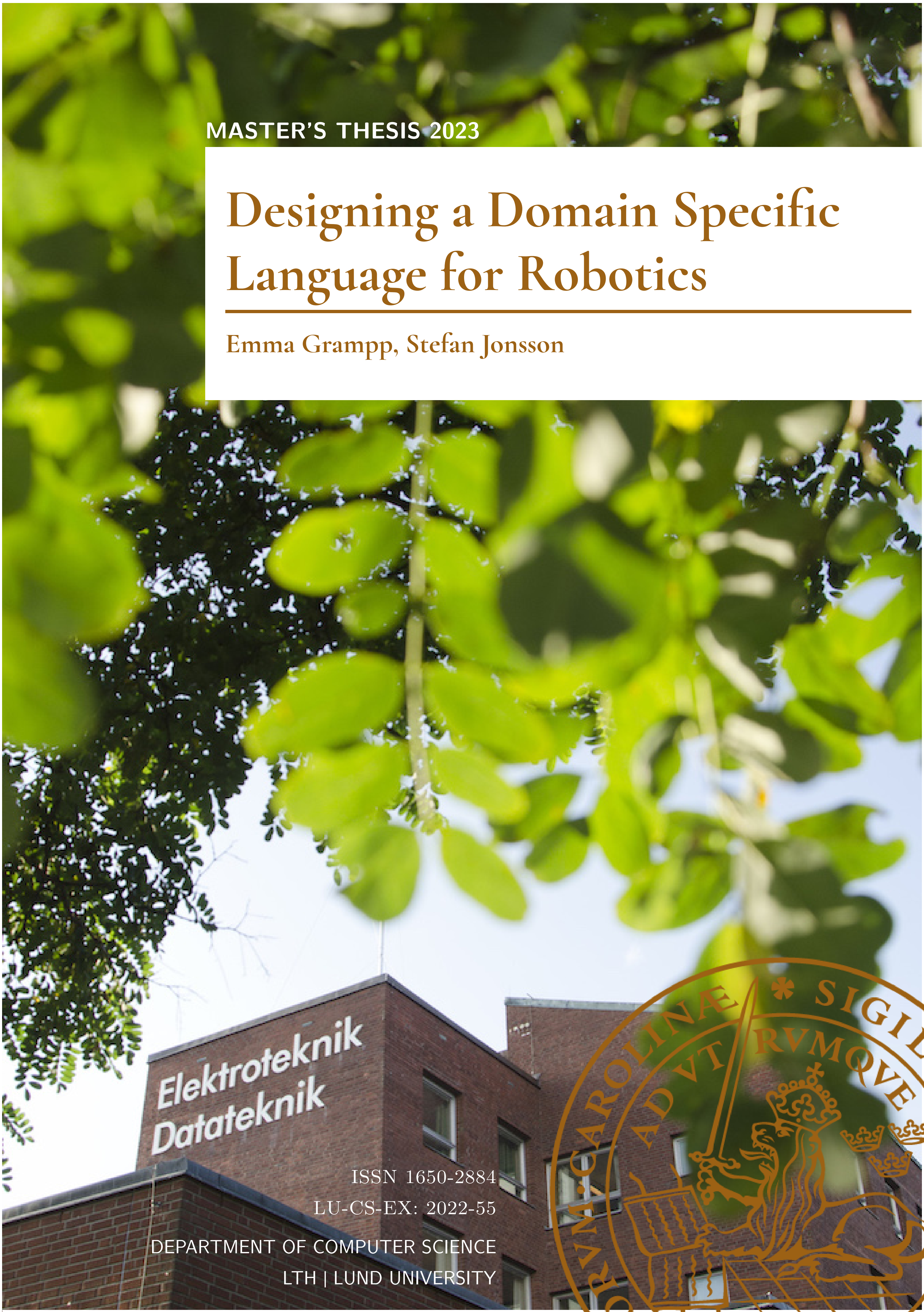

MASTER'S THESIS 2023

Designing a Domain Specific Language for Robotics

Emma Grampp, Stefan Jonsson

ISSN 1650-2884
LU-CS-EX: 2022-55

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

LU-CS-EX: 2022-55

Designing a Domain Specific Language for Robotics

Emma Grampp, Stefan Jonsson

Designing a Domain Specific Language for Robotics

Emma Grampp
`emma@grampp.se`

Stefan Jonsson
`nat14sjo.student.lth.se`

March 2, 2023

Master’s thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Elin Anna Topp, `Elin_Anna.Topp(at)cs.lth.se`
Christoph Reichenbach, `christoph.reichenbach@cs.lth.se`

Examiner: Jesper Öqvist

Abstract

As industrial robots get cheaper it becomes more important that they are also easy to program. For this purpose we have created a domain specific language that is used to describe movements and interactions of robots with multiple moveable parts. The language is inspired by an earlier language by M. Stenmark, but adds abstraction over components and procedures. In order to evaluate the language we have tried to translate a collection of programs from Stenmarks language to our language for comparison. We have also had a roboticist solve a collection of problems with the language and fill in a form about the experience. The language has some limitations (the main one being the lack of graphical user interface) but we hope that it can serve as a basis for further development and research.

Keywords: robot, robot arm, robotics, DSL, JastAdd, ROS

Acknowledgements

We would like to thank our supervisors, Elin and Christoph, for their boundless patience, and Alexander Dürr for his invaluable technical assistance to get us started.

Contents

Chapter 1
Introduction

Industrial robots are daily finding new uses across many sectors of industry. As they get cheaper it becomes more important that they are easy to program, as they are going to be used by smaller companies that might not be able to hire expert programmers. We present a Domain Specific Language (DSL) that will make robot programming more intuitive.

Our language is inspired by Dr Stenmark's language, as described in [9]. The main difference between our language and Stenmark's is that the latter is graphical whereas ours is textual (which is the main purpose behind our language). Another difference is that in her language programs are lists of commands while our language provides some control structures and abstractions to avoid code duplication.

We have tried to make our language as minimal as possible while making it easy to extend and improve upon. Our wish is that it will serve as a base for further development.

We have used as examples two robots when creating our design: Franka Emika's Panda and ABB's YuMi. The Panda robot is a single arm robot. The arm itself has seven joints and at the end of the arm has a two-fingered gripper. The YuMi robot has two seven joint arms with grippers. The fact that it has two arms makes it more interesting to program since it inevitably involves simultaneous movements and interaction between robot parts.

1.1 Requirements

Our task is to create a Domain Specific Language (DSL) for programming industrial robots. The language should provide the basic functionality found in Dr Stenmark's framework. Furthermore it should provide some abstraction to avoid code duplication. It should run on the Panda and/or YuMi robots and preferably it should be easy to extend it to work for any robot. It should also support live programming. As dual-arm robots (such as the YuMi) must often use their hands together, a semantic for synchronization and interaction between different robot parts is an important feature to have. Such semantics might also be used in the future to facilitate interactions between multiple robots controlled by the same program.

1.2 Stakeholders

This work was done for the Robotics and Semantic Systems (RSS) and Software Development and Environments (SDE) research groups of the computer science department at LTH. The former is interested in a language as infrastructure for future tools. The latter is interested in the semantics of robot control as they pertain to language development.

1.3 Research Questions

1. Does our language provide the functionality that Stenmark's language provides?
2. Which types of abstraction are needed to facilitate easy industrial robot programming while eliminating the need for code duplication? Does our language provide the necessary abstraction?
3. Can our language be used as a platform on which to construct more tools, such as graphical and interactive programming systems?

Chapter 2
Background

In this chapter we describe some of the already existing robot programming languages with their features and limitations. We also give a brief description of the tools and methods used to build our language.

2.1 Existing Robot Languages

Many robot programming languages have been created before ours. Here we are going to focus on RAPID and Dr Stenmark's programming language.

2.1.1 Rapid

Rapid is a language created by ABB in order to program their industrial robots. The language is imperative and allows a programmer to define their own procedures which can be called later in the program. There are also if statements, for loops and while loops, used to control program flow. Positions and orientation are described by vectors.

The main limitation of Rapid is that it is proprietary to ABB robots and cannot readily be adapted to other manufacturers' robots.

2.1.2 Stenmark's Language

Dr. Stenmark's language is visual rather than textual, and is designed for the YuMi robot only. Programming is done through a drag-and-drop interface to two synchronized threads. When the program is run, the arms go through their lists of actions and perform them sequentially. Synchronization between arms is done through sync actions. When an arm reaches a sync action, it waits until the other arm reaches the corresponding sync action before it continues. Synchronization can also be done with special sync-moves. In that case one arm moves, while the other arm maintains a fixed position relative to the moving arm. The language compiles to RAPID. This is a limitation as we are still relying on proprietary software that only works on ABB robots. Another limitation is that it does not provide enough abstraction to avoid code duplication, which would make complex programs too long to be easily understood. Our language provides a subset of the instructions provided by Stenmark's language but adds abstraction in the form of loops, procedures, abstract components and spatial transformations.

2.1.3 Others

There are of course other such languages in existance. Simmons' and Apfelbaum's TDL[8] is perhaps the most similar, being an extension of C++ and thus, like ours, a straightforward imperative language. Their language has a greater set of synchronization primitives. It predates ROS, which would make it difficult to integrate with contemporary research robots. LightRocks[10] by Thomas et al. is another take on the same idea. It is built on top of the MontiCore language workbench. The language is derived from a variant of UML, and is accordingly wholly graphical. XABSL[4] by Loetzsh, Risler & Jüngel is yet another robot programming language, conceptually based on layered state machines. It also predates ROS. Many more are catalogued by Nordmann's DSL survey. [5]

2.2 Poses and Movements

The pose of a robot can be described with a Joint Vector. A Joint Vector is a list of all the joint angles, i.e. the current state of the robot's actuators. In many cases it is much more desirable to describe the actual position and orientation of the hand of the robot. The position can

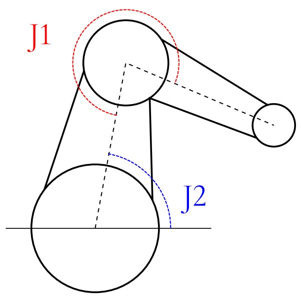

Figure 2.1: Joint-angle representation of arm position

be described with three numbers (x,y,z) and the orientation with roll, pitch and yaw. These values are usually concatenated together to form 6-dimensional vectors.

When a robot moves from one point to another it can be done in two ways:

- Specify the Joint vectors associated with the start and end pose and take the shortest path between those points in the joint space. (Joint Move)

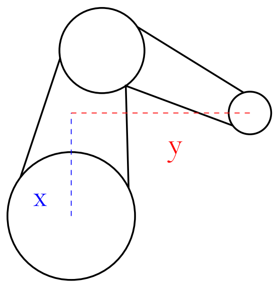

Figure 2.2: Cartesian-space representation of arm position

- Specify the Cartesian Vectors associated the start and end point and take the shortest time path between them in euclidean space, which may not be a straight line. (Cartesian Move)

2.2.1 Extra dimension in joint space and non-determinism

Both the YuMi and the Panda arms have seven joints. As mentioned in the previous section, we only need 6 numbers to specify the position and orientation of the hand. It is often desirable to use these six numbers (the Cartesian Vector) rather than the Joint Vector because

- Cartesian Vectors can be easier to understand and reason about
- It might be possible to run the same program on two different robots, since even though the robots have different joints and geometry, they might have the capability to move to the same cartesian positions.

This means that these robots (the ones with seven joints) have one extra degree of freedom that the programmer does not always care about. The way we see this in practice is that when specify a CVector, the hand can be fixed at that position while the elbow is free to move along a one dimensional curve. This means that a program only involving CVectors does not fully specify the poses and movements of the robot (the elbow ends up in arbitrary places). In practice, the position of the elbow at every step in the program will be determined by the initial position, meaning that the program might behave differently from time to time (since the initial pose will be different).

A consequence of this is that a program that works during its first run can fail during a subsequent run if it's not reset between because the elbow bumps into something and stops the program. This makes it hard to guarantee anything and instead we have to accept that programs sometimes fail even if they seemed to work before.

2.2.2 Singularities and Joint Ranges

When velocities are specified in terms of cartesian coordinates, they somehow have to be turned angular velocities for the joints in order to be put into practice. For each pose, there is a matrix that describes the relationship between the joint angles and the cartesian pose. For some particular poses, this matrix can become singular and then it becomes impossible

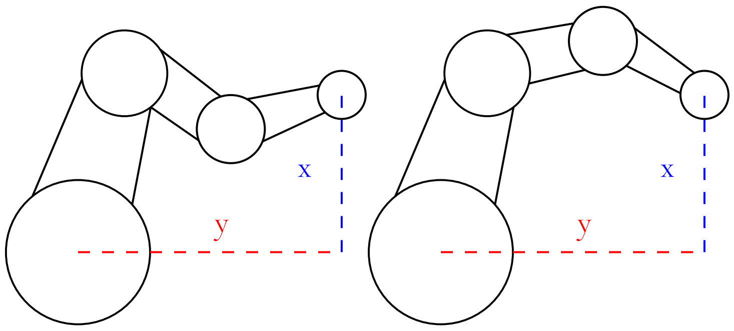

Figure 2.3: Overdimensioned robot; different joint configurations but same end position

to compute the joint angle velocities that are necessary for a given cartesian velocity. Close to such singularities the relationship becomes very unstable and the velocity controllers will therefore not function very well. Because of this, robots usually halt when they come close to singularities.

Each joint has a specific range within which it can move (like for example ±90°). It is usually a good idea to keep the robot pose far away from these limits and from singularities as these are things that can limit a robot's potential for future movement.

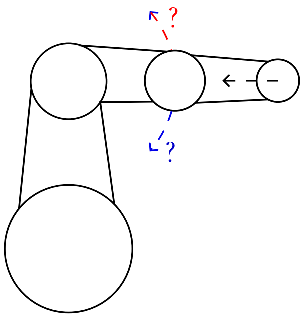

Figure 2.4: Arm at singularity: joints have lined up such that the way forward cannot easily be computed

2.3 Tools

In this section we describe the tools that we have used in the project. They consist of two major parts: the framework that we use to interface with the robot and the tool chain for building the compiler itself. Our language compiles to a so called ROS node, written in c++.

2.3.1 ROS and Moveit

ROS is an open source framework for writing robot control software in either C++ or Python. [6] Though it has not yet found widespread acceptance in industry, it is extremely popular

among robotics researchers, and is attractive to us because it provides abstraction over robot hardware.
MoveIt is a framework within ROS specifically intended to facilitate motion planning with robot arms. [2] This allows us to bypass entirely the problem of motion planning and to focus on language development.
The MoveIt project also defines the “Semantic Robot Description Format” (SRDF), which we use in our implementation to extract information about the robot. [7]

2.3.2 JastAdd, Beaver and JFlex

A compiler itself consists of the following components:

- *Scanner* - responsible for dividing the program text into "words"/tokens.
- *Parser* - responsible for checking that the program is syntactically correct and for generating the abstract syntax tree.
- *Analysis* - responsible for generating errors for things like use of undeclared functions/variables, double declaration of variables, type errors etc. The analysis step is also responsible for generating some important information that can be used in the code generation step, such as associating every use of a variable/function to the appropriate declaration.
- *Code generation* - converts the abstract syntax tree to code in the target language, in our case c++ code.

JastAdd and the Abstract Syntax

JastAdd is a tool that was developed at LTH to aid the process of creating compilers and other language tools.[3] It mainly helps with the analysis step but is also used for the code generation and to define the abstract syntax of the language. It is based on aspect oriented programming, which is an extension of object oriented programming.

The core component of a JastAdd project is the ast-file which declares the abstract syntax of the language. Each line in the ast-file declares a class that is used to represent nodes in the abstract syntax tree. The content of the ast-file is compiled to a java package with one class for each line in the ast-file. In the ast-file we declare a class called Program. The goal of the parser is to produce an instance of this class. The abstract syntax of a language is a basic description of how programs are made up of smaller components. The way we describe this is through an object oriented class hierarchy.

Chapter 3

Language Specification

In this chapter we give a description of the language that we designed. The language is procedural and is used to generate a sequence of robot movements. The main features of the language are:

- Components - refers to a part of a robot that can independently move at the same time (such as different arms).
- Functions - encapsulate reusable snippets of code.
- Transformations - makes it possible to execute snippets of code in a spatially transformed manner (mirrored, rotated, translated or scaled).

3.1 Syntax

The full syntax is found in appendix B. A number of examples are found in appendix A
The syntax is basically procedural, with newline-separated statements and block delineated by indentation level, i.e. the off-side rule. Parameters and arguments for functions are comma separated, as are numbers in vector literals Components in sync statements are space separated.

```
CVector x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
JVector y = <1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0>

left_arm.moveTo(x)
left_arm.moveTo(y)

left_arm is Arm

float Arm.f()
  Arm.moveTo(x)
  Arm.moveTo(y)
  return 1.0

left_arm.f()
```

Code example

3.2 Static Semantics

3.2.1 Placement of Statements

A statement is considered a top level statement if it is not inside a block, such as in function declarations, repeat statements and transformation statements. Function declarations and `is` statements (described below) are only allowed as top level statements. Return statements are only allowed at the end of function declarations. Functions that return void are not allowed to have a return statement. All other statements can occur in any position (both top level and inside blocks) and in any order.

3.2.2 Variable Declarations and Scope

Variables can be declared in the following places:

- As top level statements
- As parameters to functions
- As a statement inside any block

Two variables with the same name cannot be declared in the same scope. Every block is a scope. The top level statements can also be viewed as a block in this context. A parameter list counts as its own scope. In other words, it is permissible to declare a local variable in a function block even if it has the same name as a parameter of the function. If a block occurs inside another block, they still have different scopes, although we say that the scope of the inner block is a subscope of outer block. The subscope relation is transitive. In other words, if A is a subscope of B and B is a subscope of C, then A is a subscope of C.

Every time a variable is used, it refers to a variable declaration. The variable declaration has come before the variable use in the code and the variable use has to occur in the same scope or in a subscope of the scope where the variable is declared. If no such declaration exists, the variable use results in an error.

3.2.3 Function Declarations

The order of function declarations does not matter. All functions are global and can be used anywhere in the program. The signature of a function is defined by:

- An optional component declaration
- An identifier (the name of the function)
- A parameter list

Two functions with identical signatures cannot coexist. Every call expression has to refer to a function declaration with a signature which matches that of the call expression. The function declaration can be provided by the programmer or it can be one of the predefined functions of the language. A call signature matches a declaration signature if and only if

- Either both signatures have no component or the call signature has a component which inherits from the declaration signature.
- The names are identical
- The types of the provided arguments match the corresponding parameter types.

Component inheritance is discussed further in section 3.2.4. It can happen that multiple declarations match the same call. This is also discussed in section 3.2.4.

3.2.4 Components and Inheritance

Every program in the language is written for a specific robot. When the program is compiled the SRDF file for the robot has to be provided. The components that are declared in the SRDF file are available for use anywhere in the program. These components are called concrete components. There are also abstract components. These are components which do not correspond to one concrete component, but rather a set of concrete components with some common set of capabilities.

Concrete components can only be declared in the SRDF file. Abstract components can only be declared by the programmer. Abstract components are not explicitly declared, but are instead declared implicitly the first time they are used in a function declaration or an is statement.

Inheritance between components is declared using is-statements. A concrete component can inherit from an abstract component. Abstract components can also inherit from other abstract components. Inheritance is transitive. A component can occur in any number of is-statements, both on the left hand side and on the right hand side.

If a function is declared on a component, it is automatically declared on all components which inherit from that component. If a function is declared on all inheritors of a component, it is automatically declared for that component.

It is possible that a function call matches multiple declarations. There are two rules that are used to select between implementations in the cases where a function is implemented in multiple places:

1. The order of the is-declarations is used to determine which implementation is selected. For example if 'A is B' comes before 'A is C' in the program, and a function f is implemented for both B and C, the implementation for B is selected when we call A.f.

2. Implementation that is closer in the inheritance hierarchy should be used rather than an implementation further away. For example, if we have 'A is B' and 'B is C' and a function f is implemented for both B and C, then the implementation for B, is selected when we call A.f. Note that when a function actually runs, it always runs on a concrete component, so 'closer' always means further down in the hierarchy.

Rule 1 has higher priority than rule 2.

3.2.5 Types

The language has the following built in types:

- void
- single-precision float
- JVector (Joint angle vector)
- CVector (Cartesian position vector)
- Transformation (Elaborated upon below)

We allow arithmetic operations on numbers and both types of vectors, as well as vector addition and scaling.

It is not possible for the programmer to declare their own types.

Every expression has a type. Every context where an expression can occur has a set of expected types. If the type of an expression doesn't match any of the expected type, the programmer gets a type error.

3.2.6 Predefined Functions

The language provides the following predefined functions:

- float length(CVector x)
- For every component C in the SRDF file:
  void C.moveTo(JVector pose)

- For every component C in the SRDF file which is declared as a chain:
  void C.moveTo(CVector position)
- For every state s in the SRDF file which is declared as part of component C:
  void C.s()

3.3 Dynamic Semantics

3.3.1 Threading and Synchronization

All programs run single threaded. The program executes the statements one by one. When a statement corresponds to a movement, the program does not wait for the movement to complete; it places the movement in a queue and immediately moves on to the next statement. One component can only perform one movement at a time. If a component is already moving when the program reaches a statement that tells the component to move, the new move command is placed in a queue corresponding to that component. Movements for different components are performed simultaneously.

If the programmer wants components to wait for each other before starting the next movements, the sync statement is used.

```
left_arm.moveTo(A)
right_arm.moveTo(B)
sync [left_arm right_arm]
left_arm.moveTo(C)
right_arm.moveTo(D)
```

All movements that come before a sync statement in the program flow have to be completed before any movement after the sync statement can be started. In the example above, left_arm and right_arm will start moving to A and B respectively simultaneously. Then, the arm that finishes first will wait for the other arm. Then both arms will start moving to C and D respectively.

3.3.2 Transformations

Transformations are used to perform modified versions of movements (translated, scaled, mirrored or rotated). A special type called Transformation is used to represent the transformations. Values of this type are created using transformation literals. Transformations can also be combined using composition. The following example showcases this, combining mirroring of the X with rotation around the Y axis and downscaling by ¾.

```
Transformation F = MirrorX
Transformation G = RotateY 45
Transformation H = F << G
Transformation X = H << F << Scale 0.75
```

There are two syntactic forms to create transformed blocks. One just uses a transformation literal and one uses the keyword 'transform' and then an expression of type Transformation. The following example showcases both forms.

```
left_arm.moveTo(A)
left_arm.moveTo(B)
MirrorX
    left_arm.moveTo(A)
    left_arm.moveTo(B)

Transformation F = MirrorX
transform (F << RotateY 45)
    left_arm.moveTo(A)
    left_arm.moveTo(B)
```

When a moveTo(CVector) statement occurs inside a transformed block, the transformation is applied to the CVector before the moveTo is executed. This does not happen with moveTo(JVector).

When transformed blocks are nested, the transformations are composed. The innermost transformation is always applied first. If a function call happens inside a transformed block, all the (Cartesian) movements in the function are transformed, and any transformed blocks within the function are pre-composed to the outer transformation.

Here is the complete list of transformation constructors.

```
    /* Scale everything by a factor. */
    Scale([float])

    /* Translate everything by a vector.
    You should use a vector where the euler angles are set to zero. */
    Translate([CVector])

    /* Rotates everything a certain angle around the X-axis.
    This applies to both the position and orientation of robot components */
    RotateX([float])

    RotateY([float])

    RotateZ([float])

    /* Mirrors everything in the plane orthogonal to the X-axis. */
    MirrorX

    MirrorY

    MirrorZ
```

Chapter 4
Design Process

This chapter describes the process of creating the language. The main steps of the process were:

- Gathering information and talking to roboticists to get a picture of their needs and the limitations of Stenmark's language.
- A brainstorming session to get a simple initial design. (section 4.1)
- Implementation and refinement of this design.
- Compiling a list of possible additions and improvements to the language.
- Robotics workshop. (section 4.2.9)
- Implementation and refinement of selected additions and improvements. (section 4.3)
- Evaulation

4.1 Initial Design

We made an initial design of the language through simple brainstorming.

We settled on the following:

The language should be procedural. The choice to make it procedural seems natural given the domain. We are trying to make robots solve tasks through sequences of actions, where actions in turn could be described as subtasks/subroutines/functions. The choice of c-like syntax seems natural since that is what most people are used to and it will therefore help people learn our language quickly. We are not completely married to the c-syntax. We try to choose syntax that is simple, pretty and easy to understand. Instead of curly brackets, we use python style indentation to distinguish between blocks.

The initial language had the following types of statements:

- *if/else:* same syntax as C and other Pascal-like languages.
- *repeat:* repeat(x) indentation block end_indentation
- *while:* while (condition) indentation block end_indentation
- *function declaration:* [type] function [name](parameters) indentation block end _indentation
- *variable declaration:* var type x
- *reference declaration:* ref type x
- *assignment:* x = expression
- *robot component execute:* [name].[action]
- *sync:* [component].sync(label)

The following types were in the initial design of the language:

- boolean
- int
- float
- reference system/cartesian vector
- joint vector
- robot component

It also has arithmetic expressions (+, -, *, /) on numbers, logical expressions (&&, ||, !) on booleans, variables, literals (numeric and boolean) and function calls.

In the initial design, we viewed robot component execution as a special type of statement.

The language had two types of variables: var and ref. Vars work the same as the variables in most other languages. Refs "remember" the expression that they are assigned to, and are updated automatically when a variable that they depend on changes. The idea was that this could be used to set the position of one arm to be a function of the position of the other arm.

The programs in the language are single-threaded. However, multiple components can move at once. When the program reaches a robot action, it does not wait for the action to be completed before continuing the program. Because of this, multiple movements can occur at the same time. Every component can, however, only do one movement at a time. If the program reaches a robot action on a component that is already busy, the action gets placed in a queue. In order to do synchronization between components, there are special sync statements. The syntax is `[component].sync([label])`. The idea was that when one component reaches a sync action, it has to wait until the other components reach a sync action with the same label, before continuing.

In the initial design, there were lots of built in functions. The built in functions were basically the ones that can be found in Dr Stenmark's framework (moveTo, contactMove, viaPoint, etc).

4.2 First Design Iteration

4.2.1 Changes made to the design during implementation

We decided to try to implement the compiler as quickly as possible in order to have something concrete to work with. A few parts of the design were changed during implementation:

- Refs were removed. The reason for this is that our language does not allow to set the position of an arm to a variable in a way that makes the arm move automatically as the variable changes. Rather, arms are moved only with `moveTo` commands. So the example where the position of one arm can be set to a function of the position of another arm would not work, even if refs where included. Refs could be used for other things, but it was not clear why refs would be particularly useful for programming robots. Implementing refs would not be very difficult, but it would take time and make the language unnecessarily complicated.
- The 'function' keyword was removed. The reason for this was that function declarations with many parameters often could not fit within the margins. One way to fix it might be to allow newlines in parameter lists so that the programmer can split up long lines, but we figured it would be easier to just shorten the syntax a bit.
- component execute became ordinary functions. We made it possible to define functions on components with the syntax:
  float component.f(float arg) [code block]
  These functions are called with the following syntax:
  component.f(1.0)

  This makes it possible for the programmer to write their own subroutines for the different component. Built in component actions, such as moveTo, can now be ordinary built in functions.
- Component is not a type anymore. They are treated separately.

4.2.2 The first working compiler

The first working version of the compiler had the following features:

- if/else statements
- while statements
- repeat statements
- variable declarations
- variable assignment
- function declaration

- function calls
- integer arithmetic
- integers and booleans
- functions on components

The language did not have any vectors, sync instructions or anything that had anything to do with robots. It compiled to c++, but it did not use ROS.

4.2.3 Simplification of the language

In order to avoid unnecessary work, we decided to remove all features that we could not find a concrete use for. We had already decided to remove refs.

The biggest question was whether we should keep `if` and `while` statements. It is not entirely clear how the robots can get input from the environment. Many robots have cameras. However, extracting useful information from cameras will involve advanced data processing that our language will not support. Some robots have other sensors or direct IO-interfaces. It is also possible to use contact moves to get information from the environment. One possibility would be to include contact moves in the language. That would justify the existence of if/while statements, and if roboticists would like to use the cameras in the future, image processing libraries from ROS can be integrated, and then add a predefined function in our language to get the result. However, as long as the robots are running blind, if/while statements are not justified.

We wrote an example program where contact moves are used together with if and while statements to perform tasks. We confirmed with our roboticists that they would like to write programs like that.

If if/while statements are removed, booleans loose their purpose and should also be removed. Integers have one important purpose and that is in repeat statements. They could of course in theory also be used to formulate conditions for if/while but this would probably not be done very often in practice when doing robot programming. So an important question becomes: do we keep the int type, just for the sake of repeat statements or do we simplify the language? We ended up with these three alternatives:

- Remove int, because we don't want to introduce features that are barely used.
- Keep int, because repeat loops with floats are ugly and having ints there doesn't hurt.
- remove int as a type, so variables can not be of type int, functions cannot depend on int parameter etc. But require an int literal (rather than an expression) as the argument of a repeat loop.

We ended up going for the third option. One possible drawback is that now a variable cannot be used as an argument, which means that this aspect of the task cannot be parameterized. For example, the programmer might want to write a program that mixes two chemicals together and then stirs. A natural way to stir, would be to have two moveTo commands that move back and forward between two positions in a repeat loop. It seems reasonable that the number of times that you go back and forth should be a parameter of the function. It then becomes easier to reuse the function in multiple context where different amounts of stirring might be required. We asked the roboticists weather such a function would ever be written in in practice, which they believed it would.

4.2.4 Suggested additions to the language

At this stage we also discussed some features that we might want to add:

- *Infinite Repeat loop*: if repeat is not given an integer argument the loop continues indefinitely.
- *comments* [not added]
- *getter functions for robot state*: CVector currentPosition = left.getCartesianPosition() [not added]
- *setter functions for robot settings* : left.setMaxForce(10), left.setMaxSpeed(2) etc. [not added]
- *Vector arithmetic for Cartesian vectors*
- *global function for computing length of vector*
- *global variables for directions*: up,down,forward, etc [not added]
- *void return type*: at the moment a lot of functions in our example have int as their return type. We are thinking about removing integers so maybe we need void return types. This could be written as 'void' or perhaps by simply not writing anything at all. [we are using void]

The getter and setter functions are important if we are going to use contact moves to get input.

4.2.5 Better synchronization

The initial design for synchronization works well if there are only two components. When there are more components, it can't express things like "component A and B should wait for each other but C can continue". Also, compared to how little expressive power it had, it would probably still be somewhat complex to use it. You would have to remember what labels you have used and make sure that both components reach it somewhere.

We came up with three possible solutions:

- A sync instruction with no arguments that require all components to wait for each other. The semantics are simple: all instructions before sync must be completed before any instructions after sync can be started, regardless of component.
- A sync instruction that takes an arbitrary number of components as arguments. It works the same as the first solution except the only components that have to wait for each others are the one you specify.

- An asymmetric sync instruction:
  a sync b
  this means that a can not continue until b reaches the sync. However if b arrives first, it can continue without waiting for a.

We ended up going for solution 2.

4.2.6 Abstraction

Being able to define your own functions on components is not very useful unless there is some abstraction over components.

We came up with three ways of doing abstraction:

- *No abstraction*: Simpler language. Sometimes leads to code duplication.
- *component blocks*
- *component parameters*: Let functions take components as parameters.
- *Object Oriented style components*:

It is possible to combine 2-4 with each other. We ended up going with 4. We might add 3 in future.

The solution we picked involves abstract component classes, allowing the user to define functions on classes as well as on concrete components. Classes are not explicitly declared but are created when they are used. The user can declare that a class A is a subclass of B using an Is statement. Syntax:
A is B
the same syntax is used to declare that a concrete component is a member of a class.

Functions defined on a class automatically are available for all the concrete components and all subclasses of the class. If a function is available for all concrete components of a class, it becomes available on the class. This is possible since there is always a finite number of components and they are known at compile time.

Initially we allowed functions within functions and functions within loops. Since this would make the semantics of abstract functions unnecessarily complicated and since it would probably never be used, we now only allow function declarations in the outermost block of the program. The same thing is true of `is` statements.

4.2.7 Using SRDF

Every robot that can be used in MoveIt has a SRDF file. The SRDF file contains *semantic information* about the robot such as which components the robot is thought to consist of, which joints each component has, which components are made up of other components and which components can be described as kinematic chains. It also contains a number of predefined states, that is pre-set joint angles, such as open and close for the grippers.

We wrote a parser that reads the SRDF file and provides this information to the compiler. This is how the compiler can know the set of concrete components that can be used.

We also wanted to use the states and create one function for each state. For example the grippers would get one function called open() and one called close(), and they would move the grippers to the open or closed state respectively.

We have also been thinking about making the JVector type more specific. There are three options here:

- Leave it as it is. There is only one JVector type and using the wrong "type" of JVector in the wrong place will not yield an error.
- One JVector type for every size. When you call comp.moveTo(x), the compiler checks that x has the right size.
- One JVector type for every component. When you call comp.moveTo(x) the compiler checks that x was intended for comp. This might be a useful thing to do since we don't know if different components even use the same units and it would then be stupid to take a joint vector from one component and use it on another.

So far we have decided on 1.

How we should use the information about the subcomponents is still up for discussion. At the moment one queue is created for each component, including supercomponents. This is a problem since it makes it possible to execute an instruction on arm_and_gripper and arm and gripper at the same time. This should obviously not be allowed, as the gripper would be tasked with two simultaneous actions. We could remove the super components entirely or we could make sure that such instructions block both queues.

We had some ideas about being able to access subcomponents as fields of the supercomponents. It ended up not being practical. If we at some later stage make components into legitimate types (they can be return types of functions) then this can be implemented by the programmer in a way that would eliminate this potential conflict.

We also want to use the information from SRDF to know which components should have moveTo(CVector) defined on them. The idea is that components that are chains should have this function while components that are not chains should only have moveTo(JVector). This might be a problem when a component is a chain but has less than 6 joints (cannot span space). We discussed this with the robot group and confirmed treating it as a chain is still desirable behavior.

4.2.8 Integration with ROS and finalizing iteration 1

One thing that we learned is that getting and setting controller settings such as max force (needed for contact moves) cannot be done through moveit. Instead we have to do it at a lower level. In order to be able to have something working to present as quickly as possible, we decided not to include those for the moment being. We also decided not to include syncMoves.

This means that the language becomes very simple. The only interaction we have with robots is through moveTo commands. This means that we will be running blind and we should therefore drop if/while statements along with booleans. We can always add it back if it is desired (discuss with roboticists).

4.2.9 Workshop with roboticists

In order to get input into the design, we had a workshop with roboticists from the RSS research group, where we showed them the language and asked them questions about the usefulness of each feature of our language. Among the features that were implemented so far, there was nothing that the roboticists didn't expect to use. We also went through a list of features suggestions to get a perception of what the interests and priorities were among the roboticists. The complete list of suggestions can be found in appendix C. The feature that seemed most important was some sort of support for live-programming.

4.3 Iteration 2

Based on the feedback from the roboticists we started working on a second iteration of the language.

Features/tasks for consideration:

- Fix any bugs from previous iterations
- Create a Emacs mode that allows us to get the current pose of the robot and place it in a vector literal.
- Look a bit closer into dual arm manipulation.
- Find out if there are situations where arithmetic on JVectors is needed.
- Look into contact moves just to see that it's possible.
- Mirrored context and other transformations
- Proper testing and documentation
- Getters and Setters for robot state and settings
- Add support for comments in the language

4.3.1 Live Programming Support

We created a very minimal proof-of-concept implementation for a live programming system, presently consisting of an emacs mode and a pair of terminal utilities. We have not taken time, however, to test it's utility.

Chapter 5
Design Decisions

In this chapter we discuss some the various decisions that have been made concerning the design of the language. For each decision we list a few alternative solutions with some pros and cons.

The decision areas can be divided roughly into:

- Syntactic Decisions
- Included basic features
  - Program flow and control structures
  - Included types
  - Built in functions and operations
- Threading model and synchronization
- Component Model
- Transformations
- User interface

We have used the following somewhat subjective criteria when discussing pros and cons:

- Similarity to commonly used programming languages
- Simplicity
- Expressive power

The language needs to be *familiar*, because it would ultimately be used by programmers building tools upon it. We wanted it to be *simple*, to make our work developing a compiler easier. We also wanted greater *expressive power* to make our task of writing programs easier.

In the sections below we list the major decisions that have been made during the design process. Every feature that was included in the language is written in **boldface**. If the only reason that a feature didn't get included is because of shortage of time, it is written in *cursive*.

5.1 Syntax

We have borrowed most of the syntax from C and other procedural languages. The advantage of this is that the programmers might be familiar with this syntax and will therefore not have to learn anything new. We distinguish between blocks using the off-side rule, i.e. python-style indentation.

Option	Pros	Cons
Indentation	• It can make the programs look a bit less cluttered. • It forces the programmer to use correct indentation, which is good practice also in languages where brackets are used.	• It breaks with the otherwise C-like syntax. • It creates issues when tabs and spaces are mixed together.
Brackets	• In line with otherwise Pascal-like syntax.	• More clutter in code.

5.2 Included basic features

5.2.1 Input and Program Flow

We discussed including the standard set of procedural control flow features (if/else, while). We chose to not include them because currently the robots are running blind, that is, without any external sensor input. Without input, if- and while statements are not of much use.

In the future there is a possibility of adding support for input and in that case it would make sense to add if- and while statements back into the language. There are three potential ways of gathering input that might be useful in the future:

- *Contact Moves*: Move to a position which is inside of an object just to see if the object is there. If we actually reach the position we know that the object is not there.
- *Weighing objects/measuring forces*: Measure the weight of an object by sensing how much force is required to hold it. This can also be used to measure other forces. For example the amount of force required to move an object along a surface measures the friction.
- *Input from camera*: The robot might have a camera or other input device.

In order to implement the first two points we need to add communication between our runtime system and the controllers in moveit/ROS. More about this in the section about included functions.

For input from cameras, the best option is to implement the communication with the camera and the image processing in c++ and then add a predefined function in our language that retrieves the result. In order to make this more easy, it might even be worth considering adding support in our language for calling an arbitrary c++ function.

Even if no support for input is added. There is still one potential use case for if statements. It is possible in our language to write recursive procedures. With if statements, it is possible to add base cases, making this feature more meaningful.

5.2.2 Types

integers and booleans

We considered adding integers and booleans. But without if- and while statements we didn't have use cases for booleans and very few use cases for integers. Integers can still be used in repeat statements, but there are also two other options:

- Use floats in repeat statements
- Use integer literals in repeat statements (meaning int doesn't have to be added as a type)

We selected the second option. Here are some pros and cons:

Option	Pros	Cons
Keep int	• Intuitive • In case support for input is added in the future, adding if/while statements is easy.	• Add a whole new type for only one use. • Puts expectations on future developers to add if/while. Better to keep a clean slate.
Use float instead	• Avoid adding more types than necessary.	• Counter-intuitive
Int literals	• Get the pros of both solutions above.	• Decreased power. We cannot write repeat loops where the number of repetitions is given as a parameter to a function.

An example of a case where you might want to provide the number of repetitions as a parameter:

```
void shakeFlask(CVector position, int amount)
  CVector aboveTable = position + [0 0.2 0 0 0 0]
  CVector deltaX = [0.03 0 0 0 0 0]

  panda_hand.open()
  panda_arm.moveTo(position)
  panda_arm.moveTo(aboveTable)
  repeat (amount)
    panda_arm.moveTo(aboveTable + deltaX)
    panra_arm.moveTo(aboveTable)
  panda_arm.moveTo(position)
  panda_hand.open()
```

Vector Types

We have included CVectors and JVectors in the language. One idea that we discussed is to have another type to represent cartesian vectors without orientation (just the x,y,z). This would be useful mainly for translations.

Option	Pros	Cons
Use CVectors	• Fewer types. • No need to convert between types.	• Strange behaviour if the programmer accidentally translates by a vector that contains euler angles.
Add new type	• More intuitive. • Avoids strange behavior.	• Adds complexity.

5.2.3 Included Predefined Functions and Operations

Some features are so fundamental to the language that it becomes hard to reason about the pros and cons, since it is hard to imagine what the language would be like without these features. `moveTo(JVector)` and `moveTo(CVector)` are examples for this. In such cases we will focus on the pros and cons that one operation has when compared to the other. This helps us justify why we need both features and also clarifies what the use cases are for each of them. For these operations, the "cons" are not really cons, but rather limitations. The expressiveness of the language isn't reduced when we add an operation that has a certain limitation. It just means that in order to get the expressivity we want, we also need to add another operation.

Function/operation	Pros	Cons
[Component].moveTo(CVector)	• Allows for abstraction over different kinds of robots. • Interacts well with transformations.	• Only works for some components. • For components with more than 6 joints it adds non-determinism.
[Component].moveTo(JVector)	• Works for all components. • Control complete pose of arms with 7 degrees of freedom in order to add determinism.	• Harder to achieve abstraction over different kinds of robots. • Doesn't interact well with transformations.
[Component].[state]()	• Simplifies by providing predefined functions for commonly used states.	
length(CVector)	• Normalize vectors. • Test for proximity.	
Arithmetic on floats		
Arithmetic on CVectors	• More expressive than Transformations.	• More error prone than Transformations.
Arithmetic on JVectors	• Can capture incremental actions, such as turning a screwdriver	• Harder to reason about
Composition of Transformations	• More expressive than nested transformation blocks. • More compact than nested transformation blocks, especially if the same transformation is used many times.	
Vector Literals with numeric-literal arguments	• Necessary to define vector literals.	
Vector Literals with numeric-expression arguments	• More expressive than literal arguments. • More concise and intuitive than arithmetic + unit vectors.	
getters for component position	• Together with contact moves, it would allow programs to get information from the environment.	
getters and setters for component settings	• Allows programmer to adjust force and velocity. Critical for many applications.	

Example use case, moveTowards:

```
void Arm.moveTowards(CVector target, float amount)
    CVector delta = target - Arm.getPosition()
    CVector deltaNorm = delta * (1/length(delta))
    Arm.moveTo(Arm.getPosition() + delta*amount)
```

This illustrates a use for position getters. It also shows the length(CVector) function being used for vector normalization.

Example use case, contactMove as a means of retrieving information:

```
bool Arm.blockIsPresent(CVector position, float height)
    Arm.moveTo(position + [0,1.5,0,0,0,0]*height) // move to above the block
    Arm.contactMove(position)  //try to move inside the block
    //if we aren't close to where we wanted to be, the block is present
    return length(Arm.getPosition() - position) > height/2
```

This is an example of a use case where length(CVector) is used to test for proximity. This example also contains the method [Component].contactMove(CVector), which is supposed to try going to a position without stopping the program if something is in the way. The only reason we haven't included this function is because we didn't have time. This example also includes booleans. The only reason why they aren't included in the language is because they don't have any use if we don't have a way of getting information from the environment (which we don't have without contact move).

5.3 Threading model and Synchronization

A main requirement was to be able to describe simultaneous movement of multiple subcomponents, such as additional arms. This requires the language to have some form of threading model. There are many different possible threading models, but we settled on implicitly parallel movement with explicit synchronization points. We believe this is the simplest model to implement that still fulfills most requirements.

5.3.1 Alternative sync Statements

The sync statements in our language have the following syntax:

```
sync [Comp1 Comp2 ... CompN]
```

The meaning of this statement is that none of the components involved are allowed to start any action that comes after the sync statement until all their respective actions before the sync statement have been completed. We can call this sync statement the Symmetric Sync statement.

We considered two alternative solutions for synchronization:

- The Asymmetric Sync Statement

- The Universal Sync Statement

The asymmetric sync statement would have the following syntax:

```
Comp1 sync Comp2
```

and would mean that Comp1 is not allowed to start any action after the statement until all the actions for Comp2 before the sync statement have been completed.

The universal sync statement would have the following syntax:

```
sync
```

and would mean that no component can start any action after the statement until all actions before the statement have been completed.

Note that if asymmetric synchronization is enabled then symmetric synchronization is automatically also possible, by simply writing two sync statements:

```
Comp1 sync Comp2
Comp2 sync Comp1
```

If there is symmetric synchronization, universal synchronization is of course also possible, since universal synchronization is just symmetric synchronization with all components included.

To demonstrate that asymmetric synchronization is more powerful than symmetric synchronization we provide the following example:

```
repeat (5)
    left_arm.prepare_sandwich()
    right_arm sync left_arm
    right_arm.place_sandwich_in_box()
```

The corresponding program with symmetric synchronization is

```
repeat (5)
    left_arm.prepare_sandwich()
    sync [left_arm right_arm]
    right_arm.place_sandwich_in_box()
```

The difference is quite subtle. Both programs will behave the same in the first iteration. The left arm will prepare a sandwich and after it is done the right arm will place it in a box. Since there is no sync statement in the beginning or end of the loop, the left arm will move on to make the next sandwich before the right arm is done placing the first one in a box. In the next step however the two programs potentially differ. If the left arm completes its second sandwich before the right arm is done with the first sandwich, it can either wait for the right arm to complete the first sandwich before moving on to the third or it can start working on the third sandwich immediately. If we use asymmetric synchronization then the left arm moves on immediately. If we use symmetric synchronization it waits.

Solution	Pros	Cons
Universal Synchronization	• Very simple	• Not as powerful as symmetric synchronization
Symmetric Synchronization	• Simple	• Not as powerful as asymmetric synchronization
Asymmetric Synchronization	• Powerful	• Somewhat tedious to test and reason about

5.4 Abstraction over Components

In the current version of the language we abstract over components using abstract components and functions on abstract components. This is described in detailed in section 3.2.4.

Within this solution there are several decisions that had to be made:

- Whether or not to require explicit declaration of abstract components.
- Whether or not to require explicit declaration of abstract functions.
- Whether or not to allow multiple inheritance.
- What priority rules to use when selecting an implementation of a function that has multiple implementations.

Apart from these decisions there are also alternative ways to achieve abstraction over components:

- Component blocks
- Components as function parameters

Component blocks were one of the first solutions we considered, where an indented block would be run for one component only. We decided against this, because we did not want to deal with the situation where a component block could exist within a function, or decide how inheritance would work in such a situation.

We also considered having a component type which could be passed to functions. We decided against this because we wanted it to be abundantly clear what component a statement is operating on. Passing components would make sense in a robotics library for a general-purpose language, but we are designing a domain-specific robotics language.

Decision	Pros	Cons
Explicit abstract components	• Easy to read and understand programs. • Spelling errors when using components result in error rather than a new component being created.	• Slightly longer programs.
Explicit abstract functions	• Easy to read and understand programs. • Spelling errors when implementing functions result in good error message. • Forgetting to implement a function results in good error message.	• Slightly longer programs.
Multiple inheritance	• Increased power.	• Gives rise to diamond problem (there are two candidate parents with differing implementation).

Priority Rule	Pros	Cons
a) **Uppermost is-statement**	• Avoid undefined behaviour. • Gives programmer some control over which implementation is used.	• This rule is arbitrary • Easily overlooked by programmers learning the language.
b) **Most Specific Subclass**	• Makes it possible to override behaviour defined in a superclass. • Same as in most object oriented languages.	
Rule a) over Rule b)	• Easier to to reason about, since you don't have to count the number of subclasses in an inheritance chain.	• An arbitrary rule is given higher precedence than a commonly accepted rule.
Warning when a) needs to be used.	• Programmer doesn't have to spend time wondering why a certain implementation is used.	
Error when a) needs to be used. (not included)	• Programmer doesn't have to spend time wondering why a certain implementation is used. • In many cases this is indeed a programmer mistake.	• Prohibits programmer from using abstract functions in certain situations involving multiple inheritance.

Abstraction approach	Pros	Cons
Component Blocks	• Short syntax. • Avoids some of the complexity associated with inheritance.	• No semantics for restricting which operations are allowed on which components. • The static semantics becomes awkward, as some functions can only be called in specific contexts.
Components as parameters	• Makes it possible to define actions involving more than one abstract component. • Once components are treated as normal data types, other features are also enabled. see later discussion	• Would make our type system more complicated as component inheritance would translate into subtypes.
Functions on Components	• Same syntax for programmer defined functions on components as for the built in `moveTo()`.	• This means that it has to be possible to add functions on already existing components, which goes against the intuition programmers might have from other languages.

Note that any combination of these three solutions would be possible.

5.4.1 Relationships between components

It might be useful to provide the possibility to specify that two components have a specific relation to each other, for example that `left_hand` is the hand corresponding to `left_arm`.

Solution1	Solution2	Solution3
`left_arm is Arm`	`left_arm is Arm`	`left_arm is Arm`
`right_arm is Arm`	`right_arm is Arm`	`right_arm is Arm`
`left_hand is Hand`	`left_hand is Hand`	`left_hand is Hand`
`right_hand is Hand`	`right_hand is Hand`	`right_hand is Hand`
`void Hand.grab()`	`Hand Arm.hand;`	`Hand left_arm.getHand()`
`  Hand.close()`		`  return left_hand`
	`left_arm.hand = left_hand`	
`void Arm.pickUp()`	`right_arm.hand = right_hand`	`Hand right_arm.getHand()`
`  Arm.moveTo([some position])`		`  return right_hand`
`  Arm.hand.grab()`	`void Hand.grab()`	
`  Arm.moveTo([some other position])`	`  Hand.close()`	`void Arm.pickUp()`
		`  Arm.moveTo([some position])`
`left_arm.pickUp()`	`void Arm.pickUp()`	`  Arm.getHand().grab()`
`right_arm.pickUp()`	`  Arm.moveTo([some position])`	`  Arm.moveTo([some other position])`
	`  Arm.hand.grab()`	
	`  Arm.moveTo([some other position])`	`left_arm.pickUp()`
		`right_arm.pickUp()`
	`left_arm.pickUp()`	
	`right_arm.pickUp()`	

We did consider three different ways this could be done:

1. In the SRDF file we can see which components have other components as subcomponents. We could make it so that functions defined on a subcomponent automatically get defined on the components it is part of. This is the approach we chose.

2. Have a special type of declaration to declare that a component is accessible as a field of another component.

3. Change the language so that Component counts as a normal data type.

An example which is possible with solution 2 or 3 but not with solution 1:

```
Arm is Part
Hand is Part

left_arm is Arm
right_arm is Arm
left_hand is Hand
right_hand is Hand

Part Part.opposite
left_arm.opposite = right_arm
right_arm.opposite = left_arm
left_hand.opposite = right_hand
right_hand.opposite = left_hand

void Part.someAction()
    ...

void Part.trulyMirrored()
    Part.someAction()
    MirrorX
        Part.opposite.someAction()
```

Approach	Pros	Cons
SRDF connections	• No extra syntax/language constructs. • Short programs.	• Will often result in functions being defined on components that they aren't supposed to be defined on. • Not possible to access functions on the other component if they have the same name as a function on this component. • Not possible to specify relationships between components that aren't physically connected.
Components as fields	• Easy to understand. • Possible to define any relationships.	• Requires extra syntax.
Components as datatypes	• Increases the possibility for abstraction over components. • Doesn't add any new concepts to the language. • Removes a concept from the language, as Component is now a data type rather than a separate concept.	• Slightly longer syntax. • More complex type system.

5.5 Transformations

We want it to be possible to create versions of a procedure mirrored, translated. scaled pror rotated in space. There are a few ways this could be done.

- Stateful: have special methods that change the state of the program between states such as "mirrored" and "normal". It might look something like this:
  arm.moveTo(x)
  beginMirrored()
  arm.moveTo(y)
  arm.moveTo(x)
  endMirrored()
  arm.moveTo(x)

- Special syntax for function calls. When a function is called there might be some special syntax that allows us to specify that the whole function should be performed in a mirrored way.

- Transformation blocks: Introduce a special type of block where everything is mirrored.

The problem with the stateful approach is that it can easily lead to bugs where the programmer has started a transformed section and forgets to end it. The other two approaches avoid these types of bugs, because in the case of the special function syntax, the mirrored section is automatically ended when the function is done and in the case of special blocks, the compiler checks that every block is closed. There might be situations where the stateful approach is more expressive. For example if we also introduce if statements, it might be possible to start or end a mirrored section under certain conditions but continue as usual if the condition doesn't hold.

The special syntax for function calls is very similar to the trasformation block approach. They have the same expressive power and they are both intuitive and should help preventing bugs. A key difference however is that with the block approach, it is possible to mirror a small snippet of code that one might not want to move to a separate function. It can therefore be argued that the block approach is to be prefered over the function approach. We also prefer the block approach over the stateful approach, since it removes unnecessary possibilities of making errors when programming. We will see later that the lost expressive power is easily regained.

Since transformations do not always commute, it is important that the language specifies in which order the transformations are applied in nested blocks. We have decided that the innermost transformation should always be applied first.

We can't come up with any way to make sure that the transformations are applied on JVectors, since the transformations describe how the programmer wants robot position and orientation to change, not the joint state. This might however not be a problem since JVectors are primarily used for two things:

- Placing the robot in its initial position
- Moving a component to a predefined state such as hand.open() or hand.close()

The initial state will usually not be set within a transformed context. Opening and closing the gripper can often be done in a transformed context, but we don't expect those actions to be changed by the transformations.

In the future, transformations could be used to describe a relationship between components of a robot. For example, telling one arm to move a certain way, and telling the other to stay at position as the first arm except with a certain translation. This would be the same as Stenmark's SyncMoves, but it can now be generalized to any transformation, such as mirroring and rotation.

We also considered having transformations that would change speed or force. However, because of our initial assumptions about how programs should execute, and because of toolchain limitations, we could not find a way to implement this.

5.5.1 Adjusting Origin, Axes and Planes for Transformations

Since scale means that we multiply all vectors by a certain factor, it also means that they move away from the origin. The choice of the origin as the reference point that everything moves away from is arbitrary and it might be worth considering adding syntax where the reference point is provided by the programmer. We have not added this however and instead, the programmer will have to achieve this by composing scale with translations. Similar considerations need to be made for Rotations and Mirror. The best thing would probably be for the visual programming interface to provide a good visual way of creating transformations, as composing multiple transformations and trying to visualize in your head what they will do is not always so easy.

5.5.2 Representing Translations

Instead of using CVectors to represent translations we could have:

- Just let the user provide three numbers: x, y z.
- Introduced a new type that represents positions without any orientation.

Since we did use CVectors, there are a number of possible ways we can handle the case where the user provides a vector that contains non-zero euler angles:

- Automatically set the euler angles (pitch, roll, yaw) to zero.
- Throw an exception.
- Just add the euler angles together as well when doing the translations.
- Make sure that the position is added first, and then everything, both the position and the orientation is rotated in the way that is described by the euler angles.

At the moment we just add the euler angles together. This was easier to implement, but this is something we might want to change.

One thing that can be done in order to make things easier for the user is to add a predefined function *positionPart* that takes a CVector and returns a new CVector with the euler-angles set to zero.

5.5.3 Representing Rotations

Instead of providing separate transformations for rotating around the X,Y and Z axis, we could have provided one transformation that can rotate around an arbitrary axis. The rotations could be represented as:

- Quaternions
- Euler angles
- Vectors where the direction represents the axis of rotation and the length represents the angle to rotate by.

The first two options would force the user to learn new concepts that they might not have worked with before. The third option however is a bit more intuitive to learn and use and it might be worth considering this as an option. The main benefit compared to the current solution is that it is possible to rotate around any axis. This is also possible in the current solution but that requires composing multiple rotations in different directions.

Similarly, we only allow mirroring in the planes orthogonal to the x,y and z axes. We could have chosen a syntax where the user gets to specify an axis with a CVector. We didn't add this. Instead the programmer will have to achieve this by combining rotations and mirroring.

5.5.4 Transform-block using Variables or Expressions

There are two syntaxes for transformed blocks of code: Syntax1:

```
[TransformLiteral]
  arm.doSomeAction()
  arm.doAnotherAction()
```

Syntax2:

```
Transform [TransformExpression]
    arm.doSomeAction()
    arm.doSomeOtherAction()
```

An expression can be either a transformation-literal, a variable of type Transformation or a composition of transformation-expressions.

Because Transformation is a normal type, and we have variables of type Transformation that can be assigned and reassigned to, we have regained some of the expressive power we lost when we decided to use Transformation blocks rather than a stateful approach. If we ever add if-statements to the language again, we can write code like this:

```
Transformation Id = Translate([0 0 0 0 0 0])
Transformation f = Id
Transformation g = RotateY(-30)

if ([condition])
    f = g

Transform f
    arm.doSomeAction()
```

Now the transformation f is only applied if a certain condition holds and we can achieve the same result as we could with the stateful approach in the previous section.

Furthermore it is also possible for the programmers to define their own functions to construct transformations.

```
Transformation MirrorXWithOffset(float offset)
    Transformation translate = Translate([offset 0 0 0 0 0])
    return translate << MirrorX << translate
```

This reduces some of the issues discussed above, since if the programmer needs one of the features that we haven't added, they can easily add it themselves.

It would be possible to make the language even more expressive by adding the possibility to create your own transformations from scratch:

```
CVector f(CVector x)
    return 2*x + [0 1 0.5 0 0 0]
```

```
Transformation F = FromFunction(f)

Transform F
    arm.doStuff()
```

One problem with this is that it might make static analysis of the program harder. We should therefore not add this unless we find important use cases.

5.6 Live programming and user interface

We decided early on not to create any form of user interface. We nevertheless needed a way to try out live programming, that is to create the program partly from reading the robot's state. We only needed the ability to write the current position or joint angles into source code. As we had some experience with emacs lisp, we chose to create a major mode providing these functions.
Those are the only two functions this mode provides, as we found no need to explore it further.

Chapter 6
Evaluation

6.1 Unsolved Problems

Currently, concurrent motions do not work. We suspect this is due to an error in our usage of the MoveIt framework, but ran out of time to investigate.

More extensive tools for live programming were originally part of the scope of the project. This was scaled back because it proved too time-consuming to implement.

Our codebase has many very particular dependencies and is accordingly difficult to compile. Going forward, containerizing the project with a service like Docker would practically be required. The code would then be packaged with all dependencies, giving all project participants a consistent system. Again, we ran out of time, and we weren't proficient in container tools such as Docker.

6.2 Quantitative

The purpose of the quantitative evaluation is to make a systematic comparison between our and Stenmark's language in terms of features.

6.2.1 Methodology

We received a folder with programs that have been written in Stenmark's language as demos. The folder contained in total 102 programs. Out of these we have studied 13 programs. The programs were selected according to the following criteria:

- If a program is called "nyX", "workspaceX", "demoX", "roboticsWeekX" or "testX" for some number X, it is excluded. This is because it is hard to understand the purpose of those programs.

- If a program is almost or completely empty it is excluded.
- If a program seems to be an older version of another program, it is excluded unless it contains information that is not also in the newer version.

The programs come in the form of xml files with instructions. In order to make the programs easier to analyse we have simplified them down to the following format:

- Each file has:
  - A list of actions for the left arm
  - A list of actions for the right arm
  - A list of skills
  - A list of world objects
- If any of those lists are empty they are left out.
- For most of the actions we only write down the name of the action (Move, Open, ContactL, etc).
- If an action happens in relation to a world object, the name of the world object is written after the action.
- If an action is a SyncMove, a reference to the corresponding move on the other arm is written after the move.
- If a skill has been described in an earlier file, it is replaced with three dots.

Using this simplified format we have analyzed the programs to see weather or not they can be implemented in our language. When a program cannot be implemented in our language, we have noted down which features our language is missing.

program	**missing features**	**line count**
bagge	• contact move	31
exp	• master slave move	11
flavius	• locate • via move	14
give	• sync move • master slave move	23
maxSkill	• locate • contact move	23
paket	• via move	10
paket1	• via move	18
presen	• via move • sync move	27
sidefold1	• via move	30
sidefold2	• via move	30
sidefold3	• via move	32
WorkspaceMajLego	• locate • via move	17
WorkspaceMajUseMe	• locate	6

6.3 Qualitative

6.3.1 Methodology

We intended to select a sample of subjects who would learn about the language through a quick demo. The subjects were PHD students and professors who have written programs for robots in other languages/frameworks before. During the demo they received a handout containing a description of the language together with a few code examples. They also receive three programming tasks which they would complete after the demo. When they have completed the tasks, they would hand in their work and we would ask them questions about their experience of learning and using the language.
Due to scheduling difficulties we unfortunately only received a single response. While this diminishes the value of the survey, we nevertheless feel the results have some value, and therefore we chose to include them.

Questions

Appendix D contains the full list of questions. Questions A through N have been taken from the Cognitive Dimensions paper[1]. In all those questions we have replaced the term 'notation' with either 'language' or 'program' depending on weather 'notation' refers to a system of notation or a specific instance of notation in that context. We have removed the questions that are not relevant/applicable in our case.

Question H, fourth question was rephrased to make it more suitable for our purposes.

6.3.2 Results

Question	Answer1
When you need to make changes to previous work, how easy is it to make the change?	As easy or difficult as usual with a text-based programming language (i.e., somewhat tedious)
What was difficult to change?	I did not change much...
Does the language let you say what you want reasonably briefly, or is it long-winded?	A text-based language to program physical motion is awkward in any case, so I consider it long-winded and not really helpful in programming the robot directly, I see it as an interpretable abstraction layer between a user interface and the actual robot language (which is even more tedious to handle)
What sorts of things take more space to describe?	I am not sure I get this question... more space than when or where?
What kind of things require the most mental effort with this language?	Programming "in thin air" when it comes to transformations and synchronisations - exactly those things that make robot programming difficult.

Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?	Anything that you have to "combine, turn, flip, sequentialise..." in your head is a complex task, which is why we want the connection to the kinaesthetic teaching...
Do some kinds of mistake seem particularly easy to make?	Difficult to answer, since I could not test my program and thus did not get any feedback on errors I might have made.
How closely related is the program code to the actions you are describing?	Very closely.
Which parts seem to be a particularly strange way of doing or describing something?	Transformations, where I have to define a base point and then open a block of code that works under this base point assumption - took me a while that the term "transformation" was what I would have called a "reference pose".
When reading a program, is it easy to tell what each part is for in the overall scheme?	I think so.
Are there some parts that are particularly difficult to interpret? Which ones?	Again, the more complex things like transformations.
If the structure of the language means parts of the program depend on other parts, are those dependencies visible?	As far as I got and could see, yes, I think so.
In what ways can it get worse when you are creating a particularly large description?	Again, anything you have to write down explicitly is prone to have conceptual errors in it, that would propagate when you do copy and paste of program parts.
If a change in one part of the program requires a change in a second part of the program, is it easy to see what needs to be changed in the second part? If not, which changes are hard to see?	I did not get up to this level of complexity with the program.
How easy is it to stop in the middle of creating a program, and check your work so far? Can you do this any time you like? If not, why not?	Could not check at all.
Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?	Could not check at all.
Can you try out partially-completed programs? If not, why not?	Could not check / try at all.

Question	Answer1

When you are working with the DSL, can you go about the job in any order you like, or does the language force you to think ahead and make certain decisions first?	It forces me to think if I need to program based on text. If it was possible to actually program by showing and then just say "transform this to this new point" things were much easier.
If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?	If you know in advance that the code *should* be packaged in a function, you have to think carefully yourself which parts are the fixed ones and which ones are parameters that can change - in the "air", in my case.
Where different parts of the language describe similar things, is the similarity clear from the way they appear?	yes, I guess so.
Does the language force you to start by defining terms before you can do anything else?	depends on what you want to do.
Are there places where some things ought to be similar, but the notation makes them different? What are they?	Have not found any.
After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they? Could it be improved specifically for your own requirements?	A connection to a graphical / kinaesthetic teaching interface is needed, otherwise we do not win anything.
Can you give some examples of thing you would want to be able to do that can not currently be done (easily or at all) with the language?	I am not sure I explored everything, which means I have not found the limitations either.
Do you receive the feedback/validation that you need from the system regarding errors/program quality? Do you have suggestions regarding feedback/validation that can be added or changed?	Could not test / run.

Chapter 7
Conclusions

We have shown that building a DSL for robot arms on top of ROS/MoveIt is feasible. We have also proven one relatively simple way to do it through JFlex/Beaver/Jastadd targeting C++/ROS/MoveIt. Our hope is that the system can be used as a starting point for further inquiry.

Our system in its current state is still far from useful, however.

Some planned features were either not implemented or do not currently work, and thus couldn't be tested. Simultaneous movement with MoveIt is a major feature which we have not implemented correctly. Fixing the broken features would allow for a better study on the language's practical application.

More work is warranted to investigate practical usability of such a DSL. We spent very little time on the syntax and semantics of the language, so both are probably sub-optimal and would benefit from more focused research.

One of our goals was to provide a framework upon which more development tools can be built. Such tools were outside the scope of our project, but would represent a further avenue of research.

One such tool would be a GUI interface for live programming, roughly matching that of Stenmark's system. Our belief is that implementing such an interface on top of, rather than integrated in, a simple text-based system will both be easier and result in a more stable environment. This also remains to be tested.

Another useful tool would be a more advanced editor, or more likely, better support for existing editors, possibly replacing a dedicated GUI. An attempt at such a plugin was made for GNU Emacs, but abandoned at the proof-of-concept stage.

Better integration into the wider ROS ecosystem could also be worthwhile. A large number of tools and utilities already exist to help robot programmers, which could save significant development time for a future development system.

References

[1] A. Blackwell, C. Britton, Anna Cox, Thomas Green, C.A. Gurr, Gada Kadoda, Maria Kutar, Martin Loomes, Chrystopher Nehaniv, Marian Petre, C.R. Roast, Chris Roe, A. Wong, and R. Young. *Cognitive Dimensions of Notations: Design Tools for Cognitive Technology*, pages 325–341. 01 2001.

[2] Sachin Chitta Nikolaus Correll David Coleman, Ioan A. Şucan. Reducing the barrier to entry of complex robotic software: a moveit! case study. *Journal of Software Engineering for Robotics*, 5:3–16, 2014.

[3] Görel Hedin. *An Introductory Tutorial on JastAdd Attribute Grammars*, pages 166–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[4] M. Loetzsch, M. Risler, and M. Jungel. Xabsl - a pragmatic approach to behavior engineering. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5124–5129, 2006.

[5] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering in Robotics (JOSER)*, 7(1):75–99, 2016.

[6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[7] Gil Jones Ioan Sucan John Hsu Sachin Chitta, Kaijen Hsiao. http://wiki.ros.org/srdf/review.

[8] Reid Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of (IROS) IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 1931 – 1937, October 1998.

[9] Maj Stenmark. *Intuitive Instruction of Industrial Robots : A Knowledge-Based Approach*. PhD thesis, Lund University, 05 2017.

[10] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann. A new skill based robot programming language using uml/p statecharts. In *2013 IEEE International Conference on Robotics and Automation*, pages 461–466, 2013.

Appendices

Appendix A
Simplified versions of programs written in Stenmark's language

A.1 bagge

```
Right:
    BaggeSkillen

Skills:
    Majs Lego skill:
        Locate Large1
        Open
        Move relative to largeLego
        Move relative to largeLego
        Close
        Move
        Move (Tower)
        Contact (Tower)
        Move (Tilt)
        Open
        ContactL
        Move (ViaRetract)

    BaggeSkillen:
        Locate smallLego
        Move relative to smallLego
        Open
        Move relative to smallLego
        Close
        Move relative to smallLego
        Move
        Move
        ContactL
        Open
        Move
        Close
        ContactL
        Move
```

A.2 exp

```
Right:
    MasterSlave leftFlange

Left:
    Move
    MasterSlave

Objects:
    leftFlange
    rightFlange
    yellowInward
    greenOutward
    yellow2
```

A.3 flavius

```
Right:
    Locate small2
    Open
    Move relative to SmallLego
    Close
    Move
    ContactL
    Open

Skills:

    RightSkill:
        Move
        ViaMove
        Open
    MajLegoSkill:
        ...
```

A.4 give

```
Right:
    SyncMove
    Open
    SyncMove
    SyncMove
    Close
    MasterSlave relative to leftFlange
    MasterSlave relative to leftFlange
    Open

Left:
    SyncMove
    Open
    SyncMove
    SyncMove
    Close
    MasterSlave relative to armour
    MasterSlave
    Open

Objects:
    rightFlange
    pink
    blue
    armour
```

A.5 maxSkill

```
Right:
  No Actions

Left:
  No Actions

Skills:
  MajLegoSkill
    ...
  BaggeSkillen
```

```
    ...
  MaxLEGO
    Move To position pos with speed v
    Locate small Lego
    Move in relation to small Lego
    Open Hand
    Move in relation to small Lego
    Move in relation to small Lego
    Close Hand
    Move in relation to small Lego
    Move
    ContactL position, speed, desired torque
    ContactL
    Open Hand
    Move
    ContactL
    Move
```

A.6 paket

```
Right:
  ViaMove



Left:
  ViaMove

Objects:
  leftFlange
  rightFlange
  lid1
  lid2
  corner
```

A.7 paket1

```
Right:
  ViaMove
  Open
  Sync1

Left:
  ViaMove
  Open
  ViaMove
  Move
  Move
  Move
  Sync1

Objects:
  leftFlange
  rightFlange
  lid1
  lid2
  corner
```

A.8 presen

```
Right:
  Open
  ViaMove
  Move in relation to lid
  Move in relation to lid
  Sync1
  Close
  SyncMove (Slave1) in relation to leftFlange
  SyncMove (Slave2) in relation to leftFlange
  SyncMove (Slave3) in relation to leftFlange
  SyncMove (Slave4) in relation to leftFlange
  Open
  SyncMove (Slave6) in relation to leftFlange

Left:
  Open
  ViaMove
  Move (lid)
  Sync1
  Close
  SyncMove (Master1)
  SyncMove (Master2)
  SyncMove (Master3) in relation to box
  SyncMove (Master4) in relation to box
  Open
  SyncMove (Master6) in relation to box

Skills:
  MoveLidRight:
    Open
    ViaMove
    Move (lid)
    Move (lid)
    Sync1
    Close
    SyncMove (slave 11) leftFlange
    SyncMove (slave 21) leftFlange
    SyncMove (slave 31) leftFlange
    SyncMove (slave 41) leftFlange
    Open
    SyncMove (slave 61) leftFlange

  MoveLidLeft:
    Open
    ViaMove
    Move (lid)
    Move (lid)
    Sync
    Close
    SyncMove (Master11)
    SyncMove (Master21)
    SyncMove (Master31) box
    SyncMove (Master51) box
    Open
    SyncMove (Master61) box

Objects:
  leftFlange
  rightFlange
  lid
  box
```

A.9 sidefold1

```
Right:
  ViaMove
  Open
  Sync1
  Move corner
  Move corner
  HandMove
  Move corner
  HandMove
  Move corner
  Sync
  Move corner
  Move corner
  HandMove
  Move corner

Left:
  ViaMove
  Open
  ViaMove
  Move
  Move
  Sync
  Sync
  Move

Objects:
  leftFlange
  rightFlange
  lid1
  lid2
  corner
```

A.10 sidefold2

```
Right:
  ViaMove
  Open
  Sync
  Move relative to corner
  Move relative to corner
  HandMove
  Move relative to corner
  HandMove
  Move relative to corner
  Sync
  Move relative to corner
  Move relative to corner
  HandMove
  Move relative to corner

Left:
  ViaMove
  Open
  ViaMove
  Move
  Move
  Sync
  Sync
  Move

Objects:
  leftFlange
  rightFlange
  lid1
  lid2
  corner
```

A.11 sidefold3

```
Right:
  ViaMove
  Open
  Sync
  Move relative to corner
  Move relative to corner
  HandMove
  Move relative to corner
  HandMove
  Move relative to corner
  Sync
  Move relative to corner
  Move relative to corner
  HandMove
  Move relative to corner

Left:
  ViaMove
  ViaMove
  Open
  ViaMove
  Move
  Move
  Move relative to corner
  Sync
  Sync
  Move

Objects:
  leftFlange
  rightFlange
  lid1
  lid2
  corner
```

A.12 WorkSpaceMajLego

```
Right:
  Locate Large lego
  Open
  ViaMove relative to LargeLego
  ViaMove relative to LargeLego
  Move relative to LargeLego
  Close
  ViaMove relative to LargeLego
  Move
  ContactL
  Move
  Open
  ContactL
  ViaMove

Skills:
  RightSkill:
    ...
  MajLegoSkill:
    ...
```

A.13 WorkspaceMajUseMe

```
Right:
  majLegoSkill

Left:

Skills:
  majLegoSkill:
    Locate(Large lego)
```

Appendix B

Syntax

```
program --> statement*

statement --> assignment
statement --> variable_declaration
statement --> function_declaration
statement --> call_statement
statement --> return_statement
statement --> sync_statement
statement --> repeat_statement
statement --> transformation_statement
statement --> is_statement

assignment --> ID '=' expression
variable_declaration --> Type ID ['=' expression]
function_declaration --> Type [Component '.'] ID '(' parameter* ')' block
call_statement --> call_expression
return_statement --> 'return' expression
sync_statement --> 'sync' '[' Component* ']'
repeat_statement --> 'repeat' ['(' expression ')'] block
transformation_statement --> 'transform' expression block
    | Transformation_Literal block
is_statement --> Component 'is' Component
```

```
expression --> expression '<<' expression
expression --> expression '+' expression
expression --> expression '-' expression
expression --> expression '*' expression
expression --> expression '/' expression
expression --> '-' expression

expression --> variable
expression --> call_expression

expression --> NUM
expression --> CVector_Literal
expression --> JVector_Literal
expression --> Transformation_Literal

CVector_Literal --> '[' NUM* ']'
JVector_Literal --> '<' NUM* '>'

Transformation_Literal --> 'Scale' expression
Transformation_Literal --> 'Translate' expression

Transformation_Literal --> 'RotateX' expression
Transformation_Literal --> 'RotateY' expression
Transformation_Literal --> 'RotateZ' expression

Transformation_Literal --> 'MirrorX'
Transformation_Literal --> 'MirrorY'
Transformation_Literal --> 'MirrorZ'

variable --> ID
call_expression --> [Component '.'] ID '(' expression* ')'

block --> BEGIN_INDENTATION statement* END_INDENTATION
parameter --> Type ID

Type --> ID
Component --> ID
```

Appendix C

Feature Suggestions at Robotics Workshop after Iteration 1

This is a list of suggestions that we prepared for the workshop in order to see if there was interest from the roboticist. We have added their comments below each feature description. At the end we have appended a list of spontainious questions from the roboticists that were asked after the demonstration of iteration 1 but before we went through the feature list.

C.0.1 Suggested Features

Live Programming Support

Description
To get the current position of the robot while programming in order to be able to put it in a vector literal.
Response
Yes, this would be very useful. Elin would actually want a graphical user interface and that is something that might be developed in the future. But since this feature will save a lot of time for the programmer and since we don't have time to make a complete GUI, creating an Emacs Mode is a good investment.

Making Component into a legitimate data type

Description
At the moment components are treated as entirely different objects from ordinary types such as float or CVector. This means that:

- components cannot be used as arguments to functions.
- components cannot be returned from functions.

- components cannot be variables which can later be reassigned.

If we change this, two important use cases are enabled:

```
left_arm is Arm
right_arm is Arm

left_arm.pickUpBlock()
left_arm.handOver(right_arm)

void Arm.handOver(Arm other)
    ...
```

and

```
left_arm is Arm
right_arm is Arm

left_gripper is Gripper
right_gripper is Gripper

Gripper left_arm.gripper()
    return left_gripper

Gripper right_arm.gripper()
    return right_gripper

void Arm.pickUpBlock(CVector x, CVector aboveX)
    Arm.moveTo(aboveX)
    Arm.gripper().open()
    Arm.moveTo(x)
    Arm.gripper().close()
    Arm.moveTo(aboveX)
```

The second use case can also be enabled in the following way:
When we scan the SRDF file we can see which components are directly connected to other components, and make sure that the states such as open() and close() can be accessed as part of those components aswell. In that case Arm.open() is defined because:

- open() is defined on both left_gripper and right_gripper because it is found in the SRDF file.
- both left_gripper and right_gripper are directly connected to left_arm and right_arm respectively and open() is therefore defined on both left_arm and right_arm.
- Since left_arm and right_arm are the only members of Arm, open is also defined on Arm.

Response
Seems like it might be useful but it's not super important right now.

Allowing Access to Robot State and Settings

Description
In order to be able to get information from the environment it would be nice to have things like component.getPosition() etc. A similar thing that would be great to have is getters and setters for things like maximum speed and maximum force.
Response
Seems useful and important.

More control structures + boolean + int

Description
If statements and while statements. If this is added it also makes sense to add ints and booleans.
Response
This is useful and will probably be added in the future. However it is not important to add right now. Other similar languages already have if and while so this would not make our unique.

Other forms of synchronization

Description
A sync statement that syncs all components (rather than just the set that you specify).
An asymmetric sync statement (one component needs to wait for the other but not the other way around).
Response
Sync for all: seems useful but not super important.

Asymmetric Sync: Cannot come up with concrete use case on the spot, but it seems like something that would be interesting to play around with.

More specific JVector types

Description
Right now you don't get a type error if you use a JVector on a component that it is not meant for. You don't even get a type error if it is of the wrong size.

Could be a good idea to have one type of JVector for every size or maybe even for each component.
Response
Yes, that would be good. One for each component is probably best.

Explicit Declaration of abstract components and abstract functions

Description
abstract Arm
abstract void Arm.pickUp(CVector pos)

This does not add any capabilities to the language and it makes your programs a bit longer. However it might improve readability.

Response
Not super important but might make it a bit clearer.

Expressions in Vector literals

Description
Right now vector literals have to consist of float literals. Might be a good idea to allow expressions there.
Response
Cannot think of concrete use case on spot, but seems useful.

Explicit units

Description
Make it possible to write down units explicitly and they are automatically converted to whatever unit is actually needed.
JVector x = <10 deg 3.14 rad ...>
Response
Seems useful.

Comments

Description
Add ability to comment out lines so we don't have to delete them.
Response
Very important.

Global variables for directions

Description
CVectors for up, down, forward etc.
Response
Seems useful

C.0.2 Spontaneous Questions

These are some of the most important comments, suggestions and questions that we got:

- Will there be a user interface so that it is not just text based and so that live programming is possible? Jogging.
- Master Slave moves?
- Is there support for multiple robots?

- Couldn’t there be situations where it would be easier to use arithmetic on JVectors than on CVectors? For example when screwing in a light bulb, it might be easier to tell the last joint to spin a certain number of degrees than trying to use arithmetic on euler angles.
- Is it possible to switch controller?
- Would it be possible to read entire trajectories during live programming rather than just single points.
- Instead of having to write ’Sync’ every time we need to sync, would it be possible to have special blocks or something where everything is synced?
- Could we add some way to do mirrored versions
- Contact moves

Appendix D
Evaluation Survey

D.1 Questions

Questions from Cognitive Dimensions PART A ======

– How easy is it to see or find the various parts of the program while it is being created or changed? Why?

– What kind of things are more difficult to see or find?

– If you need to compare or combine different parts, can you see them at the same time? If not, why not? [not included]

PART B ======

– When you need to make changes to previous work, how easy is it to make the change? Why?

– Are there particular changes that are more difficult or especially difficult to make? Which ones?

PART C ======

– Does the language a) let you say what you want reasonably briefly, or b) is it long-winded? Why?

– What sorts of things take more space to describe?

PART D ======

– What kind of things require the most mental effort with this language?

– Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

PART E ======

– Do some kinds of mistake seem particularly common or easy to make? Which ones?

– Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

PART F ======

– How closely related is the program to the result that you are describing? Why?

– Which parts seem to be a particularly strange way of doing or describing something?

PART G ======

– When reading a program, is it easy to tell what each part is for in the overall scheme? Why?

– Are there some parts that are particularly difficult to interpret? Which ones?

– Are there parts that you really don't know what they mean, but you put them in just because it's always been that way? What are they?

PART H ======

– If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden?

– In what ways can it get worse when you are creating a particularly large description?

– Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?

- If a change in one part of the program requires a change in a second part of the program, is it easy to see what you need to change in the second part of the program? If not, which required changes are hard to see?

PART I ======

– How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not?

– Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?

– Can you try out partially-completed versions of the product? If not, why not?

PART J ====== [Not sure if this will give us very much. I don't see how this would be very different in our language compared to other languages]

– Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this?

– What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

PART K ======

– When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?

– If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

PART L ====== – Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Please give examples.

– Are there places where some things ought to be similar, but the notation makes them different? What are they?

PART M ====== – Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation?

– If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw?

– Do you ever add extra marks (or colours or format choices) to clarify, emphasise or repeat what is there already? [If yes: does this constitute a helper device? If so, please fill in one of the section 5 sheets describing it]

[I would suggest dividing this into two questions: Do you use comments, what do they look like/what do you use them for? and Do you write pseudocode before writing code and what does the pseudocode look like?]

PART N ======

[seems like we already know the answer to this one] – Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they?

[could be relevant] – Does the system insist that you start by defining new terms before you can do anything else? What sort of things?

[this seems unnecessary] – If you wrote here, you have a redefinition device: please fill in one of the section 5 sheets describing it.

Finally we ask two further questions which may evoke usability problems that are not addressed by any of the dimensions, or that may allow the respondent to express some problem that other dimensions have reminded them of: [These seem very relevant]

– Do you find yourself using this notation in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?

– After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they? Could it be improved specifically for your own requirements?

Other questions

Can you give some examples of thing you would want to be able to do that can not currently be done (easily or at all) with the language? (add runtime environment tooling)

Do you receive the feedback/validation that you need from the system regarding errors/program quality? Do you have suggestions regarding feedback/validation that can be added or changed?

Would you use all the features of the language?

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD ???

EXAMENSARBETE Designing a Domain Specific Language for Robotics
STUDENTER Emma Grampp, Stefan Jonsson
HANDLEDARE Christoph Reichenbach
EXAMINATOR Jesper Öqvist

Ett programmeringsspråk för robotarmar

POPULÄRVETENSKAPLIG SAMMANFATTNING **Emma Grampp, Stefan Jonsson**

Industrirobotar blir allt vanligare och får hela tiden nya finesser, såsom fler och smidigare armar. De är dock fortfarande knepiga att programmera. Vi har arbetat med att ta fram ett mer specifikt språk för att styra dem.

Industrirobotar har länge bara varit till för de allra största fabrikerna. Nu börjar det byggas små och mer fingerfärdiga robotar även för mindre arbetsplatser. Det råder ingen brist på enkla och repetitiva jobb som de skulle kunna ta över. Programmeringsverktygen har dock ännu inte helt hunnit ikapp. De verktyg som finns är ofta proprietära, d.v.s. de är skapade av en tillverkare och fungerar endast för dennes robotar. De är därtill sällan enkla att använda, särskilt när robotar har mer än bara en enkel arm.

På LTH har det under en tid pågått forskning om verktyg för robotprogrammering. Det har bl.a. sammanställts en större mängd standardprogram som ett system måste kunna representera. Hittills har systemen som tagits fram varit hela grafiska program, vilket gjort dem svåra att återanvända. Det finns därför en önskan om att skapa ett textbaserat grundsystem, som mer avancerade grafiska verktyg kan byggas ovanpå.

I vårt examensarbete har vi tagit fram ett programspråk för att styra industrirobotar, och byggt en kompilator med tillhörande runtime-system som kan köra programmen. Vi utnyttjar det existerande ramverket MoveIt, som är en del av projektet "Robot Operating System", förkortat ROS.

Ett program i vårt språk beskriver en sekvens av rörelser som de olika komponenterna av en robot ska genomföra när programmet körs. Flera komponenter av roboten kan vara i rörelse samtidigt. Språker erbjuder abstraktion över både komponenter och handlingar (sekvenser av rörelser). Handlingar kan även transformeras på olika sätt, så att de till exempel kan göras spegelvänt.

Vi har utvärderat vårt system dels genom att översätta en mängd standardprogram till det, och dels genom att låta robotikexperter programmera i det. Vi kom fram till att det i princip är en lovande väg att gå, men fortfarande kommer kräva mycket mer arbete för att få fram ett användbart verktyg.