in libraries that follow no common interface regarding other libraries that fulfill the same tasks but for other components. If a company decides to use another component for one of their robots, they do not want to rewrite the code for that component. In such a case it would be useful if a robot programming DSL hides platform specific behavior under a common interface. Therefore, the Code Generator or Interpreter fill the interface with the concrete library code for the component in use. This results in the following challenge:

C4) The robot programming language(s) should be holistic for many components and libraries.

Especially, from the view of the software developer, there are many components to handle and actions to compose in the right order. Therefore, mistakes caused by inattention are very probably. A DSL with context-dependent reference checks included can statically reveal flaws in a program. Since DSLs are not turing-complete, they are more precisely analyzable. For example, the software developer specify the robot to grasp an object while he is still moving. In this case, an analysis on the Abstract Syntax can find that pattern and raise a warning to the developer/programmer.

C5) Robot programming languages should be aware of context-dependent errors.

4 DSLs to suit XR-based robot programming

Regarding our seminar topic DSL are still useful to program a robotic system in a safe and multidisciplinary stakeholder friendly manner. Mixed reality is basically an abstraction of the idea to rebuild or modify the action space of a human-being in the cyberspace. Since it is difficult and less intuitive to write programs using a keyboard in the mixed reality space, the idea to facilitate graphical interactive elements is obvious. This reveals the need for a language that specify its syntax and Semantics. Moreover, the use of general purpose programming languages (GBLs) would be quite to complex to implement with graphical elements. Graphical elements fill more space than text. Additionally, a specification in a GPL needs explicit precise definition, while a DSL can presume implict knowledge that only hold for the domain of robotic systems. Therefore, we propose that a suitable DSL is needed to build an user-friendly mixed-reality robot programming environment.

5 Related Work

To aggregate the state-of-the-art in DSL maturity for robot programming, we present an overview of addressed challenges by different DSLs in Table 1. In the following two subsections, we describe DSL research for robot programming in form of short descriptions of developed approaches and languages. We separate the papers by their Semantic definition into generative and interpretative approaches (i.e. DSLs).

Table 1: An overview of DSLs for robot programming

Approach name	reference-paper	C1	C2	C3	C4	C5
LightRocks	[11]	✓	✗	✓	✓	0
FABRIC	[10]	0	✓	✗	✓	✗
CommonLang	[9]	0	✗	✗	✓	✓
NoName	[2]	✓	✗	✓	✓	✓
NoName2	[8]	0	✓	0	0	✓
vTLS	[4]	0	✓	✗	✓	✓

Robotics API	[6]	✗	✓	✗	✗	✗
Buzz	[7]	✗	✓	✗	✓	✗
NoName3	[1]	0	✓	✓	0	✓
Yampa	[5]	0	✗	✗	✗	✓

5.1 Generative Approaches to Robot Programming with DSLs

The first DSL to mention is the LightRocks DSL [11]. This language relies on an executable subset of UML the so called UML/P (C1b). UML/P together with the MontiCore language framework was developed at RWTH Aachen University. As UML itself, the notation of LightRocks is graphical. Since programming robots encodes behavior, the DSL especially relies on UML/P Statecharts. As it is typcial for many graphical DSLs, LightRocks is external. In Fig 3 an exemplary task is modelled. As the name suggests, the task advise the robot to grasp a specific object, bring it to a target location and screw it at that position. In the example, one should also mention the core element of the DSL at hand: *Skills*. *Skills* are a bunch of elemental robot commands that form one meaningful action. As an example *screwing* is given in Fig 4. As we can model tasks, we can also model skills with LightRocks. The structure of the models is also very similar. Therefore, I will explain the structure of the task specification in more detail and refer to [11] for the detailed explanation of Skills in LightRocks. As explained previously, task are composed of Skills. Skill instances are modelled by rectangles that contain several other rectangles. In the top most rectangle of a skill instance specification, one define the skill pattern that should be instantiated. The second rectangle contains input parameters to that skill instance. For example which object should be grasped. The other rectangles contain start and finish conditions depending on if they are on the right or the left side. Rectangles on the right side are finishing conditions while rectangles on the left side are start conditions. Arrows define the sequence in which the Skills are executed. To actually execute the described behavior, one can generate UML/P statecharts from the DSL and generate from these Java code. To make the program executable for a real robot, one additionally has to implement an interface the java code calls to use sensors, actuators etc. Throught the adapter pattern component code is fast exchangeable and therefore can be used with multiple robotic apis. By dividing the Task description down to: Tasks, Skills and Elemental Actions the approach address the need for discipline specific views (C1, C1a). While the Elemental Action definition can be done by component creators best which can also implement the interface to the actual component api (C4), tasks can be also created by non-programmers hence they only rely on the prior definition of skills (C3). While context-dependent specification errors are not addressed explicitly, one can imagine a typesystem to build upon that DSL to find such flaws (C5). Migration strategies for legacy code are not addressed (not C2). By means of the transformation of the DSL to statecharts the Semantics are clearly defined translational (C1b).

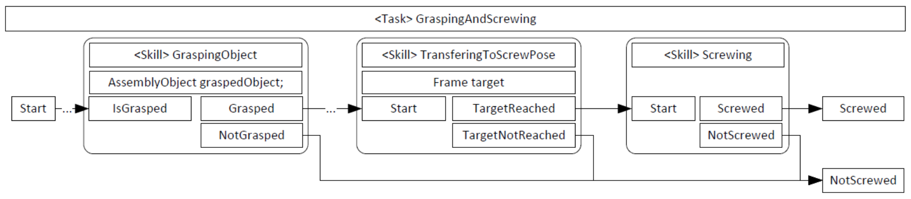


Fig. 3. Task modelling in LightRocks [11]

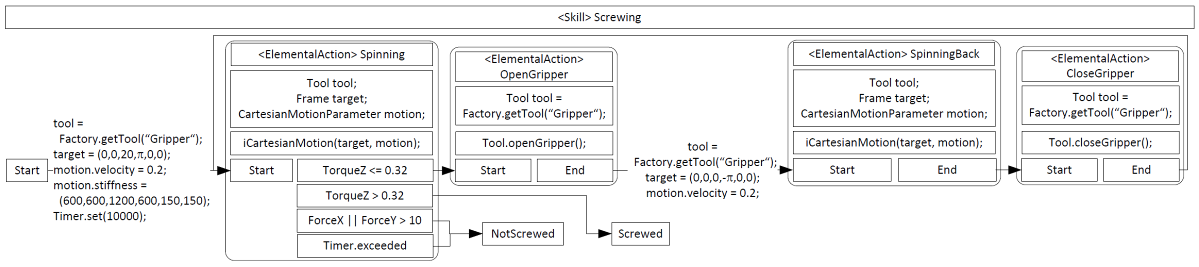

<Skill> Screwing
Start
tool = Factory.getTool("Gripper");
target = (0,0,20,π,0,0);
motion.velocity = 0.2;
motion.stiffness = (600,600,1200,600,150,150);
Timer.set(10000);
<ElementalAction> Spinning
Tool tool;
Frame target;
CartesianMotionParameter motion;
iCartesianMotion(target, motion);
Start
TorqueZ <= 0.32
TorqueZ > 0.32
ForceX || ForceY > 10
Timer.exceeded
NotScrewed
<ElementalAction> OpenGripper
Tool tool = Factory.getTool("Gripper");
Tool.openGripper();
Start
End
Screwed
tool = Factory.getTool("Gripper");
target = (0,0,0,-π,0,0);
motion.velocity = 0.2;
<ElementalAction> SpinningBack
Tool tool;
Frame target;
CartesianMotionParameter motion;
iCartesianMotion(target, motion);
Start
End
<ElementalAction> CloseGripper
Tool tool = Factory.getTool("Gripper");
Tool.closeGripper();
Start
End

Fig. 4. Skill modelling in LightRocks [11]

FABRIC is a framework to enable agent-based robot control via a DSL [10]. It facilitates final state machines to specify higher level functionality. Nevertheless, the presented approach has the claim to integrate a DSL with c++ code that is still necessary on lowest level. The final state machines are finally transformed to c++ code (code generation). The exemplary implementation of the framework is based on orocos [5] an c++ library for robot and machinery control. To implement that framework for semi-automatically generating a subsystem, one has to provide certain inputs as visualized in Fig 5. These are:

1. framework code - code that is common to all subsystems
2. a specification of the subsystem as final state machine encoded in xml. Transitions refer to orocos services. Nevertheless, it is not completely described how the fsm is encoded
3. orocos components as partitial transition functions
4. source code of simple predicates as orocos services to guard initial and terminal state transitions
5. ROS [6] message defintions used in communictation buffers

In this approach, low level details remain on code level and actual behavior can be described via state charts (partially C1). Nevertheless, they use a general purpose execution specification language and therefore are too expressive

[5] https://www.orocos.org/
[6] https://www.ros.org/

to be as close as possible to the application domain (not C1a). Semantics are translational and therefore clear (C1b). Migration is partially addressed since the framework relies on orocos. Therefore, code reuse written for that library is possible but still needs to be refactored for using as transition function or condition. The Framework theoretically can also be implemented for another robot control system and therefore is not restricted to migrate from orocos projects (C2). This fact also implies the holistic nature of the presented framework (C4). The usage process do not differ for user knowledge (C3). Since the modelling language of final state machines is general purpose expressive it is hard to find domain dependent errors (not C5).

CommonLang is a DSL to define robot behavior in terms of tasks [9]. As the

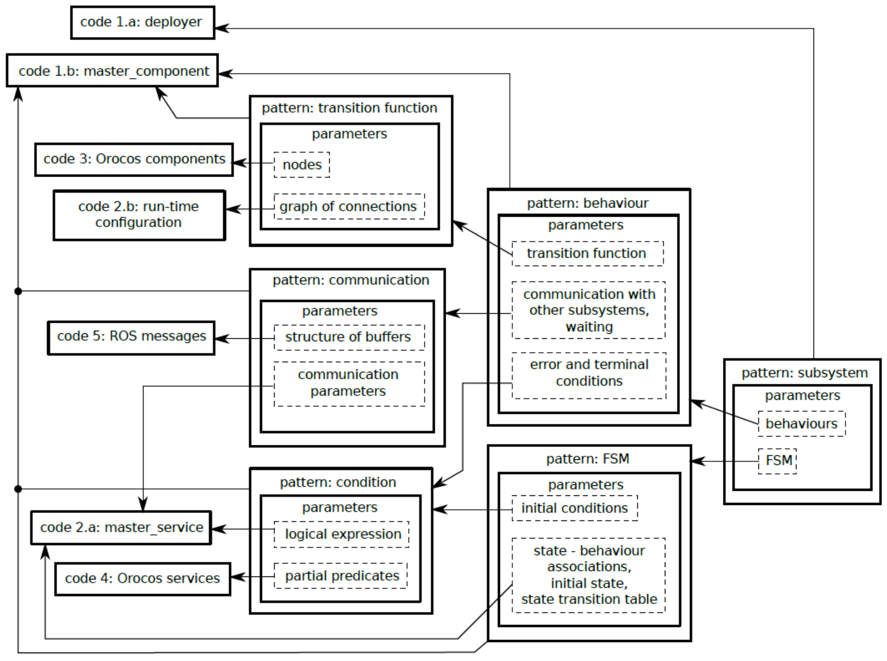
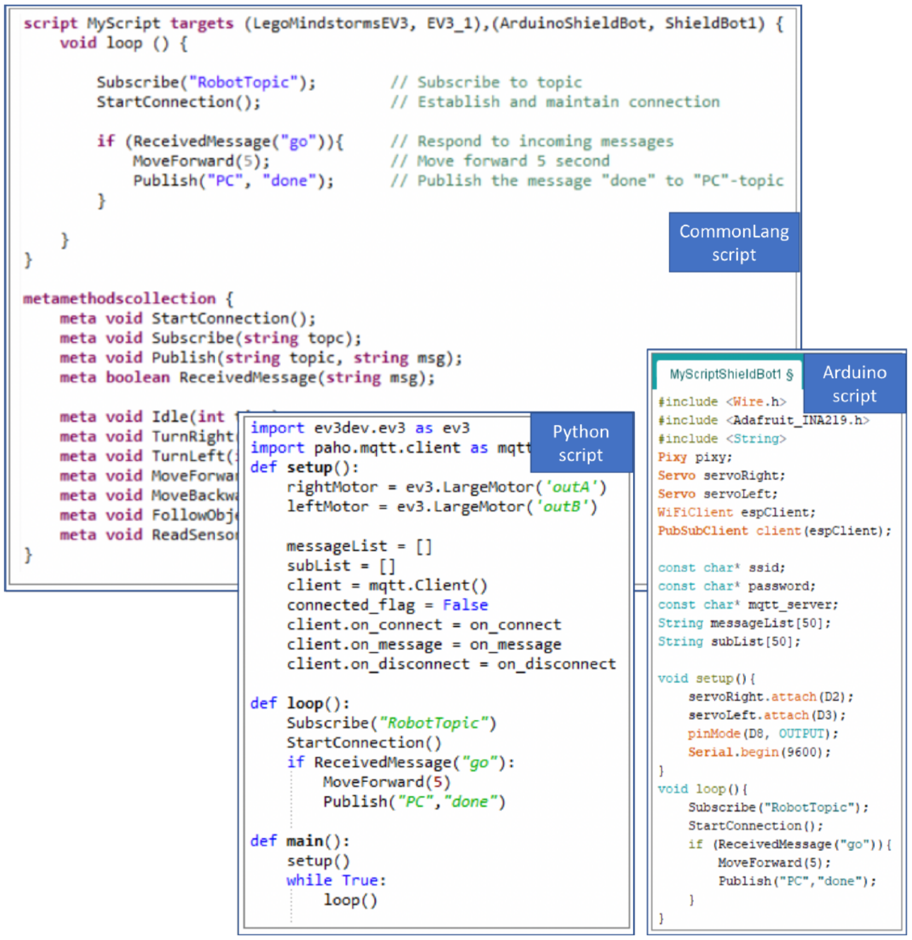


Fig. 5. FABRIC Inputs [10]

name suggests, its purpose is to abstract common task usage from actual implementation (C1a). Therefore, a transfer to different components and platforms is easy possible through the Code Generator which also define the Semantic of the DSL (C4, C1b). To define the set of common operations the metamodel is facilitated. Although, it is not mentioned, this further enables for context dependent error recognition (C5). Support for multiple roles or intellectual levels and suitable abstractions are not given (not C1, not C3). Additionally, support for migration is also not addressed (not C2). The DSL and example translations to python and C are given in Fig 6. The tasks are defined in Java-like style which is why one has additionally to define which common operations should be facilitated for defining the tasks. These common operations are listed in the *metamethodscollection*. The DSL itself builds on top of the xtext language workbench and therefore uses the EMF ecosystem. To map metamethods to actual implementation using a target library XML files are utilized. The interested reader is referred to [9]



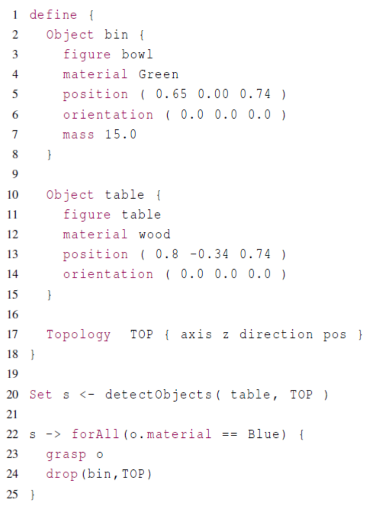
```
script MyScript targets (LegoMindstormsEV3, EV3_1),(ArduinoShieldBot, ShieldBot1) {
    void loop () {

        Subscribe("RobotTopic");         // Subscribe to topic
        StartConnection();               // Establish and maintain connection

        if (ReceivedMessage("go")){      // Respond to incoming messages
            MoveForward(5);              // Move forward 5 second
            Publish("PC", "done");       // Publish the message "done" to "PC"-topic
        }

    }
}

metamethodscollection {
    meta void StartConnection();
    meta void Subscribe(string topc);
    meta void Publish(string topic, string msg);
    meta boolean ReceivedMessage(string msg);

    meta void Idle(int
    meta void TurnRight
    meta void TurnLeft(
    meta void MoveForwa
    meta void MoveBackw
    meta void FollowObj
    meta void ReadSenso
}
```

```python
import ev3dev.ev3 as ev3
import paho.mqtt.client as mqtt
def setup():
    rightMotor = ev3.LargeMotor('outA')
    leftMotor = ev3.LargeMotor('outB')

    messageList = []
    subList = []
    client = mqtt.Client()
    connected_flag = False
    client.on_connect = on_connect
    client.on_message = on_message
    client.on_disconnect = on_disconnect

def loop():
    Subscribe("RobotTopic")
    StartConnection()
    if ReceivedMessage("go"):
        MoveForward(5)
        Publish("PC","done")

def main():
    setup()
    while True:
        loop()
```

```c
#include <Wire.h>
#include <Adafruit_INA219.h>
#include <String>
Pixy pixy;
Servo servoRight;
Servo servoLeft;
WiFiClient espClient;
PubSubClient client(espClient);

const char* ssid;
const char* password;
const char* mqtt_server;
String messageList[50];
String subList[50];

void setup(){
    servoRight.attach(D2);
    servoLeft.attach(D3);
    pinMode(D8, OUTPUT);
    Serial.begin(9600);
}
void loop(){
    Subscribe("RobotTopic");
    StartConnection();
    if (ReceivedMessage("go")){
        MoveForward(5);
        Publish("PC","done");
    }
}
```

Fig. 6. CommonLang [9]

In ”Towards Easy Robot Programming: Using DSLs, Code Generators and Software Product Lines” the authors present a DSL to enable Domain experts and end-user to program robots [2]. Moreover the authors facilitate software product line techniques to ease the exchange of soft- and hardware components. Referring to [2], the approach concretely address the need for powerful functions and abstraction to replace platform specific low level code and technical details like sensor data (C1a). It further enables support for various hard and software (C4) and finally runtime reconfiguration. Runtime reconfiguration enables the DSL to scale on complexity with more or less rich configurations (C3). Migration from previously written code is not mentioned (not C2). While not mentioned directly, through the usage of Xtext this approach can address context dependent errors and warn the user regarding their occurrence (C5). After the DSL were parsed by xtext, the related EMF model (Abstract Syntax tree) will be issued to be transformed to c++ code by acceleo model-to-text engine. Therefore, the approach gain a translational Semantic in terms of c++ (C1b). The DSL only allows to program tasks in one way and therefore not enables domain experts of different domains to program the system in their prefered way. An example of an instance of the mentioned DSL is provided in Fig. 7. It describes the task to grasp all blue objects from a table. The product line approach is implemented by utilizing feature models to describe different configurations. For example, one can configure the concrete path planning algorithm to use. For more details we refer to [2].



```
define {
  Object bin {
    figure bowl
    material Green
    position ( 0.65 0.00 0.74 )
    orientation ( 0.0 0.0 0.0 )
    mass 15.0
  }

  Object table {
    figure table
    material wood
    position ( 0.8 -0.34 0.74 )
    orientation ( 0.0 0.0 0.0 )
  }

  Topology  TOP { axis z direction pos }
}

Set s <- detectObjects( table, TOP )

s -> forAll(o.material == Blue) {
  grasp o
  drop(bin,TOP)
}
```

Fig. 7. DSL (with no name) [2]

In [8], the authors present a graphical DSL to enable robot control also for domain experts rather than programmers (C1). The approach especially focuses on parallel execution of multiple subtasks. The DSL allows to define dependencies between subtask such that a concrete sequence can be checked for violating such order constraints. Violations of order constraints can be considered as one kind of context dependent errors (C5). Additionally, in [8] the authors name the need for a DSL to assist the user in reducing failures (C5). The DSL uses an own meta model with limited elements and therefore limited expressiveness. The expressiveness is limited by the DSL configuration which declares which methods and components are available (C1a). Hence the Semantic is defined by the order of execution, which is explicitly defined as a sequence in this approach the Semantic is clear to a big extend. Only the injected code via code templates may affect execution in previously undesirable ways which can harm a clear Semantic (partly C1b). Because we are able to define actions in arbitrary languages, we should be able to easy migrate previously written code (C2). Moreover, we can distinguish between two abstraction levels the code level and the control flow level (partly C3). To consider the approach to completely address C3 it would need to address more levels of abstraction. Since it is possible to write action in arbitrary programming languages, the DSL is holistic in terms of usage of libraries and components (C4). An example DSL configuration is given in Fig. 8. The actual DSL only visualize a sequence of actions. Therefore, the heard of the approach is the DSL configuration. In the example configuration, we specify a cleaning robot. The configuration encodes different self-explaining actions and actual technical components of the robot. The reader should note the *ForbittenParallelActionTypes* definition of the *MoveFwd* action. This constraints the developer of actual program to only run the *MoveFwd* action or the *Discharge* action. Not both at the same time.

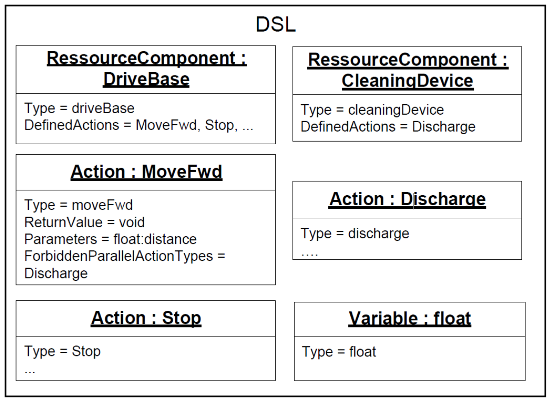
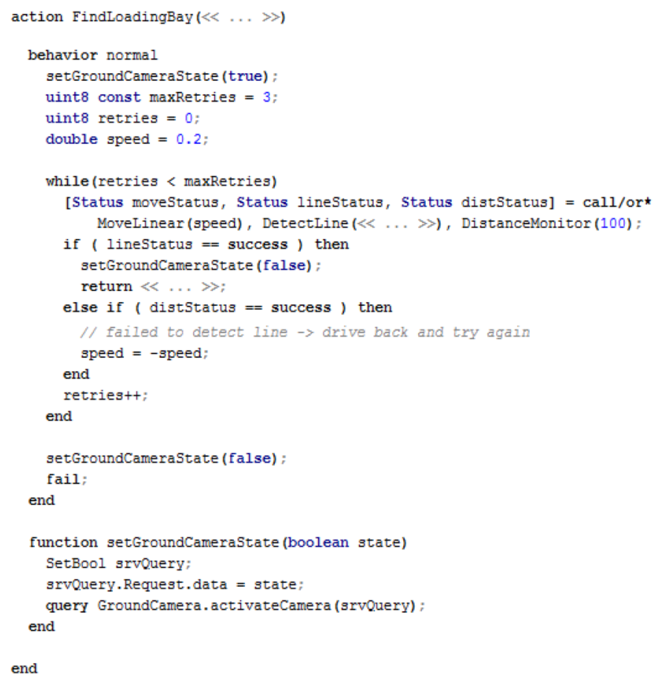

DSL
RessourceComponent : DriveBase
Type = driveBase
DefinedActions = MoveFwd, Stop, ...
RessourceComponent : CleaningDevice
Type = cleaningDevice
DefinedActions = Discharge
Action : MoveFwd
Type = moveFwd
ReturnValue = void
Parameters = float:distance
ForbiddenParallelActionTypes = Discharge
Action : Discharge
Type = discharge
....
Action : Stop
Type = Stop
...
Variable : float
Type = float

Fig. 8. DSL (with no name) [8]

vTLS is a DSL for defining verifiable robot programs [4]. The authors build a model-to-model-transformation that maps the internal model to a Promela model. The Promela model can be checked for satisfaction of a set of predefined properties by the model checker Spin. To evaluate the approach, the authors compare the DSL to the prior approach that provides a C++ library and implements the same task tree Semantic. The developers of the aforementioned approach confirm that the DSL code is better readable and the approach reduced the code size by more than 50%. The approach was also evaluated regarding the performance of the state space analysis. The runtime is exponential in relation to the embedded (ROS) components but this is reasonable to the fact that the state space itself is exponential in general for programs. Another advantage of the DSL is its natural support for concurrency. For more details, we refer to [4]. Hence the approach focus on verification of robot programs, it contains no further layers to enable domain experts to encode their knowledge (not C1). The target of the approach is verifiability of program and additionally code size reduction compared to the pure c++ approach (C1a). The language is simple enough and at the same time verifiable. Because of the transformation to a verifiable model, the Semantics can be considered as formally found (C1b). Since the DSL found on ROS nodes and services prior written code for ROS can be reused (C2). As only the software development layer of the product lifecycle is addressed the approach moreover do not scale on ease and expressiveness (not C3). The skill layer ROS is widely used and moreover substitutable with any suitable task tree based language therefore the approach is holistic for multiple components and libraries (C4). As verifiability is the major purpose of vTLS context-dependent errors will be found and can be fixed (C5). In Fig. 9 we present an excerpt from an example program in vTLS. It supports c++ types and a subset of keywords. Tasks will be specified by action which found on behavior. Behaviors form the skill layer that typically rely on ROS architecture. For further explanation we refer to [4]

5.2 Interpretative Approaches to Robot Programming with DSLs

The Robotics API is an abstraction Layer on top of general purpose programming languages (by now Java and C# [6]). The paper Ön reverse-engineering the KUKA Robot Languagëdeal with challenges of migrating DSLs that were developed to program specific robots to a generalized api called Robotics API [6]. The paper exemplary explains the process of migration from the KUKA Robot Language (KRL). The paper does not introduce a new DSL but a sematic definition of KRL. The paper authors emphasis that migrating DSLs to a common basis is a major challenge in the domain of robot programming. Therefore, programs written for one platform utilizing a DSL can be interpreted on robots for which that language was not developed. While the approach should be mainly categorised as interpretative, the authors also mention an approach to translate the DSL Abstract Syntax tree to byte code which can be interpreted even faster. Translation to bytecode can be seen as a code generation process. Therefore, the overall approach has generative aspects. An exemplary mapping of KRL to an AST and finally to bytecode to interpret is given in Fig. 10. The surface of the Interpreter implements the robotics api and translate the program to the Realtime Primitive Interface (RPI). For detailed explanation of the robotics api approach we refer to [6]. For an explanation of the Realtime Primitive Interface we refer to [12]. Hence this approach focus on the migration of multiple DSLs

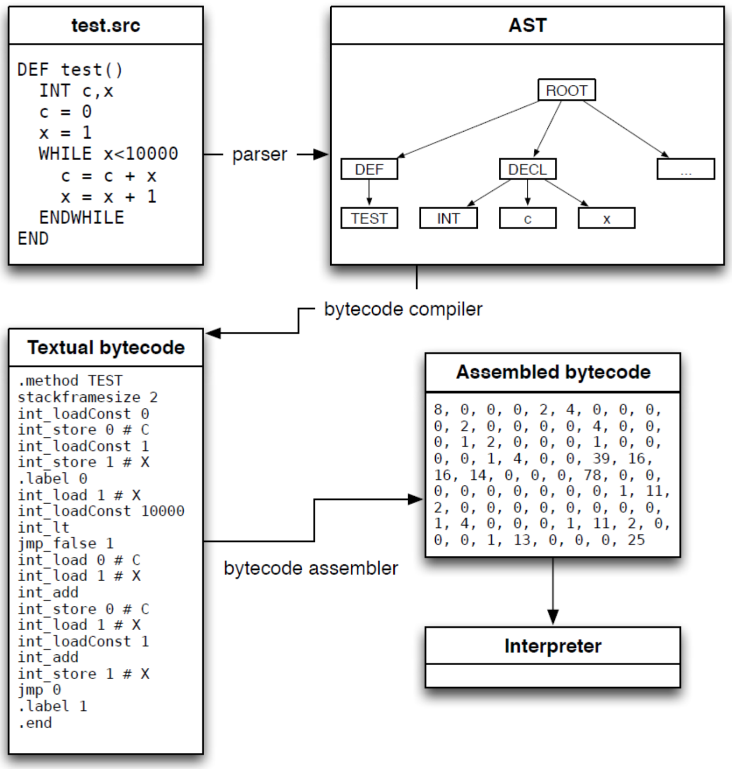


```
action FindLoadingBay(<< ... >>)

  behavior normal
    setGroundCameraState(true);
    uint8 const maxRetries = 3;
    uint8 retries = 0;
    double speed = 0.2;

    while(retries < maxRetries)
      [Status moveStatus, Status lineStatus, Status distStatus] = call/or*
          MoveLinear(speed), DetectLine(<< ... >>), DistanceMonitor(100);
      if ( lineStatus == success ) then
        setGroundCameraState(false);
        return << ... >>;
      else if ( distStatus == success ) then
        // failed to detect line -> drive back and try again
        speed = -speed;
      end
      retries++;
    end

    setGroundCameraState(false);
    fail;
  end

  function setGroundCameraState(boolean state)
    SetBool srvQuery;
    srvQuery.Request.data = state;
    query GroundCamera.activateCamera(srvQuery);
  end

end
```

Fig. 9. vTLS [4]

to a common ground and not on defining a DSL based approach to robot programming, it only addresses C2.

Buzz is a DSL to program robotic swarm behavior with primitive function for common swarm behavior[7]. The DSL execution Semantic founds on step-wise interpretation of the task description. In this way all robots stay synchronized on the task. Moreover, the language utilize a mixed imperative and functional syntax which helps users in reader that code and easy combine multiple scripts. Also mentioned in [7] the DSL was build to support multiple platforms and languages (C4). The language only allow programming on software developer abstraction level (not C1). The language seams to be very ambitious and aims to be domain spanning usable for robotic swarm programming. Therefore, it is to expressive to simplify robot task description from a domain expert point of view (not C1a). It is even possible to use C code in lambda expressions loaded into the Interpreter. It would be more elegant to focus on a specific sub domain of swarm robotics to allow easy and less loaded task descriptions. Additionally, the buzz language runtime founds on a self written Interpreter which formal Semantic is not clearly described (not C1b). The authors only state that the runtime can found on popular approaches such as ROS or OROCOS (C4). Embedding of lambda expressions and the foundation on popular approaches to robot programming serves well for migration of legacy code (C2). As discussed before, the DSL only allows to write programs on one abstraction level/level of



```
DEF test()
  INT c,x
  c = 0
  x = 1
  WHILE x<10000
    c = c + x
    x = x + 1
  ENDWHILE
END
```

```
.method TEST
stackframesize 2
int_loadConst 0
int_store 0 # C
int_loadConst 1
int_store 1 # X
.label 0
int_load 1 # X
int_loadConst 10000
int_lt
jmp_false 1
int_load 0 # C
int_load 1 # X
int_add
int_store 0 # C
int_load 1 # X
int_loadConst 1
int_add
int_store 1 # X
jmp 0
.label 1
.end
```

```
8, 0, 0, 0, 2, 4, 0, 0, 0,
0, 2, 0, 0, 0, 0, 4, 0, 0,
0, 1, 2, 0, 0, 0, 1, 0, 0,
0, 0, 1, 4, 0, 0, 39, 16,
16, 14, 0, 0, 0, 78, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 11,
2, 0, 0, 0, 0, 0, 0, 0, 0,
1, 4, 0, 0, 0, 1, 11, 2, 0,
0, 0, 1, 13, 0, 0, 0, 25
```

Fig. 10. Robotics API And KRL [6]

ease (not C3). Domain specific configuration error handling is not mentioned in the paper. Nevertheless, the authors name debugging as a part of possible future work. Therefore we assume C5 to be not addressed by now but planed for future work. In Fig. 11 we present a code example for the buzz DSL. The language is close to common programming languages. Also comments are added for different sections in the example. Therefore, we assume that no further explanation on the code example is needed. For further detailed information consider [7]. In [1] the authors present an internal DSL for defining ROS programs. The DSL approach addresses the need for refactoring of legacy programs written for ROS. The DSL offers the language mechanisms to wrap legacy code such that it can form new behavior, conforms to a new interface or modify its in and output parameters (C2). To this means, it allows also for context dependent error checks at runtime (C5). Additionally, as it is interpreted, the code can be manipulated at runtime. This further improves the ease of developing and debugging a program. This approach only addresses software developer issues and does not further involve domain experts (not C1). Its internal nature also implies a favor on completeness

```
# Group identifiers
AERIAL   = 1
GROUND   = 2
GRIPPERS = 4
# Task for aerial robots
function aerial_task() { ... }
# Task for ground-based gripper robots
function ground_gripper_task() { ... }
# Create swarm of robots with 'fly_to' symbol
aerial = swarm.create(AERIAL)
aerial.select(fly_to)
# Create swarm of robots with 'set_wheels' symbol
ground = swarm.create(GROUND)
ground.select(set_wheels)
# Create swarm of robots with 'grip' symbol
grippers = swarm.create(GRIPPERS)
grippers.select(grip)
# Assign task to aerial robots
aerial.exec(aerial_task)
# Assign task to ground-based gripper robots
ground_grippers = swarm.intersection(GROUND + GRIPPERS, ground, grippers)
ground_grippers.exec(ground_gripper_task)
```

Fig. 11. Robotics API And KRL [7]

of the DSL rather than precision. Nevertheless, the examples imply a use that needs less completeness of the language to define meaningful programs while on the other side only necessary information like ROS topics, angles etc. will be defined (partially C1a). By its internal nature its Semantic is clear by means of the python language whereon it founds (C1b). The users are able to define low level code or define composed functionality. While the first requires good programming skill and maybe knowledge about the hardware the second is abstract and can be very easy (partially C3). Nevertheless, this seems more a side effect of the architecture than an intended feature. Since the architecture found on ROS and ROS is very commonly used, the DSL is comprehensive regarding the use of components and libraries. Nevertheless, the approach does not target components and architectures that do not rely on ROS (partially C4). An excerpt of an example program written by means of that DSL is presented in Fig. 12. In this example, a mapping from one topic to another is defined. For more detailed explanations and examples we refer to [1]. Yampa is an internal DSL

```
{
nd.new.subscribe(topic = "/turtle1/pose", handler =
     showPose, msgType = Pose)
       .publish(topic = "/turtle1/cmd_vel", msgType = Twist
            )
}
```

Fig. 12. ROS DSL example [1]

to Haskell which enables writing programs for mobile robots [5]. Since Haskell is a functional programming language, Yampa also founds on functional programming and so called Monads a mathematical structure in the field of category theory. By its descriptive nature, properties of the language are easier provable than for algorithmic programming language like Java, C++ etc.. The syntax behaves as expected for functional programs in Haskell. A complete explanation of the complete language or Semantics is unfortunately out of the scope of this paper. Since the language is an internal language to the Haskell programming language, it is typically only suitable for programmers familiar with functional programming paradigm (not C1). To the same reason the language seems not as precise as possible but either complete (not C1a). The descriptive nature of Haskell is well found on category theory, therefore, the Semantics are clear (C1b) and the programs are well analysable (C5). As discussed before, programming nowadays typically happens with libraries like ROS or OROCOS. We assume that one can not (re)use code from such code bases because scala do not provide access to these platforms through known libraries (not C2 and C4). General purpose programming like programming in scala do not scale well on ease and there is no hint that especially scampa do (not C3).

6 Conclusion & Future Work

We saw that all of our challenges are addressed by different approaches but not a single approach exists that fulfil every challenge. Additionally, some approaches are quite better in addressing a challenge than others. Although, we did not and even can not address every domain specific language that addresses robot programming we give an overview of the different kinds of available approaches. Nevertheless, in practice ROS, OROCOS and other programming libraries still dominate the domain of robot programming in practice. However, as we also saw in software engineering, the trend slightly will move towards simpler and more domain specific languages such that robot programming will leave the area of low level details programming. This will finally enable end users to program their robots on suitable abstraction levels and even allow to port previously written programs to other robots.
For the future of robot programming, we emphasis the need of better support for different domain experts involved hence this is only slightly addressed nowadays (see Fig. 1). This is what finally will persuade end users and domain experts to use these tools productively without the need of an actual programmer. Additionally, if the DSL also supports electrical engineers, mechanical engineers etc. for describing their components the compatibility to different approaches will grow further. This will again extend the range of such a DSL approach very much. Verification ability in a formal way is only directly addressed by one of the analysed approaches. In realtime environments the runtime is often crucial for the success of an operation. Therefore, analysing whether a program will violate an runtime timing constraint can be very useful and is also necessary in some domains. Therefore, research in that direction should be also considered as quite important.

Additionally to technical depts and organizational needs, we remind that many approaches still lack research on the field of Pragmatics for robot programming and usability studies to actually conduce which tools and methodologies domain experts really use and understand. Since the basis seems to be well developed, we are curious of future development and maybe rather the actual transition from GPLs to DSLs in many areas of robot programming.

References

1. Adam, S., Schultz, U.P.: Towards interactive, incremental programming of ros nodes. arXiv preprint arXiv:1412.4714 (2014)
2. Baumgartl, J., Buchmann, T., Henrich, D., Westfechtel, B.: Towards easy robot programming-using dsls, code generators and software product lines. In: ICSOFT. pp. 548–554 (2013)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice, second edition. Synthesis Lectures on Software Engineering **3**(1), 1–207 (2017). https://doi.org/10.2200/S00751ED2V01Y201701SWE004, https://doi.org/10.2200/S00751ED2V01Y201701SWE004
4. Heinzemann, C., Lange, R.: vtsl-a formally verifiable dsl for specifying robot tasks. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 8308–8314. IEEE (2018)
5. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: International School on Advanced Functional Programming. pp. 159–187. Springer (2002)
6. Mühe, H., Angerer, A., Hoffmann, A., Reif, W.: On reverse-engineering the kuka robot language. arXiv preprint arXiv:1009.5004 (2010)
7. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3794–3800. IEEE (2016)
8. Reckhaus, M., Hochgeschwender, N., Ploeger, P.G., Kraetzschmar, G.K.: A platform-independent programming environment for robot control. arXiv preprint arXiv:1010.0886 (2010)
9. Rutle, A., Backer, J., Foldøy, K., Bye, R.T.: Commonlang: a dsl for defining robot tasks (2018)
10. Seredyński, D., Winiarski, T., Zieliński, C.: Fabric: Framework for agent-based robot control systems. In: 2019 12th International Workshop on Robot Motion and Control (RoMoCo). pp. 215–222. IEEE (2019)
11. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: A new skill based robot programming language using uml/p statecharts. In: 2013 IEEE International Conference on Robotics and Automation. pp. 461–466. IEEE (2013)
12. Vistein, M., Angerer, A., Hoffmann, A., Schierl, A., Reif, W.: Interfacing industrial robots using realtime primitives. In: 2010 IEEE International Conference on Automation and Logistics. pp. 468–473. IEEE (2010)
13. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013), http://www.dslbook.org