The MetaForma Language

A DSL to Program the ATRON Self-Reconfigurable Robot

Wouter van Koppen
The MetaForma Language

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Wouter van Koppen
born in Naaldwijk, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Engineering Research Group
University of Southern-Denmark (SDU)
Department of Modular Robotics Lab
Odense, Denmark
www.sdu.dk
Abstract

Self-reconfigurable modular robots are mechatronic devices that have the capability to modify their own shape. They have a promising potential as a result of being able to solve a wider range of problems. However, programming self-reconfigurable robots is in general a difficult task. General-purpose languages like ANSI-C or Java provide only general abstractions that do not facilitate tackling difficult issues such as distribution, unreliable hardware, and dynamically evolving communication topology. We believe that a domain-specific language is needed to be able to program robot behavior on a higher, more productive level. Existing DSLs and control approaches have been developed in the past, but were designed to solve problems specific to certain scenarios.

In this work, a domain analysis is used to identify and define concepts in existing approaches. Based on this, a prototype for a new DSL is implemented in the language-workbench Spoofax. The prototype is called MetaForma and comprises a compiler with Eclipse IDE support and a static runtime-system. The runtime-system acts as an abstraction on top of the ATRON robot and the USSR robot simulator. The USSR simulator is used to conduct experiments on generated Java code that simulates the behavior of the ATRON robot and justifies claims for robustness. The contribution of MetaForma is twofold. First, it supports shape-transformations and locomotion of small ATRON ensembles. Second, it supports locomotion methods for large-scale ATRON ensembles, that were published in earlier work but not implemented in a language before. Although not all solutions are yet implemented in the MetaForma runtime-system, and it currently only runs in simulation, we believe that MetaForma is a powerful DSL solution for the ATRON robot.
Thesis Committee:

Chair: dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor: dr. E. Visser, Faculty EEMCS, TU Delft
External supervisor: dr. U. Schultz, MIP, University of Southern Denmark
Committee Member: dr. Birna van Riemsdijk, Faculty EEMCS, TU Delft
Preface

This document is the result of my master-thesis project conducted at the University of Southern-Denmark (SDU) in Odense. Although I was having a great time when studying for my master degree at Delft University of Technology, I wanted to conduct my thesis project abroad, influenced by my interest in travelling. It was a guess to go alone and for almost a year to Denmark, a country that I had never been before, but at this moment I can say it was a lucky one.

First, I would like to thank my daily supervisor dr. Ulrik Pagh Schultz from the SDU, for assisting me in the rehousing to Odense, for supervising me during the entire project and for sharing his immense knowledge about programming robotics with me. He is a truly expert in this field, and this work would have not been possible without his ideas, inspiration and comments.

Second, I would like to thank dr. Eelco Visser for supervising me from The Netherlands and for his suggestion and idea to do my project in Denmark. In addition, I would like to thank the following people: dr. David Brandt for sharing his knowledge about robots and meta-module approaches, ir. Anders Clausen for his helpful and constructive feedback on my thesis, dr. Kasper Støy for his feedback on my presentation, and ir. Gabriel Konat for his support on my language implementation in Spoofax.

Third, bonus points for the makers of Spoofax, LaTeX, BibTeX, and the excellent TeXWorks IDE with integrated PDF-viewer. Also, the research-paper organising tool Mendeley makes carrying out a research significantly easier. When printing this document, perhaps it is best to print in color format, as otherwise some figures and pseudo-codes become difficult to read.

Finally, I am thankful that I had the opportunity to have met so many new friends and awesome people in Denmark, and for learning those cultural-related things that a student cannot learn at school. My stay at the international student-dormitory Rasmus Rask Kollegiet, housing international students from all over the world, has become a valuable experience in my life.

Wouter van Koppen
Delft, the Netherlands
March 12, 2013
Contents

Preface iii

Contents v

List of Figures xi

1 Introduction 1

1.1 What is a self-reconfigurable Robot? 1

1.1.1 Large-scale ensembles 2

1.1.2 Docking and splitting up 3

1.1.3 Potentials of self-reconfigurable robots 4

1.2 Programming Systems for Self-reconfigurable Robots 5

1.2.1 Automatic Programming Systems 5

1.2.2 Manual Programming Systems 6

1.3 Project Goals 7

1.3.1 Relevance for the University of Southern Denmark 7

1.3.2 Relevance for the Delft University of Technology 7

1.4 Research Questions 8

1.5 Methodology 9

1.6 Contributions 10

1.7 Thesis Overview 10

2 Programming Modular Robots 13

2.1 Modular Robots 13

2.1.1 Heterogeneous and Homogeneous Modular Robots 13

2.1.2 Chain, Lattice and Hybrid Structures 15

2.1.3 The ATRON Self-Reconfigurable Robot 16

2.2 The USSR: Unified Simulator for Self-reconfigurable Robots 18

2.2.1 The ATRONController class 19

2.2.2 A Hello World Simulation Example 20

2.2.3 The Assemble and Animate (ASE) Framework 22
## CONTENTS

2.3 Distribution of Controller Implementations ................................................. 23
2.4 Control Algorithm approaches ...................................................................... 24
  2.4.1 Off-line and On-line control algorithms .................................................. 24
  2.4.2 Centralized Control versus Decentralized Control ................................. 25
2.5 Control using a Subsumption Architecture .................................................. 25
2.6 Summary ........................................................................................................ 26

3 DSLs for Modular Robots .................................................................................. 27
  3.1 Introduction .................................................................................................... 27
  3.2 UrbiScript ...................................................................................................... 28
    3.2.1 Concurrency Support ........................................................................... 29
    3.2.2 Time Management ................................................................................ 29
    3.2.3 Event-Driven Communication ............................................................... 30
  3.3 Phase Automata Robot Scripting Language (PARSL) ................................... 30
  3.4 Locally Distributed Predicates (LDP) ............................................................. 32
  3.5 Meld ............................................................................................................. 34
  3.6 Functional Programming Languages: Proto and Regiment .......................... 34
  3.7 DynaRole: A Virtual-Machine Architecture ................................................ 35
    3.7.1 The Distributed-Control Diffusion Virtual Machine (DCD-VM) ................. 35
    3.7.2 The DynaRole Language ....................................................................... 36
    3.7.3 DynaRole using a Python-based internal DSL ........................................ 38
    3.7.4 Analysis ................................................................................................. 38
  3.8 Robust and Reversible Sequences in DynaRole .......................................... 39
    3.8.1 Syntax and Semantics of Sequences ...................................................... 39
    3.8.2 Compilation Backend ........................................................................... 40
    3.8.3 Reversibility of Sequences .................................................................... 42
  3.9 Summary ........................................................................................................ 42

4 Domain Analysis ................................................................................................ 45
  4.1 Modular Robots ............................................................................................. 45
    4.1.1 Reconfigurable vs. Self-reconfigurable .................................................. 45
    4.1.2 Existing scenarios on the ATRON robot .................................................. 46
  4.2 Shape Transformations of Small-scale ensembles ......................................... 46
    4.2.1 Identification of Modules ..................................................................... 46
    4.2.2 Sequential execution of actions .............................................................. 47
  4.3 Small-scale Locomotion: Obstacle Avoidance .............................................. 48
  4.4 Cluster-flow devised by Christensen et al. ...................................................... 50
    4.4.1 Meta-modules ....................................................................................... 50
    4.4.2 Cluster-flow: Clusters of Three Modules .............................................. 51
    4.4.3 Symmetry preservation ........................................................................ 53
    4.4.4 Cluster-flow: Clusters of Four Modules .............................................. 54
  4.5 Cluster-flow devised by Brandt et al. ............................................................. 55
    4.5.1 Meta-module moves ............................................................................. 56
    4.5.2 Defining global behavior: Strategies ..................................................... 57
4.5.3 Meta-action One: Flip-Along ........................................... 58
4.5.4 Meta-action Two: Flip-Over ........................................... 59
4.5.5 Meta-action Three: Flip-Through ................................. 61
4.6 Runtime System ........................................................... 62
4.7 Summary ..................................................................... 63

5 Language Design of MetaForma ........................................ 65
  5.1 Language Architecture ................................................ 65
  5.1.1 Domain-Specific Language vs. General-Purpose Language 65
  5.1.2 Motivation for DSL approach .................................... 66
  5.1.3 Internal DSL vs. External DSL .................................. 66
  5.1.4 Language Architecture: External DSL ......................... 66
  5.1.5 Motion and Coordination Language ........................... 67
  5.1.6 Strategy Language .................................................... 67
  5.2 Obstacle Avoidance Strategy ........................................ 68
  5.3 Two-Wheeler Car Obstacle Avoidance ......................... 69
  5.3.1 Init sequence ......................................................... 69
  5.3.2 Turn Sequence and Drive Behavior .......................... 71
  5.4 Four-wheeler Car Obstacle Avoidance ......................... 72
  5.4.1 Init Sequence ....................................................... 73
  5.4.2 Turn Sequence and Drive Behavior .......................... 74
  5.5 Cluster-Flow devised by Christensen et al. .................. 75
  5.5.1 Create Grid of Meta-modules ................................. 75
  5.5.2 High-level Strategy for Cluster-flow ......................... 77
  5.5.3 Defining Meta Neighbors ...................................... 78
  5.5.4 Module gradients ................................................. 79
  5.5.5 LiftUp sequence .................................................... 80
  5.5.6 WalkStep sequence ................................................. 81
  5.5.7 LiftDown sequence ............................................... 83
  5.6 Cluster-Flow devised by Brandt et al. ...................... 85
  5.6.1 Divide structure into meta-modules ....................... 85
  5.6.2 High-level Strategy for Cluster-flow ....................... 87
  5.6.3 Detecting meta-neighbors ................................. 87
  5.6.4 Flip-Through meta action .................................. 89
  5.7 Summary ..................................................................... 90

6 MetaForma Language Implementation .................................. 93
  6.1 The Spoofax Language Workbench ............................. 93
  6.1.1 Syntax Definition using SDF ................................. 93
  6.1.2 Program Transformation using Stratego .................. 93
  6.1.3 Name Binding using NBL ..................................... 94
  6.1.4 Editor Services ..................................................... 94
  6.2 MetaForma Implementation in Spoofax .................... 96
  6.2.1 Parsing step .......................................................... 96
9 Perspectives  
9.1 Related Work .................................................. 137
  9.1.1 Learning Systems ........................................ 137
  9.1.2 Rule-based control ...................................... 138
  9.1.3 Spatial programming ..................................... 138
  9.1.4 DSL approaches ........................................... 138
9.2 Evaluation ...................................................... 139
9.3 Conclusions .................................................... 140
9.4 Future work ..................................................... 144
  9.4.1 Support for Physical ATRON Modules ................. 144
  9.4.2 Automatic programming system for Motion and Coordination language 145
  9.4.3 3D symmetry preservation ............................. 145
  9.4.4 Sophisticated Gradient Protocol ....................... 146
  9.4.5 Algorithm for Parallel Lifting ......................... 146
  9.4.6 Collision Detection and Anticipation .................. 147
  9.4.7 Multiple Module Lookahead and Custom Variables in Neighbor Table 147
  9.4.8 Better debugging support .............................. 148
  9.4.9 Improved meta-neighbor system ....................... 148

Bibliography ....................................................... 151

A MetaForma DSL programs .................................... 157
  A.1 Cluster-flow devised by Brandt et al. .................. 157
  A.2 Implementation of the Eight2Car sequence ........... 161

B Results of Experiments ........................................ 163
  B.1 Obstacle Avoidance ....................................... 163
  B.2 Cluster-flow devised by Christensen et al. .......... 164
  B.3 Cluster-flow devised by Brandt et al. ................. 167

C Glossary .......................................................... 169
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Examples of conventional robots</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>The PolyBot Self-Reconfigurable Robot [80]</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>The ATRON Self-Reconfigurable Robot</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>Simulated large-scale ensembles of ATRON modules</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>Three single ATRON modules moving together as a meta-module (red) [28]</td>
<td>3</td>
</tr>
<tr>
<td>1.6</td>
<td>Two-wheeler ATRON car docking onto an existing robot [71]</td>
<td>4</td>
</tr>
<tr>
<td>1.7</td>
<td>Examples of Graphical Languages</td>
<td>7</td>
</tr>
<tr>
<td>1.8</td>
<td>The Molecube robot [85]</td>
<td>9</td>
</tr>
<tr>
<td>1.9</td>
<td>The M-TRAN robot [47]</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>The ODIN Heterogeneous Reconfigurable Modular Robot</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Eight ATRON connectors and infra-red communication channels (two at the back)</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Communication example between two ATRON modules</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Class Infrastructure for a simple robot simulation</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>The USSR simulating a Two-wheeler car driving to an obstacle</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Architecture of ASE [27]</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>A traditional decomposition of a mobile control system into functional modules</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>A decomposition of a module robot control system based on task achieving</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Job control in UrbiScript [6]</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Flow control in UrbiScript [61]</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Events in UrbiScript using payload [4]</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Snake-Gait simulation on PolyBot [83]</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>A “Hello World” example of PARSL: A Snake Gait</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Snake Gait example using LDP: Black and white modules are actuating a negative and positive joint angle, respectively [61]</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>ATRON Two-Wheeler</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>The Eight-to-Car self-reconfiguration sequence [65]</td>
<td>39</td>
</tr>
<tr>
<td>3.9</td>
<td>Distributed state machine, derived from the eight-to-car sequence</td>
<td>41</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>3.10</td>
<td>The global state merge function on the currently active state (s_i), the address of the active module (a_i), and the set of pending states (P_i).</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Connector sets for the ATRON robot</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>The eight to snake self-reconfiguration sequence</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Simple obstacle-avoidance behavior for a Two-Wheeler and Four-Wheeler Car</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>State diagram for basic obstacle avoidance</td>
<td>49</td>
</tr>
<tr>
<td>4.5</td>
<td>Lifting the walker up</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>A single walking step, which must be repeated until the end of the ensemble is reached</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Meta-module merges back in structure</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Symmetry issue after a self-reconfiguration sequence</td>
<td>53</td>
</tr>
<tr>
<td>4.9</td>
<td>Meta-parts assigned to modules in a meta-module, to distinguish between neighbors</td>
<td>55</td>
</tr>
<tr>
<td>4.10</td>
<td>L1,R1 and L2,R2 are the left and right lifters for meta-module 1 and 2, respectively, resulting in the problem that R1 and L2 are physically the same</td>
<td>55</td>
</tr>
<tr>
<td>4.11</td>
<td>Meta-modules devised by Brandt et al. (2007)</td>
<td>56</td>
</tr>
<tr>
<td>4.12</td>
<td>Three different basic meta-module moves. Each dark grey square represents a meta-module, while a white square represents an empty square. For the light grey squares there is no constraint upon the presence of a meta-module or not.</td>
<td>56</td>
</tr>
<tr>
<td>4.13</td>
<td>Different orientations shown for meta-actions: The Flip-Along and Flip-Through have eight different orientations, while the Flip-Over has only four.</td>
<td>57</td>
</tr>
<tr>
<td>4.14</td>
<td>A strategy for cluster-flow in northern direction. Note that in this meta-module setup, the second rule of the strategy is executed three times before rule the third rule will be applied</td>
<td>57</td>
</tr>
<tr>
<td>4.15</td>
<td>A group of eight ATRON modules performing the Flip-Along meta-action like shown in Figure 4.12 (a)</td>
<td>58</td>
</tr>
<tr>
<td>4.16</td>
<td>A group of twelve ATRON modules performing the Flip-Over meta action Figure 4.12 (b)</td>
<td>60</td>
</tr>
<tr>
<td>4.17</td>
<td>The use of a gradient for assigning groups</td>
<td>61</td>
</tr>
<tr>
<td>4.18</td>
<td>A group of eight ATRON modules performing the Flip-Through meta action like shown in Figure 4.12 (c)</td>
<td>62</td>
</tr>
<tr>
<td>4.19</td>
<td>The Runtime System provides abstractions for different control programs</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>MetaForma Language Architecture: External DSL approach</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Module identification for a Two-Wheeler car</td>
<td>70</td>
</tr>
<tr>
<td>5.3</td>
<td>Four-wheeler car</td>
<td>74</td>
</tr>
<tr>
<td>5.4</td>
<td>Connector sets for the ATRON module</td>
<td>74</td>
</tr>
<tr>
<td>5.5</td>
<td>Structure divided into grid of meta-modules</td>
<td>76</td>
</tr>
<tr>
<td>5.6</td>
<td>Cluster-Flow forward</td>
<td>78</td>
</tr>
<tr>
<td>5.7</td>
<td>Gradient and module borders</td>
<td>80</td>
</tr>
<tr>
<td>5.8</td>
<td>Steps of AssignMetaID packets that are sent in the Init sequence</td>
<td>86</td>
</tr>
<tr>
<td>5.9</td>
<td>List of sequence orientations</td>
<td>87</td>
</tr>
<tr>
<td>5.10</td>
<td>Five meta-modules moving forward</td>
<td>88</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

“If the only tool you have is a hammer, you tend to see every problem as a nail.”
Abraham Maslow (1908-1970), Psychologist

Conventional robots are built from fixed hardware and have a fixed structure. Indeed, a robot which is designed to be a car and is built to have four wheels, will stay a four-wheeled car forever. The same limitation holds for a conventional two-wheeler car robot or a conventional snake robot, like shown in Figure 1.1. A conventional robot that is created for a specific task, cannot be used for something else, which is an important limitation.

On the contrary, numerous examples of science fiction films exist, about robotics that are capable to change its own shape. For example morphing robots, liquid-metal robots and nano-robots. These are robots that can fulfil tasks no-one ever thought about. Until recently, robots that can change their own shape only played a role in science fiction films, but they now exist in real life: They are known as self-reconfigurable robots.

1.1 What is a self-reconfigurable Robot?

Instead of designing a new robot for each task, we can use a robot that is capable of changing its own shape \[79, 80\]. This architecture is beyond conventional robots, which are built...
from one piece without the possibility to change the shape of its own hardware. Thus, while conventional robots modify the problem to fit its shape, self-reconfigurable robots modify its shape to fit the problem. Hence, a single type of a self-reconfigurable robot can be used to solve a wide range of problems. Figure 1.2 and Figure 1.3 show examples of self-reconfigurable robots. The modules of a self-reconfigurable robot are referred to together as an ensemble: A group of modules that is either directly or indirectly connected.

Figure 1.2: The PolyBot Self-Reconfigurable Robot [80]

(a) A single ATRON module [17] (b) Three groups of seven modules. From left to right, the configurations are snake, cluster-walk and car [58].

Figure 1.3: The ATRON Self-Reconfigurable Robot

1.1.1 Large-scale ensembles

In addition to small-scale ensembles like a snake or a car robot, self-reconfigurable robots can also exist in large ensembles such as towers, bridges and scaffolds. This is shown in Figure 1.4 (a) and (b).

A large-scale ensemble cannot be moved in one piece, modules need to be moved piece by piece from one place to another. However, for most types of self-reconfigurable robots, single modules itself are not capable to move without help. In order to move, a module needs help from its neighboring modules. Usually, to simplify the problem of moving a module,
What is a self-reconfigurable Robot?

(a) Tower [28] (b) Bridge [28]

Figure 1.4: Simulated large-scale ensembles of ATRON modules

Figure 1.5: Three single ATRON modules moving together as a meta-module (red) [28]

a so-called meta-module approach is used. A meta-module is a group of multiple single modules and is used as a single unit. This is shown in Figure 1.5. The three red modules form a meta-module that moves itself around among the ensemble of blue modules. Note that the modules are all functionally equal, and the difference in color only illustrates the idea of a meta-module.

1.1.2 Docking and splitting up

Self-reconfigurable robots are capable of merging multiple robots into one bigger robot, which is known as docking [71]. In addition, they are capable of splitting a big robot up into multiple smaller ones. A large-scale self-reconfigurable robot, consisting of many modules, can split itself up into multiple small-scale car robots that drive to a new destination. When arrived, they merge back into a large-scale structure again. For example, multiple two-wheeler car robots can merge into each other by driving one of the cars inside the other cars connectors [71]. Both robots connect to each other and result in a new single robot. This is shown in Figure 1.6.
1. INTRODUCTION

1.1.3 Potentials of self-reconfigurable robots

Self-reconfigurable modular robots can control physical connections among modules and alter their configuration. In addition, if each module is autonomous, i.e. has the capability of information processing and communication with others, and if all the modules are identical in hardware and software, the system will have important capabilities such as robustness, adaptability, scalability [46] and low-cost [72].

Robustness Robustness is defined as the ability to handle hardware and software failures [72]. As a result of the redundant structure and small number of module types, self-reconfigurable robots have the potential to become robust to partial hardware failures. The modules can diagnose themselves and each other and can compensate for, replace, or reconfigure themselves around other modules that are malfunctioning. This enables degrading performance linearly, in contrast to conventional robots that fail catastrophically when a crucial part in the robot fails. [80].

Adaptability Self-reconfigurable robots are adaptable, because they can change their shape and adapt to the environment as needed. They are suitable for applications in unknown environments such as planetary exploration and search-and-rescue operations. The possibility to adapt to the internal environment allows self-repair and scalability: If one module fails, the robot can get rid of the module and replace it, degrading its performance linearly. Adaptation to the external environment allows the robot to change such that it solves the problem in a shape that is most efficient.

Scalability In terms of communication among an ensemble, all modules can be divided among groups such that modules only communicate with neighbors in the same group, without the need to have knowledge about modules outside its group.

Low-cost Self-reconfigurable robots are low-cost as they are built from multiple equal modules which can be mass produced and hence reduce production costs. This can potentially reduce the price of the individual modules to a level where they can compete with fixed-configuration robots [58, 72, 80].
These properties are potential properties, such that in theory it should be possible to realize them based on the concept of self-reconfigurable robots. However, in practice, the properties are often realized only to a limited degree [72], and research is needed on both the hardware and software part to improve this. In order to improve the software, we must program them appropriately, i.e. we need to program self-reconfigurable robots in such a way that we can use their theoretical properties in practice. Programming them can fundamentally be done in two different ways, which will be discussed in the next section.

1.2 Programming Systems for Self-reconfigurable Robots

A robot programming system is a way to put behavior inside a robot. Principally, robot programming systems can be divided into two main categories: manual and automatic programming systems [8]. Both methods excel at different kinds of problem domains.

In manual programming, users create the program for a robot by hand by using a computer language, hence, this approach becomes suitable to solve problems in well-known domains. In contrast, when dealing with problems in unexplored domains, an automatic programming system rather comes to mind. These systems provide little or no direct control over the robot program the robot will run [8]. Instead, the code is generated from information entered into the system by using indirect ways. Both types of programming systems are discussed in the sections below.

1.2.1 Automatic Programming Systems

When using an automatic programming system, often the robot must be active and running while putting instructions into it. However, a robot simulator can also be used for programming, simulating the active robot. Automatic programming systems can be divided into three main categories, known as learning systems, programming by demonstration systems, and instructive systems.

Learning Systems

Learning systems comprise systems that start with a basic controller provided by the user, and then converges to more optimal solutions by applying machine learning. This approach excels when applied to problems in unexplored domains, as the robot tries to learn itself how to solve the problem. In theory, a learning system can be applied to any kind of problem, even when a straightforward solution may not exist. However, as a result, sometimes no guarantee can be made on the quality of a solution, or even whether a solution will be found.

A learning system can be implemented using an artificial neural network (ANN) approach [28], particle swarm optimization (PSO), ant colony optimization (ACO), genetic algorithms (GA) [36][39] e.g. for controlling the M-TRAN robot [40][45][47], and distributed reinforcement learning [54].
1. INTRODUCTION

Programming by Demonstration Systems

In case programming a robot needs to be accessible to people without any kind of programming experience, a programming by demonstration comes to mind. In this approach, a demonstrator, which has the role of programmer, performs the task at a teach pendant. A teach pendant is a robot that is able to record its overall movement, by recording all its joint positions. As the teach pendant records its movement, a controller program that is capable to repeating the movement can be generated [59].

Instructive Systems

Instructive systems are suited for commanding robots to carry out tasks that they have already been trained or programmed to and can be considered as the highest level of programming. This approach focuses entirely on what should be done instead of how. A robot using an instruction system can be given commands in real-time. Typically, commands are encapsulated in speech or gestures. In order to understand its commands, the robot uses a speech-recognizer or a combination of a camera and gesture-detector, respectively.

Lauria et al. [48] describes a natural language system to teach the robot how to move to different locations by specified routes. Natural languages are used to communicate between humans. Hence, a natural language system facilitates the interaction between natural language and computer language.

1.2.2 Manual Programming Systems

In a manual programming system, the programmer uses a computer language to write a program by hand. This can be done in a General-Purpose Language (GPL) like ANSI-C or Java, Domain-Specific Language (DSL) or even by using a graphical language.

General-Purpose Languages

GPLs are flexible in the sense that they are created for general purpose programming. On the contrary, GPLs provide only general abstractions that do not facilitate tackling difficult issues typical for self-reconfigurable robots. Programming self-reconfigurable robots directly in a GPL has been done before, and includes the ATRON [10,19] and the M-TRAN [46] robot. Nevertheless, programming self-reconfigurable robots directly in a GPL is known as a difficult task.

Domain-Specific Languages

DSLs trade in some of the flexibility of GPLs but can provide built-in abstractions for difficult issues typical for self-reconfigurable robots. Programming in a higher-level DSL is an interesting research field, as the right domain-specific abstractions potentially can make programming more easy. Fundamentally, a distinction can be made between internal DSLs, in which a GPL is used as the host language for a DSL [46,53], and external DSLs, in which a new language with its own syntax and semantics is implemented [3,4,12,56,61,64,65,83].

3 English and Chinese are examples of natural languages, while computer languages such as ANSI-C and Java are not.
1.3 Project Goals

The goal for this master thesis project is to implement a new DSL in the language workbench Spoofax [41]. Spoofax is a language-workbench platform running on Eclipse aiming the development of textual DSLs. The new DSL must improve the shortcomings in existing DSLs [12, 63, 65] for the ATRON robot.

The project is conducted at the University of Southern Denmark, in collaboration with the Delft University of Technology. The relevance of the project is twofold, as it builds on technology developed at both universities.

1.3.1 Relevance for the University of Southern Denmark

The University of Southern Denmark (SDU) conducts research in a wide range of robotics, including self-reconfigurable robots. The ATRON self-reconfigurable robot has been developed at the SDU and is still under development in terms of software for controlling the robot [10, 12, 63, 65]. New paradigms for expressing behaviour for self-reconfigurable robots are investigated, as this field is still widely unexplored.

1.3.2 Relevance for the Delft University of Technology

The Software Engineering Research Group (SERG) at Delft University of Technology conducts research in various fields of software engineering. This research includes DSLs and tools that enable DSL approaches, such as language workbenches. The language workbench
Spoofax is developed by the SERG. Spoofax integrates SDF [37] for grammar definition and Stratego [75] for model-to-model and program transformations into one formalism, in which a DSL can be defined.

1.4 Research Questions

Programming modular self-reconfigurable robots appears to be a difficult task [2,3,10,11,24,45,53,55,61,63,65,72,80]. In Section 1.2 are multiple approaches described that can be used for programming robotics. How can we use them to make programming self-reconfigurable robots more easy? This immediately leads to the first research question:

**RQ 1:** How can programming modular self-reconfigurable robots, and the ATRON robot in particular, be made easier and more accessible to the user?

In order to develop a new DSL for the ATRON robot, we analyze existing DSLs for the ATRON and other robotic systems, and identify their strong points and their infirmities. This leads to the second research question:

**RQ 2:** What DSLs currently exist for programming robotics, what are their strong points, and what are these solutions lacking?

By identifying the strong and weak points of existing languages, we can learn something about which kind of abstractions are powerful for modular robots. Object-oriented languages such as Java and C++ use classes, methods, fields and variables as built-in abstractions. Inherent to their general-purpose, these languages also support to manually implement abstractions, that the programmer can write in the language itself. In order to create a DSL for the ATRON robot, we need appropriate domain-specific abstractions that cover its domain. This leads to research question 3:

**RQ 3:** What kind of abstractions are needed in the DSL? What abstractions should be integrated in the language, and what abstractions should be expressed in the DSL itself?

Properties such as robustness and scalability are important in self-reconfigurable robots. In order to accomplish passive robustness (robustness in terms of software) during self-reconfiguration, and to abstract these properties for the programmer, a runtime system is needed that runs on the robot. This leads to the next research question:

**RQ 4:** What kind of runtime system is needed to provide the desired properties of being scalable and robust, and how do these properties relate to the DSL?

In order to address the problem of locomotion in large-scale structures, we need to integrate cluster-flow by using meta-modules (as described in Subsection 1.1.1). This is an
important aspect to integrate into the DSL. This leads to research question 5:

**RQ 5:** How can meta-modules be used to simplify the process of self-reconfiguration on the ATRON robot, and what kind of meta-modules are used until now?

Figure 1.8 and Figure 1.9 show the Molecubes [85] and M-TRAN [47] self-reconfigurable robots, respectively. These other robotic systems share similarities with the ATRON robot, so an interesting research question is how other robotic systems can benefit from the same DSL approach. This leads to research question 6:

**RQ 6:** What is needed to make the MetaForma language applicable to other types of modular robots?

It might be the case that existing programs in other languages can be expressed more efficient in the DSL. This leads to the final research question:

**RQ 7:** Can existing scenarios on modular robots be resolved more efficiently in terms of development using the MetaForma language?

1.5 Methodology

In this work, an agile development process has been used consisting of two main iterations. Both iterations followed the following path: First we conducted a domain analysis, from which we implemented an internal-DSL prototype, and later extended the prototype to an external DSL approach.

In the first iteration, we investigated obstacle-avoidance behavior for car robots and self-reconfiguration sequences that change the shape of a robot. Based on hand-written controllers in Java, conceptual abstractions were defined and high-level DSL programs were designed for both scenarios. Based on the DSL programs, a syntax definition and ‘quick n dirty’ code generator was implemented, focussing on compilation and leaving out editor services.
1. **Introduction**

In the second iteration, we investigated meta-module solutions and experimented with
hand-written Java controllers in the robot simulator, from which we designed high-level
DSL programs. DSL programs from the first iteration were revised according to the new
DSL specification. Hence, influenced by the results of the first iteration, a new DSL was
implemented, consisting of a code generator and editor services.

1.6 **Contributions**

The contributions of this thesis are the MetaForma language, the MetaForma compiler, and
the MetaForma runtime system.

**MetaForma Language** The contribution of the MetaForma language is twofold. It com-
prises a high-level strategy language, used to define global behavior for a robot. The low-level
behavior, that can be used by strategies, is implemented in the motion and coordination
language.

The MetaForma language supports the implementation of shape-transformations and
locomotion of small ATRON ensembles. Second, it supports locomotion methods for large-
scale ATRON ensembles, that were published in earlier work [16,29] but not implemented
in a language before.

**MetaForma Compiler** The MetaForma compiler and IDE is implemented in Spoofax
[41] as an Eclipse plugin, offering IDE support to the programmer not limited to syntax
highlighting, syntax checking, code folding, an outline view, reference resolving, and hover
help.

**MetaForma Runtime System** The MetaForma compiler compiles source files to Java
controllers that use the MetaForma runtime system, which is a static system providing ab-
stractions for sequential behavior, meta-modules, regions, parallelism, symmetry preservation
and communication.

1.7 **Thesis Overview**

The remainder of this thesis is structured as follows. In [Chapter 2] we discuss self-reconfigurable
robots and how we can program and simulate their controllers. In [Chapter 3] we describe
existing DSL approaches that are used to program self-reconfigurable robots.

In [Chapter 4] a solution-independent domain analysis is conducted in which we define
concepts to be integrated in MetaForma. Thereafter, in [Chapter 5] we argue for the need for
the new MetaForma language, and describe how the previously-defined concepts resulted in
the MetaForma syntax and semantics.

In [Chapter 6] we describe how the MetaForma compiler is implemented in Spoofax.
Thereafter, [Chapter 7] describes the implementation of the MetaForma runtime system.

In [Chapter 8] we present results of experiments that are conducted on a sample of
MetaForma programs. Finally, in [Chapter 9] we describe a list of related work, evaluate
MetaForma, answer the research questions and conclude the thesis. Finally, we and present recommendations for future work.
In this chapter we describe several ways of programming modular robots. First, we discuss modular robots more thoroughly and how we can simulate robot controllers using a robot simulator. Second, we discuss methods for distributing controller implementations to each module in an ensemble. Third, off-line and on-line control algorithms are discussed. Fourth, we describe control using a subsumption architecture.

2.1 Modular Robots

Various modular robots have been built the last decades. Regarding module design, they can be divided into two main categories: Heterogeneous and homogeneous modular robots. Additionally, besides heterogeneity and homogeneity, the overall design of the robot can be divided into three different structural types: Chain, lattice and hybrid structures. First we will discuss heterogeneous and homogeneous modular robots, followed by a discussion on the different structural types.

2.1.1 Heterogeneous and Homogeneous Modular Robots

Both heterogeneous and homogeneous modular robots per definition consist of multiple modules. However, in homogeneous robots the modules are functionally equal, while in heterogeneous robots they are not.

Heterogeneous Modular Robots

Heterogeneous modular robots are modular robots built from functionally different building blocks. In Figure 2.1 is the ODIN robot shown, a recent example of a heterogeneous modular robot. The configuration in the picture consists of 21 modules packed in a lattice:
The eight black links are telescoping link modules and the six white links are rigid passive link modules. They are connected to each other by seven joint modules. The links are connected to the joints using a flexible connector.

The ODIN robot is a reconfigurable robot in the sense that the modules can be configured into different shapes. However, an important limitation of ODIN is the lack of self-reconfiguration: The robot can only be put in a different configuration with the help of humans. Hence, when the modules of a robot can be connected in several different ways to form differently shaped robots, the robot is called reconfigurable. If the robot can do this shape-transformation task on its own while it is active, without human interaction, the robot is said to be self-reconfigurable [72].

Homogeneous Modular Robots

Homogeneous modular robots are built of functionally equal modules. Figure 1.3 shows the ATRON [58] robot, which is an example of a homogeneous modular robot. As being able to change its own shape while being active, it is also self-reconfigurable.

Using functionally equal modules has the major advantage in terms of robustness: If one of the modules fails, there is a higher chance that the robot can degrade gracefully instead of crashing completely. The robot might even apply self-repair by detaching the failing module and replace it by an other module. Regarding self-repair, homogeneous modular robots excel, as in heterogeneous modular robots a spare module might not be available.

Robustness is an important design goal in modular robotics because hardware failures can occur easily and we want the robot to work independent from human assistance, and even in environments where human assistance is impossible. Robustness can be measured by performing experiments with the robot and observing in what percentage of the experiments the robot fails [72].

It appears that when a homogeneous robot is used for real tasks, specific dedicated modules
are needed, e.g. a drill module to drill holes, or a gripper module to grab things. Although in contradiction with its module design, it is acceptable if a homogeneous robot contains dedicated special modules for special task, as long as the bulk part of the robot must consists of reusable, functional equally parts [72].

2.1.2 Chain, Lattice and Hybrid Structures

Chain-type Modular Robots

Chain-type modular robots are robots that are organized in a chain structure. An example is PolyBot [78], which is shown in Figure 1.2. Chain-type modules usually have one or two rotational degrees of freedom, perpendicular to the chain. The chains can be connected in many different ways, allowing a wide range of flexible and efficient locomotion patterns, such as walking, crawling and rolling, as shown in Figure 1.2.

Lattice-type Modular Robots

Lattice-type modular robots are organized in a lattice structure. This is similar to the way in which atoms are organised in a crystal. The lattice restricts the positions and orientations of modules, implying that there is a known fixed distance and orientation between neighbor modules in a lattice-type robot. Modules move from one lattice position to a neighbor position.

It is relatively easy to make a self-reconfiguration step in a lattice structure, since the lattice guides the connector alignment and thereby simplifies the control that is required. A self-reconfiguration step in a lattice-type modular robot is a sequence of disconnect, move and connect actions. For the ATRON robot, a move action means a rotation of hemispheres relatively to each other, while at PolyBot a move action means extending or retracting a telescoping link.

As a result of movement constraints, lattice-type systems can perform only a limited form of locomotion through self-reconfiguration. Locomotion can only be done by letting modules wander from the rear of the robot to its front. This is often referred to as cluster-flow locomotion [72] p47] and can be achieved by using meta-modules. As a result of the limitations of self-reconfiguration in lattice structures, locomotion patterns designed for chain-type modular robots - such as walking and rolling - are not within reach for lattice-type modular robots.

Hybrid-type Modular Robots

Hybrid modular robots can exist in both chain and lattice form, but not on the same time and place in a robot. For example, a robot can consist of a scaffold structure in lattice configuration while an arm connected to the structure is configured in chain form. Hybrid modules can be packed in a lattice structure to ease self-reconfiguration, but they can also configure itself in a chain configuration for more efficient - but also more complex - self-reconfiguration.
Most modern modular robots are hybrid. Lattice-type robots usually can be transformed to chain-type robots by releasing its actuator constraints, whereas chain-type robots usually can become a lattice-type robot by imposing extra actuator constraints such that it agrees to the lattice.

### 2.1.3 The ATRON Self-Reconfigurable Robot

The ATRON [58] is a homogeneous hybrid-type self-reconfigurable robot. An ATRON module consist of two hemispheres that can perform unlimited rotations with respect to each other. Each module has four male and four female connectors, used to connect to neighbor modules. A male connector can be extended and grab onto a passive female connector.

Eight infra-red ports, one below each connector, are used to communicate with neighboring modules and sense distance to nearby objects.

Figure 2.2 shows a single ATRON module: It has a mass of 850 grams and a diameter of 110 mm. The even and odd numbers denote male and female connectors, respectively. The two male connectors at the north hemisphere are extended, while on the south hemisphere they are retracted.

An ATRON module can sense its relative orientation to gravity by using its tilt sensors. In addition, each module has its own power supply in terms of a battery.

### Communication

ATRON modules use infra-red channels to communicate with neighboring modules, this can work up to 9.6 kilobit per second, assuming one neighbor at a time [38]. Infra-red communication is not reliable: Messages can get lost or even picked up by non-neighboring modules. In the latter case, in which infra-red light containing the message is reflected to other modules, is referred to as crosstalk [24]. This problem is common when using infra-red communication, as infra-red light is easily reflected on almost any surface - not limited to glossy ones. Christensen et al. (2007) [24] devised an algorithm to deal with this problem.

---

Figure 2.2: Eight ATRON connectors and infra-red communication channels (two at the back)

Figure 2.3: Communication example between two ATRON modules
in a simple, computationally cheap and completely distributed way. This algorithm can run on the physical modules between the robot controller and the device drivers that control the infra-red channels.

**Actuation**

Male connectors are active in the sense that they can be extended and grab onto a female connector of a neighboring module. In terms of hardware, female connectors are just a metal bar on which a hook-shaped male connector can grip, hence female connectors are completely passive.

One single ATRON module has only a single rotational degree of freedom, therefore its ability to move is very limited. Basically, a module cannot move itself, the help of a neighboring module is always needed to achieve movement. Additionally, modules must always remain connected either directly or indirectly to prevent modules from being disconnected from the rest of the structure. Collision must be avoided and actuator strength must be respected as one module is only capable of lifting two neighbors against gravity. Lifting more modules at once is possible, but then multiple modules must work together to achieve the lifting.

**First generation ATRON modules**

Two generations of ATRON modules have been built. The first generation of ATRON [38] was built in 2004. In the first generation, each module is controlled by two Atmel ATMEGA 128 processors, one per hemisphere, running at a clock frequency of 16 MHz. The processors are mutually connected with an RS-485 serial connection. Each CPU has 128 KB of Flash memory for storing programs and 4 KB of RAM to use as data memory during program execution.

**Second generation ATRON modules**

The second generation of ATRON [58] modules was developed in 2009 and is equipped with much more powerful hardware. In ATRON-2 modules, both of the hemispheres contain an FPGA\(^1\) that can support up to eight 32-bit Microblaze CPU’s running a clock frequency of 100 MHz, equipped with 8 MB of Flash memory and 64 MB of DDR-RAM [18, 53].

In addition to direct-neighbor communication using infra-red, the second generation of ATRON modules have a wireless ZigBee [42] interface, allowing to communicate with any module in the ensemble.

\(^1\) A field-programmable gate array (FPGA) is an integrated circuit that can be described by a hardware description language
2. PROGRAMMING MODULAR ROBOTS

2.2 The USSR: Unified Simulator for Self-reconfigurable Robots

Transferring controller implementation to ensembles of physical modules is a time-consuming task [12]. In order to test robot controllers easily, several robot simulators have been developed, that can visualize how the robot operates using the controller, hence enabling robot controllers to be tested continuously during development. The Webots [30] simulator supports modular robots, but is a commercial project and not open-source. Gazebo [43] is an open-source simulator, but does not support simulating modular robots. The USSR project [25] is an open-source simulator and supports modular robots and used in this project to simulate robot controller programs.

The USSR [25] supports controller programs for the ATRON [38,58], ODIN [52] and MTRAN [55] robot. The simulator is implemented in Java and is based on the Open Dynamics Engine (ODE) [69] that simulates realistic physics, and the JMonkey (JME) game engine for 3D graphics. The graphical interface shows the behavior of the robot while running its controller. Simulation takes place into a three-dimensional world in which the robot is placed, eventual together with obstacles like boxes. Visually, each type of robot module is built from a number of geometric shapes, like spheres, cubes and cylinders. The basic shapes are glued together by visually rendering them using relative coordinates to each other.

Figure 2.4 shows a simulation of a Two-Wheeler car of ATRON modules. In order to create a new simulation, two classes need to be created. First, the abstract GenericATRONSimulation class need to be extended to define a new concrete simulation. In the example this is done by the class CarSimulation. Second, the abstract class ATRONController needs to be extended with a concrete class that implements the abstract method activate with the robot controller program, usually containing an endless while loop to attain control until the thread is killed.

The GenericATRONSimulation class is responsible for creating a separated thread for each module in the robot, which executes an instance of the ATRONController class separately.
2.2.1 The ATRONController class

The ATRONController class is the parent class for creating a controller for a single ATRON module. It enables a means to control the ATRON hardware by implementing the ATRONAPI interface, which comprises methods for actuation, controlling connectors, and communicating with neighbors. [Listing 2.1] shows a fragment of the interface.

```java
1 - connect (int) - disconnect (int)
2 - sendMessage (byte[], byte, byte) - handleMessage (byte[], int, int)
3 - rotate (int) - rotateToDegree (int)
4 - rotateContinuous (float) - stop ()
5 - isConnected (byte)
```

Listing 2.1: Fragment of the ATRONAPI, used to control connectors and the central actuator, and to communicate with neighbors

Actuation

An ATRON controller can instruct a module to extend a specific male connector by invoking the `connect` method, with the connector number as a parameter. Retracting a connector works analogous using the `disconnect` method. The connection state of a connector can be checked with `isConnected`. Rotating the north and south hemisphere with respect to each other can be done in three ways. First, the `rotate` method rotates relative to the current degree. Second, `rotateToDegree` to rotates to an absolute degree, so independent of the current position of the actuator. Third, `rotateContinuous` rotates continuously at a specified level of velocity, such that it keeps rotating until `stop` is invoked.

Communication

In order to achieve a realistic simulation, robot controllers must respect the ATRONAPI to communicate with each other. Global variables, for any other use as collecting statistic or maybe for debugging, must be avoided as using them for communication will not be a realistic representation on how it would work on the physical modules.

Communication is implemented in the ATRONAPI by the `sendMessage` and `handleMessage` methods for sending and receiving, respectively. Sending a message is done by invoking the `sendMessage` method with (a) the message payload in an array of bytes, (b) the number of the infra-red connector to send the message along and (c) a number representing the length of the message<sup>2</sup>. In order to use infra-red, the connector numbers [0..7] are used, and for ZigBee communication the number 8 is used.

As ZigBee communication is a wireless broadcast, all modules in the ensemble will receive the message. For communication over infra-red channels, the ODE physics engine checks whether a neighbor module exist at the sending infra-red channel and is properly

<sup>2</sup>For the ANSI-C implementation of the ATRONAPI, we need to provide the length of the payload as it is an array and therefore only a pointer in ANSI-C. Although the length is not necessary in the Java implementation, it is still included to keep the interface of the ATRONAPI uniform for different platforms.
aligned to a sender. In either case, when a module receives a message, the controller automatically invokes the `handleMessage` method with as parameters (a) the payload, (b) the connector number the message was received on and (c) the length of the message. Figure 2.3 visualizes how this works.

On the physical modules, packet loss can occur due to alignment errors and infra-red communication malfunction. To attain a more realistic simulation, packet loss can be simulated also in the USSR. At default, each message that is send over a connector will always be received by the neighbor connected to that connector, if any. Hence, at default, there is no package-loss, but it can be enabled by the `setCommFailureRisk` method, which will be discussed in detail in Chapter 8.

The problem of crosstalk in communication is not simulated by the USSR, as this problem can be tackled at the physical modules, between the device drivers of the infra-red channels and the robot controller [24].

### 2.2.2 A Hello World Simulation Example

In Figure 2.5 is shown a simulation of a two-wheeler car. This example shows how that effort must be made in order to implement a trivial controller program in a GPL like Java. In this simulation, a world is created with a car robot in it that has to drive straightforward, where it will meet an obstacle. The car is built from three ATRON modules: Two wheels with an axle in between. Moving the car forward is attained by continuously rotating the wheel modules.

The distance to the obstacle is measured by using the proximity sensor on the north (blue) hemisphere of the central axle module. All proximity sensors are read and the maximum value is taken, and as it increases the value is printed out to the console.

![Figure 2.5: The USSR simulating a Two-wheeler car driving to an obstacle](image)

(a) 2 seconds: 0.0 0.0 0.0 0.10078055 0.10078055 0.20208347 0.20208347 0.30249572 0.4026808 0.5026808
(b) 7 seconds: 0.0 0.10078055 0.20208347 0.30249572 0.4026808 0.5029478
(c) 10 seconds: 0.0 0.10078055 0.20208347 0.30249572 0.4026808 0.5029478

Figure 2.5: The USSR simulating a Two-wheeler car driving to an obstacle
In Listing 2.2 is shown the implementation of the car controller. We remind that for all three modules a separated thread is created that runs a different instantiation of the same controller program. The activate function is invoked by the simulator when the modules are created; from this point the controller becomes active.

```java
public class ATRONCarController extends ATRONController {
    public void activate () {
        super.setup ();
        delay (1000);
        float lastProx = Float.NEGATIVE_INFINITY;
        String name = module.getProperty ("name");

        if(name.contains("Wheel1")) rotateContinuous (-1);
        if(name.contains("Wheel2")) rotateContinuous (1);

        while (true) {
            if (name.contains("axle")) { // Only run at axle module
                float max_prox = Float.NEGATIVE_INFINITY;
                for(Sensor s: module.getSensors()) {
                    if(s.getName().startsWith("Proximity")) {
                        max_prox = Math.max(max_prox, s.readValue ());
                    }
                }

                if(Math.abs(lastProx-max_prox)>0.1) {
                    System.out.println(max_prox);
                    lastProx = max_prox;
                }
            }
            yield (); // Always call yield sometimes
        }
    }
}
```

Listing 2.2: Controller Program in Java that shows Proximity Sensing for a Two-Wheeler

First the `setup` function is invoked which initializes the actuators and connectors. Then, the `delay` method is invoked to make sure everything is properly initialized before continuing. The proximity variable is initialized to $-\infty$ such that later on we can print out its value when increased.

Consecutively, the main while loop is entered, reading out the sensors of the `Axle` module and when a higher value is received it is printed out. The invocation of `yield` makes the current thread wait for the simulator to make a simulation step, so it ensures some degree of balanced scheduling between the controllers and the simulator. Because all threads have equal priority, the simulator thread might not get enough CPU time if 100 of modules (and hence 100 controller threads) are used in a simulation.

Although the simulator is written in Java, it can also simulate controllers implemented in ANSI-C by using the JNI (Java Native Interface). This interface allows the programmer to write a controller in ANSI-C to be used at a physical robot, while still being able to run the same controller in the USSR simulator.
2.2.3 The Assemble and Animate (ASE) Framework

To ease the process of developing uniform controllers, an implementation framework has been developed that can be used at both a physical robot and in the USSR simulator. This framework is called the Assemble and Animate (ASE) framework [27] and is a control framework that is written in ANSI-C and targets the ATRON, ODIN and M-TRAN robots. The objective of ASE is to provide a flexible and extendible control framework such that controller software can be reused more easily. Figure 2.6 shows the architecture of ASE.

![Figure 2.6: Architecture of ASE [27]](image)

**Application Layer** The Application layer contains the ANSI-C implementation of the robot controller. Instead of programming directly on device drivers, it only uses components from the ASE layer, making the application independent on the implementation of the platform on which it is running, which could be a physical robot or the USSR.

**ASE Layer** The ASE Layer contains an abstract API with robot-independent functionality to be used by controllers in the Application Layer. For example, a function to get the hardware id of a module or a function to send a message to a neighbor module make part of this abstract API, as all modular robots have this functionality, with a robot-specific implementation. To maintain portability between physical and simulated robots the interface of the Target API must be identical for both the simulated and physical target, and the Application Layer must not use the Kernel Layer directly, it should access it through the Target API.
Multiple robot-specific APIs are implemented with functionality that is dedicated to a specific type of robot. For example, the ATRONAPI implements this target API and has functions for rotating hemispheres. This function only applies to the ATRON robot since not all modular robots consist of hemispheres.

**Kernel Layer** The kernel layer can be a full OS, simple firmware or a physics-based simulator like the USSR. A robot controller in the Application layer can be simulated in the USSR by compiling the controller and linking it to the USSR API from the Kernel layer.

The same controller implementation can be loaded into a physical robot by after compilation linking it to a Kernel Layer targeting the physical robot.

**Hardware Layer** The hardware layer contains functionality to control actuators, sensors and communication devices.

### 2.3 Distribution of Controller Implementations

When simulating a robot controller in a simulator, we do not have to transfer the implementation of the robot controller manually to each module in the ensemble. As the simulation of the complete ensemble is executed at one computer, the controller implementation is directly available to every module. Indeed, when we run a simulation, all modules in the ensemble are automatically executing the appropriate controller program.

For the physical modules, this does not hold; as it is a distributed network of modules in which each module has its own hardware and CPU. Hence, once a controller is implemented, we need to transfer the implementations to each module in the ensemble. This can be done using the following ways.

**Manual controller distribution** In manual distribution, the programmer transfers the code manually to each module in the ensemble. As software development for modular robots is often an incremental process and the time needed to reboot all modules, this process incurs a significant burden to the developer [53]. Even the smallest modification in a controller program needs a reprogramming of the ensemble, which quickly becomes a tedious job for module structures consisting of more than three modules.

**Distribution by using a boot-loader** A boot-loader can boot a binary controller-program image in the module, after it has been propagated through the structure. Hence, this allows a single point of programming in the ensemble of modules. However, after reprogramming, the robot needs to be restarted in order to activate the new controller on the modules, impeding a running system [12].

**Automatic controller distribution** In automatic controller distribution, each robot in the ensemble is programmed with a simple operating system [49, 63] that is capable to execute behavior that is propagated through the ensemble. Behavior can vary from byte-code instructions [12] to compressed source-code [53], depending on the robots constraints in
terms of communication and processing power. Hence, while each module runs the same static controller, behavior is propagated through the ensemble of modules, enabling a single point of programming and even live-update of controllers while they are running.

2.4 Control Algorithm approaches

A number of ways exist to implement a control algorithm, which we discuss in this section. First, we compare off-line and on-line control algorithms. Second, we discuss centralized control and decentralized control.

2.4.1 Off-line and On-line control algorithms

Robot controller algorithms can be implemented using an off-line and on-line control algorithm. In an off-line algorithm, the execution path is already known and determined at compile time, while in an on-line algorithms it is not.

Off-line control When using an off-line control algorithm, the robot executes an algorithm which execution path is already computed at compile time. Hence, it has a limited means to adapt to the environment.

When something unexpectedly happens, such as an error, the robot has a limited means to handle it, as it has been programmed with a fixed, predetermined algorithm. An option could be to reverse its algorithm, resetting the robot back to its initial state, and then try to execute the same algorithm again.

On-line control In purely on-line distributed algorithms, the robot adapts itself to the environment, and the execution path of the algorithm is computed at runtime. Hence, robust communication is essential to determine the position of modules with respect to each other. A malfunctioning communication channel, even in only one way, is problematic for the controller. Therefore, on-line programming systems typically require communication to always works bidirectionally in the hardware it is running on.

The Proteo [81] system, for example, uses an on-line algorithm, by using a model that describes constraints on how modules can move along in the lattice structure. Based on this model, Proteo computes move transitions for modules in the structure. It requires two-way communication between all modules, because it uses temperature gradients in which the heat must be maintained when transferred to a neighbor.

Hence, in on-line algorithm, decisions are made at runtime and modules can make wrong decisions if they have wrong or obsolete information. Therefore, on-line control algorithms are flexible, but trade in robustness; they are fragile against unreliable communication.

In order to preserve heat in the Heat-based method, the heat sent to a neighbor must be subtracted from the sender, but only if correctly received by the neighbor. Hence, an acknowledgement must be send, requiring two-way communication.
2.4.2 Centralized Control versus Decentralized Control

In order to realize the desired global behavior in terms of individual, local, low-level programs, we need a means of coordination, either centralized or decentralized.

In centralized control, a single entity has all knowledge about what is going on in the system, and controls how the other modules should act based on this global knowledge. In decentralized control, often referred to as distributed control, there is a lack of an all-seeing central entity, and all modules are autonomous.

Both centralized and decentralized control have potentials, but an important drawback of centralized control is that the controller does not scale well in terms of the number of modules in the system [26]. Both on-line and off-line algorithms can be implemented distributed or centralized. Typically, distributed is harder for the human programmer, but it can be more robust and more efficient.

2.5 Control using a Subsumption Architecture

A control system for a completely autonomous mobile robot must perform many complex information processing tasks in real time. Brooks et al. (1986) [20] describes a subsumption architecture, which divides the robot control problem in terms of behaviors (horizontal layers) rather than in terms of functional modules. This provides a way to incrementally build and test a complex robot control system.

The layered behavioral division works as follows: The bottom layer describes the behavior at the most specific level. Layers are stacked on top of each other, which means that a layer can suppress the layer beneath it and override its output. Therefore, layers do not need to be modified once created; after creation they are debugged thoroughly and never modified. In case additional behavior is needed, a new layer is created that overrides the layer underneath it when needed. The system can be partitioned at any level, and the layers below form a complete operational control system.

Figure 2.8 shows (a) a subsumption architecture for a mobile control system, and (b) how higher-level behavior layers subsume the output of lower-level layers. For comparison, Figure 2.7 shows a traditional architecture for a mobile control system.

![Figure 2.7: A traditional decomposition of a mobile control system into functional modules](image)

In the architecture showed in Figure 2.8 a mobile car robot needs to explore the world around it without colliding to obstacles. If we solve this problem using a subsumption architecture, we first create a bottom layer (layer zero) that defines behavior for contact-
2. Programming Modular Robots

(a) Horizontal control layers

(b) Higher level layers subsuming the roles of lower level layers when they wish to take control [20]

Figure 2.8: A decomposition of a module robot control system based on task achieving behavior [20]

avoidance with objects (Figure 2.8 (a)). After implementing, we debug this layer thoroughly and do not modify it afterwards. It is able to examine data from the bottom layer and is also permitted to inject data into the internal interfaces of the bottom layer, suppressing the data flow. This is shown in Figure 2.8 (b). Next we implement a second control layer, that instructs to wander aimlessly around without hitting things. Consecutively, the third layer explores the world by seeing places in the distance which look reachable and heading for them. The fourth layer builds a map of the environment and plans routes from one place to another. [20]

2.6 Summary

Programming modular robots using a traditional GPL, like ANSI-C and Java, is in general a difficult task, for the following reasons. First, the hardware of modular robots on which the software runs, is unreliable. Realistic situations include changing battery levels, message communication problems, connector failures, and even modules resetting from time to time due to software or electrical failure [65].

Second, modular robots consist of multiple modules situated in a distributed architecture, resulting in a lack of global knowledge among the modules. Third, the modules make part of a dynamically evolving communication topology, meaning that the structure of modules changes during time and modules do not have fixed neighbors.

Robot controllers can be tested using a robot simulator such as the USSR simulator. Using the ASE framework, we can simulate controllers implemented in ANSI-C to be executed on the physical robots, hence releasing the need of a Java and ANSI-C implementation.

Although GPLs can be used to program self-reconfigurable robots, additional effort from the programmer is needed in order to attain properties such as robustness and scalability, as GPLs provide only general abstractions. We believe a DSL is needed to abstract overcome these problems.
Chapter 3

DSLs for Modular Robots

“Once we accept our limits, we go beyond them”
Albert Einstein (1879-1955), Physicist, Scientist, Inventor

In order to tackle problems typical to modular robots, researchers developed DSLs that can provide domain-specific abstractions for the programmer. In this chapter, we will first discuss the notion of a DSL and which properties make them interesting for modular robots. Thereafter, we will discuss a list of DSLs created for modular robots.

3.1 Introduction

First, we take a closer look to what a DSL is. Some of the definitions are (but not limited to) the following:

- Martin Fowler: “A computer programming language of limited expressiveness focused on a particular domain.” [32]
- Eelco Visser: “A domain-specific language (DSL) is a high-level software implementation language that supports concepts and abstractions that are related to a particular (application) domain.” [76]
- Arie van Deursen et al.: “A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.” [31]

Hence, a Domain-Specific Language (DSL) is, like the name implies, a language specially created for a small, specific domain. For this specific case, the domain is the self-reconfigurable robot, therefore, the following properties are interesting to integrate in a language:
3. DSLs for Modular Robots

- **Control at ensemble-level** This allows the programmer to develop controller software on the level of an entire robot ensemble \[8,12,56,61,65,83\], instead of individual module level.

- **Small-scale ensemble support** This enables expressing self-reconfiguration transformations \[65\], or locomotion such as driving a car \[12\] or crawling snake. Ensembles that exist in small-scale cannot afford to overlook reliability issues, as in a low-redundant context this is a more important matter\(^1\) compared to ensembles comprising hundreds or thousands of modules \[65\].

- **Large-scale ensemble support** In ensembles that can comprise hundreds or thousands of modules \[16,29\], scalability is an important property to attain. Moreover, in order to attain locomotion on large-scale ensembles, we need self-configuration by using meta-modules.

- **Proximity sensing** When a robot controller has a means to detect objects around it \[12\] or blocked actuators, obstacle-avoidance and search algorithms can be implemented.

- **Sequential and parallel actuation** Self-reconfiguration actions are usually dependent on other actions, i.e. action \(B\) is dependent on action \(A\) if action \(B\) can only be started when action \(A\) has finished. An action can be a actuated move, or the extension/retraction of a connector. Actions that depend on each other must be executed in sequential order. In contrast, to minimize execution time, independent actions can be executed in parallel, as actuation is usually a slow process.

- **Communication** In order to define meta-modules and implement gradients \[57\], modules need to communicate with neighbors.

- **Integration with a simulator** When a robot simulator, e.g. the USSR is integrated in a DSL, we can add debugging code to DSL programs. The debugging code can be compiled different for simulated and physical target, such as a coloring of modules in the simulator and a LED pattern at the physical modules.

3.2 UrbiScript

UrbiScript (formerly the Urbi programming language) was among the first languages specially developed for robotics. It is a parallel and event-driven scripting language \[4,9\] for robots, and has a distributed component architecture. From syntactical point of view, the language is inspired by ANSI-C, C++ and JavaScript \[4\]. Like most scripting languages it is dynamically typed. UrbiScript is an Object-Oriented Language (OOL), hence, values are treated as objects. Instead of being class-based, like Java, Python and Ruby, is UrbiScript prototype-based. Hence, it contains no classes, and instantiation of objects has been replaced by cloning an object.

\(^1\)In low-redundant context there is a limited possibility for self-repair
UrbiScript is specially dedicated to robots, hence it has built-in support for concurrency, time management and event-driven communication.

### 3.2.1 Concurrency Support

UrbiScript has been designed as a concurrent language. Functions in UrbiScript can be executed as jobs in separated threads, while a global scheduler takes care of giving each thread its fair share of time. Tags can be used to communicate with jobs running in the background. This is illustrated in Figure 3.1. A tag is created, and consecutively associated with two jobs that run in the background. By invoking `stop` on the tag object, the job execution stops.

UrbiScript also supports customized flow control: Statements can be executed sequentially by separating them with a semicolon, while statements separated by a comma are executed in parallel. This is illustrated in Figure 3.2.

```urbi
var t = Tag.new;
[00000010] Tag<tag_8>

  t : every (1s) echo("tick"),
    [00000019] +++ tick!
  sleep(0.5s);
  t : every (1s) echo("tack"),
    [00000020] +++ tack!
  sleep(2s);
  [00000021] +++ tick!
  [00000022] +++ tack!
  [00000023] +++ tick!
  [00000024] +++ tack!
  t.stop;
  sleep(2s);
  // Nothing else runs.
```

Figure 3.1: Job control in UrbiScript.

Figure 3.2: Flow control in UrbiScript.

### 3.2.2 Time Management

Built-in time management support allows the user to write time-related code easily in a concise manner. The standard UrbiScript library provides assignment modifiers to generate trajectories, which is useful when controlling motors. For example, if we want to move to position 10 in five seconds, we use the following code:

```urbi
motor.val = 10 time: 5s;
```

The assignment modifier `time` facilitates in this case a controlled actuation to position 10. If want to secure any possible load that is attached to the motor and use limited acceleration, we can use the following statement:

```urbi
motor.val = 0 acc: 0.1
```
This moves the motor back to position 0, and makes sure that its axle does not accelerate faster than 0.1 unit/s². Oscillation of a motor is also possible, this can be expressed conveniently by using [4]:

```cpp
motor.val = 1 sin:pi ampl:5
```

This will let the motor oscillate around 1 with a period of \( \pi \) and an amplitude of 5.

### 3.2.3 Event-Driven Communication

UrbiScript supports event-driven communication [4, 5]. The programmer can define events and install event-handlers that are executed when the corresponding event is emitted. Figure 3.3 shows how events are created and handled. It shows that events can contain payload which is simply a collection of one or more values that belong to the event. When an event is thrown, its payload is first matched against the conditions that are specified in the event handler. If case of a match, the event handler is invoked.

```cpp
// Create a new event called "e".
var e = new Event;
// Install an event handler for "e".
at (e(5, (var b))) { echo("Received e(5, " + b + ")"); };
// Emit event "e" twice with different values.
emit e(1, 3); // No match, nothing is printed.
emit e(5, 7);
[00000000] *** Received e(5, 7)
```

Figure 3.3: Events in UrbiScript using payload [4].

UrbiScript provides powerful abstractions that ease the development of robot control programs, e.g. for concurrency, time management and event-driven communication. However, the language focuses entirely on non-reconfigurable robots with a high degree of freedom, for example humanoid robots. As self-reconfigurable robots consist of multiple single modules, UrbiScript can not target them at whole robot-level, but only at single module level. This is an important limitation when applying the language to the domain of self-reconfigurable robots.

### 3.3 Phase Automata Robot Scripting Language (PARSL)

The Poly-kinetic Phase Automata Robot Scripting Language (PARSL) [83] from Zhang et al. separates specification from implementation by providing a language for the definition of behavioral gaits, i.e. strategies to move a robot forward. PARSL is based on XML and allows researchers to share locomotion gaits in the research community, like walking, crawling or even rolling. Figure 3.4 shows the simulation of a snake-gait on PolyBot.

The language targets chain-type modular robot systems with a high degree of freedom. The traditional way of achieving gait behavior is by using a gait table [84]. A gait table is a two-dimensional array, with one axle representing modules, and the other axle representing
time, the entries of the table are joint angles. The rows of the table are traversed through
time, each module moves to the commanded angle at the corresponding time. Upon reaching
the last row, execution returns to the top. As the size of the gait table grows linearly with
increasing the number of modules, it is not scalable.

PARSL tackles this problem by exploiting a main property of locomotion gaits: Gaits are
periodic, and the behavior of all modules is usually equal except the difference of a phase
shift. Hence, the programmer can express a state automation that realizes the locomotion
gait. This is explained using the following example, showing how a snake-gait behavior can
move a robot in a snake-structure forward. Figure 3.5 shows (a) the PARSL implementation,
(c) the phase automation that it implements, and (b) how the automation results in a gait.

```xml
<?xml version="1.0" ?>
<program>
  <robot>
    <structure>
      <group1 elements="0,2,4,6" />
      <group2 elements="1,3,5,7" />
    </structure>
  </robot>
  <main>
    <simpleWave />
  </main>
  <behaviors>
    <simpleWave>
      <wave group="group1" phase="0.25" period="1" />
    </simpleWave>
  </behaviors>
  <automata>
    <wave states="up, down" />
  </automata>
  <states>
    <up angles="30" />
    <down angles="-30" />
  </states>
</program>
```

(a) PARSL implementation [83]

(b) The top and bottom graph show the
    phase for group1 and group2, respectively [83]

(c) Two-state cyclic phase automaton that
    realizes the gait [83]
This program works as follows. Two groups (group1 and group2) are defined for the modules 0, 2, 4, 6 and 1, 3, 5, 7, respectively. Consecutively, a wave behavior is defined which is invoked in the main section. This wave behavior sets the phase offset to 0.25 for the first group of modules. Both groups have a phase period of 1, causing both groups to trigger the up and down state in an alternating manner. The up and down state are linked to different actuation angles, causing the snake to crawl.

Besides snake-gait behavior, rolling and walking can also be expressed using PARSIL. It allows the programmer to describe gait tables in a declarative way. A drawback is that PARSIL requires the modules to have a pre-programmed hardware id, in order to be able to identify them in the language. In addition, creating groups of modules to use in the gait table is hard-coded in the language, precluding scalability.

3.4 Locally Distributed Predicates (LDP)

Locally Distributed Predicates (LDP) is a distributed programming language that is used to specify lists of actions and corresponding predicates [61]. LDP is specifically developed to target ensembles composed of spherical micro-robots [65], known as programmable matter [35].

In LDP, the modules in a robot are continuously queried against the defined predicates. If a predicate matches on a specific module, the corresponding actions are executed. An LDP program consists of data declaration rules and a series of statements, in which each statement has a predicate clause and a collection of action clauses. The predicate clause acts as the condition for the actions, if the condition evaluates as true the actions are executed.

A predicate begins with a declaration for each module involved in the statement. These modules are searched for in the robot and must be directly connected in order for the predicate to match. The condition itself is composed of state variables (which are expressed as module.variableName), offset operators (like prev() and next()), and topology restrictions (by using the neighbor relation neighbors (moduleA, moduleB)).

The core of the LDP execution model is the PatternMatcher. This is a data structure that is propagated constantly along the entire ensemble, containing the expression tree that is stated in the predicate clause. Each module fills in its variables in the expression and propagates it further, until it either matches at some module or fails to match.

This is illustrated in the following example program, illustrating a snake-structure of modules that crawls forward like shown in Figure 3.6. The program is described part by part, and the complete program can be found at [61].

The first part of the program defines the variables. This is shown below. Each module has an id and a parent integer variable, identifying itself and its parent, respectively. The offset2 variable is used to assign each module a successive value such that a phase can be computed from it.

```plaintext
int id; // the id number of module. Read-only.
int parent = -1; // the id number of the previous module in the chain
```

2In the LDP paper, this variable is later on used using the name phase, we think this is an error [61]
float time; // the current time at the module. Updated by runtime
float offset = 0; // the module’s phase offset
float angle = 0; // the joint angle of the module’s central joint

Listing 3.1: Variable declarations for the LDP snake-crawl program [61]

The second part of the program contains four predicates. They are shown below.

First we need a predicate that causes module 1 to recognize itself as the leader. This is
done by the following LDP code [61]:

forall (a) where (a.id == 1) do a.parent = a.id;

Consecutively, we need to link successive neighboring modules, so it will form a chain
from head to tail while setting phase offsets. These offsets are successive integers in the
ensemble [61]:

forall (a,b) where (a.prev(1).parent != a.parent) & (a.id < b.id)
do b.parent = a.id, b.phase = a.phase + 0.1;

Third, we set joints to bend positively or negatively at the indicated phase offsets by the
following two predicates. The modules are triggered to actuate their angle by first adding
the time and the phase, consecutively taking the modulo value, and comparing this to a
constant [61]:

forall (a) where ((a.time + a.phase) % 1.0 == 0.5) & (a.parent != -1)
do a.angle = 15.0;
forall (a) where ((a.time + a.phase) % 1.0 == 0.0) & (a.parent != -1)
do a.angle = -15.0;

Figure 3.6: Snake Gait example using LDP: Black and white modules are actuating a negative
and positive joint angle, respectively [61]

In LDP, distributed state comparison and simple temporal relationships can be quite
naturally expressed. However, LDP supports sequential behavior only to a very limited
degree. Although being designed for running on modules with severe resource constraints,
they currently run in simulation only, and there are no specific considerations on how to
implement the language runtime for these languages [53].
3.5 Meld

Meld \cite{3} is a logic programming language inspired by P2 \cite{51}, that on its turn is a logic programming facility, based on the logic programming language Datalog \cite{21}. In Meld, programs are expressed using logic rules, hence certain properties of Meld programs can be proven for correctness \cite{61}.

The intention of P2 is to greatly simplify the process of selecting, implementing, deploying and evolving an overlay network design for a distributed system \cite{51}. However, unless self-reconfigurable robots are distributed systems, P2 provides only general abstractions for them, and typical abstractions for self-reconfigurable robots, such as actuation, are lacking. Moreover, self-reconfigurable robots are in general a highly-mobile node topology, causing facts to obsolete, while nodes in a wireless sensor network, that P2 addresses, in general do not move \cite{3}.

Meld has support for movement of modules, together with a means for deletion of obsolete facts. Indeed, as nodes are capable of moving, facts representing neighbor relations can become obsolete once a module has moved, and must be deleted.

Meld allows the programmer to specify global logic rules, at whole-robot level. Those rules are matched against facts available in each individual module in the robot. Two types of facts exist, inherent to logic programming: First, Base facts, e.g. local neighborhood information, about which modules are adjacent to the module in question, automatically generated by the runtime system. Second, produced facts, which are produced by rules that fired on already existing facts. If sufficient facts are produces by rules, modules can decide to move itself to another neighbor, hence attaining emergent behavior.

Analogous to LDP, Meld also targets ensembles of programmable matter \cite{65}, but with a main difference: Meld uses rule-based control in the form of logic rules that have no means of traversing neighbors like the PatternMatcher in LDP, and hence trades in expressiveness for provability \cite{61}.

3.6 Functional Programming Languages: Proto and Regiment

Proto \cite{7} and Regiment \cite{56} are functional logic programming languages. Proto targets large-scale ensembles of stationary and resource-constrained nodes that communicate wirelessly \cite{3}. A global behavior description written in Proto is compiled automatically into locally executed code that produces emergent phenomena matching the global description \cite{7}.

Regiment is based on the concept of functional reactive programming (FRP). The language allows the programmer to develop complex applications with in-network computation without explicitly dealing with low-level features of node programming \cite{56}.

Both Proto and Regiment are specialized for data aggregation, analysis \cite{3} and sharing knowledge among distributed systems and are less focussed on geometry and actuation \cite{61}. Hence, it is hard to express sequences of distributed actions in these languages.
3.7 DynaRole: A Virtual-Machine Architecture

DynaRole (Dynamically update-able Role-based programming language) [12] is a role-based DSL for the ATRON robot, layered on top of a virtual-machine based architecture. In DynaRole, ensembles are programmed at single-module level by defining behavioral roles. Each module in the ensemble can be assigned a role, such that the module adopts the behavior corresponding to this role. DynaRole compiles programs to low-level byte-code instructions. [Section 2.3] describes that reprogramming all modules in a robot to provide them with a new controller image quickly becomes a tedious job. Automatic controller distribution significantly diminishes this issue by requiring only a single module to be reprogrammed, instead of all modules in the ensemble. To attain automatic controller distribution, a simple OS or virtual-machine is needed that can propagate behavior while the robot is active.

As the resource-constrained ATRON modules have only 4 KB of RAM, a general-purpose virtual machine like the Java Virtual Machine cannot realistically be used for executing instructions inside the robot [12]. Hence, a new virtual-machine architecture has been designed, called the Distributed-Control Diffusion Virtual Machine (DCD-VM) [63].

3.7.1 The Distributed-Control Diffusion Virtual Machine (DCD-VM)

The Distributed-Control Diffusion Virtual Machine (DCD-VM) [63] consist of two main components: An interpreter for interpreting byte-code instructions and a network stack for propagating controller programs among the ensemble of modules.

The Interpreter The interpreter executes byte-code instructions produced by the DynaRole compiler, and is implemented using a simple switch-case interpreter. The byte-code instruction set is dedicated to the ATRON robot: It includes instructions for conditional branching, logical operators, comparison operators, actuator control, sensor value interpretation, program termination, and program migration to another module.

The instruction set of the byte-code language is by design very simple. Currently, it is very limited for realistic scenarios involving physical hardware, resulting in reduced scalability in terms of behavior. [63]

The Network Stack The network stack realizes the propagation of mobile byte-code programs by using distributed control diffusion. This concept enables controller code to be selectively (and hence efficiently) deployed to modules where a specific behavior is needed. Modules are identified by their role, which is a behavioral title for a module depending of its properties, including position, connectivity to other modules and current behavior.

Control diffusion works as follows: In order to spread the roles and their corresponding behavior, two message traversals are carried out. The first traversal determines for each module the role and which behavior is needed. The second traversal transfers the required behaviors in terms of byte-code programs to each module. When behavior for a specific role is needed, the DCD-VM implementation assumes that at least one module contains the required byte-code program, such that it is always available in the ensemble when needed.
3. DSLs for Modular Robots

Messages are addressed using roles and are only sent in the direction where that role is found; in a cyclic structure this is the same as broadcasting. In acyclic structures, it sends messages in the relevant direction, e.g., the central module of a snake ensemble knows which direct neighbor is in the path to the head-module or the tail-module of the snake.

Program transmission has to be done through infra-red communication, which is typically slow. The complete program to drive a two-wheeler car, as described in this section, takes up 156 bytes after compilation [53].

The DCD-VM realizes portability and automatic controller distribution, but implementing directly in the byte-code language is tedious and error-prone. The DynaRole language provides an extra layer of abstraction for the programmer, such that the programmer can implement behavior in a higher, more productive level.

3.7.2 The DynaRole Language

DynaRole uses roles as the main abstraction to express behaviors for modules in the ensemble. In addition, DynaRole supports primitives for simple decision making, but complex computations must be implemented in external code. This is usually implemented in a low-level language like ANSI-C.

Roles can be assigned to specific modules in the ensemble based on invariants, which we will see in the following example.

Example Program: Controlling a Two-wheeler Car

DynaRole can be used to implement simple drive-forward behavior in a Two-Wheeler car, as described below. Figure 3.7 shows a Two-Wheeler car of ATRON modules, together with their corresponding roles, i.e., LeftWheel, RightWheel and Head. To drive the car forward, both the wheels must rotate continuously, and in reverse direction as they are positioned in opposite to each other. Listing 3.2 defines an abstract role to be used for both wheels, as both wheels are functionally equal except the direction of rotation.

In this role, we declare three variables to be used in both wheels, which store the direction of driving, evading and the position on which the module is placed in the ensemble. Subsequently, in Listing 3.3 we create the sub-roles extending the abstract role Wheel and assign specific values to them. Note the opposite-value difference, to attain drive-behavior in forward direction.

The require clause is used to link a role to a specific module. This works like a pre-condition, if satisfied at a module, it will assign the role to that module.

```plaintext
abstract role Wheel extends Module {
    connected_direction; turn_direction; evasion_direction;
    require (self.center == $EAST_WEST);
    require (sizeof(self.connected(connected_direction)) == 1);
    behavior move(_) {
        self.$TURN_CONTINUOUSLY(turn_direction);
    }
}```
Both wheels implemented an `evade` command, facilitating the drive behavior. To invoke this command and activate the driving behavior, a `Head` role is created for the module situated between both wheels, shown in [Listing 3.4](#). We define a `startup` command in this role that will be invoked automatically when the role is assigned, thus triggering the drive behavior by invoking the `evade` command on both of the wheels.

In order to assign the appropriate role to each module, roles include a `require` clause. Thus, the `Head` role is assigned to each module that is facing in north-south direction. In case of the two-wheeler car, the central module is facing in that direction and thus will be assigned the `Head`. After becoming active, the `initialize` command of the role is invoked automatically, activating the drive behavior.

### Compilation

DynaRole programs are compiled into mobile program-fragments of DCD-VM byte-code that can be diffused along the module ensemble. This compilation works as follows: First, the contents of super-roles are copied to their sub-roles. Second, the role and its containing behavior is compiled to byte-code, which can be diffused among the modules in the ensemble. When compiled, behaviors are either packed in messages and stored in RAM, or stored in Flash memory.
3.7.3 DynaRole using a Python-based internal DSL

Moghadam et al. (2011) devised a Python-based internal DSL implementation [53] for DynaRole. Due to both the required implementation effort and the severe memory restrictions of the first generation of ATRON modules, the DynaRole language lacks basic constructs such as local variables, loops, and functions.

**ATRON support** The second generation of ATRON modules was equipped with powerful hardware, hence running a Linux kernel on the ATRON modules became possible. Running Linux on the modules provides features are immediately useful in the context of almost any robotic application [53]. Moreover, it enables the Python interpreter to be run on the modules and Python can be used as the host language for an internal DSL.

**Syntax** The proposed syntax for the DSL is based on DynaRole, enriching the original implementation with local variables, loops and functions. Listing 3.5 implements the definition for the Head role similar to the original DynaRole program (Listing 3.4). This shows that from a language design point of view, the python-based DSL includes a significant amount of syntactic noise compared to the original DynaRole syntax.

```python
class Head(Module):
    @require
    def dir(self):
        return self.center == NORTH_SOUTH
    @handle([PROXIMITY_1, PROXIMITY_3])
    def proximity(self):
        Wheel.evade()
        self.sleepcs(25)
```

Listing 3.5: Python syntax for the Head role definition [53]

In order to diffuse the Python program through all modules in the structure, it is first compressed in gzip format and then transmitted. The complete Python variant of the original DynaRole program shown in Section 3.7 takes up 350 bytes in compressed format. Although inducing a significant overhead compared to transmitting DCD-VM bytecode (156 bytes), transmitting compressed source-code is still feasible.

Both the host language and its operating system provide benefits for the embedded DSL. Some of the features required for a embedded DSL for modular robot programming can be provided by a language such as Python, but a significant number of features are also provided by the underlying operating system. The more flexible and powerful the language and operating system, the easier it becomes to implement the DSL abstractions required for the specific domain [53].

3.7.4 Analysis

As a result of roles as main abstraction without an abstraction to express sequential behavior, it is difficult to implement sequences of actions in an efficient manner. Sequential behavior can be implemented in DynaRole using roles and event handlers, meaning it will be done in
a purely asynchronous way. This approach however has multiple drawbacks with respect to
development, resulting in programs that are difficult to maintain and having a hard to follow
control flow, even for trivial algorithms [10].

To allow expression of distributed sequential behavior in a more straightforward manner,
an extension has been created on DynaRole: The Robust and Reversible Sequences [65].

3.8 Robust and Reversible Sequences in DynaRole

A distributed scripting language has been developed as an extension of DynaRole, to en-
able the execution of self-reconfiguration sequences [65] on unreliable hardware. A self-
reconfiguration sequence is a list of actions that is distributively executed among all modules
in a robot, transforming it from an initial configuration, e.g. an ‘Eight’ to a target configura-
tion, e.g. a Car. Figure 3.8 visualizes an ensemble of ATRON modules that is executing the
eight2car sequence.

![Figure 3.8: The Eight-to-Car self-reconfiguration sequence](image)

The language extension provides three main improvements on DynaRole: First, se-
quenences are compiled to a robust and efficient implementation based on a distributes state
machine. Second, dependencies between operations are explicitly stated to allow independent
operations to be performed in parallel, while dependent operations are executed sequentially.
Third, the language is reversible. This means that for any self-reconfiguration sequence
expressed in the language, the corresponding reverse sequence is automatically generated.
Reversible self-reconfiguration is useful in many cases as the reverse sequence is often just
as usable as the original one.

3.8.1 Syntax and Semantics of Sequences

A DynaRole sequence is basically a list of actions. Each action is prefixed with a role, identi-
fying the module on which the action needs to be executed. Consecutively, each statement is
3. DSLs for Modular Robots

...terminated either with a semicolon, meaning sequential execution, or an ampersand, meaning parallel execution. Table 3.1 shows the actions supported in the DynaRole sequence API.

M.Connector[n].retract()
Retracts connector number n (releasing a connected module, if any).

M.Connector[n].extend()
Extends connector number n (connecting to an appropriately positioned module, if any).

M.Joint.rotateFromToBy(f,t,d,s)
Rotates the main joint from f degrees to t degrees in direction t at speed s.

| sequence eight2car {
| M0.Connector[0].retract() & M3.Connector[4].retract();
| M3.rotateFromToBy(0,324,false,150);
| M4.rotateFromToBy(0,108,true,150);
| M4.Connector[0].extend() & M1.rotateFromToBy(0,324,false,150) &
| M6.Connector[2].retract();
| M4.rotateFromToBy(108,216,true,150) & M6.rotateFromToBy(0,108,true,150);
| M0.Connector[0].extend();
| M6.Connector[6].retract();
| M0.rotateFromToBy(0,324,false,150) & M1.rotateFromToBy(324,0,true,150);
| M0.rotateFromToBy(324,0,true,150);
| M5.Connector[0].extend() & M2.Connector[4].extend() &
| M1.Connector[4].extend();
| M4.Connector[0].retract();
| M3.Connector[6].retract();
| M1.rotateFromToBy(0,108,true,150) & M3.rotateFromToBy(324,0,true,150);
| M1.Connector[6].extend();
| M1.rotateFromToBy(108,216,true,150);
| }

Listing 3.6: DynaRole implementation of the ‘eight’ to car self-reconfiguration sequence [65]

3.8.2 Compilation Backend

The DynaRole compiler has a backend supporting compilation to a nesC [34] implementation, which can be executed in a TinyOS [49] environment. The compiler also supports backends for an ANSI-C or Java implementation, to be executed in the USSR simulator. Figure 3.9(a) shows a fragment of the generated nesC code, compiled from the eight2car sequence. The generated code implements a distributed state machine allowing the sequence to be executed reliably under real-life imperfect world conditions, such as changing battery levels, communication problems and even module resets by a watchdog due to software or hardware failure [65]. Figure 3.9(b) shows the corresponding state diagram.

TinyOS is a small operating system targeting severe constrains embedded systems.
The variable \textit{state} contains the local active state. Module \textit{M0} starts at state 1. This module first adds its own state to the pending state, such that other modules know it has not finished yet. Second, \textit{M0} starts retracting connector 0 and immediately advances to state 2. In this state, \textit{M0} instructs \textit{M3} to advance to state 4 and hence allowing a parallel action. \textit{M3} thereby immediately start with retracting connector 4, while \textit{M0} is also still retracting connector 0. When \textit{M0} is finished, it removes itself from the pending state. Finally, when \textit{M3} is finished with retracting, it advances to state 5. In this state the sequence waits for the pending state to be empty, i.e. a finished retract operation at \textit{M0} before continuing. Thereafter, \textit{M3} start rotating its hemispheres and the sequence continues.

```
switch(state) {
    case 1: /* Module M0 */
        call distState.addPendingState(1);
        call Connector.retract[CONNECTOR_0]();
        state = 2; break;
    case 2:
        call distState.sendState(4,3);
        state = 3; /* fall-through */
    case 3:
        if(call Connector.get[CONNECTOR_0]() != MALE_CONNECTOR_RETRACTED) break;
        call distState.removePendingState();
        state = 255; /* inactive state */
        break;
    case 4: /* Module M3 */
        call Connector.retract[CONNECTOR_4]();
        state = 5; break;
}
```

(a) Fragment of compiled nesC code [65] (b) State diagram of program flow [65]

Figure 3.9: Distributed state machine, derived from the eight-to-car sequence

Local state \( G_0 = (s_0, a_0, P_0) \) \hspace{1cm} Resulting state \( G_2 \)

Incoming state \( G_1 = (s_1, a_1, P_1) \) \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm}

\[
G_2 = \begin{cases} 
(s_0, a_0, \{ p \in P_0 \mid p > s_1 \vee p \in P_1 \}, \quad s_0 > s_1 \\
(s_0, a_0, P_0 \cap P_1), \quad s_0 = s_1 \\
(s_1, a_1, \{ p \in P_1 \mid p > s_0 \vee p \in P_0 \}, \quad s_0 < s_1
\end{cases}
\]

Figure 3.10: The global state merge function on the currently active state \((s_i)\), the address of the active module\((a_i)\), and the set of pending states \((P_i)\). [65]

Each module in the structure has its own local state, and shares this continuously among all its neighbors. Figure 3.10 shows the merge function that each module uses to merge a received state with its own local state.
3. DSLs for Modular Robots

In DynaRole’s state-sharing diffusion algorithm, synchronisation of parallel operations can only be done globally by sharing among the complete ensemble which modules are finished. For large ensembles this requirement can become an overhead [65].

The distributed state machine approach used by DynaRole an off-line control algorithm, meaning that the execution path is predetermined, reducing the ability to handle unexpected changes such as errors (Section 2.4), but gaining robustness. In addition, as the behavior of a sequence is known at compile time, the compiler can automatically generate the reversed sequence, which in theory is just as useful as the original one.

3.8.3 Reversibility of Sequences

All necessary knowledge for reversing a sequence is already known at compile time, hence the inverse of a sequence can be computed quite easily. This is performed in two steps.

First, the sequence is rewritten from the form of \(<S_1, S_2, ..., S_n>\) to \(<S_n, S_{n-1}, ..., S_1>\). An S represent a statement that may contain one single action or multiple parallel actions. Second, connect and disconnect actions are swapped, and rotateFromToBy(f, t, d, s) is simply replaced to rotateFromToBy(t, f, ¬d, s). In the latter replacement, the ‘from’ and ‘to’ angles are swapped and the direction is reversed. For example, the sequence \(<S_1 & S_2; S_3>\) will be reversed to \(<S_3; S_2 & S_1>\).

The approach is highly inspired by Janus [82], which is a reversible general-purpose programming language by Yokoyama et al. It allows algorithms to be implemented and automatically reversed. In Janus, functions can be called for normal execution, just like in other programming languages. In addition, a function be uncalled in order to execute it in reverse. For example, Fibonacci series can be computed back to the start instead of only from the start, using this approach.

3.9 Summary

First, we argued that a DSL to program modular robots should allow the programmer to target the robot as a whole, instead of targeting multiple, single-modules. In addition, the language should both support small-scale and large-scale ensembles, as both have their own important properties. Moreover, we need support for sequential and parallel actuation, proximity sensing, and a means to debug controllers either in a simulator or on a physical robot.

Second, we discussed a number of robot DSLs. UrbiScript has interesting features in terms of concurrency, but is not dedicated to expressing behavior at a whole-robot level that we would like to attain on modular robots. PARSL [83] focusses on locomotion gaits at whole-robot level for chain-type modular robots, but this language does not support self-reconfiguration.

The logic programming language Meld [3] and the declarative language LDP [61] target large-scale ensembles of programmable matter in a highly redundant context, but both languages lack support for reliable self-reconfiguration of small-scale ensembles.

Functional languages such as Regiment [56] and Proto [7] target large-scale ensembles of stationary modules and are less focussed on geometry and actuation, hence it is difficult to program a sequential list of actions in these languages.
DynaRole was created to support both behavioral patterns such as driving a car, and self-reconfiguration to transform between different robot configurations. The design of DynaRoles external DSL [12] and Python-based internal DSL [53] allows the programmer to work at a modular level in terms of roles and at a whole-robot level in terms of sequences [65]. However, these two aspects of the language are poorly integrated and were in each case designed to solve problems specific to certain scenarios.

Hence, we believe that a new DSL is needed for the ATRON robot that incorporates the scenarios that were developed on a case-by case basis.
Chapter 4

Domain Analysis

“Logic will get you from A to B. Imagination will take you everywhere.”
Albert Einstein (1879-1955), Physicist, Scientist, Inventor

In this chapter we study the domain of controlling modular robots, and define concepts as the fundamental requirements for a new language. For each type of modular robot, we need specific low-level control that targets its specific hardware. Therefore, the specific domain of the ATRON robot is chosen, in which we analyse the main control scenarios of what has been done so far. At the end, we generalize the domain other types of modular robots.

4.1 Modular Robots

4.1.1 Reconfigurable vs. Self-reconfigurable

A distributed topology is inherent to reconfigurable robots, as they are composed of multiple, connectable modules that can be placed in different configurations. A fundamental distinction in types of modular robots can be made between the ability for a robot to change its own configuration, i.e. a distinction between reconfigurable of self-reconfigurable robots. [72]

The Odin robot [52] (Figure 2.1) is modular and reconfigurable, i.e. its shape can be changed, but the robot cannot do this by itself. The ATRON robot [38] however, is capable of changing its own shape, meaning that it is a self-reconfigurable robot.

In self-reconfigurable robots, we want to exploit the possibility to do automatic morphology changes in the robot, but without causing unreliability. In order to realize reliability, robustness is a property that must be attained in the software in order to deal with hardware failures. In large-scale ensembles of homogeneous modules, the inherent redundancy of modules potentially results in the ability to deal with partial hardware failures.
4. Domain Analysis

4.1.2 Existing scenarios on the ATRON robot

Three known scenarios are analysed that capture fairly what has been done with the ATRON robot so far. First, shape-transformations of small-scale robots using self-reconfiguration, e.g. transformations between Car, Eight and Snake structures [65]. Second, behavior for specific small-scale ensembles, e.g. obstacle-avoidance for a Car [12, 66] robot or crawl behavior for a Snake. Third, locomotion of large-scale robot ensembles.

In order to realize locomotion on large-scale ATRON structures, we cannot instruct the ensemble to drive, walk or crawl itself forward in one piece, as a result of the single degree of freedom in each module, and its limited actuator strength. Hence, if we want to move the structure forward, we have to move it piece by piece, which can be done in several ways.

Rule-based control (Section 1.2.1) can be used, although this approach has several shortcomings: First, global behavior is stored in many non-transparent rules, which do not reflect the global behavior, and are difficult to modify afterwards. Second, many rules have to be created, either by using programming by demonstration [59] or manual programming [19]. Third, modules can get deadlocked in the self-reconfiguration process [15].

A major complication in the process of self-reconfiguration is the motion constraint induced by the single degree of freedom of a single module (Subsection 2.1.3). As a result, simple transformations of shape must be performed by using complex sequences of moves, if the transformation is even possible at all. Motion constraints can be reduced by using so-called meta-modules.

Therefore, as discussed in Chapter 1, meta-modules are an important helping tool in order to move large structures of modules. Fundamentally, two main meta-module approaches are used for the ATRON robot, which are devised by Brandt et al. (2007) [16] and by Christensen et al. (2006) [29].

4.2 Shape Transformations of Small-scale ensembles

Shape transformations are used to change the shape of a robot, from its source configuration into a new target configuration. Figure 3.8 shows a transformation of an ensemble of ATRON modules from an ‘Eight’ structure into a Car. In order to write a program for any kind of transformation, we need a means to identify modules in the ensemble.

4.2.1 Identification of Modules

Once each module in an ensemble is identified, we can assign behavior to each module by referring to its id. Identification can be done using a pre-programmed id at compile time, or by using a dynamically assigned id at runtime.

- Identification at compile time assumes each module in the ensemble to have a pre-programmed id, before the controller becomes active [65, 103]. When applied to a physical robot, this implies the need to transfer the corresponding controller image individually per module.
Identification at runtime does not assume a pre-programmed id. Modules are identified at runtime using their relative position in the ensemble [12].

In a non-robotic passenger car used by humans, all four wheels are homogeneous and connected in the same manner to their axle, regardless whether a wheel is placed at the left or right side, or at the front or back axle. The wheels can even be interchanged without changing the behavior of the car. By avoiding the use of a pre-programmed module id in controllers, the homogeneity of ATRON modules is respected also in terms of software, hence allowing equal controller images for each module in the ensemble. This leads to the first concept:

**Concept 1:** Module identification

“A means to identify modules in the robot at runtime, without using any kind of hardware-id or pre-programmed value, such that in a program modules can be referred to using an identifier”

Modules are connected to neighbors using specific connectors, depending on the physical placement of the module in the ensemble. In order to exploit this property, i.e. to assign the appropriate id to each module depending on its neighbors, connector sets are defined. Figure 4.1 shows the six connector sets, and how they are formed relative to its compass, at a Four-Wheeler car and a (part of an) ‘Eight’ ensemble. This leads to a new concept:

**Concept 2:** Connector sets

“Six different groupings of connectors, that each group connectors that are situated at the same surface area, defined as North, South, East, West, Up and Down”

![Figure 4.1: Connector sets for the ATRON robot](image)

4.2.2 Sequential execution of actions

Sequences should support both sequential and parallel actions, as executing all actions necessary in a transformation sequentially would cost an unnecessary high amount of time.
visualizes the eight2snake sequence, which transform an ‘eight’ ensemble into a snake. First, both disconnect statements are executed in parallel in one action. When finished, both rotate statements are executed in parallel, as they are non-conflicting.

We use a sequence to facilitate the execution of a list of distributed actions, each consisting of either a single statement or multiple statements to be executed in parallel. Before the next action is started, the statements in the current action have to be completed.

This leads to the next concept:

**Concept 3: Sequence**

“A list of distributed actions that is executed in sequential order among the ensemble, in which a single action can contain multiple statements to be executed in parallel”

### 4.3 Small-scale Locomotion: Obstacle Avoidance

Obstacle avoidance is the task of moving a robot forward and to avoid it from colliding with obstacles, by changing the direction of driving. Figure 4.3 shows a simple obstacle-avoidance algorithm for the ATRON robot.

As is not known at compile time when an obstacle will be encountered, we need a means to adapt the robots behavior to the environment. This is in contrast to fixed shape-transformations like the eight2car sequence, in which the execution is determined at compile time. Hence, in order to be able to implement obstacle avoidance, we require a means such that the robot can adapt its behavior to the environment. This leads to:

**Concept 4: Event handling**

“A means to implement actions to be executed when a specific condition is satisfied”
When the proximity sensors of the Car sense an obstacle, it should stop driving and avoid a collision, either by driving back \[66\], or turning itself and continue driving in another direction. The \textbf{sequence} concept can be used in this scenario, by creating a sequence that implements drive behavior and a sequence that implements the avoidance behavior. However, we need an additional means that facilitates transitions between sequences. The transition to \textit{Turn} must only be made from the \textit{Drive} sequence, and not from the \textit{Turn} sequence itself, as this would trigger an endless loop. Hence, it must be possible to relate sequence transitions to the current sequence.

\textbf{Concept 5: Sequence transition}

“A sequence transition is a discontinuation of the current sequence and the start of a new one.”

```plaintext
sequence Drive {
    when (AxleFront.prox > PROX_TRIGGER) {
        execute Turn
    }
    do {
        rotating (LeftWheel,CW)
        rotating (RightWheel,CCW)
    }
}
```

Listing 4.1: Sequence implementation for the Drive state

Figure 4.4 shows the state diagram for basic obstacle avoidance, and Listing 4.1 implements the Drive sequence that can be used on both the Two-Wheeler and Four-Wheeler. Inherent to the different hardware structure of both cars, specialized low-level control for turning each car is needed. Hence, the Turn sequence is implemented differently for both cars. Listing 4.2 and
4. Domain Analysis

Listing 4.2: Turn implementation for Two-Wheeler

```java
sequence Turn { // 2-wheeler
    do {
        rotating (LeftWheel, CW)
        rotating (RightWheel, CCW)
    } until (backwardsTime)
    do {
        rotating (LeftWheel, CW)
        rotating (RightWheel, CW)
    } until (turnTime)
    execute Drive;
} // CW and CCW refer to Clockwise and Counter-clockwise, respectively.
```

Listing 4.3: Turn implementation for Four-Wheeler

```java
sequence Turn { // 4-wheeler
    do {
        rotate (AxisFront steerDegrees)
        rotate (AxisBack -steerDegrees)
    }
    do {
        rotate (LeftWheels, CW)
        rotate (RightWheels, CCW)
    } until (backwardsTime)
    do {
        rotate (AxisFront, 0)
        rotate (AxisBack, 0)
    }
    execute Drive;
}
```

Listing 4.3 implement the Turn sequences for the Two-Wheeler and Four-Wheeler, respectively.

In the sequence we use statements as `connect`, `disconnect` and `rotate`, `rotating` in order to control the modules connectors and actuator, respectively. These functions need to be integrated in the DSL in order to pass control to the ATRONAPI (Section 2.2).

**Concept 6:** Hardware control primitives

“Primitive language constructs to control hardware such as actuators and connectors”

4.4 Cluster-flow devised by Christensen et al.

The meta-module approach of Christensen et al. (2006) [29] can be used for building structures such as towers [28], but also for moving ‘flat’ ensembles of ATRON modules. In this analysis, we analyze cluster-flow of a flat ensemble, and generalize later to the tower-building scenario.

4.4.1 Meta-modules

A meta-module is a small group of modules that acts as a single unit. As a result of containing multiple individual modules, a meta-module has a higher degree of freedom compared to a single module. This extra degree of freedom results in reduced motion constraints and a reduced complexity of motion planning, hence simplifying self-reconfiguration.

However, meta-modules can also increase the granularity of the system. By treating meta-modules as a single unit, moving them in one piece to their destination can be difficult or impossible. This problem is somewhat related to the problem of moving large furniture inside a small house.
Hence, a balance needs to be found between low granularity and a high degree of freedom. Christensen et al. (2006) has shown that ATRON meta-modules generally achieve the best performance when consisting of three modules\(^1\)\(^2\). Typically, the size of meta-modules does not exceed four modules, as a result of actuation constraints imposed by gravity. This leads to the concept:

**Concept 7:** Meta-module

“A meta-module is a group of individual modules that work together, to simplify the problem of self-reconfiguration.”

### 4.4.2 Cluster-flow: Clusters of Three Modules

In cluster-flow, sometimes referred to as scaffolding, we entitle a meta-module to walk over a scaffold of other modules\(^3\). It starts at the back, then walks to the front of the ensemble, and after arriving re-merges in the ensemble.

This is executed in three steps. Figure 4.5 shows the first step, in which we elect a meta-module at the back of the ensemble and lift it up. Figure 4.6 shows the second (iterative) step: The meta-module has to continue walking until reaching the front of the structure. Figure 4.7 shows the third step, in which the meta-module has to be lifted down to re-merge in the ensemble.

```
1 do { disconnect (Walker.Left, Lifter) }
2 do { retract connector Lifter diagonal to Walker.Left } // To enable rotation
3 do { rotate (Walker.Right, 90) }                       // Lifts up 2 modules
4 do { rotate (Lifter, 90) }                            // Lifts up 1 module
```

![Figure 4.5: Lifting the walker up](image)

In cluster-flow, a distinction in functionality can be made between walking modules and modules in the scaffold, which logically results in two module groups: Walker and Scaffold. Integrating this into the DSL allows the programmer to express behavior using referring to groups. Programmers can then instruct modules in the ensemble by using its id or group, such

\(^1\)Not all scenario’s have been tested in these experiments, although meta-modules consisting of three modules are shown to be a good starting point.
4. Domain Analysis

while Walker.Right is not at the end:
  do { connect (Walker.Right, Scaffold)
  do { disconnect (Walker.Left, Scaffold)
  do { rotate (Walker.Top, -45)
  do { rotate (Walker.Top, 180)
  do { rotate (Walker.Top, 45)
  do { swap identifiers of Walker.Left and Walker.Right

Figure 4.6: A single walking step, which must be repeated until the end of the ensemble is reached

that each module checks whether its id or group matches at runtime. This leads to the concept:

**Concept 8: Module group**

“A dynamically assigned group name to a collection of single modules that represent
the same function in the ensemble”

Actions in a sequence are assigned to modules by referring to their module identification[1] or module groups[8]. In order to attain a rotate action, modules can simply check individually
whether it matches to the id or group, and if the case perform the rotate action. However, for
connect and disconnect actions, modules must have knowledge about which neighbor is situated
at which connector number. As self-reconfigurable robots consist of modules in a dynamically
evolving communication topology, each modules neighborhood is not fixed and changes during
time. In order for modules to gain knowledge about their neighborhood, modules can propagate
their own id and group to neighbors through infra-red messages such that neighbors can store
this in combination with the connector on which the message was received.

Using this mechanism, each module can translate a neighbor id or group to a connector
number, and check whether this is a male connector. If this is the case, it extends or retracts the
connector, depending on to the neighbor. In case of a female connector, it waits for the neighbor
module to connect.

This leads to the definition of the next concept:

**Concept 9: Module neighbor table**

“A translation table that stores information from neighboring modules, like its id and
group, together with the connector number that the neighbor is connected to”

This scaffolding scenario assumes certain constraints on the size of the scaffold. First, the
length must be odd, as we need a Lifter module at both sides of the structure. Second, the length cannot be less than seven, as a result of gravity constraints: At a length of five, the Lifter module is the center of actuation hence it becomes an equilibrium point. Hence, in this case the Lifter module cannot make the difference between lifting up the meta-module or the rest of the structure. Therefore, a length of seven modules is the minimum length. Second, in order to apply meta-walkers consisting of three modules, the width of the ensemble must also be three.

### 4.4.3 Symmetry preservation

After executing a transformation, modules are moved relatively to each other and the symmetry of the ensemble is no longer maintained. Figure 4.8 shows (a) the initial ensemble and (b) the ensemble after one meta walk. as the three modules that have walked to the front have symmetry disruptions. These modules are flipped, meaning that e.g. its north-south hemispheres are swapped.

**Module flips** When assuming a two-dimensional grid and that the grid structure is maintained, i.e. the modules actuator is either 0 degrees or 180 degrees rotated, an ATRON module can be flipped in eight different ways.
In the cluster-walk scenario, the following combinations of symmetry disruptions occur. First, the north and south hemisphere of both the Walker.Left and Walker.Right are flipped. Second, the east and west sides are flipped of the Walker.Middle module, when the walker has made an odd amount of steps. Hence, in this scenario it depends on the amount of walking steps, and thus the length of the ensemble, whether the Walker.Middle is out of symmetry. Indeed, when there is an even amount of walking steps, the flips neutralize each other in this case. But in a scenario with one more walking step, the module Walker.Middle will be out of symmetry.

As connector sets of each module rely on the relative position to neighbors in the structure, this becomes problematic. Symmetry disruptions occur after the execution of almost any sequence.

Solving symmetry issues by performing self-reconfiguration is in theory a solution, as it can reorganise the module structure in such a way that each modules connector number matches the uniform grid structure. However, this only possible to a limited degree; i.e. when a module is flipped in east-west, because in that case rotating one or both of the hemispheres can correct the disruption.

A north-south flip cannot trivially be repaired by the structure itself without external interaction, as they are both situated on a different hemisphere. Additionally, even when mechanically repairing is possible, it is not a dexterous solution as it needs time and battery power to self-reconfigure, and the programming effort by the user to accomplish.

Although it is possible to let the programmer include information in the transformation program to preserve symmetry, for this specific case, it will include a means to keep track of the number of walked steps. Figuring out the information needed for each sequence becomes quickly a tedious job. Hence, it is more desirable to integrate an automatic symmetry preservation mechanism into the language. This leads to a new concept:

**Concept 10: Symmetry preservation**

“A mechanism that abstracts over symmetry issues that arise after executing a self-reconfiguration sequence”

### 4.4.4 Cluster-flow: Clusters of Four Modules

The meta-module approach of Christensen et al. was applied on Scaffold structure having a width of three modules. If the Scaffold structure has a bigger width, the problem of cluster-flow becomes more difficult, as it implies at least some of the walkers to contain dummy modules, needed to move all modules at the back of the structure, over its complete width. This introduces two new problems.

First, Figure 4.9 shows that the location of a neighboring meta-module can no longer be determined based on its connector only. The figure shows two identical pairs of connectors, from which one connects to a meta-module at the top and the other at the top-left.

In order to identify neighbors on the meta-module level, we need an additional means to distinguish between different meta-neighbors. Thus, neighboring information on connector
numbers alone is not sufficient, we need an additional label to assign to modules in a meta-module indicating their relative position in the meta-module.

The meta-part facilitates the distinction between single modules that are in an equal way connected to different meta-neighbors and need to determine meta-neighbor information. In addition, the meta part can help forming the grid of meta-modules in the structure.

This leads to a new concept:

**Concept 11:** Meta part

“A label that is assigned to each single module inside a meta-module, representing its relative position."

Second, Figure 4.10 shows the need for parallel actuation, as four modules cannot be lifted by one single neighbor, and meta-modules need to assist each other in lifting neighbors. Moreover, the ‘lonely’ modules at the front of the structure are needed for lifting the walkers down, but cannot be integrated in an equally-sized grid.

### 4.5 Cluster-flow devised by Brandt et al.

Brandt et al. \[16\] describes a meta-module approach for the ATRON robot. It enables arbitrary two-dimensional structures at a meta-module granularity of four individual modules. Figure 4.11 shows (a) one single meta-module and (b) a grid of meta-modules. When using this approach, it is assumed that the ensemble is already formed at meta-module granularity, i.e. each element in the grid has either zero or four single modules. There are no constraints on which parts of the grid are filled with meta-modules and which parts are not, as long as all modules in the ensemble are either directly or indirectly connected.
4. Domain Analysis

(a) One single meta-module [16]  (b) A grid-structure of meta-modules [16]

Figure 4.11: Meta-modules devised by Brandt et al. (2007)

4.5.1 Meta-module moves

Due to the high degree of symmetry of the meta-modules and its simple motion constraints, only three basic moves are sufficient to facilitate all possible moves of a meta-module in the grid. In Figure 4.12 are shown the three basic moves.

(a) Flip-Along [16]  (b) Flip-Over [16]  (c) Flip-Through [16]

Figure 4.12: Three different basic meta-module moves. Each dark grey square represents a meta-module, while a white square represents an empty square. For the light grey squares there is no constraint upon the presence of a meta-module or not. [16]

Using these three basic moves, all possible grid transitions can be made, respecting the grid shown in Figure 4.11(b). In order to achieve grid transitions on the meta-module level, we need a means to express it. Ideally, this should be done using local knowledge, to enhance scalability and parallelism. Each meta-module should be treated as one single autonomous module.

In order to be able to apply the same meta-action but in a different orientation, we need a means to rotate / mirror it, in order to avoid the need of having to re-implement the low-level behavior of it. The Flip-Along and Flip-Through sequence have eight different orientations, namely 0 degrees, 90 degrees, 180 degrees and 270 degrees, and for all four different angles also the mirrored version. For the Flip-Over sequence the rotations are equal to the mirrorings, hence it only has four rotations.

This leads to the following concept:

Concept 12: Sequence orientation

“A sequence orientation parametrizes a sequence with an orientation parameter, such that a generic sequence can be executed in multiple orientations”
Cluster-flow devised by Brandt et al.

Figure 4.13: Different orientations shown for meta-actions: The Flip-Along and Flip-Through have eight different orientations, while the Flip-Over has only four.

4.5.2 Defining global behavior: Strategies

In order to realize global behavior, e.g. cluster-flow in a certain direction, we need a high-level means that assigns behavior to modules at meta-module granularity. We do this by using a so-called strategy. A strategy is a collection of rules, that work on the neighborhood information of a meta-module.

Figure 4.14 shows a strategy that implements cluster-flow in northern direction, using three rules. Meta-module moves are triggered by the availability of the eight neighbor squares of each meta-module in the grid. Each neighbor squares is either assigned to a set represented by ■ or □, in which the former set contains the occupied squares while the latter contains the free spots.

Figure 4.14: A strategy for cluster-flow in northern direction. Note that in this meta-module setup, the second rule of the strategy is executed three times before rule the third rule will be applied.

Each of the three rules checks when a meta-move is possible, i.e. when the corresponding white spots are free and dark grey spots are occupied. The rule annotated with (1) checks whether a meta-module has a neighbor in the north position, and whether its west and north-west positions are free. If this is the case, it executes the Flip-Through meta-move. Once completed, the rule annotated with (2) will fire on the meta-module, and it will execute the Flip-Along in northern direction, until it reaches the front of the ensemble.

To allow the programmer to express these moves, we need a high-level means of expressing
these strategies in the DSL. This leads to the following concept:

**Concept 13: Strategy**

“A strategy is a high-level program that defines behavior using meta-modules, regions and sequences”

A strategy works on the meta-module level, and is treated as a single unit. As a strategy uses the meta-neighbor information of a meta-module, we need a means to store and share this information inside a meta-module, among all individual modules. A meta-module variable is stored among all its single modules and automatically synchronised when its value changes. This leads to the next concept:

**Concept 14: Meta-module variable**

“A meta-module variable is a variable stored in all individual modules that make part of the meta-module, and automatically shared and updated when changed.”

Now we have defined a global means to assign meta-actions using a strategy, we need to define the concepts for the low-level control. Hence, all three meta-actions for this approach are described in the next three sections.

### 4.5.3 Meta-action One: Flip-Along

Figure 4.15 shows the Flip-Along meta-action, and the corresponding pseudo-code is shown in Listing 4.4. Only one of the three participating meta-modules is moving in this meta-action, the remaining two meta-modules are stationary.

It is important to mention that the stationary meta-modules are not inactive, as have to be involved in the entire meta-action. In action 4, *Clover.North* need to connect to the structure, but it will be aligned to the structure with a female connector, hence the structure needs to make the connection to the meta-module at this point. In action 6, *Clover.South* needs to be disconnected from the structure, and also in this case the structure has to disconnect from the module. Therefore, the two stationary meta-modules need to be involved in the meta-action.

![Figure 4.15](image)

Figure 4.15: A group of eight ATRON modules performing the Flip-Along meta-action like shown in Figure 4.12(a).
Module groups  In this example we see that using module groups help to specify actions concise. In actions 6,7,8 and 9, we instruct to (dis)connect to the structure in general, regardless what the exact id of the module is. Without module groups, we need to assign an id to it, and we need to remember to which exact id we have to (dis)connect.

For the Clover meta-module, all Structure modules can be treated equally, as in each action we just want to connect or disconnect a specific Clover module to its Structure module that it is aligned to. Thus, we need a means in the language to create groups of modules, that can be used to assign behavior to.

Limitation of DynaRole sequences  The pseudo-code for this algorithm is at a higher level compared to the eight2snake sequence described in Subsection 4.2.2. Indeed, connect and disconnect constructs are expressed using only module identification, without needing a connector number. Expressing behavior at this level of abstraction is a limitation of DynaRole sequences [65], as the programmer need to figure out connector numbers manually. Moreover, the programmer has to know which of the two modules has the male connector that connects to the other, as female connectors are completely passive and the module having the male connector must take the action.

4.5.4 Meta-action Two: Flip-Over

Figure 4.16 shows the Flip-Over meta-action. In this move, the ATRON modules apply exchange of single modules between meta-modules, forced by the granularity of the grid. Listing 4.5 shows pseudo code for the Flip-Over meta-action.

Module groups  In this example we see that using module groups help to specify actions concise. In actions 6,7,8 and 9, we instruct to (dis)connect to the structure in general, regardless what the exact id of the module is. Without module groups, we need to assign an id to it, and we need to remember to which exact id we have to (dis)connect.

For the Clover meta-module, all Structure modules can be treated equally, as in each action we just want to connect or disconnect a specific Clover module to its Structure module that it is aligned to. Thus, we need a means in the language to create groups of modules, that can be used to assign behavior to.

Limitation of DynaRole sequences  The pseudo-code for this algorithm is at a higher level compared to the eight2snake sequence described in Subsection 4.2.2. Indeed, connect and disconnect constructs are expressed using only module identification, without needing a connector number. Expressing behavior at this level of abstraction is a limitation of DynaRole sequences [65], as the programmer need to figure out connector numbers manually. Moreover, the programmer has to know which of the two modules has the male connector that connects to the other, as female connectors are completely passive and the module having the male connector must take the action.

4.5.4 Meta-action Two: Flip-Over

Figure 4.16 shows the Flip-Over meta-action. In this move, the ATRON modules apply exchange of single modules between meta-modules, forced by the granularity of the grid. Listing 4.5 shows pseudo code for the Flip-Over meta-action.

In Figure 4.12 the dark squares represent a meta-module each, and the region that executes the meta-action should cover all dark squares.
Grouping meta-modules in a region

This meta-action is assigned to meta-modules by using a strategy, that checks when meta-modules are p. In this example, we see that a meta-module can only execute one meta-move at the same time. Therefore, we need a means to lock meta-modules to prevent them from being captured by another meta-module when still being active. Therefore, prior to executing the transformation in a meta-action, the participating meta-modules should be placed in an isolated region. When the meta-action has finished, the region can be disabled and the meta-modules in the region are free for the strategy to be reassigned new behavior.

In addition, we can limit propagation of information between modules to only propagate to neighbors within the same region. This allows multiple regions to exist at runtime, without the need of having knowledge about each others existence, solving the state-sharing scalability issue from DynaRole [65] as described in Section 3.8. This leads to the following concept:

**Concept 15: Region**

“A region is an isolated unification of meta-modules that can be enabled and disabled at runtime, to lock meta-modules that are working together to execute a meta-action.”

Grouping functionally-equal modules

The Flip-Over meta-action can be executed by making two groups in the region. The first group are the yellow/purple modules and is called Inside. The second group are the yellow/magenta modules and is called Outside.

Groups can by assigned by using the module’s relative position in the ensemble, by checking which connector sets are connected to neighbors. However, that would require to write a connector rule for every single module in the group, as every module has a different relative placement. A gradient [57] allows to assign groups by first specifying a two-dimensional gradient, and thereafter the range of gradient values from which the modules must be added to
the group. This leads to a new concept:

**Concept 16: Gradient**

“A gradient is a function that assigns a number to modules in a successive way, starting at one or multiple source modules with the number 0. All other modules are assigned the number of modules that it needs to cross in order to reach the source.”

**Figure 4.17** represents how a gradient is used to assign both module groups in **Figure 4.16** by (a) showing the assigned gradient values and (b) the group assignment function for each module. The modules at the right and the bottom are defined as the source for the horizontal and vertical gradient, respectively.

$$
\begin{bmatrix}
1 & 3 & 0 & 3 \\
2 & 2 & 0 & 2 \\
3 & 1 & 2 & 1 \\
3 & 0 & 2 & 0 \\
\end{bmatrix}
$$

(a) Gradient value assignment (x,y) for each module in **Figure 4.16**

$$
group = \begin{cases}
\text{outside} & \text{if } (x = 0 \lor y = 0) \\
\text{inside} & \text{if } (x = 1 \land y > 0 \lor y = 1 \land x > 0)
\end{cases}
$$

(b) Group assign function

**Figure 4.17**: The use of a gradient for assigning groups

In order to communicate with neighbors and to allow the programmer to define gradients in a generic way, a communication system is needed in the language. Using this communication system, the programmer can send packets to specific neighbors, and neighbors have a means to process received packets. In order for neighbors to receive packets that are sent, we can use event handling (4) to execute specific behavior to process the packet.

This leads to the next concept:

**Concept 17: Communication system**

“A system that enables communication between modules, by sending packets and providing a means to process received packets.”

4.5.5 Meta-action Three: Flip-Through

The Flip-Through is very similar to the Flip-Over. The only difference is that the Flip-Over uses three meta-modules while the Flip-Through uses only two. In contrast to the Flip-along action, this action does not have inactive meta-modules during the meta action. In addition, individual modules need to exchanged between meta-modules in this move. **Listing 4.6** shows pseudo code for the Flip-Through meta-action.

```
1 do { disconnect (Lifter.Right, Dummy.Left), disconnect (Lifter.Left, Dummy.Right) }
```
4. Domain Analysis

Figure 4.18: A group of eight ATRON modules performing the Flip-Through meta action like shown in Figure 4.12(c).

Listing 4.6: Pseudo code for the Flip-Through meta-action

```plaintext
2 do { rotate (Lifter.Left, 90), rotate (Lifter.Right, 90) }
3 do { disconnect (Dummy.Left, Lifter.Left) }
4 do { rotate (Lifter.Top, -180) }
5 do { rotate (Lifter.West, 90), rotate (Lifter.East, 90) }
6 do { rotate (Lifter.South, -180) }
7 do { reconnect all modules }
```

4.6 Runtime System

In order to make abstractions for the programmer that incorporates all concepts defined in this chapter, we need a so-called runtime system. This is a library or ‘operating system’ for the robot control programs, from which the robot control programs can use functionality. Figure 4.19 shows how the runtime system relates to the robot control programs.

Figure 4.19: The Runtime System provides abstractions for different control programs

62
In the case of an external DSL approach, the robot control program is generated by a
compiler. Hence, the compiler can generate code that use the abstractions provided by the
runtime system, reducing the size and complexity of the generated controller, which is often
beneficial [33]. In this case of an internal DSL approach, a GPL is used as a host language and
the runtime system can be implemented in the same GPL.

This leads to the next concept:

**Concept 18: Runtime system**

“A static system that implements abstractions for the concepts in the language”

### 4.7 Summary

The following concepts are identified to be integrated in the new DSL.

Modules can be referred to using module identification [1], or by using a module group [8].
Assigning the appropriate id or group to each module can be done by checking its relative
position in the ensemble using its connector sets [2] or by using a gradient [16]. Modules
propagate their id, group and other information to neighbors such that they can store the
association between connector numbers and neighbor module properties in a module neighbor
table [9].

A sequence [3] is defined as a list of distributed actions, to be executed in sequential order
among the ensemble, in which a single action can contain multiple statements to be executed in
parallel, and can even be switched to each other by using a sequence transition [5].

Sequences can be implemented in a generic way, to be able to execute it in different angles
and directions, using a sequence orientation [12]. After executing a sequence, symmetry
disruptions can be corrected using symmetry preservation [10]. Actions in a sequence use
hardware control primitives [6] in order to control the hardware of the robot.

In order to express problems at a higher level of abstraction, a meta-module [7] is used.
which are equally-sized groups of single modules, each having their own meta part [11]. It
can contain meta-module variables [14] that are automatically synchronized across all single
modules in the meta-module.

A strategy [13] implements the high-level behavior in the robot, such that multiple strategies
can use the same low-level control implementation. A region [15], created by a strategy, is an
isolated group of meta-modules in the ensemble, allowing multiple groups of meta-modules to
work in parallel.

In order to send and receive packets of information, a communication system [17] is used
for propagating information through the robot. We can use event handling [4] to process
received packets, but also execute specific actions when a condition is satisfied.

Finally, a runtime system [18] is used to implement all conceptual abstractions.
Chapter 5

Language Design of MetaForma

This chapter describes how the MetaForma language syntax and semantics are derived from the concepts defined in the domain analysis (Chapter 4). First, we argue for using a DSL approach. Second, we present the language using implemented examples, derived from the scenarios in the domain analysis.

5.1 Language Architecture

5.1.1 Domain-Specific Language vs. General-Purpose Language

Arguments for a DSL approach A DSL approach allow solutions to be expressed at the level of abstraction of the problem domain. Hence, domain experts can understand, validate, modify, and often even develop DSL programs [31]. As a result of the higher level of abstraction, productivity [31, 60, 76] and maintainability [31] are enhanced in comparison to GPL approaches.

DSL programs are concise, self-documenting to a large extent, can be reused for different purposes [31], and can abstract over low-level implementation details and can eliminate the need for boilerplate code [50].

Portability [31], flexibility [31, 60], reliability [31, 60] and usability [60] are increased as a result of higher-level domain abstractions. DSLs contain domain knowledge, which enables the conservation and reuse of domain knowledge [31]. Moreover, as domain knowledge is integrated in the DSL compiler, validation and optimization at the domain level [31] becomes possible.

Counter-arguments for a DSL approach The costs of designing, implementing and maintaining a DSL [31] and the cost of education for DSL users [60] is a disadvantage.

Finding the proper scope for a DSL [31] and finding the proper balance between domain-specificity and general-purpose programming language constructs [31] appears to be difficult. DSLs easily evolve too big, in the direction of becoming a GPL, which is known as the Language Ghetto Problem [33].

Although IDE support for DSLs has been improved significantly the last decade, advanced debugging support in the IDE [50] and other necessary tasks for DSL users is still lacking [60, 77].
Finally, even although DSL code can be *optimized at the domain level*, the lower-level code that is generated by the compiler has a *potential loss of efficiency* compared to a semantically-equivalent manual implementation in a GPL [31].

### 5.1.2 Motivation for DSL approach

A common objection against DSLs is known as the Language Cacophony Problem [33], arguing that languages are hard to learn, therefore using multiple languages is more complicated compared to using one language. According to Fowler *et al.* (2010) [33], two main misconceptions are made here.

First, projects will always have complicated areas that are hard to learn. In a GPL, these areas are usually covered by libraries, meaning that the interface to these libraries still has to be understood in order to use them.

Second, learning a DSL is usually not as hard as learning a GPL. Furthermore, the whole point of a DSL is to make the model easier to understand; the argument that learning a new language is more difficult than using an existing language does not hold in general.

Programming self-reconfigurable robots is in general a difficult task [2, 3, 10, 11, 24, 45, 53, 55, 61, 63, 65, 72, 80]. GPLs like ANSI-C or Java provide only general abstractions that do not facilitate tackling difficult issues such as distribution, unreliable hardware, and a dynamically evolving communication topology. In order to implement global sequences on a distributed network of nodes using unreliable communication, a distributed state machine has shown to be useful [65]. And different execution modes than imperative, like a state machine, are a strong motivation for using a DSL approach [33]. Hence, we believe that a DSL approach is needed to be able to program robot behavior on a higher, more productive level.

### 5.1.3 Internal DSL vs. External DSL

An *internal DSL* is implemented inside a GPL, in which the GPL acts as a host language for the DSL. The DSL uses a subset of the GPL language-constructs in DSL programs, meaning its syntax is constrained to the syntax of the host language.

An *external DSL* is not implemented using a host language, and can therefore have its own syntax and semantics. Fundamentally, four main ingredients for the construction of an external DSL exist [41]: A parser, static analysis, a target code generator and a language IDE. Traditionally, each of these ingredients needed to be implemented, hence requiring a significant effort. However, the uprising trend of Language Workbenches, e.g. Spoofax [41], ease the development of external DSLs significantly by incorporating all traditional ingredients into one single tool.

### 5.1.4 Language Architecture: External DSL

As described in [Subsection 2.1.3](#), the first generation of ATRON modules are highly resource-constrained; this directly affects the range of possible host languages to choose from that can be used on the physical ATRON modules. Typical host languages for internal DSLs like Java
or Ruby cannot realistically be used on the physical modules [63]. ANSI-C can be used on the physical modules, but using ANSI-C as a host language for an internal DSL will probably introduce syntactic noise in the DSL programs [33]. Furthermore, semantic editor services for general-purpose ANSI-C are significantly limited to editor-services available in external DSL approach.

Figure 5.1 shows the architecture of the MetaForma language. The language is implemented as an external DSL, consisting of a Strategy language to implement high-level control, and a Motion and Coordination Language to implement low-level control. The syntax of MetaForma is inspired by Java, in order to minimize the Language Cacophony Problem [33].

5.1.5 Motion and Coordination Language

The Motion and Coordination Language is used to implement a library of reusable MetaForma code for a specific scenario. First, it comprises the Init sequence that is automatically executed when a module is started. A typical task of the Init sequence is creating a grid of meta-modules in the ensemble, like shown in Figure 4.10 and Figure 4.11, for meta-module behavior. Once the Init sequence is finished, the strategy automatically gets control.

Second, transformation sequences like the Turn, Flip-Along, Flip-Over and Flip-Through (described in Chapter 4) are also implemented in the Motion and Coordination Language. As shown in Figure 5.1, once a sequence is finished, it automatically hands over control to the strategy, either directly or by first re-executing the Init sequence.

5.1.6 Strategy Language

The strategy language incorporates the concept of a strategy [13] and a region [15] to express global behavior. Fundamentally, a strategy is a sequence, but with a special meaning: Unless implemented otherwise, the strategy automatically gets control once the current sequence is finished. Therefore, the strategy language can be used to start a specific sequence determined by a satisfied condition, which will return control to the strategy when finished. Hence, after several executed sequences, global behavior will emerge from the ensemble.
5. LANGUAGE DESIGN OF MetaFORMA

```cpp
float FORWARD_VEL = 1.0 // forward velocity
float BACKWARD_VEL = -1.0 // backward velocity
float PROX_TRIGGER = 0.15
float TIME_BACKWARD = 5.0

sequence Strategy { 
  when (module.id == Axle.Front && module.proximity() > PROX_TRIGGER) { 
    execute Turn(RIGHT);
  }
  meta.drive(FORWARD_VEL, FORWARD_VEL);
}
```

Listing 5.1: High-Level Strategy for Obstacle Avoidance

5.2 Obstacle Avoidance Strategy

In this section we describe a MetaForma program that implements high-level obstacle-avoidance behavior for both a Two-Wheeler and Four-Wheeler car. The low-level implementation that is used by the strategy is implemented in Section 5.3 and Section 5.4 for each type of cars. Listing 5.1 implements a high-level strategy that attains obstacle-avoidance behavior, that can be used on both the Two-Wheeler or Four-Wheeler car. Furthermore, it defines algorithmic constants to be used in the implementation of the low-level control.

The strategy contains one event and one sequential action. The sequential action is a function call to meta.drive, passing two parameter constants indicating the rotational direction of the left and right wheel(s). This function is manually defined and will be discussed later, in enables different drive behavior implementation for different types of cars.

**Event handler** The strategy uses an event to detect obstacles during driving forward, this is implemented using a when construct. When an obstacle is detected by the Axle.Front module, the when condition is satisfied and the car will execute the Turn sequence.

The event is included within the scope of the strategy, meaning that the condition in the when is only evaluated when the strategy is active, as evaluating the event handler in the Turn sequence as well would cause an endless loop. Event handlers can also be included within the scope of specific sequential actions, or within the scope of the program. The former means that the event handler will be evaluated when the specific action in the sequence is active. The latter means that the event handler is only evaluated constantly, regardless which sequence is active.

**Sequence transition** A new sequence can be started by using a sequence transition, implemented using the execute construct. As a result of the condition in the when, the statement in the event is only executed at the Axle.Front module. However, all modules belonging to the same meta-module will automatically receive new state updates, hence all modules in the car will start executing the new sequence. By specifying a sequence orientation as a parameter to the Turn sequence, we can later implement a generic sequence that can be used for turning a car in multiple directions.
The *meta.drive* function, and the *Init* and *Turn* sequences are implemented individually for each type of car in the *Motion and Coordination language*, as for these low-level aspects of control a specific implementation is needed.

5.3 Two-Wheeler Car Obstacle Avoidance

In this section we will present the *motion and coordination control* of the Two-Wheeler car algorithm. Below we define a package for the Two-Wheeler implementation using the *package* construct, which is the first construct of a MetaForma source file. This scopes all definitions written in the file with the package name, hence it allows definitions to be spread over multiple files, without interfering with definitions in other packages.

```
package TwoWheeler
```

Furthermore, we declare *module identification* and *module groups* for the Two-Wheeler by declaring two groups: a *Wheel* group, for the left and right wheel, and a group *Axle*, for the front module. Inside the declaration of a module group, we declare the module names that belong to that group. Once declared, the ids and groups can be assigned to modules. Later, we can refer to modules using the groups and ids.

```
1 group Wheel [Left,Right]
2 group Axle [Front]
```

5.3.1 Init sequence

The *Init* sequence is the sequence that automatically gets executed on a module once the *runtime system* becomes active. [Listing 5.2](#) defines the *Init* sequence for the Two-Wheeler, to initialize the robot. This sequence contains two sequential actions. In the first action, we create a meta-module of all modules in the ensemble by using the *enable meta* construct. This enables communication inside the meta-module, as we need sequential behavior among all modules in the ensemble. In the second action, the ids and groups are assigned to the appropriate module.

**Module Reference Operator** In order to use module groups and ids declared by the *group* construct, the *module reference operator* is used. It can refer to complete groups, but also specific ids. In order to refer to all modules within a group, the group name must be specified. Furthermore, in order to refer to a unique module, the group name with a period follow it by the module id must be specified. Thus, for the module-reference operator, a group is required, and the module name is optional. Below we see an example:

```
@Group @Group.moduleName
```
In Figure 5.2 is shown a Two-Wheeler car and its module ids, that are assigned by the Init sequence implemented in Listing 5.2. By checking each module whether it matches a specific, relative placement in the ensemble, we can dynamically assign the appropriate id to every module. We do this by using the module neighbor table (9), as it allows to refer to connectors in a uniform way. The id of a module is stored in the primitive variable module.id.

Neighbor Count Operator This returns the number of neighbors in the module neighbor table (9), that match eventual specified condition(s). Conditions can be combined by separating them with a comma, resulting in a conjunction. The operator has the following format:

```
#(condition1:val1, condition2:val2, ..)
```

To query the neighbor table using connector sets, two conditions can be used: connOwn and connNb. The former affects connectors adjacent at the current module, while the latter affects connectors adjacent to the neighboring module(s). More conditions can be specified to query on other module properties, which we will see later in this chapter.

At each module, the runtime system evaluates three if statements, in order to assign the appropriate module id. Each module checks by using the neighbor table whether it has one east or one west connector adjacent to that neighbor. In the former case, we assign the module id Wheel.Left, in the latter case we assign Wheel.Right. The runtime system also checks for any module to have two neighbors, and that module is assigned the Axle.Front id.

As during self-reconfiguration the neighborhood of modules change, the module neighbor table (9) needs to be up to date with the correct neighbors before the sequence continues with the next sequential action. Therefore, we use a do..until block, such that the runtime system (18) executes the statements repeatedly for a fixed amount of time, represented by the runtime-system constant assignTime. During execution of a do..until action, the runtime system in each module communicates with neighbor modules, to update the module neighbor table.

**Syntax and semantics of sequential actions** Sequential actions are executed by the runtime system and can be implemented in different ways. In Listing 5.3 are shown the syntax and semantics of sequential actions. We make a distinction between do..wait(postcondition)
and `do..until(postcondition)` actions. The former is used to execute an action once after which the runtime systems waits for `postcondition` to be satisfied, the latter is used to execute continuously until the postcondition is satisfied.

Valid postconditions comprise (a) `consensus()`, which is satisfied when consensus among all modules in the meta-module is achieved, and (b) a specified float value `t` which is satisfied after waiting for `t` seconds. Consensus is achieved when all modules in the meta-module have finished their current action. \(\text{(Section 7.3.3)}\)

The postcondition can be omitted as syntactic sugar for `consensus()`. Moreover, the `do` construct can be omitted as well if the action contains only one statement.

### 5.3.2 Turn Sequence and Drive Behavior

In this subsection we present the behavior for driving and turning the Two-Wheeler.

The previously defined strategy in \[\text{Listing 5.1}\] and the Turn sequence use the `meta.drive` function in order to attain rotating wheels. In \[\text{Listing 5.4}\] is implemented both the Turn sequence and the `meta.drive` function. The sequence comprises a list of four sequential actions for turning the Two-Wheeler.

First, the runtime function invokes the `stop` function on all modules in the `Wheel` group. Second, when the wheels are stopped, the `meta.drive` function is invoked. This attains the car to drive backward, for the time defined by `TIME_BACKWARD` in the strategy. Third, the car is turned, by rotating the wheels in opposite direction. After commanding the wheels to rotate continuously, we wait for the constant `TIME_TURNAROUND`. Fourth, the wheels are stopped and as the sequence is finished the Strategy automatically gets back the control.

Using the `parts` construct in the meta-module we set the amount of modules in the meta-module to three. This means that after we have invoked `meta.enable` in the `Init` sequence, a meta-module of three modules is created.
5. LANGUAGE DESIGN OF METAFORMA

Listing 5.4: Two-Wheeler Turn Sequence and meta-module functionality implementation

```plaintext
sequence Turn {
  stop(@Wheel);
  do {
    meta.drive (BACKWARD_VEL, BACKWARD_VEL);
  } wait(TIME_BACKWARD);
  do {
    if {LEFT} meta.drive (BACKWARD_VEL, FORWARD_VEL);
    if {RIGHT} meta.drive (FORWARD_VEL, BACKWARD_VEL);
  } wait(TIME_TURNAROUND);
  stop(@Wheel); // Stop the wheels, before strategy gets control again
}

parts 3; // Three modules
meta {
  void drive (float dirLeft, float dirRight) {
    rotating(@Wheel.Left,-dirLeft); // Rotate continuously
    rotating(@Wheel.Right,dirRight); // at velocity of ∈ [-1,...,1]
  }
}
```

Actuation In order to control the rotational actuator, a number of hardware control primitives can be used. The primitive function rotate attains rotation for a number of degrees relative to the current degree of the module, and rotateAbsolute attains rotating to an absolute degree. The rotating primitive realizes continuous rotation at a specified velocity, which can afterwards be stopped using the stop primitive. To rotate back to the origin at zero degrees, we use the rotateOrigin primitive.

Listing 5.5: Functions in MetaForma to control the central actuator

```plaintext
// Rotate modules referred by r relative/absolute deg degrees
rotate (ModuleRef r, int deg);
rotateAbsolute (ModuleRef r, int deg);
// Rotate modules referred by r to 0 degrees
rotateOrigin (ModuleRef r);
// Rotate modules referred by r continuously at velocity vel
rotating (ModuleRef r, float vel);
// Stop continuous rotation at modules referred by r
stop (ModuleRef r);
```

A ModuleRef represents a reference to either a module group or module identification.

5.4 Four-wheeler Car Obstacle Avoidance

In this section we will present the motion and coordination control of the Four-Wheeler car algorithm. In order to attain obstacle-avoidance behavior at a Four-Wheeler car, the same previously-defined strategy in Listing 5.1 is used.
A Four-Wheeler has two functionally-equal wheels on each side. As we need to refer to both of the two left and both of the two right wheels at the same time, we now use groups instead of ids. In addition, we define an Axle group, for the three axle modules.

```plaintext
1 package FourWheeler
2 group Axle [Driver, Front, Back]
3 group LeftWheel
4 group RightWheel
```

In Figure 5.3 is shown how the groups and ids are assigned to each module, which is done in the Init sequence.

### 5.4.1 Init Sequence

In this subsection we present the implementation of the Init sequence for the Four-Wheeler, which is implemented in [Listing 5.6](#). Analogous to the implementation for the Two-Wheeler, we use two sequential actions; first we enable the meta-module, and second we assign the appropriate module ids and groups. The group of a module is stored in the primitive `module.group` variable.

The function `module.lying` is defined by the programmer and computes whether a module is lying (horizontally) or standing (vertically), by reading out its tilt sensor for the Z-coordinate of the ATRON module. When a module is standing the tilt on the Z-coordinate equals zero, whereas for a lying module it equals 90.

Using this function, we define conditions to assign the appropriate id and group to each module, depending on its relative position to other modules. The conditions for the Four-Wheeler are more complicated compared to the Two-Wheeler, mainly because we need to identify each module in a bigger and three-dimensional ensemble, and the module neighbor table allows to only detect direct neighbors. It might be an interesting project for future work to improve the means to address modules based on their relative position in the ensemble (Subsection 9.4.7).

The runtime system checks continuously at each module whether it is lying or not, and whether one of the if conditions is satisfied by using the neighbor count operator. If a condition is satisfied on a module, the runtime system sets its corresponding module id and group.

In addition to only connector numbers, information from neighbors such as module ids and groups is stored in the module neighbor table as well. Therefore, the Module Reference Operator can be included, meaning that we can filter on neighbors on which the specified module id or group matches. In order to identify Axis.Driver, the runtime system checks whether it has two Axle neighbors.

### Connector Sets

Figure 5.4 shows (a) the connector sets that are defined as built-in constants in MetaForma and (b) which connectors they represent on the modules. An ATRON module has eight connectors, therefore connector sets are treated like a single byte. Sets can be combined using bitwise arithmetic including `∥` and `&` to compute their union and intersection.
5. LANGUAGE DESIGN OF MetaFORMA

Listing 5.6: Module ids and groups assignment for a Four-Wheeler Car

```java
sequence Init {
    enable meta;
    do {
        if (module.lying()) {
            if (#(connOwn:NORTH&WEST)==1 && #()==3)
                module.id = @Axle.Front;
            if (#(connOwn:NORTH&EAST)==1 && #()==3)
                module.id = @Axle.Back;
        }
        else {
            if (#(connNb:MALE&EAST|FEMALE&WEST)==1 && #()==1)
                module.group = @LeftWheel;
            if (#(connNb:FEMALE&EAST|MALE&WEST)==1 && #()==1)
                module.group = @RightWheel;
            if (#[@Axle]==2)
                module.id = @Axle.Driver;
        }
    } until (config.assignTime);

    module {
        boolean lying() | return module.tiltZ() > 50; }
}
```

Figure 5.3: Four-wheeler car

(a) Connector sets and their byte representation  
(b) Graphical representation

Figure 5.4: Connector sets for the ATRON module

5.4.2 Turn Sequence and Drive Behavior

In this subsection we present the behavior for driving and turning the Four-Wheeler. The Turn sequence uses six sequential actions and is implemented in Listing 5.7. By using a conditional check to the sequence orientation[12], either a turn to the left or the right is realized.

In order to reuse behavior, in the meta class are three functions implemented: The drive function attains continuous rotation of the wheels, the steer function rotates the front and
back axle of the Four-Wheeler car such that the car drives in an angle, and the halt function stops the wheels. The parts construct in the meta class defines the amount of modules in the meta-module.

```cpp
1 sequence Turn {  
2     meta.halt(); // Stop the wheels  
3     do {  
4         meta.drive (BACKWARD_VEL,BACKWARD_VEL);  
5         wait(TIME_BACKWARD*0.25);  
6         do {  
7             if (LEFT) meta.steer(STEER_DEGREES);  
8             if (RIGHT) meta.steer(-STEER_DEGREES);  
9         }  
10         do {  
11             meta.drive (BACKWARD_VEL,BACKWARD_VEL);  
12         } wait(TIME_BACKWARD*0.75);  
13         meta.halt();  
14     }  
15     do {  
16         if (LEFT) meta.steer(-STEER_DEGREES);  
17         if (RIGHT) meta.steer(STEER_DEGREES);  
18 }  
19     meta {  
20         parts 7; // Four-wheeler Car = 7 modules  
21         void steer (int degrees) {  
22             rotate(@Axle.Front,degrees);  
23             rotate(@Axle.Back,-degrees);  
24         }  
25         void drive (float dirLeft, float dirRight) {  
26             rotating(@LeftWheel,-dirLeft);  
27             rotating(@RightWheel,dirRight);  
28         }  
29         void halt () { stop(@LeftWheel); stop(@RightWheel); }  
30     }
```

Listing 5.7: Implementation of the Turn Sequence for the Four-Wheeler

### 5.5 Cluster-Flow devised by Christensen et al.

This section describes the implementation of cluster-flow using the meta-module approach of Christensen et al. [29], as implemented in Section 4.4. First, we need to divide the structure into a grid of meta-modules. Using the grid, we can express cluster-flow at meta-module granularity using a high-level strategy.

#### 5.5.1 Create Grid of Meta-modules

Creating a grid of meta-modules in an ensemble is not a trivial task; it needs to be done in a systematic way. The following algorithm is used, which is visualized in Figure 5.5.
Algorithm  The algorithm is divided in two steps.

- It starts at single modules with at least two neighbors and a connected neighbor at EAST\&MALE\&SOUTH. If this is the case, it starts the new meta-module, sets its \textit{meta part} to \texttt{LeftPart} and sends an \texttt{AssignMetaID} packet to the neighbor at EAST\&MALE\&SOUTH.

- When the \texttt{AssignMetaID} packet is received by a neighbor, the module sets its meta id to the meta id in the packet. If the counter in the packet equals zero, the meta part of the module is set to \texttt{MiddlePart} and the packet is propagated further to the right with an increased counter. When the counter equals one, the module sets its meta-part to \texttt{RightPart}, and the meta-module is successfully created.

To implement the distributed algorithm, we first define a packet to use for sending and receiving messages. We add two bytes to the packet, to be included in the message as payload: The \texttt{newID} is used to transfer the new meta id to the neighbor.

\begin{verbatim}
packet AssignMetaID { byte newID, counter; }
\end{verbatim}

In \textbf{Listing 5.8} is the algorithm implemented in the \texttt{Init} sequence. We use one sequential action in a \texttt{do..until} block, as we want to continuously execute the algorithm, packets that are sent might not be received by the neighbor, so it needs to keep sending the...
packets until the meta-module is correctly created. To implement both steps of the algorithm, two event handlers are used.

The first event handler checks whether a module has a connected neighbor at the EAST\&MALE\&SOUTH connector. If this is the case, it sets its meta id to its module number, which is a byte representation of the module identification[1], making the meta id unique.

The second event handler processes received packets and propagates new packets to the right, by using the Neighbor Operator $(). It has the same semantics as the neighbor count operator #(), but instead of returning the amount of neighbors it returns the set of neighbors. The mirror primitive computes the connector number at the mirror side of the module$^1$.

In Figure 5.5 is shown how the packet travels to the right in two steps. When a meta-modules

Communication System Packets can be sent and received using the communication system[17], by using the send.. and when (receive ..) primitive, respectively. A packet can contain so-called payload that is included in the packet during transmission, their values can be assigned between parenthesis after the send command. Currently only byte variables are supported. Received packets by neighbors can be processed by using event handling[4]. Upon reception of the packet by a neighbor, the payload variables can be accessed in its event handler. Also the connector number on which the packet was sent and received can be accessed using the connNb and connOwn primitives, respectively.

Received packets are processed by event handlers, the programmer can specify whether a received packet must be processed or ignored by including it in the package, sequence or sequential action scope. This means that a received packet is (1) always processed when received, or only processed if the runtime system of the receiver is executing the (2) sequence of the event handler or (3) the sequence and sequential action of the event handler.

As the event handler in Listing 5.8 is included in the sequential action scope, received packets are only processed by the runtime system if the module is executing the first sequential action of the Init sequence, otherwise they are ignored.

5.5.2 High-level Strategy for Cluster-flow

In this subsection we present the strategy to realize cluster-flow in forward direction. In Listing 5.9 is implemented the strategy to attain this. It contains an event handler that is used to detect meta-neighbors, and one sequential action. The strategy is executed among every meta-module[7] until one of the three conditions is satisfied for lifting a walker up, walking a step or lifting the walker down. The strategy works as follows.

If a meta-module is lying on the floor while not having a neighbor at the bottom, it is lifted up to become a walker. As a walker cannot lift itself up to the scaffold, it uses a region[15] incorporating its meta-neighbor at the top. Both meta-modules work together in the lifting. A delay of twenty seconds is used before the lift takes place, to give the previous walker time to walk away before the new walker rises up.

If a meta-module is already placed on top of the scaffold in order to walk to the front, it checks whether it already reached the front, or another step forward can be made. In the former

$^1$The boolean argument specifies whether to mirror in east-west(true) or in north-south(false) direction
5. LANGUAGE DESIGN OF METAFLORA

```c
sequence Strategy {
  when (receive {Packet p}) {
    meta.neighborHook {p};
  }
  do {
    if (meta.type == LYING) {
      wait (20) // wait for next walker!
      try region GetUp {TOP_LEFT} including
        Top excluding Bottom,Up;
    }
    else if (module.id == @Walker.Right) {
      if (#@X,inRegion: false) == 2)
        try region LiftDown including Bottom;
      if (#@X,inRegion: false) == 3)
        execute WalkStep;
    }
  } until (false);
}
```

Listing 5.9: High-level Strategy to Attain Cluster-flow forward

Figure 5.6: Cluster-Flow forward

In the latter case, it creates a region with its bottom neighbor, to execute the LiftDown sequence.

Creating regions A region can be created by the `try region` construct, which automatically tries to create a region from the meta-neighbors specified in the `including` clause. The optional `excluding` clause specifies meta-neighbors that must be absent before the region can be created. Once the region is successfully created, it automatically executes the sequence using the `sequence orientation` that is specified in the command.

5.5.3 Defining Meta Neighbors

A strategy makes decisions at meta-module granularity, using meta-neighbors and, but a meta-module consists of individual ATRON modules. we need to implement a means to make this knowledge available in every single module. The meta class declares the meta-module variables and meta functions. In Listing 5.10 is defined the meta class; we define three meta-neighbors `Top`, `Bottom` and `Up` using a meta-module variable, to store the meta id of neighboring meta-modules at that position. The `type` variable stores whether the meta-module is lying on the floor or standing on the scaffold.

In order to assign the appropriate neighboring meta id to each meta variable, we define the `neighborHook` function. This function is invoked by the runtime system at the reception of any packet, and based on the meta part of the module it was received on, the runtime system can determine the position of the originating meta-module. The variable `p.metaID` contains the meta id of the sender of the packet. In order to store the appropriate neighboring meta id to each meta-module variable, we first filter out zero (no meta-module) and equality to its own
meta { 
    parts [LeftPart, MiddlePart, RightPart];
    byte Top, Bottom, Up; // Meta neighbors
    byte type; // LYING or STANDING
}

void neighborHook (Packet p) {
    if (p.metaID != module.metaID && module.metaID != 0 && p.metaID != 0 &&
        p.sourceGroup != @Walker) {
        if (meta.type == LYING ||
            (module.id != @Walker.Left)) {
            if (module.metaPart == MiddlePart) Bottom = p.metaID;
            if (module.metaPart == RightPart) Top = p.metaID;
        }}}

Listing 5.10: The implementation of the meta class

meta-id. If the packet is received at the MiddlePart or RightPart, the runtime system assigns the received meta id to the meta-variable Bottom or Top, respectively.

In the Strategy, we add an event handler to call this function at the reception of any kind of packet. As the receive construct uses the generic Packet type, it will match all packets that are received when the strategy is active.

5.5.4 Module gradients

In this subsection we implement a two-dimensional gradient for the grid that is created in Listing 5.8, that we can later use for module identification (1). First, we define two module variables gradPri and gradSec, for a primary and secondary gradient. Second, we define a function that initializes the gradient and sets both variables to MAX_BYTE.

Third, we define a function to propagate the gradient. How the gradient is formed depends on the current sequence orientation (12): In the strategy in Listing 5.9 we invoke with the TOP_LEFT orientation, meaning that the primary border is TOP and the secondary border is LEFT. As the gradPri and gradSec variables are stored in every module, gradientPropagate function assigns 0 to gradPri in modules at the TOP, and assigns 0 to gradSec in modules at the LEFT. Finally, we broadcast the Gradient packet to share the local values with neighbors, after assigning the pri and sec payload variables to the local values of gradPri and gradSec of the module.

The runtime system of each module determines according to Listing 5.12 whether it is situated at a specific border, by checking the relative placement of the module in the ensemble.

We define a Gradient packet with one payload variable per gradient and an event handler for handling received packets from neighbors, as shown below.

when (receive (Gradient p)) {
    if (p.pri + 1 < module.gradPri || p.sec + 1 < module.gradSec) {
        module.gradPri = min (p.pri + 1, module.gradPri);
        module.gradSec = min (p.sec + 1, module.gradSec);
        module.gradientPropagate(); // Re-propagate in case of a value change
    }
}
5. LANGUAGE DESIGN OF MetaFORMA

```c
module {
    byte gradPri, gradSec;  // Two-dimensional gradient
}

void gradientInit() { // Part 1 of gradient: initialize
    gradPri = MAX_BYTE;  // Set highest value
    gradSec = MAX_BYTE;
}

void gradientPropagate () { // Part 2: propagate to all nbs
    if (module.atBorder(orient.primary)) gradPri = 0;
    if (module.atBorder(orient.secondary)) gradSec = 0;
    send Gradient (pri=gradPri, sec=gradSec);
}
```

Listing 5.11: Functions for initializing and propagating a gradient

```c
borders []// Is a module situated at a border?
    LEFT: #(connOwn: EAST&MALE)>0;
    RIGHT: #(connOwn: WEST&MALE)>0;
    TOP: #(connOwn: SOUTH&MALE)==1 && #(connOwn: SOUTH&FEMALE)==0;
    BOTTOM: #(connOwn: EAST&FEMALE)==0 && #(connOwn: WEST&FEMALE)==0;
};
```

Listing 5.12: Module borders specification

If the runtime system receives a gradient packet, it takes the minimum of its own local gradient variable and the value it receives plus one. In case the local gradient values change, the packet is re-propagated immediately in order to update neighbors.

Hence, in order to use this gradient, first on all modules the runtime system must execute the `gradientInit` function, and consecutively the `gradientPropagate` function a sufficient amount of times, to make sure all modules are assigned their appropriate values. In Figure 5.7 are the values of \((\text{gradPri}, \text{gradSec})\) shown on each module, after the gradient is completed.

**5.5.5 LiftUp sequence**

This subsection presents the LiftUp sequence that is invoked by the strategy in Listing 5.9. In Figure 4.5 is the sequence shown. It is implemented in Listing 5.13 and consists in total of fourteen sequential actions. The first four actions attain module identification, the following nine actions realize the self-reconfiguration and the final action ends the sequence.

In the first part, we attain module identification by using the gradient shown in Figure 5.7. The primitive `module.storeID` stores a copy of the current module id, such that afterwards we can change its value temporarily, and at the end restore it to the original value by using `module.restoreID`.
A gradient to assign the appropriate module ids in the region. First, we invoke `gradientInit` to initialize all gradient variables. Second, we propagate the gradient for `config.propagationTime`, which is a primitive constant from the runtime system. Third, based on the `gradPri` and `gradSec` variables, we assign the appropriate module id to each module.

**Connecting and disconnecting neighbors**  The MetaForma language contains primitive functions to instruct the runtime system to control the connectors on each module. They are expressed in a procedural-oriented style, meaning that no receiver object is used in contrast to DynaRole sequences (Section 3.8). The main reason is that a receiver object is not intuitive in combination with module identification [1], as we need to specify two module-reference operators for the `connect` and `disconnect` action. As it is not known at compile time whether the receiver object or the parameter should take the action, it is more intuitive to use a procedural approach and pass both module-references as a parameter.

```plaintext
1 // Mutually (dis)connect modules referred by r1 between mod. referred by r2
2 connect (ModuleRef r1, ModuleRef r2);
3 disconnect (ModuleRef r1, ModuleRef r2);
4 // Extend/retract the single connector c at modules referred by r
5 extend (ModuleRef r, byte c);
6 retract (ModuleRef r, byte c);
7 // Extend/retract the connectors in set s at modules referred by r
8 extendSet (ModuleRef r, byte s);
9 retractSet (ModuleRef r, byte s);
```

Listing 5.14: Functions in MetaForma to control connectors

**Ending a sequence**  After the final sequential action in a sequence, by default the strategy is given back the control without disabling the meta-module it was executing in. In case the sequence is executed in a region [15], we can disable it by the `disable region` construct, i.e. splitting up the region back to meta-modules. In case the meta-module must be destroyed as well, it can be disabled by the `disable meta` construct, this resets all meta-module variables [14] and thus also the region.

In the LiftUp sequence, at the end the runtime system must make a distinction between the walker and lifting meta-modules. The walker must be maintained such that it walk further according to the strategy and the WalkStep sequence, thus only its region is disabled. The lifting meta-module, on the contrary, is not needed any more and must be disabled to reset its meta-module variables.

**5.5.6 WalkStep sequence**  The WalkStep sequence attains a single walking step by the walker. At the end of the sequence, it does not disable the meta-module so control is directly given back to the strategy. Instead, the runtime system is instructed to swap the `Walker.Left` and `Walker.Right` module ids, such that a new walking step can be made. The sequence is implemented in [Listing 5.15]
5. LANGUAGE DESIGN OF METAFORMA

Listing 5.13: MetaForma implementation of the LiftUp sequence

```c
int HALF = 180 // Constant declaration
int QUART = 90
int EIGHT = 45

sequence LiftUp {
    module.storeID();
    module.gradientInit();
    do {
        module.gradientPropagate();
    } until (config.propagationTime);
    do {
        if (gradPri == 2 && gradSec == 2) module.id = @Walker.Left;
        if (gradPri == 3 && gradSec == 1) module.id = @Walker.Middle;
        if (gradPri == 2 && gradSec == 0) module.id = @Lifter.Main;
        if (gradPri == 0 && gradSec == 0) module.id = @Lifter.Help;
    }
    do {
        disconnect(@Lifter.Main, @Lifter.Help);
        disconnect(@Lifter.Main, @Walker.Left);
    }
    rotate(@Walker.Right, QUART);
    rotate(@Lifter.Main, QUART);
    connect(@Walker.Left, @X, true); // Also connect to modules in other region
    disconnect(@Walker.Right, @Lifter);
    do {
        rotateOrigin(@Lifter.Main);
        rotateOrigin(@Walker.Right);
    }
    do {
        connect(@Lifter.Main, @Lifter.Help);
        rotate(@Walker.Middle, -EIGHT);
    }
    rotate(@Walker.Left, HALF);
    rotate(@Walker.Middle, EIGHT);
    if (module.metaID == meta.regionID) { // I am in the walker meta-module
        meta.type = STANDING;
        disable region; // The runtime system keeps the meta-module intact
    } else { // I am in the lifting meta-module
        module.restoreID();
        disable meta; // The runtime system disables the meta-module as well
    }
}
```
Cluster-Flow devised by Christensen et al.

1 sequence WalkStep {
2   connect (@Walker.Right, @X);
3   disconnect (@Walker.Left, @X, true); // Also disconnect to other regions
4   rotate (@Walker.Middle, -EIGHT);
5   rotate (@Walker.Right, -HALF);
6   rotate (@Walker.Middle, EIGHT);
7   if (module.id == @Walker.Left) module.id = @Walker.Right;
8   else if (module.id == @Walker.Right) module.id = @Walker.Left;
9 }

Listing 5.15: MetaForma implementation of the WalkStep sequence

5.5.7 LiftDown sequence

In this subsection, we present the sequence to re-merge the walker back in the scaffold. The sequence is implemented in Listing 5.17 and can roughly be defined in three parts: Absorbing the ‘lonely’ module at the front of the scaffold, the actual self-reconfiguration and finally the symmetry preservation.

Absorbing the lonely module

The possibility to lift meta-modules downwards such that they can merge back in the structure, requires a ‘lonely’ module at the front of the structure. This forces the structure not to be divided in equally-sized meta-modules of size three, as shown in Figure 5.5. Meta-modules need a means to absorb this lonely module, in order to be able to use the module in the execution of the sequence for lifting down.

Therefore, the first action in the sequence is the invocation of meta.absorb. This function sends Absorb packets to all its neighbors, also the neighbors outside the region. Importantly, it sets the meta.sizeExtra primitive variable to 1, as the size of the meta-module will be increased with the absorbed module.

1 void absorb() {
2   if (module.id == @Walker.Right)
3      send Absorb() to $(inRegion: false);
4   meta.sizeExtra = 1;
5 }

Listing 5.16: Broadcast an absorb packet

An event handler is used to process the received Absorb packet.

1 when (receive {Absorb p}) {
2   if (module.metaID == 0) {
3      module.metaID = p.metaID;
4      module.storeID();
5      module.id = @Lifter.Main;
6   }
7}

83
5. LANGUAGE DESIGN OF METAFORMA

The ‘lonely’ module at the front of the structure is the only module without a meta id, hence it adds itself to the meta-module by setting its meta id to the sender. Furthermore, it sets its module id to Lifter.Main, to be used later in the LiftDown sequence.

Self-reconfiguration

The self-reconfiguration part of the LiftDown sequence realizes the walker to merge back into the scaffold. The retractSet primitive is used to retract all female connectors on the north hemisphere of the Lifter. The Walker.Right module connects itself in the same action to that hemisphere, hence allowing it to rotate. In the LiftUp sequence this is done using a Lifter.Help module, but here we do not have a straightforward means available to identify this module.

Symmetry preservation

At the end of the execution of the sequence, we have to attain symmetry preservation\[^{10}\]. We do this by invoking the module.fixSymmetry function, and specify an expression that can identifies a module that has not been moved in the sequence. The runtime system will evaluate the expression on every module in the region. As the expression identifies the module that has not been moved, other modules can adjust their symmetry to it, restoring the global symmetry. The function is implemented in the runtime system\[^{18}\], and is described in Subsection 7.6.2.

After the symmetry is restored, we disable the meta-module. This resets the modules to their initial state, where it starts re-executing the Init sequence.

```
1 sequence LiftDown {
  2   do { meta.absorb(); } until(consensus());
  3   do {
  4     connect(@Walker.Right,@Lifter.Main);
  5     retractSet(@Lifter.Main, NORTH&FEMALE); // Also females can be used
  6   }
  7     rotate(@Lifter.Main, QUART);
  8     rotate(@Walker.Right, QUART);
  9     connect(@Walker.Left, @Lifter.Main);
 10     disconnect(@Walker.Right, @Lifter.Main);
 11   do {
 12     rotateOrigin(@Lifter.Main);
 13     rotateOrigin(@Walker.Right);
 14   }
 15   do {
 16     connect(@Walker.Right, @Lifter.Main);
 17     connect(@Lifter.Main, @X);
 18   } module.fixSymmetry ( module.metaID != meta.regionID && module.atBorder( BOTTOM));
 19   disable meta;
 20 }
```

Listing 5.17: Implementation of LiftDown sequence
This section presents part of the implementation of the cluster-flow algorithm devised by Brandt et al. First, we will discuss how the grid of meta-modules is created. Second, we present a high-level strategy to attain cluster-flow forwards.

5.6.1 Divide structure into meta-modules

First, to be able to work with meta-modules, the module structure needs to be divided into groups of four single modules. In Figure 4.11 is the grid shown that we need to create. First we describe the algorithm, followed by the implementation in MetaForma.

**Algorithm** An AssignMetaID packet is used to travel in a circle among a group of four modules. When the packet arrives back at the sender, the meta-module is created. Creating the grid of modules must start in a corner of two border of the structure and, after each new meta-module continue working towards the center, such that the lines of the grid are respected among the complete ensemble. Figure 5.8 shows how the packets travel, the numbers represent the counter variable in the packet and the letters represent the direction of how the groups are created.

**Implementation** We implement this algorithm in the Init sequence, shown in Listing 5.18. It uses three event handlers: The first event handler sets its meta id to its module number and sets its meta part to West, when at the left of it are either no modules or modules already packed in a meta-module.

The second event handler is the continuation for the first handler: We need to propagate packets continuously, but the condition in the first event handler is falsified when it is fired, as it changes the module.metaID variable.

The third handler processes the received AssignMetaID packets, and sets the meta id and appropriate meta part. It propagates the packet further by the primitive deflect function, which computes the deflected connector number, relatively to the incoming connector. When the counter equals four, the meta-module is successfully created.

**Conditions for Neighbor Operator** The following conditions can be used in both the Neighbor Count Operator () and Neighbor Operator (), to make a selection over specific neighbors.

```java
// Request the amount of neighbors matching the given ...
1 # (in: @Walker) //.. id or group name ("in:" can be omitted)
2 # (metaPart: Dummy) //.. meta-part
3 # (connected: true) //.. connection state to myself (false="floating")
4 # (inRegion: true) //.. regionID being equal to my own region
5 # (metaID: x) //.. metaID being equal to x
6 # (connNb: x) //.. the connector number of the neighbor matches x
7 # (connOwn: x) //.. the connector number of myself matches x
```

Listing 5.19: Neighbor conditions
sequence Init {
    do {
        when (module.metaID == 0 && #(connOwn: EAST\&MALE, metaPart: None) == 2 
            && #(connOwn: WEST, metaPart: None) == 0) {
            module.metaID = module.number;
            module.metaPart = West;
        }
        when (module.metaPart == West) {
            send AssignMetaID (newID = module.metaID, counter=1) to $(connOwn: EAST\&MALE\&NORTH);
        }
        when (receive (AssignMetaID p)) {
            if (module.metaID == 0) {
                module.metaID = p.newID;
                if (p.counter==1) module.metaPart = North;
                if (p.counter==2) module.metaPart = East;
                if (p.counter==3) module.metaPart = South;
            }
            if (p.counter==4) {
                enable meta;
                execute Strategy;
            }
            send AssignMetaID (newID=p.newID, counter=p.counter+1) to $(connOwn: EAST\&MALE\&NORTH);
        }
    } until(false);
}

Listing 5.18: The Init sequence: Divide structure in meta-modules of size 4

Figure 5.8: Steps of AssignMetaID packets that are sent in the Init sequence.
5.6.2 High-level Strategy for Cluster-flow

This section presents the high-level strategy\textsuperscript{13} to attain cluster flow in forward direction. The strategy consists of two sequential actions:

- As the strategy immediately gets control after the $\text{Init}$ sequence, the meta-neighbors, implemented as \textcolor{black}{meta-module variables}\textsuperscript{14}, are still empty. Therefore, we need to assign the meta-neighbors their appropriate values. We do this by invoking the \textcolor{black}{meta.broadcastNeighbors} function continuously for $\text{config.propagationTime}$. This makes sure that the \textcolor{black}{module neighbor table}\textsuperscript{9} will be filled and the meta-neighbor variables will be set to their value.

- Consecutively, now meta-modules have knowledge about their neighbors, they can make a decision whether to create a \textcolor{black}{region}\textsuperscript{15} with other meta-modules to execute a self-reconfiguration sequence in that region. Three rules are defined by the $\text{try region}$ construct. This construct corresponds to the strategy in \textcolor{black}{Figure 4.14}, as the \textcolor{black}{including} clause defines the neighbors that must be non-zero, and thus are verified to exist, and the \textcolor{black}{excluding} clause represents neighbors that must be zero, and thus are verified to be empty.

At this point, the global behavior is defined in terms of \textcolor{black}{meta-modules}\textsuperscript{7} and \textcolor{black}{regions}\textsuperscript{15}, in which a \textcolor{black}{sequence}\textsuperscript{3} with a corresponding \textcolor{black}{sequence orientation}\textsuperscript{12} is assigned. The following twelve sequence orientations can be used to parametrize sequences:

- LEFT\_BOTTOM
- RIGHT\_BOTTOM
- BOTTOM\_LEFT
- TOP\_LEFT
- LEFT\_TOP
- RIGHT\_TOP
- BOTTOM\_RIGHT
- TOP\_RIGHT
- LEFT
- RIGHT
- BOTTOM
- TOP

\textbf{Figure 5.9:} List of sequence orientations

The orientation assigned to a sequence is accessible by the \textcolor{black}{orient} operator. In combination with conditions, the behavior of the sequence can be generic, as we saw at obstacle-avoidance behavior in \textcolor{black}{Listing 5.4} and \textcolor{black}{Listing 5.7}. But sequence orientations are even more powerful for self-reconfiguration sequences.

If we use an orientation with an underscore in it, it is automatically separated between a primary and secondary orientation, and can be accessed by the \textcolor{black}{orient.primary} and \textcolor{black}{orient.secondary} variables. When implementing \textcolor{black}{gradients}\textsuperscript{16}, instead of relating the start of the gradient at modules at a specific border (e.g. bottom or left), we can relate the start at modules at the border of the primary or secondary orientation. In \textcolor{black}{Listing 5.11} is the gradient implementation shown that uses the orientation of a sequence.

5.6.3 Detecting meta-neighbors

This subsection implements the meta-neighbors system. In \textcolor{black}{Listing 5.21} we have implemented the used \textcolor{black}{meta parts}\textsuperscript{11} and \textcolor{black}{meta-module variables}\textsuperscript{14}.
Collecting direct neighbors

In addition, we declared the `neighborHook` function, to automatically detect direct meta-neighbors such as `Top`, `Bottom`, `Left` and `Right` when a packet from them is received. It assigns the meta id of direct meta-neighbors to their appropriate meta-module variables. However, indirect neighbors, such as `TopLeft`, `BottomLeft`, `TopRight` and `BottomRight` are also needed by the strategy, but as we cannot communicate directly with them, we share them via-via.

Sharing diagonal indirect neighbors

The `meta.broadcastNeighbors` function sends `AddNeighbor` packets to all direct meta-neighbors in two phases. It sends to top and bottom the values of `Left` and `Right`, and to left and right the values of `Top` and `Bottom`.

When an `AddNeighbor` packet is received, the runtime system checks to which neighbor it belongs using its own stored meta-neighbors. For example, meta-module `C` in Figure 5.10 has a neighbor at the top, left and bottom. Meta-module `D` below `C` has no direct way of knowing about the existence of `B`. But `C` sends the meta id of `B` to `D`, and as `D` knows it is at the top of `C`, it sets the meta id of `B` to `TopLeft`.

Having only knowledge about direct meta-module neighbors is not sufficient. In order to make a decision for a meta-module to create a region with other meta-modules, it also needs knowledge about its diagonal neighbors. These values are represented by `TopLeft`, `TopRight`, `BottomLeft` and `BottomRight`.

---

**Listing 5.20**: Strategy to move an ensemble in forward direction, based on Figure 4.14

```java
sequence Strategy {
    when (receive (Packet p)) {
        meta.neighborHook (p);
    }
    do {
        meta.broadcastNeighbors();
    } until (config.propagationTime);
    do { // Move in forward direction
        try region FlipThrough(LEFT.BOTTOM) including Right excluding Top, TopRight; // B
        try region FlipAlong(RIGHT.BOTTOM) including Right, TopRight excluding Top;
        try region FlipThrough(BOTTOM.RIGHT) including Top excluding TopLeft, Left, Right,
            Bottom, BottomLeft, BottomRight; // E
    } until (false);
}
```

**Figure 5.10**: Five meta-modules moving forward
Cluster-Flow devised by Brandt et al.

```c
meta {
  parts [North, West, South, East];
  byte Top, Bottom, Left, Right, TopLeft, TopRight, BottomLeft, BottomRight;

  void neighborHook (Packet p) {
    if (p.metaID != module.metaID && module.metaID != 0) {
      if (p.connNb==5 || p.connNb==6) meta.Right = p.metaID;
      if (p.connNb==2 || p.connNb==7) meta.Top = p.metaID;
      if (p.connNb==0 || p.connNb==3) meta.Left = p.metaID;
      if (p.connNb==1 || p.connNb==4) meta.Bottom = p.metaID;
    }
  }

  void broadcastNeighbors () {
    send AddNeighbor(first=Left, second=Right) to $(metaID: [Top, Bottom ], inRegion:false);
    send AddNeighbor(first=Top, second=Bottom) to $(metaID: [Left, Right ], inRegion:false);
  }
}

when (receive (AddNeighbor p)) {
  if (p.metaID == meta.Left) {
    meta.TopLeft = p.first; meta.BottomLeft = p.second;
  }
  if (p.metaID == meta.Right) {
    meta.TopRight = p.first; meta.BottomRight = p.second;
  }
  if (p.metaID == meta.Top) {
    meta.TopLeft = p.first; meta.TopRight = p.second;
  }
  if (p.metaID == meta.Bottom) {
    meta.BottomLeft = p.first; meta.BottomRight = p.second;
  }
}
```

Listing 5.21: Fill neighborhood of meta neighbors

### 5.6.4 Flip-Through meta action

This subsection presents the MetaForma implementation of the Flip-Through meta-action. The MetaForma implementation is shown in [Listing 5.22](#). As we saw in [Figure 4.13](#), eight different sequence orientations are possible for the Flip-Through meta-action. In order to not need to implement all eight orientations in eight different sequence implementations, we implement a generic sequence of the Flip-Through to be used on all eight orientations. For each orientation, this has the following impact on the implementation of the sequence:

- Module ids must be assigned to different modules: This is solved by using a gradient that is relative to the sequence orientation. In [Figure 5.11](#) is shown how the module ids relate to two different orientations.
- Modules must use different connectors to connect/disconnect to their neighbors during
the self-reconfiguration. The adjacency of male and female connectors can also be different, meaning that it is not pre-determined which module must execute a connect or disconnect action. This is solved by the MetaForma runtime system: It uses an on-line algorithm for computing the correct connector to each neighbor. The module neighbor table\(^9\) stores the appropriate connector number for each neighbor.

- In the sequence implementation, we define whether a module’s actuator should rotate 90 or 180 degrees for actuation. In order to respect the orientation, modules must use either a positive or negative degree value. Therefore, we define two module variables QUART and HALF, and by using a check condition on the current sequence orientation, we either assign a positive or negative value to it. Later in the sequence we use these variables in the hardware control primitives\(^6\) to attain actuation in the correct direction.

Hence, as a result of a sequence orientation\(^12\), sequences can be generalized to higher-level behavior; it allows the region to execute the same sequence in another direction, without the need to write a new sequence.

For the complete MetaForma implementation of the meta-module approach devised by Brandt [16], we refer the reader to Section A.1.

### 5.7 Summary

MetaForma is an external DSL comprising a high-level Strategy Language to implement global behavior, and a Motion and Coordination language to implement low-level control.

First, we argued for the MetaForma language using an external DSL approach.

Second, we described the syntax and semantics of MetaForma by using the following implemented examples:

- Obstacle-avoidance on a Two-Wheeler and Four-Wheeler car
- Meta-module cluster-flow devised by Brandt et al. [16]
- Meta-module cluster-flow devised by Christensen et al. [29]
\begin{verbatim}
if (BOTTOM_RIGHT, RIGHT_TOP, LEFT_BOTTOM, TOP_LEFT) {
  QUART = 90;  HALF = 180;
} else {
  QUART = -90;  HALF = -180;
}
module.gradientInit();
do { module.gradientPropagate(); } until (config.propagationTime);
do {
  if (gradPri==1 && gradSec==1) module.id = @Lifter.Right;
  if (gradPri==2 && gradSec==0) module.id = @Lifter.Left;
  if (gradPri==1 && gradSec==0) module.id = @Dummy.Left;
  if (gradPri==0 && gradSec==0) module.id = @Lifter.Top;
  if (gradPri==2 && gradSec==1) module.id = @Dummy.Right;
  if (gradPri==3 && gradSec==1) module.id = @Lifter.Bottom;
  isRef = gradPri==3 && gradSec==0;
}
do {
  disconnect (@Lifter.Right, @Dummy.Left);
  disconnect (@Lifter.Left, @Dummy.Right);
}
do {
  rotate (@Lifter.Right, QUART);
  rotate (@Lifter.Left, QUART);
} disconnect (@Lifter.Right, @Dummy.Right);
rotate (@Lifter.Bottom, HALF);
do {
  rotate (@Lifter.Right, QUART);
  rotate (@Lifter.Left, QUART);
} rotate (@Lifter.Top, HALF);
module.restoreID();
connect (@X, @X);
module.fixSymmetry(isRef);
disable meta;
\end{verbatim}

Listing 5.22: MetaForma implementation of the Flip-Through Sequence
Chapter 6

MetaForma Language Implementation

This chapter describes how the MetaForma compiler and its IDE are implemented. It is implemented in the language workbench Spoofax, as Spoofax offers the desired features such as advanced editor services, and the author was experienced to work with this language workbench.

6.1 The Spoofax Language Workbench

Spoofax/IMP [41] is a language workbench that is integrated as a plugin into Eclipse. Spoofax/IMP incorporates SDF [37] to specify the syntax of a language, and Stratego [75], a declarative language for program transformation, into IMP [22]. IMP is a meta-tooling platform in Eclipse for creating language-specific IDEs.

6.1.1 Syntax Definition using SDF

The Syntax Definition Formalism (SDF) is a formalism for syntax definition covering both the lexical and abstract-syntax aspect. In addition, it offers a standard interface between lexical and context-free syntax. Furthermore, SDF has a powerful means to disambiguate between ambiguous parse trees and it uses an efficient incremental implementation which accepts arbitrary context-free syntax definitions [37].

Spoofax generates a scanner-less generalized LR (SGLR) parser from the grammar defined in SDF. SGLR parsing can be applied on the complete set of context-free grammars. In contrast, when using LR(x) and LL(x), and therefore also SLR and LALR(1) parsers, shift-reduce conflicts can occur when parsing a program in the context-free grammar family. Hence, these parsers are not powerful enough to be used for the entire collection of context-free grammars.

6.1.2 Program Transformation using Stratego

Stratego [75] is a transformation system aiming to support a wide range of program transformation. It uses programmable rewriting strategies that operate on a tree structure of ATerms [14] (Annotated Terms). A Java implementation of Stratego is integrated in Spoofax.
6. **MetaForma Language Implementation**

Fundamentally, Stratego supports three approaches for code generation:

- **Direct string generation** for the target language by using string interpolation. The benefits are as follows: High-performance, easy to implement. Drawbacks: No syntactical check for target program, and reduced possibility to format layout in target code.

- **Model-2-model transformation** in which an AST model for the source language is rewritten to an AST model for the target language, and then execute a pretty printer at the result to generate the target program. Benefits: Target code will be properly formatted, but abstract syntax (used to build the target model) in the rewrite rules can be very long-winded.

- **Direct string generation by using Concrete Syntax** for the target language. This is somewhat equal to AST model conversion, with the difference that concrete syntax is used instead of abstract syntax. The benefit is that concrete syntax is much more concise and a syntactical check can be made, serving as an extra check for syntax errors in the target program. The drawback is that is more complicated to use, e.g. the target language grammar must be combined with the Stratego grammar.

The MetaForma compiler uses the first approach: **Direct string generation**, as it can be implemented quickly.

### 6.1.3 Name Binding using NBL

The Name Binding Language [44] (NBL) is an external DSL incorporated in Spoofax, facilitating **name binding**, which is the problem of connecting references to their appropriate declarations. NBL is a declarative language that is compiled to Stratego code.

### 6.1.4 Editor Services

Spoofax provides a means to implement editor services that can assist the programmer in writing code, which is usually common functionality in IDEs for GPLs like Java and C++. Editor services can be distinguished between syntactic services and semantic services. The former are static and work on syntax level, while the latter are dynamic and work on the ‘meaning’ of the program.

**Syntactic Editor Services**

**Syntax checking** is performed by the parser; if during parsing at a certain point no production rule in the grammar can be applied, this part is skipped\(^1\) in the program and a syntax error is shown in the editor. **Syntax highlighting** is the relation of a specific color to a specific type of literals and lexical syntax. Spoofax has a default colorization generated from literals and

\(^1\) Skipping allows the parser to continue instead of stopping at the first syntax error, hence it will be able to show multiple syntax errors instead of only the first one found [I p73/p74].
lexical syntax in the grammar. It can be extended with custom colors for lexical syntax and AST constructors.

Code folding and the Outline view are specified by selecting grammar productions to be foldable or shown in the outline view window of Eclipse. By default, Spoofax uses heuristics to automatically derive a generated folding descriptor from the SDF grammar; this can be overridden by the user. Automatic bracket insertion, bracket matching and automatic bracket indentation can be activated inside an editor by specifying pairs of brackets in the implementation.

Syntax completion can be triggered in the editor by pressing \texttt{ctrl+space} while typing, the editor will a list with matching language constructs to complete. \cite{[41]}

**Semantic Editor Services**

Reference resolving enables the cursor to jump to the position of the declaration in the editor using \texttt{ctrl+click}, e.g. when done at a variable reference, the cursor will jump to the place where the variable is declared.

Hover help shows a tool-tip when hovering over a reference, showing information about the declaration it refers to.

Type checking makes sure that the program respects the type system of its language. In case a type mismatch is detected, an error is shown.

Content completion is presented to the programmer in the same list as syntax completion, and when pressing \texttt{ctrl+space} while typing, it will show identifier proposals that match in the current scope. For example, if a user types the word ‘ret’ and in the same scope an integer named ‘returnVal’ is defined, it displays this to the user.
6. MetaForma Language Implementation

6.2 MetaForma Implementation in Spoofax

The MetaForma compiler uses four steps to compile a MetaForma program into its corresponding Java controller. The first phase is the Parse phase, in which the program is converted to a tree data structure to enable traversal, called an Abstract Syntax Tree (AST). Second, the desugar phase removes syntactic sugar from the AST and rewrites it to a more generic format. These two steps are executed individually for each MetaForma source file.

Third, the analyze step is executed using the Name Binding Language (NBL), which tries to link all references that are used in a MetaForma program, such as variables, to their appropriate declaration. This step works together with the index, which is a database in which all previously parsed/desugared files are stored, as a variable reference may point to a declared variable in another file.

Fourth, the generate phase retrieves all declarations from the index using the package name of the MetaForma source file that the compiler is invoked on. The Strategy sequence is also lifted from this source file, as this sequence is not stored in the index. This allows the programmer to write a package in MetaForma code containing lower-level implementation, such as self-reconfiguration sequences and functions, eventual divided among multiple files. Multiple strategy sequences that implement the high-level behavior can use the same package, and if we want to execute a strategy we only need to invoke compilation from the MetaForma source file that it is implemented in. In Figure 6.2 is shown how the different phases relate to each other.

6.2.1 Parsing step

The MetaForma grammar, that specifies the MetaForma language syntax, is implemented in SDF. In Listing 6.1 is shown a Backus–Naur Form (BNF) of the grammar, to enable a global understanding of the syntax. For simplicity, disambiguation rules and list separators are omitted.
Also, so-called constructor names, which are included in SDF to convert the concrete-syntax tree (CST) to an AST, are not visible in BNF.

```
1 PROGRAM ::= DEBUG? MAINDEF*
2 DEBUG ::= debug {SHOW? VISUALIZE?}
3 SHOW ::= show [ID*]
4 VISUALIZE ::= visualize { north [VIS*] south [VIS*] }
5 VIS ::= MODREF : COLOR; | ID : COLOR;
6 MODREF ::= @ ID | @ ID . ID
7 COLOR ::= black|white|red|orange|yellow|green|cyan|blue|purple|brown|
8 MAINDEF ::= sequence ID {EVENT* ACTION*} | packet (byte ID*) | TYPE ID = EXP
9 MAINDEF ::= module (BRDRS? FIELD* FUNC*) | meta (PARTS? FIELD* FUNC*)
10 PARTS ::= parts [ ID+ ]
11 BRDRS ::= border [BORDER+]
12 BORDER ::= ID : EXP;
13 FUNC ::= TYPE ID (PARAM*) {VAR* STMT*}
14 PARAM ::= TYPE ID
15 TYPE ::= void | byte | int | boolean | float
16 EVENT ::= when(receive(PACKET ID)) {VAR* STMT*} | when(EXP) {VAR* STMT*}
17 ACTION ::= do PRECON? (EVENT* VAR* STMT*) POSTCON?
18 PRECON ::= wait(EXP)
19 POSTCON ::= wait(PRECON) until(POST)
20 POST ::= consensus() | EXP
21 STMT ::= enable meta | disable meta | disable region
22 STMT ::= send ID (ASSIGN*) to NBS_OP
23 STMT ::= if (EXP) (STMT*) else (STMT*) | if (EXP) (STMT*)
24 STMT ::= execute SEQ_REF;
25 STMT ::= try region SEQ_REF including ID+ EXCL?
26 EXCL ::= excluding ID+
27 SEQ_REF ::= TOP|LEFT|BOTTOM|RIGHT|TOP_LEFT|TOP_RIGHT|BOTTOM_LEFT|
|BOTTOM_RIGHT|RIGHT_TOP|RIGHT_BOTTOM|LEFT_TOP|LEFT_BOTTOM;
28 OBJ ::= meta. | module. | ε
29 FUNCALL ::= OBJ? ID (EXP*)
30 FUNCALL ::= ACTUATION (EXP*)
31 ACTUATION ::= connect | disconnect | extend | extendSet | retract |
| retractSet | rotate | rotateOrigin | rotating | rotateAbsolute | stop
32 VARREF ::= OBJ? ID
33 EXP ::= VARREF | NBS_COUNT | FUNCALL | EXP OP EXP | CONST
34 EXP ::= module.id | module.metaID | meta.regionID
35 CONST ::= NORTH | SOUTH | WEST | EAST | MALE | FEMALE | MAX_BYTE
36 OP ::= * / & | && ||
37 NBS_COUNT ::= #(NBS_SPEC ID*)
38 NBS_OP ::= $(NBS_SPEC ID*)
39 NBS_SPEC ::= metaPart | connected | inRegion | metaID | connNb | connOwn
```

Listing 6.1: Simplified MetaForma grammar definition in BNF, list separators like comma’s have been omitted for simplicity.
6. MetaForma Language Implementation

parsed. Two facets have been highlighted, to save space, as an AST takes more space than the original program. We see that the Strategy sequence has two actions, an assignment and a do until();

### Listing 6.2: MetaForma Demo Program

```plaintext
package P
module {
  byte val, val2;
}
sequence Strategy {
  module.val = module.number;
  do {
    when (receive (Spread p)) {
      val = max(p.val, val);
    }
  }
  send Spread(val=val);
} until(config.propagationTime);
```

### Listing 6.3: Fragment of Parsed AST

```plaintext
Module( None(), [FieldDeclList(Byte(), ["val", "val2"])], [])
```

6.2.2 Desugar step

The desugar step is implemented using declarative Stratego rules. A Stratego rule has a name, a precondition and a rewrite action. Rules are applied to ATerms in the AST. If the precondition matches on a specific ATerm, the rewrite rule is executed and the ATerm is replaced in the tree. In order to match a rewrite rule to the appropriate ATerms in an AST, a strategy is used.

#### Desugaring Class Declarations

In Listing 6.3 we see that the module class is parsed to the ATerm highlighted in yellow. In order to rewrite the Module class and Meta class to a uniform format, we define a rewrite rule `desugar`, shown in Listing 6.4. This rule desugars a type into ‘single type, multiple fields’ declarations. Later, we will traverse the entire AST and try to match the `desugar` rule to every ATerm.

### Listing 6.4: Desugaring Rules for Rewriting the Module Class and its Fields

```plaintext
// Desugar Module Class, for the Meta class the rule is omitted
desugar: Module(bor*, var*, fun*) -> Class(Module(), bor*, decl*)
where decl* := <unify>(var*, fun*)
```

### Desugaring Actions

As the MetaForma syntax specifies that each sequential action can have an optional precondition and postcondition, they are either parsed as `None()` when omitted or as `Some(..)` when included.
An omitted precondition means that the statements in the action can be executed immediately when entering its corresponding state. An omitted postcondition means that we want to wait for the entire region or meta-module to be completed with the current action before proceeding to the next. In order to unify the format of an action, the precondition must be either `None()` or `Wait(..)`, and the postcondition must be either `Wait(..)` or `Until(..)`, and `Wait(Consensus())` when omitted.

The actions in a sequence need to be distinguished using an ordinal measuring scale, such that they can be related to the sequence counter. This is solved by placing a successive number to each successive action, starting at zero in a sequence. In order to accomplish this, we define a second rewrite rule `desugar` that can rewrite a sequence with to a sequence with

```plaintext
1 desugar: Sequence{n, evts, acs*} -> Sequence<wrap>n, evts, acs4*
2 where
3   acs2* := <conc> {acs*, [Execute(SequencePrimRef("Strategy"))]};
4   acs3* := <map(try(add-action);try(uni-action))>acs2*;
5   nrs := <upto><dec><length>acs3*; // nrs = [0,1,2,...,n-1]
6   acs4* := <map(swap-counter)>zip>(nrs, acs3*) // number each action
7
8   add-action: term -> Action(None(), [], [], [term], None())
9   where not (<eq>(<aterm-name>term, "Action"))
10
11   uni-action: Action(pre, evt*, var*, acs*, post) -> Action(pre', evt*, var*, acs*, post')
12   where
13     pre' := <try(\$Some(x)->x\)>pre;
14     post' := <try(\$Some(x)->x \+\ None())->Wait(Consensus()) \)}>post
15
16   swap-counter: (i, Action(precon, evt*, var*, action*, postcon)) -> Action{
17     i, precon, evt*, <conc>(var*, action*), postcon
18   }
19
20   wrap = switch id
21     case ?"Init": !Init("Init")
22     case ?"Strategy": !Strategy("Strategy")
23     otherwise: !Plain(<id>)
24   end
```

Listing 6.5: Desugaring rules for rewriting actions in a sequence

In order to apply the `desugar` rules defined above, we need to apply them to their corresponding terms\(^2\). As the precondition of each rule specifies which ATerms we want to rewrite with that particular rule, we can just execute all defined `desugar` rules on all terms in the AST.

We do this by defining a strategy `desugar-all` that will be applied to the complete AST of a parsed MetaForma source file. We use a so-called `topdown` strategy, meaning that we start at the top of the AST (the `Package(..)` term) and end at the leaves of the AST. We use a `try` strategy in the `topdown`, as we want to keep traversing the AST when a rewrite rule can not be matched. Hence, the main strategy becomes:

\[
desugar-all = topdown(try(desugar))
\]

\(^2\)The MetaForma compiler contains around 30 desugar rules, for simplicity only the two main rules are shown. For the complete set we refer to GitHub [73].
Invoking this method on the complete AST, will traverse the tree from the root to the leaves, and will try to rewrite every term it visits. Listing 6.6 shows the result of desugaring the AST of the program in Listing 6.2 limited to the parsed fragments shown in Listing 6.3.

<table>
<thead>
<tr>
<th>Class()</th>
<th>Module(), None()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[FieldDecl(Byte(), &quot;val&quot;), FieldDecl(Byte(), &quot;val2&quot;)]]</td>
</tr>
<tr>
<td>Action(0, None(), [], [Assign(Ref(&quot;val&quot;), FieldPrimRef(Module(), &quot;number&quot;))])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait(Consensus())</td>
</tr>
</tbody>
</table>

Listing 6.6: Fragment of Desugared AST

<table>
<thead>
<tr>
<th>Class()</th>
<th>Module(){{Class(),Module(),&quot;P&quot;}}, None(),</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[FieldDecl(Byte(), &quot;val&quot;{[Field(), &quot;val&quot;, Module(), &quot;P&quot;]}), FieldDecl(Byte(), &quot;val2&quot;{[Field(), &quot;val2&quot;, Module(), &quot;P&quot;]})]</td>
</tr>
<tr>
<td>Action(0{[Action(), 0,&quot;Strategy&quot;,&quot;P&quot;]})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>, None(), []</td>
</tr>
<tr>
<td></td>
<td>,Assign(Ref(&quot;val&quot;{[Field(), &quot;val&quot;, Module(), &quot;P&quot;]}))</td>
</tr>
</tbody>
</table>
|         | ,FieldPrimRef(
|         | ,Module(), "number"{})) |
|         | ,Wait(Consensus()) |

Listing 6.7: Fragment of Analyzed AST augmented with URIs

### 6.2.3 Analyze step

The task of the analyze step is to link references to their appropriate declarations, respecting scope rules. Scopes are defined in NBL using namespaces and `declaration` rules. A namespace is a name that represents a sub-tree of the AST. These sub-trees are defined by declare rules, that match on the specified ATerm of a declaration.

After we have defined declarations and their scope, we use `reference` rules to link references to their appropriate declaration. In Figure 6.3 is shown which scopes are used in MetaForma. Each scope is visualized as a box. The `Package` namespace scopes all other namespaces, meaning we can e.g. use two sequences with the same name in different packages independently from each other. When we use two sequences with the same name in one single package, an error is generated as sequences must be unique in the `Package` scope.

![Figure 6.3: Diagram showing nested scopes in MetaForma. Each rectangle represents a namespace, a nested rectangle is scoped by its surrounding (transitively)](image)

100
Binding object references to their appropriate class

Declare rules are used to identify the declarations in the AST. In Listing 6.8 is a fragment shown of the declaration rules used in MetaForma. We want to scope fields and methods inside a class in order to make them unique inside a class using their name, but to allow declarations with the same name in another class. We do this by specifying an NBL rule for the Class \((n, _, _)\) ATerm, that matches both the module and meta class as we desugared them using the rule in Listing 6.4.

The first rule links the Module() term, which is the result from parsing the ‘module’ keyword, to the declaration of the module class.

<table>
<thead>
<tr>
<th>Class ({n, _, _, _}):</th>
</tr>
</thead>
<tbody>
<tr>
<td>// n ∈ {Module(), Meta()}</td>
</tr>
<tr>
<td>defines Class n of type n</td>
</tr>
<tr>
<td>scopes Field, Method</td>
</tr>
<tr>
<td>Action ({i, _, _, _, _}):</td>
</tr>
<tr>
<td>defines Action i</td>
</tr>
<tr>
<td>scopes EventHandler, Var</td>
</tr>
</tbody>
</table>

Listing 6.8: Fragment of Name Binding declaration rules

<table>
<thead>
<tr>
<th>Module(): refers to Class Module()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta(): refers to Class Meta()</td>
</tr>
<tr>
<td>Ref(n):</td>
</tr>
<tr>
<td>// Actions have no references!</td>
</tr>
</tbody>
</table>

Listing 6.9: Fragment of Name Binding reference rules

The Stratego rules that are generated from the NBL rules will augment all declaration ATerms with a URI, a Unique Resource Identifier, which is stored as a list in the augmentation. Listing 6.7 shows these URIs compared with the initial AST shown in Listing 6.6.

Scoping Events and Variables in Actions

Reference rules are used to link references to the corresponding declaration. In Listing 6.9 is a fragment shown of the reference rules that are used.

The first rule defines the declaration for an action. This rule adds a scope on event handlers and variables, as each action can have its own variable declarations. Also, event handlers that are included inside an action, must be scoped by it such that we can distinguish between event handlers in the global, sequence and action namespace. For the complete list of NBL rules, the reader is referred to GitHub [73].

The second rule binds Ref constructs to a declaration. However, there is a catch here, as a Ref can refer to a declaration in multiple namespaces. Indeed, a Ref is simply an identifier, and the parser cannot distinguish between a reference to a Var or a Field, as they are syntactically the same. This works the same in Java when using an identifier inside a method. It is first matched against to a local variable, and if not found to a class field variable. In MetaForma, a Ref is first matched to a Var, if not found to a Field, consecutively to a MetaPart and finally to a Constant. This is implemented in NBL by using multiple refers to clauses, as shown in Listing 6.9.
6. MetaForma Language Implementation

6.2.4 Generate step

All Java code inside the MetaForma.gen package is generated using the MetaForma compiler. For each DSL program, a single Java file is used to generate code and is placed in a MetaForma.gen.<Program> package. The MetaForma compiler uses Java as a backend. This is shown in Figure 6.4.

The following classes are generated:

- A Controller class, with the name prefix of the compiled MetaForma file. that contains an init method, which is invoked by the runtime system to initialize the controller. It contains the handleStates method that is invoked by the runtime system in the main control loop. In this method is the implementation generated for the strategy, and all sequences and their corresponding actions.

- A GenModule and GenMeta class, which are the compilation targets of the module and meta classes in the MetaForma language. In both classes can methods and fields be defined, which are directly compiled to its corresponding Java class.

- A number of packet classes: Each packet that is defined in the DSL program is compiled to its own Java class that represents the packet. Payload variables in a packet are generated as fields in the class.

- For all sequences, groups, modules and meta-parts that are defined in the DSL program are four Java enums generated, containing all corresponding elements for each type.

Generated init method

The init function of the generated controller is automatically invoked by the runtime system when the controller is activated. Fundamentally, its goal is to instantiate the generated module and meta class in the generated controller, and to initialize debug information.

Generated handleStates method

The handleStates method is invoked by the runtime system in the main control loop, to execute the current sequence. It uses a state manager to link sequences and their sequential
actions to the current state in the runtime system. In [Listing 6.10] is shown a the generated handleStates() method from the MetaForma code in [Listing 6.2]. It contains generated code for the variable assignment and the sending of the Spread packet. Note that the event handler[4] is not included in the handlestates() method, as the runtime system cannot receive packets in the main control loop.

```java
1 public void handleStates() {
2     if (mfStateMngr.at(SequenceGeneric.Strategy)) {
3         if (mfStateMngr.doWait(0)) { // val = module.number;
4             module().setVar("val",module().getNumber());
5             mfStateMngr.postState();
6         }
7         if (mfStateMngr.doUntil(1)) {
8             // do {send Spread(val=val);} until( config.propagationTime);
9             if (ctrl.freqLimit("Spread0",mfConfig.getPropagationRate())) {
10                ctrl.send((PacketSpread)
11                    new PacketSpread(ctrl).setVar("val",
12                        module().val),ctrl.nbs().nbsFilterInRegion(true));
13                mfStateMngr.spend(ctrl.config().getPropagationTime());
14         }
15     }
16 }
```

Listing 6.10: Generated handleStates method

Compilation of hardware control primitives

Below are three MetaForma statements using hardware control primitives[6] shown [Listing 5.14] and [Listing 5.5].

1 retractSet(@Lifter.Main, NORTH&FEMALE);
2 connect(@X, @X);
3 rotate(@Lifter.Right, QUART);

These are (as separated examples each) compiled to method invocations of the MfActuation class (Section 7.4), as shown below.

1 ctrl.getActuation().retractSet(Mod.Lifter_Main,(MfController.NORTH &
2     MfController.FEMALE));
3 ctrl.getActuation().connect(Group.X,Group.X);
3 ctrl.getActuation().rotate(Mod.Uplifter_Right,module().QUART);

Compilation of operators

Below is a sample of the Neighbor Count Operator shown. The expression returns the amount of neighbors that is situated at the WEST connector set[2], and not having an assigned meta part[11].

1 #(connOwn: WEST, metaPart:None)
All conditions in the Neighbor Count Operator and Neighbor Table Operator are conjuncted, and are compiled to a fluent interface on the MfContext. Compilation results in the following Java code:

```java
ctrl.nbs().nbsFilterInRegion(true).nbsFilterConnOwn(MfController.WEST).nbsFilterMetaPart(MetaPart.NONE).size()
```

The `ctrl.nbs()` invocation in the runtime system returns the complete module neighbor table. Afterwards, each condition in the operator (Listing 5.19) is prefixed with `nbsFilter..`, and compiled to a separate method call, forming a chain. The `inRegion` by default is true, meaning that by default the operator only takes neighbors in the same region into account. Note the `.size()` invocation at the end of the method chain, returning the amount of neighbors subject to specified conditions.

**Generated Event Handlers for receiving packets**

The runtime system automatically invokes the `receivePacket` method when a packet is received from a neighbor, with the de-serialized packet as object parameter. Therefore, for every packet declaration one corresponding `receivePacket` method is generated. Event handlers for receiving packets are scope sensitive, i.e. event handlers can be included in the `Package`, `Sequence` or `Action` scope.

Below is shown the generated Java receive method for the MetaForma event handler as shown in Listing 6.2. The compiler automatically generates checks such that the code in an event handler is later on by the runtime system only executed when the current state matches the sequence in which the event handler was included.

```java
public boolean receivePacket(PacketSpread p) {
  boolean handled = false;
  // Only process packet when in appropriate state, otherwise ignore
  // when (receive (Spread p)) {val = max(p.val, val);}
  if (mfStateMngr.check(p,new State(SequenceGeneric.Strategy,1))) {
    module().setVar("val", max(p.val, module().val));
    handled = true;
  }
  return handled;
}
```

Listing 6.11: Generated method for an event handler

**Generated enums**

Modules can be referred to by using their unique `module identification` or by using their `module group` name. This is facilitated using the Mod enum and the Group enum, respectively. Both enums are generated by the MetaForma compiler and implement the IModuleRef interface.

The Sequence enum contains the list of sequences that is used in the DSL program, except for the Init and Strategy sequence, as they are included in the SequenceGeneric enum in the runtime system. The MetaPart enum contains the list of meta parts that is defined in the DSL program (Listing 5.21).
6.3 Editor Services

Colors are defined for the syntax of the language by assigning a color to a sort. Sorts are defined in the SDF grammar and can be lexical or context-free, in the former case it is a fixed string e.g., representing a single keyword, in the latter case it is a complete reduction rule. In Figure 6.5 is a small fragment of a MetaForma program shown with different colors. The SysObj sort represents the module, meta and region objects, which are colored in darkblue. A ModHolderRef, which can refer to a single module or a module group, is colored in darkyellow. Finally, primitive fields are colored in darkgreen.

```
module.gradientInit();
module.gradientPropagate();
do { 
  if (module.gradPri == 0 && module.gradSec < 3 ||
      module.group = @outside; 
} 
if (module.gradPri > 0 && module.gradSec > 0) { 
  module.group = @Inside; 
}
```

Figure 6.5: Different syntax colors

Content Completion

Syntax completion works using static, syntactic templates to ‘complete’ static language constructs. Completion rules are composed of static strings denoting a language construct and place-holder expressions, that denote values the programmer has to fill in manually. It is implemented by writing static rules. Each rule starts with a sort, which refers to a grammar reduction rule in the SDF implementation in . After the sort, a textual template is written, encapsulated by quotes, and variables that the user must write manually are within brackets. The construct (cursor) denotes the position of the cursor in the template when completed, and the (blank) construct is an extra condition that the template may only be triggered at a blank line.

Below are two of the complete rules shown. The first rule shows a proposal for “package” when the programmer triggers code completion in an empty file. The sort Start is used as the “package” clause is only used as the first non-empty line in a file.

The second rule shows a proposal for “sequence” and uses the sort MainDef such that the template will only be triggered at a place where the parser expects a MainDef declaration.

```
1 completion template Start : "program " <p> "\n\n" (cursor) (blank)
2 completion template MainDef : "sequence " <s> " { (cursor) "\n\n"
```

When implementing, it appeared that in Spoofax 1.0.9.0 unstable the sort is not always taken into account correctly: inside a MainDef, other MainDef templates are proposed, while the SDF grammar does not allow them to be nested. In addition, when the cursor is placed at the end of a MetaForma source file, no proposals are given, while it should propose MainDef declarations.
An algorithm for content completion is generated by NBL. As scoping is defined by using NBL rules, and every declaration is suffixed with a URI, the AST is searched for declarations with the same URI as the URI of the declaration the cursor is currently in.

Outline View

The Outline view is specified by a list of Sorts and Constructors that should be showed in the , but it cannot be specified which string is shown per item. It appears that for the Debug constructor "Debug" is shown in the outline, which is correct. But for the Meta constructor, instead of showing "Meta" it shows the name of the first field variable (so that value is shown twice), this is probably due to a bug in the Spoofax Unstable.

Hover Help and Reference Resolving

Hover help is implemented by using the index library. When moving the cursor on a declaration, the editor-hover function is invoked with as a parameter the current term and its URI. This URI is converted to a string by first retrieving the type from the index library and second concatenating the URI, from which we first take the tail and then the reverse. The following URI [Field,"Right",Meta,"Brandt"] is converted to the hover help string in Figure 6.6.

An implementation for reference resolving is automatically generated by NBL.

![Figure 6.6: The MetaForma IDE showing (1) Reference resolving and (2) Hover help](image-url)
Chapter 7

The MetaForma Runtime System

This chapter describes how the MetaForma runtime system is implemented. The runtime system bridges the gap between the generated controller programs by the MetaForma compiler, and the target platform of the ATRON robot.

7.1 Architecture

In Figure 7.1 is shown the architecture of the MetaForma runtime system. Currently, it is implemented in Java for the USSR simulator. Its core class is the MfController, extending the ATRONController class and implementing the ATRONAPI. As each module in the ensemble executes its own controller program, each module also runs its own runtime system. The MfController instantiates the following classes in the runtime system:

- The MfStateManager is responsible for attaining a distributed execution of sequences and the sequence transitions.
- The MfActuation class bridges the gap between ensemble-level hardware control primitives from the generated code, and the module-level functions in the ATRON-API (Section 2.2).
- The MfContext contains the module neighbor table and attains symmetry preservation.
- The MfScheduler executes various tasks for the runtime system in a repeating interval, which is mainly the propagation of packets among the ensemble.
- The ModuleCore, MetaCore and RegionCore. The module, meta and region construct available in DSL programs are compiled to method invocations on these classes (Subsection 6.2.4).

The following classes are simulation-supporting; their functionality is not needed on the physical modules: The MfSimulation class is a parent class for all concrete simulations of generated MetaForma controllers, to initialize the simulation. The MfBuilder class is used by
7. The MetaForma Runtime System

Figure 7.1: Architectural Diagram of MetaForma Runtime System

The *MfSimulation* and has methods for building specific ensembles, such as a Car, Snake or various types of scaffolds to the simulation.

The *MfVisualizer* provides a means to assign the appropriate color to each hemisphere and connector in the module. Furthermore, it fills and formats the debugging window in the USSR, showing specific debugging information per module. For the complete source-code of the runtime system the reader is referred to GitHub [74].

7.2 The MfController

The MfController has two main tasks. First, it processes the main control loop of the runtime system. Second, if facilitates communication with neighbors.

7.2.1 Main Control Loop

In [Listing 7.1] is the activate method shown, which is invoked by the USSR when the simulation is started. It has an analogous format as the Car controller shown in [Listing 2.2]; both controllers have a setup() call for the initialisation and a while(true){..} as main control loop. In the main control loop of the runtime system, the following things are happening:

First, if the central actuator is not rotating continuously, it instructs the ATRONAPI to rotate the absolute current angle value that is maintained in the angle variable. This improves connector alignment, as the actuator actively and automatically corrects any erroneous changes in the angle. Second, the MfScheduler is triggered, such that it can check whether it needs to execute one of its scheduled task. Third, the invocation of handleStates executes the generated
public void activate() {
    setup();
    // Packet loss configuration from Schultz et al. (2011) [65]
    setCommFailureRisk(0.25f, 0.25f, 0.98f, 0.125f);
    this.init();    // Init the generated controller
    // Using intervals from MfConfig
    mfsScheduler.enable("module.broadcastConsensus");
    mfsScheduler.enable("module.discover");
    mfsScheduler.enable("meta.broadcastVars");

    while (true) { // Main control loop
        if (mfActuation.isContinuousPositioningEnabled())
            rotateToDegreeInDegrees(angle % 360); // Actively reposition actuator
        mfsScheduler.trigger(); // Check for scheduled tasks
        this.handleStates();    // Execute current state
        mfsStateMngr.merge();   // Merge incoming state with local state
        yield();    // Give other threads CPU
        mfsVisual.colorize();
        MfStats.getInst().writeLog(getTime());    // Update statistics
    }
}

Listing 7.1: Controller initialization and main control loop in the MfController

public synchronized void yield() {
    System.out.println(module.getID() + ". before yield " + getTime());
    while (module.getSimulation().isPaused()) Thread.yield();
    module.getSimulation().waitForPhysicsStep(false);
    System.out.println(module.getID() + ". after yield " + getTime());
}

Listing 7.2: The yield() method in the ATRONController

code of the controller (Listing 6.10). Fourth, the state manager merges the received state with its own local state. Fifth, yield() is invoked, to give the other existing threads also CPU time. Sixth, the MfVisual assigns the appropriate colors to the connectors and hemispheres of each module. Seventh, the static MfStats.getInst() is invoked to update the statistics file, which happens ten times per second.

7.2.2 Communication

The communication system[17] is implemented in the MfController class. It contains the send method for sending a packet to direct neighbors, and it contains the receive method for processing received packets.

Packets

The MetaForma compiler generates a Java class for each declared packet in the language. Packets have a standard header of seven bytes. In Table 7.1 are shown the specific fields that are
7. The MetaForma Runtime System

<table>
<thead>
<tr>
<th>Field</th>
<th>Range</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>moduleID</td>
<td>[0..255]</td>
<td>8 bits</td>
<td>Module id of the sender (serialized)</td>
</tr>
<tr>
<td>metaID</td>
<td>[0..255]</td>
<td>8 bits</td>
<td>Meta id of the sender</td>
</tr>
<tr>
<td>regionID</td>
<td>[0..255]</td>
<td>8 bits</td>
<td>Region id of the sender</td>
</tr>
<tr>
<td>connNb</td>
<td>[0..7]</td>
<td>3 bits</td>
<td>Connector number of the sender</td>
</tr>
<tr>
<td>connNbConn</td>
<td>false/true</td>
<td>1 bit</td>
<td>Is the neighbor-connector connected?</td>
</tr>
<tr>
<td>type</td>
<td>[0..15]</td>
<td>4 bits</td>
<td>Type of the packet</td>
</tr>
<tr>
<td>dir</td>
<td>req/ack</td>
<td>1 bit</td>
<td>Request / acknowledgement flag</td>
</tr>
<tr>
<td>metaPart</td>
<td>[0..7]</td>
<td>3 bits</td>
<td>Meta-part of sender</td>
</tr>
<tr>
<td>state</td>
<td>State</td>
<td>20 bits</td>
<td>Current state (Figure 7.2) of sender</td>
</tr>
</tbody>
</table>

Table 7.1: Variables that are included in the header of every packet, together comprising exactly seven bytes.

Sending packets

The MetaForma send primitive (Section 5.5.1) is compiled to a method call send on the MfController with as argument the instance of the packet that it sends and the module neighbor table (9) that contains the destination modules of the packet. The send method is implemented below:

```
1 public void send (PacketBase p, INeighborTable nbSet) {
2    for (IModule nb:nbSet.modules()) {
3        buildHeader(p);
4        p.setConnConnected(isMALE(nb.conn) && mfContext.isConnConnected(nb.conn));
5        sendMessage(p.serialize(), (byte) p.serialize().length, mfContext.rel2abs(nb.conn));
6    }
```

Listing 7.3: Implementation of the send method to send packets to neighbors

This method first iterates through all neighbors from the neighbor table, and retrieves their connector number. Second, it serializes the packet object to a byte array and dispatches it to the sendMessage method of the ATRONAPI. Note that the mfContext first converts the relative connector number to an absolute number.

Processing received packets

Once a packet is received from a neighboring module, the handleMessage method is invoked on the ATRONController. The MfController overrides this method and first converts the absolute connector number that the packet was received on to a relative connector number, which is shown in Listing 7.4. Thereafter, makePacket de-serializes the received array of bytes into a packet object and passes it through to the receivePacket method (Listing 6.11).
7.3 The MfStateManager

In this section we describe the MfStateManager class. This class is responsible for attaining a distributed execution of sequences and sequence transitions. It uses a state variable for storing its own program state, and in order to synchronize its program state with neighboring modules.

As all single modules within a meta-module or region need to be updated with state information, broadcasting is used inside meta-modules and regions. It are typically small collections of modules and more importantly, every module in them needs to be updated.

7.3.1 The state of a module

In order to synchronize the execution among multiple modules, we first define the notion of a state. A state is a numerical representation of the current execution point of a module, identifying the current sequence, its sequence orientation, and the current sequential action. When a generated controller is executed, the runtime system checks by using the MfStateManager which sequential action it must execute.

To accomplish synchronisation between modules, the state manager implements a flowchart and state machine in a single, distributed algorithm. A state machine differs from a flowchart in the sense that a state machine performs its actions in response to triggers. A flowchart, on the contrary, does not need triggers and uses transitions from node to node in a graph when activities are completed. A flowchart can be compared with a factory assembly line; it describes the progression from beginning to end, while in a pure state machine progression cannot be measured.

However, the runtime system needs the notion of progression in its state, in order to be able to merge its local state with the received state from neighbors.

Flowchart-based execution of a sequence

In MetaForma sequences, actions are always executed in a successive way, as loops and jumps are not supported. Therefore, the actions of a sequence can be measured at an ordinal measurement scale. In order to get a numerical representation for each action in a sequence, the compiler assigns successive numbers to all sequential actions. Hence, a module can distinguish a newer action from an older action by simply comparing their action numbers. The number that represents the current sequential action is included in the state as the ActNr variable.
A state machine for sequence transitions

In order to share the current sequence and its sequence orientation with neighbors, the SeqID and OrientID variables are used, respectively. This is a numerical representation of the sequence, assigned by the runtime system. The Init and Strategy sequences are respectively represented by 0 and 1, the custom sequences defined by the programmer are assigned a successive and therefore unique number. The sequence orientations as defined in Figure 5.9 are assigned a unique number as well.

In contrast to a flowchart, a pure state machine does not describe progression, as from the current sequence any arbitrary new sequence can be started. As a result, the numerical representation of a sequence can only be measured at a nominal measurement scale. In order to distinguish a newer from an older sequence the state needs an additional means: A state counter Cntr that is increased by one when a new sequence starts executing.

Hence, the state of a module is represented by its state counter Cntr, its current sequence id SeqID, its current sequence orientation id OrientID, and its current sequential action number ActNr. In Figure 7.2 is the state defined as a quadruple that includes these variables.

\[(Cntr, SeqID, OrientID, ActNr)\]

State var | Range | Size
--- | --- | ---
Cntr | [0..127] | 7 bits
SeqID | [0..15] | 4 bits
OrientID | [0..15] | 4 bits
ActNr | [0..31] | 5 bits

Figure 7.2: Representation of a state using four variables

7.3.2 Merging the local state with incoming states

In order to attain synchronized execution among all individual modules in a meta-module or region, all modules need to share their own state with neighbors. Every module has its own local state and shares this state with direct neighbors in the same meta-module or region.

The runtime system continuously broadcasts packets, where the state variable is included in the header of the packet. When a module receives a packet, it checks whether it belongs to the same meta-module or region as the sender, and if this is the case it automatically merges its local state with the received incoming state.

In Figure 7.3 is defined the state merge function, that the state manager in each module uses to merge the incoming state with its current local state. It is inspired by the state merge function used in DynaRole (Section 3.8).

The first and the second clause in this formula assign the incoming state to the current state. In the first clause the incoming state counter is higher, and in the second clause the state counter is equal but the action number is higher. The third clause is used to reset the state of a meta-module module back to the Init sequence with a state counter Cntr of zero.

Resetting the state of a module to \((0, Init, 0, 0)\) is needed when the programmer instructs a meta-module to disable itself, and the Init sequence needs to be re-executed on the individual modules. This cannot be attained by increasing the state counter to a higher value, for the
The MfStateManager

\[
S_c = \begin{cases} 
  S_i & \text{if } Cntr_i > Cntr_c \\
  S_i & \text{if } Cntr_i = Cntr_c \land \text{ActNr}_i > \text{ActNr}_c \\
(0, \text{Init}, 0, 0) & \text{if } S_i = (0, \text{Init}, 0, 0) \land \text{SeqID}_c \notin \{\text{Init, Strategy}\} \\
S_c & \text{otherwise}
\end{cases}
\]

The current state \(S_c\) is denoted as \((Cntr_c, \text{SeqID}_c, \text{OrientID}_c, \text{ActNr}_c)\)

The incoming state \(S_i\) is denoted as \((Cntr_i, \text{SeqID}_i, \text{OrientID}_i, \text{ActNr}_i)\)

Figure 7.3: The state merge function that is used by the runtime system of each module to merge an incoming state with its own local state

following reasons:

First, the state counter \(Cntr\) cannot be implemented as infinite, and for each successive new state, a successive sequence counter has to be used, implying that it eventually will overflow.

Second and more importantly, it might happen that the runtime-system on modules in meta-module \(A\) initiate a region with modules in meta-module \(B\), to execute a new sequence. If meta-module \(A\) has not increased its state from the beginning, but meta-module \(B\) has already executed some sequences, \(A\) will have a sequence counter of zero while the sequence counter of \(B\) will be higher than zero.

But \(A\) initiated the region and need to spawn the sequence and sequence orientation to \(B\). Hence, it needs to use a higher sequence counter than \(B\), otherwise \(B\) will ignore the new state from \(A\). Note that the modules in \(B\) cannot just set his sequence counter to \(A\), as \(B\) consist of multiple single modules and if they are not updated at exactly the same time, which is likely to be the case, they will re-merge their own state instead of the state of \(A\). Therefore, when disabling a meta-module, the state of all its single modules must be reset to \((0, \text{Init}, 0, 0)\).

7.3.3 Parallel execution of statements in the current action

Each sequential action in a sequence can comprise multiple statements for parallel execution, to speed up the self-reconfiguration process. When a module is finished with executing its statements in the current action, it invokes the \texttt{postState()} method of the \texttt{MfStateManager} (Listing 6.10), this sets its module bit in the consensus set and shares it with neighbors. The state manager waits for other modules to finish their current sequential action before advancing to the next state.

In order to supply the runtime system in all individual modules information about which modules have finished the current action, a consensus set is used at each module. This set is implemented using a BitSet of 256 bits, the bit at position \(n\) represents whether the module with id \(n\) is finished executing its statements in the current action.

In contrast to the state, the consensus set is not included in the header of each packet and thus must be propagated separately. Each module continuously shares its own local consensus set in the payload of primitive Consensus packets. When a module receives a consensus update
7. The MetaForma Runtime System

```java
void merge () {
    stateCurrent.merge(stateIncoming); // using formula in Figure 7.3
    if (consensusReached())
        gotoNextState(); // Go to the next sequence or sequential action
}

boolean consensusReached () {
    byte moduleCount;
    if (meta.regionID == 0)
        moduleCount = metaPart.count + meta.sizeExtra;
    else if (meta.regionID != 0 && meta.regionID == module.metaID)
        moduleCount = region.size * metaPart.count;
    return meta.isEnabled && consensus.bitCount() >=
        moduleCount * (1 - (secondsSpentInState * config.downgradeConsensus));
}
```

Listing 7.5: Pseudo-code for the consensusReached function: This function checks whether
the size of the consensus set is high enough to advance to the next state

from a neighbor, it checks whether the sender is in the same state, if this is the case the runtime
system adds all modules in the received set to its own local consensus set.

State transition

A state transition is realized when all modules in the sequence are finished with executing their
current sequential action. If the consensusReached method returns true, the local state will
be increased by the runtime system.

The amount of modules that is executing the same sequence is computed by the runtime
system. There are basically two cases: (1) only a single meta-module, or (2) multiple
meta-modules inside a region. In the former case, the amount of modules is the amount
specified by the metaPart parts construct, plus the sizeExtra that can be specified by the
programmer (Listing 5.16). In the latter case, the amount of modules is computed by multiplying
the amount of meta-modules in the region with the size of a single meta-module.

If the computed amount of modules is equal or higher than the size of the consensus set, a
state transition can be made, as all modules are finished with their current sequential action. As
a final means to attain robustness against partial hardware failure, a consensus degradation is
used.

Consensus degradation

During the execution of a sequence, modules can fail for various reasons like unreliable com-
munication or connector misalignment. Waiting for a crucial part in the robot to complete
its action might be justified; however, endless waiting for a non-crucial part must be avoided.
Therefore, consensus takes a downgrade percentage into account that allows, as time progresses,
advancing to the next state when not all modules are finished with their current action, i.e. not
all modules have committed. This avoids endless waiting on partially unfinished actions. The `secondsSpentInState` variable contains the amount of seconds that the current action is active. As this value increases during time, at a certain point the `consensusReached()` method will return true.

As an example, in [Listing 4.4](#) is described the Flip-Along meta-action. In action 4 and 5, the Clover tries to connect to the Structure by using two connectors. If action 4 fails, as a result of connector misalignment or communication problems, continuing to action 5 can be beneficial. If move 5 succeeds, in contrast to move 4, the Walker is still connected to the Scaffold and the sequence can probably be finished successfully.

In general, if after some amount of time it becomes likely that the complete consensus will not be met, it is better to continue with the next action in the sequence. The uncompleted action from the previous action might was crucial, causing the sequence to hang in a later stadium, but waiting forever also hangs the execution.

### 7.4 The MfActuation

In this section we present part of the implementation of the `MfActuation` class. It contains functionality for hardware control primitives [6], such as actuation and controlling connectors.

This class is used by the `MfController` and bridges the gap between ensemble-level actuator/connector control, and the module-level functions in the ATRONAPI (Section 2.2). Ensemble-level control functions use `ModuleRefs` (Section 6.2.4), which represents a reference to either a module group [8] or a module identification [1].

#### 7.4.1 Relation with MfStateManager

Actions that use hardware actuation are slower in comparison to actions using only software computation. Indeed, an actuator is an electrical motor that needs time to rotate, while software statements are only flipping bits.

In order to attain parallelism in a module, the runtime system does not use so-called synchronous blocking code, that blocks the controllers execution until the actuator is finished. This would preclude one module from extending multiple connectors at the same time. Therefore, asynchronous non-blocking code is used and the control flow continues during actuation.

However, the use of non-blocking code results the runtime system to execute the `postState()` method when it has finished executing all statements at the end of a sequential action ([Listing 6.10](#)). This would allow the MfStateManager to advance to the next state, while the actuators are not finished with their action.

Therefore, the MfStateManager needs knowledge about which modules are using actuation in a sequential action, such that it can wait until all modules are finished, before advancing to the next state. The `MfActuation` provide knowledge about the completeness of the current actuation by first checking whether its own actuator must execute the current action. If this is the case, the runtime system precludes itself from automatic commit, and at the `postState()` call of the `MfStateManager` the runtime system does not commit for the current action.
7. **The MetaForma Runtime System**

Hence, when any action is invoked on the *MfActuation*, each module checks whether it is referred by the provided *ModuleRef* by checking its *module identification* or *module group*.

### 7.4.2 The connect and disconnect methods

The *connect/disconnect* methods (Listing 5.14) connects/disconnect modules referred to by *r1* to modules referred to by *r2* and vice versa. The *insideRegionOnly* parameter specifies whether modules outside the region should also be considered for the action, if any.

Below we discuss the implementation of the *connect* method. The *disconnect* method has an analogous implementation and is therefore omitted. First, the runtime system checks whether it has to take action, in that case it precludes itself from automatic commit. Second, the connection method is invoked with both *r1* and *r2* swapped, as they should mutually connect.

```java
1 connect(ModuleRef r1, ModuleRef r2, boolean insideRegionOnly) {
2   if (r1 refers to module.id or r2 refers to module.id):
3     mfStateMngr.commitNotAutomatic() // I am involved in the move
4     connection(r1,r2, true, insideRegionOnly)
5     connection(r2,r1, true, insideRegionOnly)
6 }
```

The *connection* method first checks whether *r1* refers to the module it is executed on. If this is the case, it loops through all neighbors that it is aligned to with a male connector. It checks for connected or disconnected connectors in case we want to disconnect or connect, respectively. Thereafter, it loops through the neighbors to check whether it is completed; if this is the case it commits manually. This is shown below.

```java
1 connection(ModuleRef r1, ModuleRef r2, boolean connect, boolean insideRegion) {
2   if r1 refers to module.id or r2 refers to module.id:
3     connection(nb, connect); // connect myself to nb
4     if nbs (inRegion: insideRegion, in:r2, isConnected:-connect) = 0 ∧ nbs {insideRegion, in:r2, isConnected:connect} ≠ 0 // Am I finished?
5     mfStateMngr.commit() // Set module bit in consensus set
6 }
```

### 7.4.3 The extend, extendSet, retract and retractSet methods

The implementation of the *extendSet* (Listing 5.14) is shown below. The implementation of the *extend, retract* and *retractSet* is analogous and therefore omitted.

```java
1 extendSet (ModuleRef r, int set) {
2   connection(r, set, true);
3 } // the retractSet, retract and extend methods are analogous and therefore omitted
```
All four methods use the connection function as shown below. This method works as follows, first the runtime system checks whether the module that it runs on must take action. If this is the case, it loops through all neighbors that are connected to it with a male connector belonging to the connector set \( \text{set} \), and performs the action on that connector.

Furthermore, it checks whether the neighbor table subject to the inverted action is empty, i.e. whether the action is completed, and then commits. To make sure also female connectors that belong to \( \text{set} \) are also considered, the runtime system loops through

\[
\begin{align*}
\text{connection} \ (\text{ModuleRef} \ r, \ \text{int} \ \text{set}, \ \text{boolean} \ \text{connect}) \ {\{} \\
\quad \text{if} \ (r \ \text{refers to module}.id) \\
\quad \quad \text{for} \ \text{nb} : \ \text{nbs} (\text{MALE}, \ \text{connected} : \neg \text{connect}, \ \text{connOwn} : \text{set}) \\
\quad \quad \quad \text{connection} (\text{nb}, \ \text{connect}) \ // \ \text{perform action on male connectors} \\
\quad \quad \text{if} \ \text{nbs} (\neg \text{connect}, \ \text{connOwn} : \text{set}) = \emptyset \\
\quad \quad \quad \text{mfStateMngr}.\text{commit}() \\
\quad \quad \text{else} \\
\quad \quad \quad \text{for} \ \text{nb} : \ \text{nbs} (\text{MALE}, \ \text{connected} : \neg \text{connect} \ \text{in} : r, \ \text{connNb} : \text{set}) \\
\quad \quad \quad \text{connection} (\text{nb}.\text{connOwn}, \ \text{connect}) \\
\{ 
\end{align*}
\]

Below is the connection method shown that the runtime system uses to pass connector through to the ATRONAPI. When disconnecting, the runtime system sends out discover messages to update the female connector cache (Subsection 7.6.4) on the neighbor. This is used to store the connection state of female connectors and when a connector is retracted the cache needs to be updated.

\[
\begin{align*}
\text{connection} (\text{byte} \ \text{connector}, \ \text{boolean} \ \text{connect}) \ {\{} \\
\quad \text{if} \ \text{connect} \ \&\& \ !\text{isConnConnected} (\text{connector}) \\
\quad \quad \text{ctrl}.\text{connect} (\text{rel2abs} (\text{connector})) \ // \ \text{dispatch to ATRONAPI} \\
\quad \text{else if} \ !\text{connect} \ \&\& \ !\text{isConnConnected} (\text{connector}) \\
\quad \quad \text{ctrl}.\text{disconnect} (\text{rel2abs} (\text{c})) \ // \ \text{dispatch to ATRONAPI} \\
\quad \text{module}.\text{discover} () \ // \ \text{notify disconnection to neighbor} \\
\{ 
\end{align*}
\]

## 7.5 The MetaCore and RegionCore

### 7.5.1 The MetaCore class

The main task of the MetaCore class is meta-module variable synchronisation among all single modules in a meta-module. As the value of a variable need to be available in all single modules inside a meta-module, a synchronisation algorithm is needed. In addition, if a meta-variable is modified inside one single module, it must send an update to all other modules in the meta-module. As messages can get lost, the runtime system propagates each meta-variable constantly in a fixed interval, in addition to when a variable changes, then it will be propagated immediately. In order to distinguish new values from old values, each variable is associated with a sequence number during storage and transmission.

The following things are needed to achieve variable synchronisation:
7. The MetaForma Runtime System

1. A table storing variable numbers together with their value and sequence number.

2. A packet to send updated entries in the entire meta-variable list, and sending the whole list every timespan.

The regionID is a primitive meta-variable used by the runtime system, and the programmer can define their own variables as well (Listing 5.10) which will automatically be synchronized.

A MetaVarSyncPacket is used to synchronize variables inside a meta-module. This packet is independent on the state of the sender, i.e. it can be processed always when received, which is handled by the runtime system. It contains the synced variable id, its value, and its sequence number. When merging the value at the destination, the value of the highest sequence number is used.

7.5.2 The RegionCore class

The task of the RegionCore class is to facilitate enabling and disabling regions[^15]. In the future, it can be extended with support for region variables, that are automatically synchronized among all single modules in an entire region.

Enabling a region

Enabling regions is performed by the try region construct (Listing 5.20), specifying an including \( M_{in} = \{\text{metaID}_1,..\} \) and optional excluding \( M_{ex} = \{\text{metaID}_1,..\} \) clause. First it is checked whether \( M_{in} \) does not contain zeros and \( M_{ex} \) does not contain non-zeros, as zero represents a non-existing metaID. If these constraints are fulfilled, the region is created.

This works as follows. Every single module in the meta-module initiator checks whether it has a neighbor belonging to a meta-module in \( M_{in} \), i.e. a meta-module that needs to make part of the region. If this is the case, it sends a RegionPacket to that module. The runtime system check whether the meta-modules in the region it creates are all adjacent to the originating meta-module. If this is not the case, the runtime system include the non-adjacent metaID in the RegionPacket that is sent to the neighboring meta-module. Currently only one non-adjacent meta-module is supported, but this can be extended in the future. Also, try region needs to be invoked on the module that is adjacent to the meta-module, as this reduces the complexity of the need to propagate RegionPackets in its own meta-module, which can have a cyclic structure.

A RegionPacket is used to enable regions from groups of meta-modules. In order to enable regions, all participating meta-modules must be reached by communication to assign them to a region. Unfortunately, this does also include non-adjacent meta-modules (see Figure 4.15). To reach a non-adjacent meta-module, its meta id is included in the packet. When received, the adjacent neighboring module propagates the RegionPacket further to the other module.

Another option would be using meta-packets, i.e. packets that are sent and received by meta-modules. This has advantages pertaining robustness, but also induces the following extra complexities: (1) A TTL mechanism must be used as meta-modules contain cyclic communication paths and (2) a routing protocol that directs meta-packets to the right meta-module. Although meta-packets are highly robust pertaining packet-loss, as a packet is send...
over multiple physical connectors, the benefits are not worth the costs in this case. For future work it can be worth to reconsider, especially when the need arises for assigning gradients at meta-module granularity.

**Disabling a region**

Disabling a region includes the following: First, the runtime system waits for \textit{preRegDisableTime} (Table 7.2) to make sure all modules in the region get the most recent state update, prior to disabling the region. Second, each module sets its regionID to zero individually, and hereafter, it does not share state information any more with neighbors outside the meta-module. Hence, it is important all modules in the region have the same state before modules start to disable the region, otherwise it can happen that isolated modules remain after disabling.

To make sure no two regions can exist at the same time with the same region id, before enabling a new region, the runtime system also has to wait for \textit{preRegDisableTime}, as a conservative approach. Otherwise, in case the same meta-module enables a new region and the two regions are adjacent, they will interfere each other with their current state variable.

### 7.6 The MfContext

The main task of the \textit{MfContext} is to realize symmetry preservation in the ensemble. It exploits the hardware property of the ATRON module that both the north and south hemisphere and both the east and west parts are functionally equal in hardware, pertaining connector design. The runtime system uses so-called relative connector numbers, that are relative to the eight different disruptions that can occur in a two-dimensional grid, as described in Subsection 4.4.3.

Each module maintains how its symmetry is disrupted relatively to other modules, by using three boolean variables \textit{flipNS}, \textit{flipEWN} and \textit{flipEWS}.

#### 7.6.1 Relative angle and connector numbers

The MfContext gets knowledge about neighbors when messages are received from neighbors, i.e. when the \textit{handleMessage} method is invoked by the \textit{ATRONController}. Before the call is processed by the runtime system, the MfContext first converts the absolute connector number to a relative number (Listing 7.4). Therefore, all stored connector numbers in the runtime system are automatically relative. The function is implemented as follows:

\[
\text{abs2relConnector} : \mathbb{N} \rightarrow \mathbb{N} \text{ such that:}
\]

\[
\text{abs2relConnector}(abs) = \begin{cases} 
abs = \text{abs} + 2 \pmod{4} & \text{if } \text{flipEWN} \land (0 \leq \text{abs} \leq 3) \\
abs = \text{(abs} + 2 \pmod{4}) + 4 & \text{if } \text{flipEWS} \land (4 \leq \text{abs} \leq 7) \\
abs = \text{abs} + 4 \pmod{8} & \text{if } \text{flipNS}
\end{cases}
\]

At the place where the MetaForma runtime system communicates to the \textit{ATRONAPI}, such e.g. the \textit{sendMessage} invocation (Listing 7.3), each provided relative connector is mapped back to its absolute value. This is performed using the following function:
7. The MetaForma Runtime System

rel2absConnector : \mathbb{N} \rightarrow \mathbb{N} \text{ such that:}

\begin{align*}
\text{rel2absConnector}(\text{rel}) &= \begin{cases} 
\text{rel} = \text{rel} + 4 \pmod{8} & \text{if } \text{flip}_{\text{NS}} \\
\text{rel} = \text{rel} + 2 \pmod{4} & \text{if } \text{flip}_{\text{EW}} \land (0 \leq \text{rel} \leq 3) \\
\text{rel} = (\text{rel} + 2 \pmod{4}) + 4 & \text{if } \text{flip}_{\text{EW}} \land (4 \leq \text{rel} \leq 7)
\end{cases}
\end{align*}

The rotateOrigin function (Listing 5.5) must take the symmetry into account when rotating back to zero degrees. Therefore, the degree value from the rotational actuator is mapped from the locally stored relative degree value to an absolute degree, before the runtime system invokes the ATRONAPI. This is shown below. Note that a absolute to relative mapping for degrees is not needed, as the rotational actuator only rotates as controlled by the ATRONController.

abs2relAngle : \mathbb{N} \rightarrow \mathbb{N} \text{ such that:}

\begin{align*}
\text{abs2relAngle}(\text{abs}) &= \begin{cases} 
\text{abs} & \text{if } \text{flip}_{\text{EW}} \oplus \text{flip}_{\text{NS}} \\
\text{abs} + 180 \pmod{360} & \text{otherwise}
\end{cases}
\end{align*}

7.6.2 Distributed Symmetry Preservation

In this subsection we present a distributed algorithm that the runtime system uses to attain symmetry in the overall ensemble. This algorithm consist of (1) a basis step, and (2) an inductive step.

1. **Traversal initialization**: One module from which the symmetry is still preserved, starts the traversal by continuously broadcasting SymmetryRepair packets along its connectors.

2. **Traversal propagation**: As the SymmetryRepair is eventually received by neighbors, they update their own symmetry relatively to the sender using the algorithm shown in Listing 7.6. Consecutively, they keep broadcasting the packet further to their neighbors.

```java
void fixSymmetry (boolean isRef) { // basis step
    if (isRef) // One of the modules initiates the traversal
        broadcast PacketSymmetry();

    // A symmetry packet is received
    when (receive PacketSymmetry (byte connOwn, byte connNb)) { // inductive step
        // First fix own local symmetry
        if (connOwn == FEMALE)
            if (connOwn == WEST) \oplus (connNb == SOUTH)
                flipNorthSouth();
            connNb = connNb + 4 \pmod{8}
            flipEastWestHem ((connOwn == NORTH) \oplus (connNb == WEST), connNb == SOUTH)
    }
```

120
else if connOwn ∈ MALE

if (connOwn ∈ WEST) ⊕ (connNb ∈ NORTH)

flipNorthSouth ()

connNb = connNb + 4 (mod 8

flipEastWestHem ((connOwn ∈ NORTH) ⊕ (connNb ∈ EAST), connNb ∈ SOUTH)

broadcast PacketSymmetry () // Afterwards, propagate the packet further

\*

Listing 7.6: Pseudocode for the distributed symmetry preservation algorithm used by the MetaForma runtime system

The flipNorthSouth function is trivial, in the sense that it flips the \textit{flipEWN} boolean and updates the connector numbers in the module neighbor table\footnote{9}. On the contrary, the flipEastWestHem function needs to know on which hemisphere the packet arrived and whether there is a symmetry mismatch.

- If there is no symmetry mismatch on the received hemisphere, the module checks whether it has rotated zero or 180 degrees. In the latter case, the other hemisphere is out of symmetry, thus it perform an east-west flip at that specific hemisphere.

- If there is a symmetry mismatch, the module performs an east-west flip at the hemisphere the packet was received at. Consecutively, it checks whether it has rotated zero or 180 degrees. In the former case, the other hemisphere is also out of symmetry and needs to be flipped.

The symmetry-preservation protocol in this section can in theory be used to correct symmetry after docking\footnote{71} a robot, for example a car, into another robot ensemble\footnote{Subsection 1.1.2}. One of the modules in the structure that the car docks onto must initiate the protocol, and each module in the car robot changes its connector numbers match those used in the structure. Support for docking is considered future work for MetaForma, but the symmetry repair protocol
already solves a part of the docking problem: to get an overall symmetry in connector sets in each module in the ensemble.

### 7.6.3 The Neighbor Table

The \textit{NeighborTable} class implements the concept of the \textit{module neighbor table}. The neighbor table contains an entry for each connector that a packet is received on, it is updated by the runtime system when any packet is received (Listing 7.4).

When using actuation or communication to neighbors, the \textit{Neighbor Operator} can be used. In addition, in order to identify a module based on its relative position to neighbors, the \textit{Neighbor Count Operator} is used. These operators query the neighbor table by using a fluent interface. Each method returns a reference to the new refined neighbor table. The original neighbor table is not modified when queried, instead; a new instance of a refined neighbor table is returned.

A \textit{DiscoverPacket} is used to fill the \textit{module neighbor table}, by broadcasting periodically to discover neighbors. When executing a sequence, at the beginning of a sequential action it is send more frequently, to make sure the neighbor table is filled.

The following query operations are available on the neighbor table. They form the basis for the \textit{Neighbor Operator} and \textit{Neighbor Count Operator} conditions, as shown in Listing 5.19 and compiled in Section 6.2.4. Each condition is compiled to invocations of the following methods:

```java
public boolean isConnected(int c) {
    if (c == -1) return false;
    if (MfController.is(FEMALE, c)) // For female connectors,
        return femaleConnectorCache[c]; // use the connector cache
    else // For male connectors, use the ATRONAPI
```
The MfScheduler

7 return ctrl.isConnected(rel2abs(c));
8 }

Therefore, instead of using the isConnected method for male and female connectors, the runtime system retrieves from the female connector cache whether a female connector is connected, such that it can also work on the physical modules. The MfContext has the isConnConnected(c) method, that is used in the overall runtime system instead of the isConnected method from the ATRONAPI.

Thus, when disconnecting a neighbor module, the runtime system sends a number of DiscoverPackets such that the neighbor module knows it is no longer connected to the sender. Clearing the female connector cache when entering a new state is a conservative approach, but does not work pertaining e.g. gradients that use the neighbor table in order to determine the origin.

7.7 The MfScheduler

The task of the MfScheduler is to execute certain tasks for the runtime system in a repeating interval, mainly propagating information among the ensemble. The MfController can request the MfScheduler to schedule methods to be invoked at a specific interval.

To accomplish this, the method trigger() is invoked in the main control loop (Listing 7.1), that checks constantly which methods needs to be invoked, according to their scheduled interval.

Methods are scheduled for: periodically broadcasting (a) DiscoverPackets, to make sure the module neighbor table stays up to date, (b) ConsensusPackets, such that every module has an up-to-date consensus set, and (c) MetaVarSyncPackets, to update meta-module variables at neighbors that might not received the packet that was sent when a meta-module variable was updated.

The intervals for these tasks are configured in the runtime constants broadcastDiscover, broadcastConsensus and broadcastMetaVarSync, respectively (Section 7.8).

7.8 Runtime System Configuration

In Table 7.2 are the parameters shown that are used by the runtime system.
### 7. The MetaForma Runtime System

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcastDiscover</td>
<td>6 s</td>
<td>Interval to automatically broadcast a Discover packet</td>
</tr>
<tr>
<td>broadcastConsensus</td>
<td>10 s</td>
<td>Interval to synchronize consensus</td>
</tr>
<tr>
<td>broadcastMetaVars</td>
<td>4 s</td>
<td>Interval to synchronize meta-variables</td>
</tr>
<tr>
<td>propagationRate</td>
<td>0.3 s</td>
<td>Minimum interval between sending same packet again</td>
</tr>
<tr>
<td>propagationTime</td>
<td>5 s</td>
<td>DSL config constant</td>
</tr>
<tr>
<td>assignTime</td>
<td>5 s</td>
<td>DSL config constant</td>
</tr>
<tr>
<td>downgradeConsensus</td>
<td>0.1%</td>
<td>Downgrade the amount of modules every second using this factor</td>
</tr>
<tr>
<td>priorStateDiscTime</td>
<td>1 s</td>
<td>Time to refill neighbor table prior to starting an action</td>
</tr>
<tr>
<td>preRegDisableTime</td>
<td>5 s</td>
<td>Time to propagate state before disabling a region</td>
</tr>
</tbody>
</table>

Table 7.2: Configuration parameters for the runtime system
Chapter 8

Experiments

This chapter describes the results of experiments that have been conducted to test the MetaForma compiler and runtime system by running simulations in the USSR simulator.

First, we compare the MetaForma implementation of the Eight2Car sequence to the existing DynaRole \cite{65} implementation, in terms of code layout and performance. Second, we conduct experiments on obstacle avoidance with both a Two-Wheeler and Four-Wheeler car. Third, we verify the MetaForma implementation of the cluster-flow approaches of Brandt et al. \cite{16} and Christensen et al. \cite{29}.

8.1 Preparation

For each experiment, we compiled the MetaForma program to a Java controller to be executed in the USSR simulator. Consecutively, we created a simulation class in Java, that initializes the robot ensemble and eventual obstacles such as boxes, and assigns the generated controller to each module in the ensemble. In each experiment we refer to the used Java simulation class, using a relative URL\footnote{Relative URL prefix: https://github.com/wvankoppen/ussr/blob/wouter/src/ussr/samples/atron/simulations/} to GitHub \cite{74}. Each experiment can be redone by starting a USSR simulation using the specified main class.

The goal of the experiments is twofold. First, we check whether the MetaForma compiler and runtime system work. We check this by verifying whether the generated Java controller attains the behavior as specified in the corresponding DSL implementation.

Second, we check whether the MetaForma implementation respects the constraints for the first generation of physical ATRON modules. As described in Subsection 2.1.3 these constraints are as follows: Unreliable communication that is only possible to direct neighbors (which is already attained by not making use of the radio in the USSR) and with a limited bandwidth of 1.2 KB/s per module, female connectors that are completely passive, limited actuator strength, and modules that are prone to spontaneous lock-ups.


8. Experiments

8.1.1 Unreliable communication over limited bandwidth

The USSR simulator simulates by default ‘perfect communication’ between modules. Hence, every packet that is sent to a neighbor is also received by that neighbor. In order to realize a resemblance with the communication performance on the physical modules, we explicitly enable packet loss in the simulator. We use the configuration shown in Table 8.1, these values are found to be representative for the degree of packet loss on the physical modules [65].

For a given connector, there is a 12.5% risk of being ‘bad’, as 98% of the packets that are sent through will be dropped. The remaining connectors have a general dropping rate of 25% of the packets that are sent.

In order to check for the limited communication bandwidth, all sent packets are logged and at the end of an experiment we check whether the total communication exceeds on average 1.2 KB/s per module, as a worst-case approximation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>generalRiskMin</td>
<td>0.25</td>
<td>Minimum chance that a packet is dropped (25%)</td>
</tr>
<tr>
<td>generalRiskMax</td>
<td>0.25</td>
<td>Maximum chance that a packet is dropped (25%)</td>
</tr>
<tr>
<td>totalDegree</td>
<td>0.125</td>
<td>12.5% chance that a connector perform ‘bad’</td>
</tr>
<tr>
<td>totalDegreeRisk</td>
<td>0.98</td>
<td>98% chance that packets that will be dropped on bad connector(s)</td>
</tr>
</tbody>
</table>

Table 8.1: Packet-loss Configuration for the USSR Representing Physical ATRON modules [65]

8.1.2 Respected limitation of actuator strength

At the physical modules, the actuator of one single module is capable to lift at maximum two other modules against gravity. However, the actuation strength imposed by USSR does not accurately match the actuation strength of the physical modules. Therefore, we check visually in each simulation whether no more than two modules per actuator are lifted against gravity.

8.2 Eight to Car Transformation

The Eight2Car sequence is both implemented in DynaRole (Listing 3.6) and in MetaForma (Listing A.4). As the same Eight2Car scenario is implemented in both DSLs, and both compilers can compile to a Java controller to be executed in the USSR simulator, we can compare the running time and communication statistics of both scenarios. The following Java classes were used:

DynaRole Eight2Car Simulation: contrib/robustReversible/EightToCarRobustnessExperimentParallelStd.java
MetaForma Eight2Car Simulation: metaforma/EightToCarModRefSimulation.java
We ran the experiment ten times on both the DynaRole and MetaForma implementation. In Table 8.2 and Table 8.3 are the results shown, respectively. In the simulations of the MetaForma implementation, five simulations ended in a deadlock before completion.

In Listing 8.1 is shown the last part of the console output of one of the failed simulations. We located the deadlock to the main control loop of the runtime system, where \texttt{yield()} is invoked (Subsection 7.2.1). It shows that the controllers in all seven threads invoke \texttt{yield()}, but do not continue afterwards. In Listing 7.2 is the implementation of the \texttt{yield()} method in the USSR shown, that prints out a line to the console before and after yielding the thread.

As the runtime system invokes \texttt{yield()} in the USSR but does not get the control afterwards, we argue that the deadlocked simulation is a result of a concurrency bug in the USSR, and not a bug in the MetaForma runtime system.

Analysis

Performance  When we compare the running times of both approaches it is clear that the DynaRole implementation has a lower execution time than the MetaForma implementation: All ten simulations ended within 34.44 s while the MetaForma implementation took at max 58.77 s. However, the average communication in the DynaRole implementation is higher compared to the MetaForma implementation: 4.83 KB/s against 3.52 KB/s, both for the total ensemble. On average per module, the communication is 0.503 KB/s for the MetaForma implementation. In Table 8.4 are detailed packet statistics shown from the first simulation of the MetaForma sequence.

As we measured the communication statistics per ensemble and computed the average bytes per module and also on average per second, we cannot make any hard claim whether this fits within the upper-bound of 1.2 KB/s per physical ATRON module.

<table>
<thead>
<tr>
<th>Sim. #</th>
<th>Running Time</th>
<th>Bytes</th>
<th>Packets</th>
<th>Bytes / Packets</th>
<th>Packets / Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.49 s</td>
<td>142.24 KB</td>
<td>16184</td>
<td>9.00</td>
<td>4.82 KB / s</td>
</tr>
<tr>
<td>2</td>
<td>36.43 s</td>
<td>176.2 KB</td>
<td>20048</td>
<td>9.00</td>
<td>4.84 KB / s</td>
</tr>
<tr>
<td>3</td>
<td>29.54 s</td>
<td>142.66 KB</td>
<td>16232</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>4</td>
<td>29.84 s</td>
<td>144.14 KB</td>
<td>16400</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>5</td>
<td>29.79 s</td>
<td>143.72 KB</td>
<td>16352</td>
<td>9.00</td>
<td>4.82 KB / s</td>
</tr>
<tr>
<td>6</td>
<td>29.78 s</td>
<td>143.72 KB</td>
<td>16352</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>7</td>
<td>30.50 s</td>
<td>147.16 KB</td>
<td>16744</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>8</td>
<td>34.44 s</td>
<td>166.36 KB</td>
<td>18928</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>9</td>
<td>29.64 s</td>
<td>143.16 KB</td>
<td>16288</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
<tr>
<td>10</td>
<td>29.70 s</td>
<td>143.23 KB</td>
<td>16296</td>
<td>9.00</td>
<td>4.82 KB / s</td>
</tr>
<tr>
<td>AVG</td>
<td>30.91 s</td>
<td>149.26 KB</td>
<td>16982.40</td>
<td>9.00</td>
<td>4.83 KB / s</td>
</tr>
</tbody>
</table>

Table 8.2: Ten simulations of the DynaRole Eight2Car sequence (AVG deviation is 1 decimal)

Code layout  The MetaForma Eight2Car sequence (Listing A.4) is more expressive than the DynaRole sequence (Listing 3.6) that uses connector numbers, as the latter assumes a specific
8. Experiments

<table>
<thead>
<tr>
<th>Sim. #</th>
<th>Running Time</th>
<th>Bytes</th>
<th>Packets</th>
<th>Bytes / Packets</th>
<th>Packets / Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.02 s</td>
<td>199.83 KB</td>
<td>18471</td>
<td>11.08</td>
<td>3.50 KB / s</td>
</tr>
<tr>
<td>2</td>
<td>56.79 s</td>
<td>200.83 KB</td>
<td>18571</td>
<td>11.07</td>
<td>3.54 KB / s</td>
</tr>
<tr>
<td>3</td>
<td>58.06 s</td>
<td>205.71 KB</td>
<td>19054</td>
<td>11.06</td>
<td>3.54 KB / s</td>
</tr>
<tr>
<td>4</td>
<td>58.77 s</td>
<td>208.56 KB</td>
<td>19297</td>
<td>11.07</td>
<td>3.55 KB / s</td>
</tr>
<tr>
<td>5</td>
<td>57.30 s</td>
<td>198.57 KB</td>
<td>18448</td>
<td>11.02</td>
<td>3.47 KB / s</td>
</tr>
<tr>
<td>AVG</td>
<td>57.59 s</td>
<td>202.70 KB</td>
<td>18768.20</td>
<td>11.06</td>
<td>3.52 KB / s</td>
</tr>
</tbody>
</table>

Table 8.3: Five successful simulations out of total ten simulations the MetaForma Eight2Car sequence. The five unsuccessful simulations hanged as a result of a deadlock in the simulator (AVG deviation is 2 decimals)

<table>
<thead>
<tr>
<th>Packet</th>
<th>Communication (total ensemble)</th>
<th>Communication (total / # modules)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>800</td>
<td>18.71 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>6599</td>
<td>45.11 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>11072</td>
<td>136.01 KB</td>
</tr>
<tr>
<td>Total</td>
<td>18471</td>
<td>199.83 KB</td>
</tr>
</tbody>
</table>

Table 8.4: Detailed Communication statistics from the MetaForma implementation of Eight2Car sequence

module orientation for each module in the ensemble.

This is shown by the Eight2Car sequence implementation in both languages. DynaRole sequences use a completely off-line control algorithm, that controls connectors at ‘connector-number’ level, and thus a symmetry disruption of a module becomes problematic. MetaForma sequences are not completely computed off-line; the actions in the sequence are predetermined but the connector numbers are computed at runtime. Hence, it does not assume a module to be in symmetry. MetaForma trades in robustness for flexibility, but this experiment shows that the potential loss of robustness is not of a significant degree.

8.3 Car Obstacle Avoidance

This section describes two experiments to test the implementation of obstacle avoidance algorithms in MetaForma. The generated Java-controller from the MetaForma obstacle-avoidance programs for the Two-Wheeler and Four-Wheeler car, as shown in Section 5.2, are executed in the USSR simulator.

The simulation is executed ten times, for both the Two-Wheeler and the Four-Wheeler. The cars need to escape from a circle with a 25% gap, this gap is positioned at a random place in each situation. The simulation is stopped when the car successfully escaped from the obstacle or when it might crash. In Figure 8.1, are shown both (a) the Two-wheeler and (b) the Four-wheeler, each situated in a circle with a 25% gap at a random place.
8.3.1 Two-wheeler obstacle avoidance

Java Simulation main-class: `TwoWheelerSimulation.java`

Table 8.5 shows the results of running the obstacle avoidance experiment ten times on the TwoWheeler. All cars were able to escape through the gap from the circle of obstacles. In 60% of the scenarios, the Two-Wheeler drove out of the circle with its Axle.Front at the back instead of at the front, as a result of having bumped onto one of the obstacles. However, this appears not to be a problem, as the proximity sensor of the Axle.Front module could still be used to continue obstacle avoidance. Table B.1 shows a detailed view on the first simulation, it shows (a) the time each sequence took to complete and (b) the module communication statistics.

8.3.2 Four-wheeler obstacle avoidance

Java Simulation main-class: `FourWheelerSimulation.java`
### Table 8.5: Results of ten simulations of the Two-Wheeler Car (AVG deviation is 1 decimal)

<table>
<thead>
<tr>
<th>Sim.</th>
<th>Total</th>
<th>Avg per try</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>Bytes sent</td>
</tr>
<tr>
<td>1</td>
<td>94.7 s</td>
<td>12.5 KB</td>
</tr>
<tr>
<td>2</td>
<td>283.0 s</td>
<td>34.3 KB</td>
</tr>
<tr>
<td>3</td>
<td>44.8 s</td>
<td>5.7 KB</td>
</tr>
<tr>
<td>4</td>
<td>21.0 s</td>
<td>2.4 KB</td>
</tr>
<tr>
<td>5</td>
<td>286.1 s</td>
<td>31.4 KB</td>
</tr>
<tr>
<td>6</td>
<td>97.7 s</td>
<td>12.2 KB</td>
</tr>
<tr>
<td>7</td>
<td>340.6 s</td>
<td>45.8 KB</td>
</tr>
<tr>
<td>8</td>
<td>104.2 s</td>
<td>12.5 KB</td>
</tr>
<tr>
<td>9</td>
<td>119.8 s</td>
<td>15.5 KB</td>
</tr>
<tr>
<td>10</td>
<td>72.4 s</td>
<td>6.6 KB</td>
</tr>
<tr>
<td>AVG</td>
<td>146.43 s</td>
<td>17.89 KB</td>
</tr>
</tbody>
</table>

Table 8.6 shows the result of running the experiment ten times using the Four-Wheeler car. One out of ten simulations failed; the Four-Wheeler drove forwards onto an obstacle and felt on his back, therefore we restarted the simulation to get complete data. In all nine other simulations, the Four-Wheeler successfully escaped from the circle of obstacles. Table B.2 shows a detailed view on the first simulation of the experiment, it shows (a) the time each sequence took to complete and (b) the module communication statistics.

### Table 8.6: Results of ten simulations of the Four-Wheeler Car. In one of the simulations, the four-wheeler felt on its back and was not able to escape, hence the simulation was restarted (AVG deviation is 1 decimal)

<table>
<thead>
<tr>
<th>Sim.</th>
<th>Total</th>
<th>Avg per try</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>Bytes sent</td>
</tr>
<tr>
<td>1</td>
<td>181.9 s</td>
<td>40.9 KB</td>
</tr>
<tr>
<td>2</td>
<td>80.2 s</td>
<td>20.7 KB</td>
</tr>
<tr>
<td>3</td>
<td>73.3 s</td>
<td>23.7 KB</td>
</tr>
<tr>
<td>4</td>
<td>19.4 s</td>
<td>3.7 KB</td>
</tr>
<tr>
<td>5</td>
<td>164.8 s</td>
<td>32.7 KB</td>
</tr>
<tr>
<td>6</td>
<td>85.7 s</td>
<td>13.5 KB</td>
</tr>
<tr>
<td>7</td>
<td>198.1 s</td>
<td>40.7 KB</td>
</tr>
<tr>
<td>8</td>
<td>37.1 s</td>
<td>4.9 KB</td>
</tr>
<tr>
<td>9</td>
<td>230.9 s</td>
<td>50.0 KB</td>
</tr>
<tr>
<td>10</td>
<td>21.5 s</td>
<td>4.0 KB</td>
</tr>
<tr>
<td>AVG</td>
<td>109.29 s</td>
<td>23.48 KB</td>
</tr>
</tbody>
</table>
8.3.3 Analysis

For the Two-Wheeler 100% and for the Four-Wheeler 90% of the experiments successfully escaped from the circle of obstacles. At 10% of the experiments for the Four-Wheeler the Four-Wheeler felt on its back, as it drove onto an obstacle. However, neither the robot, neither its controller crashed, they were still functioning, only the car was not programmed to flip itself back on its wheels.

The PROX_TRIGGER constant in the MetaForma program is set to 0.1, thus fairly sensitive, to sense the obstacle quickly and avoid the car from bumping into the obstacle. The event handler that detects the obstacle is triggered in the Axle.Front module, and in order to stop the wheels we first have to propagate the new state that tells the wheels to stop. This costs time, and the runtime system also need to refill the module neighbor table at each state transition, i.e. needs time to wait for received packets.

It appears to be difficult in the USSR simulator to make a car robot turn in a specific angle, as wheels may slip on the ground when turning. But on the physical ATRON modules, the same problem happens. Therefore, we not rely on a specific angle, but turn for a fixed time, and drive again in forward direction until a new obstacle is encountered.

8.4 Cluster-flow devised by Christensen et al.

Java simulation main-class: ChristensenSimulation.java

This experiment tests the MetaForma implementation of cluster-flow as devised by Christensen et al. [29], described in Section 4.4 and implemented in Listing 5.9. The meta-walker can be applied on ensembles having a width of three modules. We experiment with two ensemble lengths, of seven and fifteen modules. The goal of this experiment is to start the simulation and determine how many meta-walkers can move forward in the ensemble. At the ensemble with fifteen modules length, meta-actions must be automatically executed in parallel. For the seven-modules length example, meta-actions are executed sequentially, as there is no space to execute meta-actions in parallel. We test symmetry preservation, sequence transition, the enabling and disabling of a region.

In Table 8.7 and Table 8.8 is shown for both simulations the time the simulation ran, the amount of bytes that was sent on average per module, and the amount of meta-walkers that were successfully moved from the back to the front of the structure. For both experiments, in Table B.3 and Table B.4 is a more detailed view of sequence and communication statistics shown of the first simulation of each experiment.

Although the state manager is robust against packet-loss on the communication channels, in order for the Walker.Right to know its neighboring modules at least a single packet needs to be received from the scaffold modules beneath it. Indeed, when executing the Strategy sequence, the Walker.Right module checks whether it has two or three modules beneath it (Figure 4.6). In the former case, it has to walk another step, while at the latter case it must lift itself down.

Packet loss is dependent on the level of connector misalignment, which is in its turn is increased by the Walker by wandering over the scaffold. Therefore, at a certain point, the
Table 8.7: Results of ten (Christensen et al. (2006) [29]) cluster-flow simulations on a 7x3 ensemble.

<table>
<thead>
<tr>
<th>Sim. #</th>
<th>Sim. time</th>
<th>Avg bytes sent / module</th>
<th>Lifts up</th>
<th>Avg Time / LiftDown</th>
<th>Avg Bytes / LiftDown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1110.6 s</td>
<td>153.9 KB</td>
<td>6</td>
<td>5</td>
<td>222.12 s / liftdown</td>
</tr>
<tr>
<td>2</td>
<td>1027.7 s</td>
<td>149.6 KB</td>
<td>6</td>
<td>5</td>
<td>205.54 s / liftdown</td>
</tr>
<tr>
<td>3</td>
<td>912.9 s</td>
<td>127.4 KB</td>
<td>5</td>
<td>4</td>
<td>228.23 s / liftdown</td>
</tr>
<tr>
<td>4</td>
<td>1040.8 s</td>
<td>189.5 KB</td>
<td>6</td>
<td>5</td>
<td>208.16 s / liftdown</td>
</tr>
<tr>
<td>5</td>
<td>1474.3 s</td>
<td>254.5 KB</td>
<td>8</td>
<td>7</td>
<td>210.61 s / liftdown</td>
</tr>
<tr>
<td>6</td>
<td>1583.8 s</td>
<td>190.2 KB</td>
<td>6</td>
<td>5</td>
<td>316.76 s / liftdown</td>
</tr>
<tr>
<td>7</td>
<td>1288.8 s</td>
<td>186.5 KB</td>
<td>9</td>
<td>8</td>
<td>161.10 s / liftdown</td>
</tr>
<tr>
<td>8</td>
<td>1516.7 s</td>
<td>213.3 KB</td>
<td>10</td>
<td>9</td>
<td>168.52 s / liftdown</td>
</tr>
<tr>
<td>9</td>
<td>1411.2 s</td>
<td>187.0 KB</td>
<td>8</td>
<td>7</td>
<td>201.60 s / liftdown</td>
</tr>
<tr>
<td>10</td>
<td>1473.5 s</td>
<td>187.6 KB</td>
<td>8</td>
<td>7</td>
<td>210.50 s / liftdown</td>
</tr>
<tr>
<td>AVG</td>
<td>1284.03 s</td>
<td>183.95 KB</td>
<td>7.2</td>
<td>6.2</td>
<td>213.31 s / liftdown</td>
</tr>
</tbody>
</table>

connector misalignment is that big that the communication-channels throughput becomes zero. In that case, the instruction that connects the walker to the scaffold cannot succeed any more, as it has no way of knowing which connector to use. During the walk, this does not have to be a problem, as two connectors can grip on the scaffold and only one is necessary. So if one of the two channels is blocked, it can use the remaining connector. However, when lifting the walker down, it needs to connect to the Lifter module, and without communication the walker can only wait for a packet. After some time, the consensus downgrade algorithm simply tries to continue with the next sequential action. In this specific case, it results in the walker to fall off. Figure 8.2 shows the connector misalignment problem and Figure 8.3 shows the fallen-off walker.

At module Walker.Right in the LiftDown sequence, at sequential action #1, packets from Lifter.Main were received also on connector 0 instead of only connector 3, as shown below:

1 $513.89$ #1 .addNeighbor ( Walker_Right , 0 ,NONE ,4,4) //Incorrect conn
2 $513.89$ #1 .addNeighbor ( Walker_Right , 3 ,NONE ,4,4) //Correct conn
3 .... (module id , connector , meta-part , meta id , region id)

This poisons the module neighbor table[9], hence as a result, it tries to connect to Walker.Right using connector 0, while the neighbor is situated at connector 3. Therefore it waits, until the walker behind it becomes blocked, and the simulation must be stopped.

The error is a result of crosstalk, in which packets are received by infra-red channels that are not aligned to the sender [Subsection 2.1.3]. Christensen et al. (2007) devised an algorithm to tackle this problem, and we assume that this algorithm is used on the physical modules [24].
Cluster-flow devised by Brandt et al.

![Simulation](image1)

(a) Simulation

<table>
<thead>
<tr>
<th>Sim. #</th>
<th>Sim. time</th>
<th>Avg bytes sent / module</th>
<th>Lifts up/down</th>
<th>Avg Time/LiftDown</th>
<th>Avg Bytes/LiftDown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>457.6 s</td>
<td>50.8 KB</td>
<td>4 2</td>
<td>228.80 s / liftdown</td>
<td>25.40 KB / liftdown</td>
</tr>
<tr>
<td>2</td>
<td>509.7 s</td>
<td>53.6 KB</td>
<td>4 2</td>
<td>254.85 s / liftdown</td>
<td>26.80 KB / liftdown</td>
</tr>
<tr>
<td>3</td>
<td>558.8 s</td>
<td>58.5 KB</td>
<td>4 2</td>
<td>279.40 s / liftdown</td>
<td>29.25 KB / liftdown</td>
</tr>
<tr>
<td>4</td>
<td>467.3 s</td>
<td>50.4 KB</td>
<td>4 2</td>
<td>233.65 s / liftdown</td>
<td>25.20 KB / liftdown</td>
</tr>
<tr>
<td>5</td>
<td>563.7 s</td>
<td>60.5 KB</td>
<td>5 2</td>
<td>281.85 s / liftdown</td>
<td>30.25 KB / liftdown</td>
</tr>
<tr>
<td>6</td>
<td>565.4 s</td>
<td>83.0 KB</td>
<td>5 3</td>
<td>188.47 s / liftdown</td>
<td>27.67 KB / liftdown</td>
</tr>
<tr>
<td>7</td>
<td>525.6 s</td>
<td>80.6 KB</td>
<td>5 3</td>
<td>175.20 s / liftdown</td>
<td>26.87 KB / liftdown</td>
</tr>
<tr>
<td>8</td>
<td>648.8 s</td>
<td>92.4 KB</td>
<td>5 3</td>
<td>216.27 s / liftdown</td>
<td>30.80 KB / liftdown</td>
</tr>
<tr>
<td>9</td>
<td>312.0 s</td>
<td>70.0 KB</td>
<td>3 1</td>
<td>311.99 s / liftdown</td>
<td>69.95 KB / liftdown</td>
</tr>
<tr>
<td>10</td>
<td>361.7 s</td>
<td>95.7 KB</td>
<td>4 2</td>
<td>180.87 s / liftdown</td>
<td>47.87 KB / liftdown</td>
</tr>
<tr>
<td>AVG</td>
<td>497.06 s</td>
<td>69.55 KB</td>
<td>4.3 2.2</td>
<td>235.13 s / liftdown</td>
<td>34.01 KB / liftdown</td>
</tr>
</tbody>
</table>

Table 8.8: Results of ten (Christensen et al. (2006)) cluster-flow simulations on a 15x3 ensemble.

Figure 8.2: After ten meta-walkers, connector alignment errors start to get non-negligible. As a result, communication channels become blocked or connectors simply refuse to connect.

Figure 8.3: Meta-Walker has fallen-down from the scaffold, as a result of connector misalignment errors.

8.5 Cluster-flow devised by Brandt et al.

Java simulation main-class: [BrandtSimulation.java](#)

This experiment tests the MetaForma implementation of cluster-flow as devised by Brandt et al. ([16]), described in Section 4.5 and implemented in Section A.1.

The goal of this experiment is to start the simulation and determine how many meta-modules...
can move in the specified direction in the strategy. We want to verify that multiple high-level strategies each can attain specific global behavior, while using the same motion and coordination implementation. Therefore, we conduct two experiments, the first one to realize cluster-flow in forward direction, and the second one in backwards-left direction. For the cluster-flow itself, both strategies use the same motion and coordination implementation.

### 8.5.1 Moving in forward direction

In Listing 5.20 is shown the MetaForma strategy that we use in this experiment. Figure 5.10 shows the initial state of the ensemble in the simulation, with the difference that at the start all meta-modules in the ensemble are situated in one straight line.

Table B.5 (a) shows that the simulation ran for 2889.9 seconds and nineteen successful meta-actions were executed, achieving a cluster-flow of 76 individual ATRON-modules in forward direction. At that time, a meta-module deadlock occurred, visualized in Figure 8.4. Meta-module D compromises C and E in order to move forward, but has no knowledge about C, that must move forward first in order to avoid the deadlock. It is a small chance that this happens, as a race condition must be satisfied, but it can occur.

The current implementation does not support the check for the existence of B, as the metaneighbor variables are assigned the appropriate values due to receiving packets, and once B has moved it cannot tell D any more that it has left. We describe a possible fix for this in Subsection 9.4.9.

![Figure 8.4: Deadlock: Meta-module D has compromised both C and E in order to move itself forward, not knowing that B needs C first.](image)

Table B.5 (b) shows the communication statistics. On average, each module sent 434.2 KB along its communication channels in 2888.9 s, resulting in an average communication load of 0.15 KB/s per module. The table shows that Discover, Consensus and MetaVarSync packets, with respectively 177 KB, 128.4 KB and 121.1 KB, are the highest load on the communication channels.

### 8.5.2 Moving in backwards-left direction

The strategy in Listing 8.2 is used to attain cluster-flow in backward-left direction. The regions in the strategy do not mutually exclude each other, as region 1 and 2 can ‘fire’ at the same time. In this experiment we explicitly use an ensemble of three meta-modules to check whether this does not result in failures. Table B.6 (a) shows that the FlipThrough(RIGHT, TOP) sequence
Assessment

precedes the FlipThrough(TOP_RIGHT). The conditions fire at the same time, but the regions with the highest id wins, hence one of them has to retract and wait for its turn.

When the ensemble executed the FlipThrough in (RIGHT_TOP)orientation, the final two rotations were executed in the wrong direction, meaning that they pulled the ensemble up while rotating. This means there is still an issue in the sequence orientation implementation of this sequence.

Table B.6(a) shows that 48 meta-actions were completed successfully in the 6535.5 seconds simulation, as a single meta-move moves 4 single modules it achieves a cluster-flow of 192 individual ATRON-modules in bottom-left direction. Table B.6(b) shows that on average 1024.8 KB of packets is sent per module, resulting in an average communication load of 0.16 KB/s per module.

```plaintext
sequence Strategy {
    when (receive (Packet p)) {
        meta.neighborHook (p);
    }

    do {
        meta.broadcastNeighbors();
        until (config.propagationTime);
    }

    do {// moving backward left
        try region FlipThrough(TOP_RIGHT) including Bottom excluding Left, Top;
        try region FlipOver(TOP_RIGHT) including Bottom, Left excluding Top;
        try region FlipThrough(RIGHT_TOP) including Left excluding Top, Bottom, BottomLeft;
        until (false);
    }
}
```

Listing 8.2: Strategy to move an ensemble in backward-left direction

Figure 8.5: Three meta-modules moving in backward-left direction

8.6 Assessment

The knowledge that we gained by running these experiments is as follows. First, the MetaForma implementation of the Eight2Car sequence on average uses less communication compared to the DynaRole version, and also has on average a longer completion time. We cannot make a hard claim whether the MetaForma implementation exceeds the 1.2 KB/s per module. However, as the DynaRole implementation has been demonstrated to work on the physical ATRON modules [65], we believe the capacity of the communication channels of the physical ATRON modules to be sufficient for the MetaForma implementation of the Eight2Car sequence.

Second, the MetaForma language can be used to implement obstacle avoidance, but the algorithms are still primitive: There is no support for driving around an obstacle and afterwards continue in the same direction, only driving in a new direction is supported.
8. Experiments

Third, the MetaForma language can be used to implement various methods for cluster flow. We showed that in the implementation of cluster-flow devised by Christensen et al. [29] walkers can work in parallel. This is facilitated by regions [15], as the average of the simulation time divided by the amount walkers that were lifted down is only slightly higher on a 15x3 scaffold compared to a 7x3 scaffold. Ideally the values should be equal, but at the end of the simulation, walkers are still on the scaffold without being lifted down and thus without being counted.

Ideally, all packet statistics and running times should have been logged with at least 3 decimals, to reduce deviation errors. In addition, the packet statistics are shown on average per ensemble, instead of the max per module.

8.7 Summary

We have conducted experiments on the MetaForma implementation of the Eight2Car sequence, and compared this with the Eight2Car sequence implementation in DynaRole. The time to complete the sequence were not significantly different in both implementations, in contrast the communication statistics. Although the MetaForma implementation requires a higher communication bandwidth in several orders of magnitude, this does not exceed the capacity of the physical ATRON modules. We showed that MetaForma sequences do not require pre-programmed module ids and modules to be in symmetry prior to execution.

We implemented high-level obstacle-avoidance behavior in a strategy, and showed that the same behavior could be used by both a Two-Wheeler and Four-Wheeler car, while having a specific implementation for driving and turning around. In the experiment, the car was positioned in a circle of obstacles with a randomly-placed gap. The two-wheeler managed to escape in all ten simulations, the four-wheeler in only nine; in one simulation it fell on its back. We argued this is not a bug in MetaForma and can be solved in the implementation of the obstacle-avoidance algorithm.

The implementation of Christensen et al. [29] cluster-flow was tested to attain walk-behavior in forward direction on both a 7x3 and a 15x3 scaffold ensemble. On the latter, multiple walkers were able to walk in parallel. As the implementation assumes no global knowledge, both scenarios could use an identical implementation of the cluster-flow algorithm. A cluster-flow up to ten walkers, hence 30 single modules, was achieved.

The implementation of Brandt et al. [16] cluster-flow was tested to run in forward direction, showing that meta-moves can be executed in parallel, and although in only rare cases, the meta-module grid can become deadlocked. The cluster-flow in downward-left direction showed that even conflicting regions can be used in a strategy, in which one region waits for the other to complete.

All experiments showed not to exceed the maximum communication load of 1.2 KB per second and respected the limited actuator strength of lifting maximum two modules against gravity, with the exception of the issue in the TOP_RIGHT sequence orientation [12] in the Flip-Through sequence. In addition, the passiveness of the female connectors at the physical ATRON module is respected, as described in Section 7.6.

---

2E.g. by lowering the PROX_TRIGGER value, this will cause the car to stop and turn more early when driving to an obstacle.
Chapter 9

Perspectives

In this chapter we first present related work. Thereafter, we evaluate the MetaForma language, and answer the research questions that were posed in Section 1.4. Finally, we conclude the thesis.

9.1 Related Work

9.1.1 Learning Systems

Christensen et al. (2006) showed that ATRON modules can build a tower structure by using a meta-module approach controlled by an artificial neural network (ANN) approach [28]. It comprises three neural networks, in which each ANN has the reachable space of the meta-module as input. The ANNs compute a decision whether a meta-module should emerge, when it should be stopped, and which fitness value every reachable self-reconfiguration path has. The latter is influenced by attraction points that attracts modules in the right direction (Figure 1.4).

Experiments on this approach [23] by using a simple transition-based simulator show that neural networks can be used as an automatic programming approach to program the ATRON self-reconfigurable robot.

It was shown by Kamimura et al. (2003) [39] that various types of M-TRAN walker configurations can learn a locomotion pattern to move in a chosen direction. A 4-leg and 6-leg walker and a caterpillar were tested. The computation of the locomotion pattern is executed during simulation, and afterwards the pattern is transferred to the M-TRAN modules. Each module uses a Genetic Algorithm (GA) to optimize the walk regarding minimizing power consumption versus maximizing walking distance.

In addition, it was shown by Kurokawa et al. [47] that large ensembles of M-TRAN modules can be split, in which separated walkers are generated. Hence, these separated walkers can move individually to their destination, where they are merged back in a large ensemble.

1Meta-modules in this approach can get deadlocked when used for building three-dimensional structures [15]
9. Perspectives

9.1.2 Rule-based control

Manually implemented rules  Brandt et al. (2004) presents a rule-based control approach to move modules from the rear-end of an ensemble to the front. Manually written, local rules were applied to each module in a scaffold of simulated ATRON modules. Rules were written for (a) disassembling the rear-end, (b) moving the freed modules forward and (c) assembling the front end.

Three types of rules were considered, from specific Point-to-Point rules to more general Wild-Card rules, to even more general Java Rules. It was shown that the more general Java Rules reduced the amount of rules necessary to attain movement, and they also resulted in a faster global behavior compared to specific rules.

Programming by Demonstration  Østergaard et al. (2004) used a rule-based control approach (Section 1.2.1) to program behavior into the ATRON robot using a simple transition-based robot simulator. By selecting a module and moving it to a new destination, a precondition → action rule is generated at the module that detects a change in the rotation of its hemispheres. The pre-condition consists of its connectivity to neighbors before the action, and the action contains the amount of degrees that it was rotated.

After repeating this process many times, a collection of rules is created that eventually will be shared among all modules in the robot. Each module matches the preconditions in this collection to its own connectivity to other modules. As soon as a module finds a match, the rules fires and the action is executed, i.e. the module rotates to the specified degree.

Programming using this approach is a tedious job, as many rules have to be created in order to attain global behavior [59]. In addition, a program is stored as a collection of non-transparent rules, such that it is difficult to change the behavior afterwards.

9.1.3 Spatial programming

The Mechatronics Modelling Language (M3L) is an approach to model the geometrical description and connector topology of modular robots. It tackles the issue that different types of modular robots have different connector topologies, different degrees of freedom and different spatial properties. M3L supports modelling various modular robots including the ATRON and M-TRAN. When M3L is used in combination with spatial labels, it allows morphology-independent programming of modular robots.

If applied to the ATRON robot, one hemisphere is assigned as the root or origin. Thereafter, hemispheres connected to the root compute their placement in the ensemble based on the origin hemisphere. This process is repeated until all hemispheres have updated its compass direction with respect to the origin.

9.1.4 DSL approaches

The following DSL approaches for modular robots were described in Chapter 3.

PARSL focusses on locomotion gaits at whole-robot level for chain-type modular robots. The logic programming language Meld and the declarative language LDP target large-scale ensembles of programmable matter in a highly redundant context. Functional languages
such as Regiment \[56\] and Proto \[7\] target large-scale ensembles of stationary modules and are less focussed on geometry and actuation.

DynaRole was created to support both behavioral patterns such as driving a car, and self-reconfiguration to transform between different robot configurations. The design of DynaRoles external DSL \[12\] and Python-based internal DSL \[53\] allows the programmer to work at a modular level in terms of roles and at a whole-robot level in terms of sequences \[65\].

### 9.2 Evaluation

In this section we evaluate the MetaForma language using the DSL evaluation criteria by Visser (2008) \[76\].

#### Expressivity

In general, as a result of the abstraction for module groups\[8\], sequences can be coded concisely in MetaForma, in comparison with DynaRole sequences \[65\]. In addition, strategies\[13\] allow controller programs to be concisely written using libraries of low-level behavior.

On the contrary, some parts of the low-level behavior implemented in MetaForma is still complex to understand:

- The definition on which the runtime system of a module belongs to a specific border in the ensemble (Listing 5.12).
- The algorithm to divide an ensemble in a grid of meta-modules (Listing 5.8 and Listing 5.18).
- The Neighbor Operator (Listing 5.19) allows multiple conditions only to be conjuncted, meaning that when the programmer needs a disjunction he has to use multiple neighbor operators (Section 5.4.1). Therefore, expressions to address specific modules can become long-winded, as we see in the borders section of Listing A.1 (line 105 t/m 108).
- The choice to implement meta-neighbors using meta-module variables\[14\] the DSL, results in the need for many conditional statements, to as shown in Listing 9.1.
- It is difficult to reuse behavior, sequences cannot easily be extended by other sequences as a result of the sequential flow of actions. For example, it is difficult to re-use the implementation for the 3-module cluster-flow of Christensen et al. \[29\] (Listing 5.8), to implement cluster-flow for a 4-module walker, while their solutions are related.

#### Coverage

The MetaForma language supports obstacle avoidance algorithms for cars, shape-transformation sequences, and horizontal cluster-flow using meta-modules. Vertical cluster flow, e.g. for building towers is only supported to a limited degree, as a means to preserve symmetry in 3D-lattice structures is lacking. This is mainly because modules have no connector sets\[2\] for Up and Down (Section 5.4.1), to simplify symmetry preservation\[10\]. In order to also support
3D-lattices, a revision of the connector set and the incorporation of Kinematic Configurations [13] in the symmetry preservation algorithm is considered future work (Subsection 9.4.3).

Docking of an ensembles onto another ensemble is also not supported by the language (Subsection 1.1.2).

**Completeness**  The MetaForma language implementation is complete from the point of view that it is not necessary to write additional external code, except for a simulation class to run the generated controller in the USSR. The language has been designed to not use any external code, as this would add complexities when multiple backends are used.

Furthermore, it is possible to add a debug section to MetaForma programs (Listing A.1) to provide a means for debugging in the USSR.

**Portability**  Currently, the MetaForma compiler backend compiles to Java and only targets the USSR simulator. An ANSI-C backend to target the physical modules has been postponed to future work (Subsection 9.4.1) and therefore no hard claims can be made about the quality of the MetaForma abstractions.

**Code quality**  Experiments have shown (Chapter 8) that the generated code by the MetaForma compiler is syntactically correct and working.

**Maintainability**  A DSL solution such as MetaForma is in comparison to using only a GPL approach, an extra software system that must be maintained. However, most of the abstractions that MetaForma offer are implemented in the MetaForma runtime system[18] (Chapter 7) and are static, and the DSL only provides an additional access layer to it. Moreover, a GPL solution does not subduct the need for a runtime system, hence the level of maintainability is not significantly increased.

### 9.3 Conclusions

This thesis presents the design and implementation of MetaForma, a DSL for the self-reconfigurable ATRON robot.

**RQ 1:** How can programming modular self-reconfigurable robots, and the ATRON robot in particular, be made easier and more accessible to the user?

Modular self-reconfigurable robots can be programmed by using an automatic or manual programming system, as described in Chapter 1. The former approach varies from e.g. learning, demonstration or instruction systems, and the latter approach comprises GPL and DSL approaches.

A DSL solution allows the programmer to program at a higher and more productive level using domain-specific abstractions, as the programmer no longer needs to tackle difficult issues typical for self-reconfigurable robots and can work on a more productive compared to a GPL
approach. Textual DSLs provide flexibility compared with graphical DSLs, while a graphical DSL can provide a higher level of abstraction.

It is difficult to claim that a DSL solution makes programming modular self-reconfigurable robots easier in any scenario. However, to tackle problems with a known optimal solution, such as cluster-flow, and shape-transformations between a car, snake and eight ensemble, a DSL approach is a fruitful choice. As it is already known how the sequences of actions that are needed looks like, we know how to implement solutions to these problems.

On the contrary, in unknown domains like building large-scale structures such as towers [28], AI approaches (or automatic programming systems in general) rather come to mind, as we do not know a straightforward solution yet to solve these problems.

**RQ 2:** What DSLs currently exist for programming robotics, what are their strong points, and what are these solutions lacking?

Chapter 3 describes the following DSLs used for programming robotics. UrbiScript [4] has interesting features in terms of concurrency, but is not dedicated to expressing behavior at a whole-robot level that we would like to attain on modular robots. PARSL [83] focusses on locomotion gaits at whole-robot level for chain-type modular robots, but this language does not support self-reconfiguration.

The logic programming language Meld [3] and the declarative language LDP [61] target large-scale ensembles of programmable matter in a highly redundant context, but both languages lack support for reliable self-reconfiguration.

Functional languages such as Regiment [56] and Proto [7] target large-scale ensembles of stationary modules and are less focussed on geometry and actuation, hence it is difficult to program a sequential list of actions in these languages.

DynaRole was created to support both behavioral patterns such as driving a car, and self-reconfiguration to transform between different robot configurations. The design of the DynaRole external DSL [12] allows the programmer to work at a modular level in terms of roles and at whole-robot level in terms of sequences [65]. However, these two aspects of the language are poorly integrated and were in each case designed to solve problems specific to certain scenarios.

**RQ 3:** What kind of abstractions are needed in the new DSL? What abstractions should be be integrated in the language, and what abstractions should be expressed in the DSL itself?

Fundamentally, we need a high-level strategy language incorporating strategies [13] to implement high-level, reusable control behavior for a robot, and a motion and coordination language incorporating low-level behavior. Moreover, we need support for small-scale and large-scale ensembles, and support to program behavior at meta-module [7] and region [15] granularity.

In Chapter 4 is defined a list of abstractions that is integrated in MetaForma. We distinguish between automatic abstractions, which are integrated in MetaForma, and manual abstractions, which can be implemented using the MetaForma language.
Automatic abstractions  First, we need sequences to implement lists of distributed, sequential actions. To enable sequences to be executed in different directions, sequence orientations are needed. Second, we need a module neighbor table to translate module identification to connector numbers. This supports sequences and enables modules to be identified in the ensemble based on its relative placement to neighbors. To help the programmer in identifying connectors, connector sets and symmetry preservation are needed.

To enable expressing problems at higher granularity, a meta-module is needed, and a meta part to distinguish between individual modules in meta-module. We need meta-module variables variables that are automatically synchronized among all modules in the same meta-module. We need a means to specify region borders, that can be related to the origin for gradients. In order to group meta-modules in an isolated area, we need regions.

We need a communication system to communicate with neighbors, and event handling to facilitate asynchronous communication, which can also be used to process specific conditions.

We need an abstraction to integrate debugging, which is implemented using a debug section. The programmer can assign colors to module ids, module groups and sequences, such that modules in the simulator are colored according to their properties.

Manual abstractions  We need an abstraction to create meta-modules manually, which enables experimenting with different kinds of meta-modules instead of sticking to a single existing approach. Hence, meta-neighbors are manually implemented using meta-module variables, allowing neighbors not to be fixed. We need gradients, which are implemented manually using variables and the communication system, to enable support for different gradient patterns.

RQ 4: What kind of runtime system is needed to provide the desired properties of being scalable and robust, and how do these properties relate to the DSL?

We need a runtime system that provides the previously defined abstractions at whole-robot level for the DSL, while it is running on a distributed topology of multiple modules.

To realize scalability, the runtime system must use decentralized control and provide a means to divide an ensemble in multiple groups of modules that can operate independently from each other. MetaForma supports modules to be grouped in a region, such that each group can communicate without the need to have knowledge about modules in other groups.

In order to attain robustness, the MetaForma runtime system must provide abstractions to deal with packet loss and partial hardware failures. The state manager realizes these properties by using a state merge function and a consensus degradation algorithm, respectively.

Moreover, the implementation of the runtime system respects hardware constraints of the physical ATRON modules.

RQ 5: How can meta-modules be used to simplify the process of self-reconfiguration on the ATRON robot, and what kind of meta-modules are used until now?

A meta-module has a higher degree of freedom compared to a single module, resulting in
Conclusions

Reduced motion constraints and a reduced complexity in motion planning, hence simplifying the problem of self-reconfiguration (Subsection 4.4.1).

First, Christensen et al. [29] investigated meta-modules in automatic programming systems, such as AI approaches [28] (Section 4.4). Second, Brandt et al. [16] devised a new metamodule approach for the ATRON robot (Section 4.5). This approach was found to be useful by conducting experiments in a transition-based simulator, but has not been incorporated in any language before.

RQ 6: What is needed to make the MetaForma language applicable to other types of modular robots?

Modular robots exist in different hardware designs, and differ in terms of dimension (two- or three-dimensional), degrees of freedom (typically one, two or three degrees), number of connectors (active and passive), method of actuation (perpendicular, parallel or rotational) and method to neighbor attachment (typically mechanical) [72].

Currently, the ATRON robot is the only target platform that MetaForma supports, as the language assumes each module to have two rotational hemispheres, a single degree of freedom, and four male and four female connectors with each of them a communication channel (Subsection 2.1.3). In theory, every self-reconfigurable robot that satisfies these constraints can be programmed using MetaForma. However, as the ATRON is the only robot satisfying these constraints, we discuss a number of requirements that is needed for a generalization of the domain to other types of modular robots: In order to adopt MetaForma to other types of modular robots, we need the following:

First, we need appropriate hardware control primitives for the actuators and connectors, by adding extra primitives that control the different actuators and connectors, or by introducing parameters for it. In order to support robots with two degrees of freedom, e.g. the M-TRAN [45], the language needs a means to control the second actuator.

Second, and more importantly, a revision of the connector sets is needed such that they match the connectors of the modules in the target robot. Moreover, in order to attain symmetry preservation, it might be beneficial to integrate a plugin system in MetaForma, such that specialized language plugins are used for specific types of modular robots.

Both aspects need to be incorporated in the MetaForma language (Chapter 5), compiler (Chapter 6) and runtime system (Chapter 7).

The Molecube [85] (Figure 1.8) robot shares the following similarities with the ATRON robot: three-dimensional, one degree of freedom, and an actuator angle of 45°. The difference is that Molecule modules only have two connectors, which are both active. In order to generalize MetaForma to another self-reconfigurable robot, the Molecule is a likely choice, as a result of its similarities with the ATRON robot.

The M-TRAN [45] (Figure 1.9) robot also has similarities with the ATRON, however, it also has important differences: It has a second degree of freedom, six connectors and a different lattice structure. As a result, it will be harder to generalize MetaForma to support the M-TRAN. In particular, more than one degree of freedom per module makes symmetry preservation
9. Perspectives

more difficult.

**RQ 7:** *Can existing scenarios on modular robots be resolved more efficiently in terms of development using the MetaForma language?*

First, when using DynaRole sequences [65] the programmer needs knowledge about which connector number to use for connect and disconnect actions (Listing 3.6). Furthermore, it assumes the ensemble to be in symmetry at the start, and each module to have a pre-programmed hardware id. In MetaForma, the Eight2Car (Listing A.4) sequence can be implemented more high-level and efficient, as the programmer does not need knowledge about connector numbers. Moreover, initial symmetry and pre-programmed module ids are no longer required, with the exception of one module that acts as a point of reference for the symmetry.

Second, DSLs used to implement obstacle avoidance algorithms [12, 66] for the ATRON robot lack a means to generalize global behavior that can work on different car ensembles. MetaForma allows a single library of low-level behaviors to be reused by multiple concise high-level behaviors (Section 4.3).

Third, in order to attain locomotion by using cluster-flow, rule-based control has been used on ensembles [19, 59] in a transition-based simulator. Without support for physics in the simulation, the speed and robustness of this approach is difficult to compare with the MetaForma implementation that was tested in the USSR simulator [25]. However, we can compare both approaches in the way of how programs are represented: Rule-based control uses many non-transparent rules that are difficult to modify afterwards, while the MetaForma cluster-flow implementation (Listing 5.9) has a more intuitive representation of its behavior.

To sum up, it is not possible to argue that all scenarios for the ATRON robot can be resolved more efficiently, as only a subset was subject to conducting experiments in the USSR simulator. However, we showed that by using MetaForma (a) obstacle-avoidance for the Two-Wheeler and Four-Wheeler, (b) cluster-flow by using meta-modules, and (c) the Eight2Car self-reconfiguration sequence can be resolved more efficiently in terms of development.

9.4 Future work

In this section we present interesting ideas for future work.

9.4.1 Support for Physical ATRON Modules

Currently, the MetaForma compiler has only a backend for Java to be used in the USSR [25] simulator. In order to port the MetaForma language to the physical ATRON modules, an ANSI-C backend must be added to the MetaForma compiler, and the runtime system must be ported from Java to ANSI-C. This migration can be done in two stages by incorporating the ASE framework [27] (Subsection 2.2.3).

First, the ANSI-C backend must be added to the compiler, that incorporates the ASE framework in its generated controllers. The generated ANSI-C controllers can be tested in the USSR by using the Java-implementation of the MetaForma runtime system.
Second, after testing the ANSI-C backend, the MetaForma runtime system can be ported to an ANSI-C implementation as well, such that it can also run on the physical ATRON modules.

After a migration of both the MetaForma backend (Chapter 6) and the MetaForma runtime system (Chapter 7), the programmer can experiment with new MetaForma controller programs in the USSR, and afterwards they can directly be transferred to the physical ATRON modules. As an ANSI-C implementation of the MetaForma runtime system can both be used on the physical modules and the USSR, only one ANSI-C implementation must be maintained.

9.4.2 Automatic programming system for Motion and Coordination language

The Motion and Coordination language can be extended with an automatic programming system (Subsection 1.2.1). By using an instructive system, the programmer can instruct the ensemble on how meta-modules should be formed, such that complex low-level parts are automatically generated. As the algorithm for creating a grid of meta-modules (Listing 5.18) is still complex, and clicking the communication path through a module grid is more a more intuitive approach.

In addition, the low-level meta-actions like the Flip-Over, Flip-Along and Flip-Through (Section 4.5) could be generated by an instructive system. Once the low-level implementation is generated, the MetaForma strategy language can be used to assign them to regions that match the rules in the strategy. An advantage of an instructive system is that low-level implementation can be generated, including connector numbers, but as a result different orientations of the same sequence must probably be generated individually.

Automatic generation of low-level behavior The MetaForma runtime system can be extended such that while running a sequence it generates code for low-level DynaRole sequences [65]. Using this approach, the programmer can implement high-level behavior in MetaForma code and execute this in the USSR simulator. While exethe planning algorithm, which now is executed at runtime, can . The low-level code can hence be loaded into physical robot

9.4.3 3D symmetry preservation

In Figure 4.1 are the connector sets shown that were conceptually defined. Currently, MetaForma has connector sets implemented only for NORTH, SOUTH, EAST, WEST, MALE and FEMALE (Figure 5.4). The connector sets for UP and DOWN are not implemented in order to simplify symmetry preservation. As a result, self-reconfiguration in 3D-lattices is only supported to a limited degree.

In terms of future work, incorporating Kinematic Configurations [13] in the MetaForma runtime system allows each module to have a 3D compass that can automatically be updated according to one source hemisphere. Consecutively, each module can derive appropriate connector sets from this compass.
It is worth noting that the symmetry can likely not be preserved on the fly, i.e. during every rotational move, as the module neighbor tables need to update their connector numbers as well. This will require a synchronisation algorithm.

### 9.4.4 Sophisticated Gradient Protocol

In the current implementation, gradients are propagated in all directions from one or more source modules, and each module takes the minimum value of the received gradient value increased by one, and its own gradient value. Spatial information is not included in the propagation, hence it cannot be detected whether all modules have received the correct gradient value. At the moment, this is dealt with by continuously propagating gradients for a fixed time, until eventually one of the packets gets through. It can happen that the propagation time is insufficient to propagate correctly and deflections will occur. If some of the infra-red communication channel are completely broken, it will result in a deflected gradient, which presumably results in incorrect module identification.

However, in theory it is possible to assign correct gradient values based on sending out specific values over each connector. In this case, gradient deflections cannot happen as either a module receives a correct gradient value, or it does not receive a gradient value and the runtime system can detect this using the consensus algorithm (Listing 7.5). This causes gradients to be immune against malfunctioning connectors.

In [Figure 9.1](#) is shown how (a) the current implementation performs on an ensemble having one completely malfunctioning connector, and (b) how an algorithm that takes spatial information into account would perform. In the former and current approach, the top-left module has no way of knowing whether it is situated only one module away from the left-bottom module, as the communication path between them is broken. In the latter approach, the top-right module tells the top-left module that it is at its right side, resulting in a correct gradient.

![Figure 9.1: Propagation of gradient values](image)

(a) In the current implementation, a completely malfunctioning communication path results in a deflected gradient

(b) Improved implementation

### 9.4.5 Algorithm for Parallel Lifting

In self-reconfiguration, sometimes more than two modules need to be lifted against gravity in one move, e.g. in the Flip-Over meta-action. In this move, three individual modules need to
be lifted against gravity at the same time. However, ATRON modules have limited actuator strength and one module can only lift two other modules against gravity (Subsection 2.1.3). Hence, two modules must work together in the actuation to facilitate the lifting move, referred to as parallel lifting.

Currently, parallel lifting is supported by MetaForma by simply commanding two modules to rotate by using two `rotate` calls in one sequential action. State diffusion facilitates the parallel behavior: As soon as a module receives a new state which tells it to rotate, it starts immediately. However, as a result of packet loss, not all modules will receive a state update at exactly the same time. Hence, one module might start earlier with rotating than the other.

In the simulator, this has never shown to be an issue, as when a time difference occurs the first module starts slightly wringing, until eventually the second module joins and from that point the lift continues smoothly. However, for the actuators on the physical modules it might be a problem.

In that case a distributed countdown state-sharing algorithm is needed, that propagates the new state update for a certain amount of time before actually advancing to the next state. The propagated packets include a counter, thus every module that receives at least one of the updates will advance to the next state at a pseudo-equal time\(^2\), hence tackling the problem of packet loss.

### 9.4.6 Collision Detection and Anticipation

Collisions between modules can occur during self-reconfiguration, e.g. in the meta-module approach devised by Christensen et al. [29], a walker can collide with the walker in front of it. Currently, MetaForma does not support a means to detect actuators that become blocked in their rotation, which happens in a collision. Moreover, MetaForma does not support a means to deal with a collision if it was able to detected it (and currently, neither does the USSR simulator).

It was shown by Christensen (2007) [23] that physical ATRON modules are capable of rolling back their action when colliding. As a result, the combination of trying certain moves and rolling-back in case of a collision becomes powerful, it allows meta-modules to find their way around unknown obstacles in their environment. It also allows multiple meta-modules to exist in parallel, without needing knowledge about neighbors.

An interesting project is to extend MetaForma with something like a `try { sequence1 } onCollision(){ sequence2 }` construct, in which the programmer has a means to express behavior to react in case of a collision. It e.g. first executes actions in `sequence1`, and if a collision occurs, it reverses and re-executes the finished actions until the start of `sequence1`, and then continues with the actions in `sequence2`. Also, in case the `onCollision()` clause is omitted, it can reverse the program to the start of `sequence1`, and start the same sequence of moves again. This can be iteratively executed, until one sequence succeeds and the Strategy is given back control.

### 9.4.7 Multiple Module Lookahead and Custom Variables in Neighbor Table

An interesting project in terms of future work would be to extend the `module neighbor table`\(^9\) such that it can store information about indirect neighbors. This increases the possibilities

\(^2\)Two clocks will never be equal, thus, with pseudo-equal we mean an insignificant difference in time
9. Perspectives

for modules to identify itself based on its relative position in the ensemble. Currently, the MetaForma runtime system provides modules to have only direct neighbor visibility, and this limits the possibilities to identify modules based on their relative position in the ensemble.

Moreover, currently the neighbor table only stores fixed variables for each neighbor, such as the module id, module group, meta id and region id. If the programmer can specify custom variables to be propagated and stored in the neighbor table, such as gradients, this becomes useful in cluster-flow scenarios. As the meta-module can ‘read’ the gradient value of the modules it walks on in the scaffold, it can follow the gradient to enable walking in a specific direction.

9.4.8 Better debugging support

**Debugging window** Currently, the USSR simulator has a debugging window that is shown when clicked on the module. This window shows all variables from the module, and a log showing the execution path and communication statistics, which is valuable information for debugging. Currently, in order to show information we are looking for, specific for the problem we are trying to debug, there is a MetaForma construct to specify a list of packet types we want to show information from. For example, we can choose to only print Region packets such we can quickly trace bugs specific to enabling and disabling regions.

However, the decision whether a packet send/receive action is logged, is now made at compile time. If we narrowed down the problem, but after that need information that is not logged, we have to restart the simulation with a bigger list of packet types to show. This slows down debugging significantly. Hence, a possibility to specify what is shown in the log at runtime, such as which packets, actions, and variable changes, would speed up debugging.

**Module Coloring** Currently, modules can be assigned two colors in the USSR, one for each hemisphere. Two colors are fairly limited for the information that needs to be represented for a module: The module id, meta id, region id, meta part, active sequence, while not even mentioning gradient values, consensus progress and symmetry corrections. A solution could be a number of colored rings along a module, for example a ring representing a module to belong to a certain region, where the ring is colored according to the modules region id. Also, the possibility to print a number on top of it will increase the speed of debugging.

9.4.9 Improved meta-neighbor system

Up-to-date knowledge about neighbors is crucial when making decisions in a strategy. In case a strategy makes a decision on outdated neighbor information, an incorrect meta-action will be started, and meta-modules might wrench into each other.

During the development of MetaForma, the neighborHook function to implement meta-neighbors for the approach devised by Brandt [16] became complicated, as a result of various controller modifications as shown in Listing 9.1.

First, the function needs to check whether the connector on which the packet is received is connected (line 2). Second, it needs to check whether the neighbor, stored as meta-module variable, differs from the received meta id (line 4). In that case the specific neighbor variable has to be propagated to other meta-neighbors such that they can update their indirect
neighbor values. Currently, all these conditions resulted in additional checks in the language, as shown below:

```
void neighborHook (Packet p) {
    if (#(in:p.sourceID, connected:true, inRegion:false) == 1) { // Connected?
        if (p.metaID != module.metaID && module.metaID != 0 && p.metaID != 0){
            if (p.connOwn==5 || p.connOwn==6) && meta.Right != p.metaID) {
                meta.Right = p.metaID; broadcastNeighbors();
            // three other direct neighbors omitted
        }
    }
}
```

Listing 9.1: Fragment of the neighborHook function to detect meta neighbors
Bibliography


Appendix A

MetaForma DSL programs

A.1 Cluster-flow devised by Brandt et al.

```java
package Brandt;

debug {
    visualize {
        north {
            @Clover.North: white;
            @Clover.South: black;
            @Clover.West: yellow;
            @Clover.East: green;
            @OutsideLifter: cyan;
            @InsideLifter: magenta;
            @Outside: blue;
            @Inside: red;
            @Lifter.Top: yellow;
            @Lifter.Bottom: magenta;
            @Lifter.Left: green;
            @Lifter.Right: yellow;
            @Dummy.Left: white;
            @Dummy.Right: black;
            @X: blue;
        }
    } south {
        Init: white;
        Strategy: red;
        FlipOver: yellow;
        FlipThrough: cyan;
        FlipAlong: magenta;
    }
}

packet AssignMetaID {
    byte newID, counter;
}
```
packet AddNeighbor {
    byte first, second;
}

packet Gradient {
    byte pri, sec;
}

when (receive (AddNeighbor p)) {
    if (p.metaID != 0) {
        if (p.metaID == meta.Left) {
            meta.TopLeft = p.first;
            meta.BottomLeft = p.second;
        }
        if (p.metaID == meta.Right) {
            meta.TopRight = p.first;
            meta.BottomRight = p.second;
        }
        if (p.metaID == meta.Top) {
            meta.TopLeft = p.first;
            meta.TopRight = p.second;
        }
        if (p.metaID == meta.Bottom) {
            meta.BottomLeft = p.first;
            meta.BottomRight = p.second;
        }
    }
}

when (receive (Gradient p)) {
    if (p.pri + 1 < module.gradPri || p.sec + 1 < module.gradSec) {
        module.gradPri = min (p.pri + 1, module.gradPri);
        module.gradSec = min (p.sec + 1, module.gradSec);
        module.gradientPropagate();
    }
}

meta {
    parts [North, West, South, East];

    void neighborHook (Packet p) { // added p.connected == false
        if (#(in: p.sourceID, connected: true, inRegion: false) == 1 && p.metaID != module.metaID && module.metaID != 0 && p.metaID != 0) { // also check for p.metaID != 0, as when nb is at init state (thus metaID == 0) it is better to keep old value
            if ((p.connOwn==5 || p.connOwn==6) && meta.Right != p.metaID) {
                meta.Right = p.metaID; broadcastNeighbors();
            }
            if ((p.connOwn==2 || p.connOwn==7) && meta.Top != p.metaID) {
                meta.Top = p.metaID; broadcastNeighbors();
            }
            if ((p.connOwn==0 || p.connOwn==3) && meta.Left != p.metaID) {
                meta.Left = p.metaID; broadcastNeighbors();
            }
        }
    }
}
if (p.connOwn==1 || p.connOwn==4) && meta.Bottom != p.metaID) {
    meta.Bottom = p.metaID; broadcastNeighbors();
}

void broadcastNeighbors () {
    module.discover (); // we need to do this explicitly here!
    send AddNeighbor(first=Left, second=Right) to $(metaID: [Top, Bottom], inRegion:false);
    send AddNeighbor(first=Top, second=Bottom) to $(metaID: [Left, Right], inRegion:false);
}

module {
    borders {
        LEFT: #(connOwn: EAST&MALE) == 2 && #(connOwn: WEST&NORTH&MALE) == 0 ||
                #(connOwn: WEST&FEMALE) == 2 && #(connOwn: EAST&NORTH&FEMALE) == 0;
        RIGHT: #(connOwn: WEST&MALE) == 2 && #(connOwn: EAST&SOUTH&MALE) == 0 ||
                  #(connOwn: EAST&FEMALE) == 2 && #(connOwn: WEST&SOUTH&FEMALE) == 0;
        TOP: #(connOwn: WEST&MALE) == 2 && #(connOwn: EAST&NORTH&MALE) == 0 ||
               #(connOwn: WEST&FEMALE) == 2 && #(connOwn: EAST&SOUTH&FEMALE) == 0;
        BOTTOM: #(connOwn: EAST&MALE) == 2 && #(connOwn: WEST&SOUTH&MALE) == 0 ||
                  #(connOwn: EAST&FEMALE) == 2 && #(connOwn: WEST&NORTH&FEMALE) == 0;
    }
}

byte gradPri, gradSec;
boolean isRef;
int QUART, HALF, EIGHT;

void gradientPropagate () {
    if (module.atBorder(orient.primary)) gradPri=0;
    if (module.atBorder(orient.secondary)) gradSec=0;
    send Gradient(pri=gradPri, sec=gradSec);
}

void gradientInit () {
    gradPri = MAX_BYTE;
    gradSec = MAX_BYTE;
    module.storeID();
}

// Init and FlipThrough sequences and the Strategy are omitted

Listing A.1: Implementation for cluster-flow devised by Brandt [16]
A. MetaForma DSL programs

Listing A.2: The FlipAlong sequence implementation

group InsideLifter [Top, Bottom]
group OutsideLifter [Top, Bottom]
group Inside

group Outside

sequence FlipOver {
  if (TOP_LEFT, LEFT_TOP, RIGHT_TOP, TOP_RIGHT) {
    HALF = -180; QUART = -90;
  }
  else {
    QUART = 90; HALF = 180;
  }
}
A.2 Implementation of the Eight2Car sequence

```plaintext
sequence Eight2CarModRefs {
  do { disconnect(@Eight[0], @Eight[2]);
  disconnect(@Eight[3], @Eight[4]);
  rotate(@Eight[3], -90);
  rotate(@Eight[4], -90);
  connect(@Eight[4], @Eight[3]);
  do { rotate(@Eight[1], -90);
  disconnect(@Eight[6], @Eight[5]);
  do { rotate(@Eight[4], 90);
  rotate(@Eight[6], 90);
  connect(@Eight[0], @Eight[6]);
```
A. MetaForma DSL programs

```c
disconnect (@Eight[6], @Eight[4]);
do{ rotate (@Eight[0], -90);
    rotate (@Eight[1], 90);
}
rotate (@Eight[0], 90);
do{ connect (@Eight[5], @Eight[6]);
    connect (@Eight[2], @Eight[6]);
    connect (@Eight[1], @Eight[4]);
}
disconnect (@Eight[4], @Eight[3]);
disconnect (@Eight[3], @Eight[1]);
do{ rotate (@Eight[1], 90);
    rotate (@Eight[3], 90);
}
connect (@Eight[1], @Eight[3]);
do{ disconnect (@Eight[3], @Eight[5]);
    disconnect (@Eight[3], @Eight[2]);
}
rotate (@Eight[1], 90);
```

Listing A.4: Implementation of the Eight2Car sequence

```c
meta {
void gradientInit () {
    gradPri = MAX_BYTE;
    gradSec = MAX_BYTE;
}
void gradientPropagate () {
    if (#{connOwn: MALE&WEST}==2 && #()==2) gradPri = 0;
    if (#{connOwn: FEMALE&EAST}==2 && #()==2) gradSec = 0;
    send Gradient (pri=gradPri, sec=gradSec);
}
void gradientAssignEight () {
    byte nr = (gradPri/2)*3;
    if (gradSec%2==0) nr = nr+gradSec/2 + 1;
    module.id = @Eight[nr];
}
}
sequence Eight2CarModRefsNoSymm {
    module.fixSymmetry (module.id == @X[0]); // module 1 basis for symmetry
    module.gradientInit();
do {
    module.gradientPropagate();
} until (config.propagationTime);
    module.gradientAssignEight();
execute Eight2CarModRefs; // Listing A.4
}
```

Listing A.5: Extension of the Eight2Car sequence, making it independent of initial symmetry and a pre-programmed module id in the ensemble
Appendix B

Results of Experiments

This chapter shows detailed information about the results of various experiments on programs written in MetaForma.

B.1 Obstacle Avoidance

Ten experiments have been conducted on the obstacle avoidance implementation for both the two-wheeler and the four-wheeler car. This section shows detailed results of the first experiment in both series.

<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4 s</td>
<td>18.0 s</td>
<td>9.6 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>18.0 s</td>
<td>30.7 s</td>
<td>12.7 s</td>
<td>Turn(RIGHT)</td>
</tr>
<tr>
<td>30.7 s</td>
<td>40.2 s</td>
<td>9.4 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>40.2 s</td>
<td>53.4 s</td>
<td>13.2 s</td>
<td>Turn(RIGHT)</td>
</tr>
<tr>
<td>53.4 s</td>
<td>65.8 s</td>
<td>12.4 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>65.8 s</td>
<td>78.8 s</td>
<td>13.0 s</td>
<td>Turn(RIGHT)</td>
</tr>
<tr>
<td>78.8 s</td>
<td>94.7 s</td>
<td>15.9 s</td>
<td>Strategy</td>
</tr>
</tbody>
</table>

(a) Sequence statistics: After four tries, the car escaped successfully through the gap

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>384</td>
<td>8.0 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>2420</td>
<td>16.5 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>1120</td>
<td>12.8 KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3924</strong></td>
<td><strong>37.4 KB</strong></td>
</tr>
</tbody>
</table>

(b) Packet statistics: 12.5 KB is sent on average per module (avg 1 decimal deviation error)

Table B.1: Statistics for the Two-Wheeler obstacle avoidance from the first out of ten experiments. This experiment ran for 94.7 s
B. RESULTS OF EXPERIMENTS

<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4 s</td>
<td>18.9 s</td>
<td>10.5 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>18.9 s</td>
<td>43.1 s</td>
<td>24.2 s</td>
<td>Turn (RIGHT)</td>
</tr>
<tr>
<td>43.2 s</td>
<td>61.2 s</td>
<td>18.0 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>61.2 s</td>
<td>85.2 s</td>
<td>24.0 s</td>
<td>Turn (RIGHT)</td>
</tr>
<tr>
<td>85.2 s</td>
<td>100.2 s</td>
<td>15.0 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>100.2 s</td>
<td>124.7 s</td>
<td>24.5 s</td>
<td>Turn (RIGHT)</td>
</tr>
<tr>
<td>124.7 s</td>
<td>140.9 s</td>
<td>16.2 s</td>
<td>Strategy</td>
</tr>
<tr>
<td>140.9 s</td>
<td>164.9 s</td>
<td>24.0 s</td>
<td>Turn (RIGHT)</td>
</tr>
<tr>
<td>164.9 s</td>
<td>181.9 s</td>
<td>17.0 s</td>
<td>Strategy</td>
</tr>
</tbody>
</table>

(a) Sequence statistics: After five tries, the car escaped successfully through the gap

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>1736</td>
<td>40.7 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>9588</td>
<td>65.5 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>8552</td>
<td>180.4 KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19876</strong></td>
<td><strong>286.6 KB</strong></td>
</tr>
</tbody>
</table>

(b) Packet statistics: 40.9 KB is sent on average per module (avg 1 decimal deviation error)

Table B.2: Statistics for the Four-Wheeler obstacle avoidance from the first out of ten experiments. This experiment ran for 181.9 s

B.2 Cluster-flow devised by Christensen et al.

Ten simulations have been executed on the cluster-flow devised by Christensen et al. [29] for moving forward a 10-modules scaffold and also ten simulations on a 22-modules scaffold. This section shows detailed results of the first simulation in both series, in Table 8.7 and Table 8.8.
<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Meta id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.8 s</td>
<td>82.6 s</td>
<td>47.9 s</td>
<td>3</td>
<td>LiftUp</td>
</tr>
<tr>
<td>91.9 s</td>
<td>147.7 s</td>
<td>55.9 s</td>
<td>3</td>
<td>LiftDown</td>
</tr>
<tr>
<td>186.7 s</td>
<td>235.8 s</td>
<td>49.0 s</td>
<td>2</td>
<td>LiftUp</td>
</tr>
<tr>
<td>240.7 s</td>
<td>285.9 s</td>
<td>45.3 s</td>
<td>2</td>
<td>LiftDown</td>
</tr>
<tr>
<td>319.4 s</td>
<td>357.3 s</td>
<td>37.9 s</td>
<td>1</td>
<td>LiftUp</td>
</tr>
<tr>
<td>366.5 s</td>
<td>410.8 s</td>
<td>44.3 s</td>
<td>1</td>
<td>LiftDown</td>
</tr>
<tr>
<td>444.3 s</td>
<td>492.5 s</td>
<td>48.2 s</td>
<td>3</td>
<td>LiftUp</td>
</tr>
<tr>
<td>497.7 s</td>
<td>547.8 s</td>
<td>50.0 s</td>
<td>3</td>
<td>LiftDown</td>
</tr>
<tr>
<td>586.6 s</td>
<td>649.0 s</td>
<td>62.4 s</td>
<td>2</td>
<td>LiftUp</td>
</tr>
<tr>
<td>658.1 s</td>
<td>708.3 s</td>
<td>50.2 s</td>
<td>2</td>
<td>LiftDown</td>
</tr>
<tr>
<td>742.9 s</td>
<td>781.6 s</td>
<td>38.7 s</td>
<td>1</td>
<td>LiftUp</td>
</tr>
</tbody>
</table>

(a) Meta-action statistics: Five walkers were successfully moved

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>15056</td>
<td>395.8 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>97169</td>
<td>664.2 KB</td>
</tr>
<tr>
<td>Absorb</td>
<td>396</td>
<td>2.7 KB</td>
</tr>
<tr>
<td>Symmetry</td>
<td>108</td>
<td>0.7 KB</td>
</tr>
<tr>
<td>AssignMetaID</td>
<td>176</td>
<td>1.5 KB</td>
</tr>
<tr>
<td>Region</td>
<td>132</td>
<td>1.4 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>41376</td>
<td>461.9 KB</td>
</tr>
<tr>
<td>Gradient</td>
<td>1152</td>
<td>10.1 KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>155565</strong></td>
<td><strong>1538.5 KB</strong></td>
</tr>
</tbody>
</table>

(b) Packet statistics: 153.9 KB is sent on average per module (avg 1 decimal deviation error)

Table B.3: Christensen’s [29] cluster-flow on a 7x3 scaffold (10 modules)
## B. Results of Experiments

<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Meta id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>55.2 s</td>
<td>102.2 s</td>
<td>47.0 s</td>
<td>7</td>
<td>LiftUp</td>
</tr>
<tr>
<td>103.5 s</td>
<td>120.0 s</td>
<td>16.5 s</td>
<td>7</td>
<td>WalkStep</td>
</tr>
<tr>
<td>121.2 s</td>
<td>135.9 s</td>
<td>14.7 s</td>
<td>7</td>
<td>WalkStep</td>
</tr>
<tr>
<td>137.1 s</td>
<td>151.8 s</td>
<td>14.6 s</td>
<td>7</td>
<td>WalkStep</td>
</tr>
<tr>
<td>153.0 s</td>
<td>169.1 s</td>
<td>16.1 s</td>
<td>7</td>
<td>WalkStep</td>
</tr>
<tr>
<td>158.2 s</td>
<td>203.8 s</td>
<td>45.6 s</td>
<td>6</td>
<td>LiftUp</td>
</tr>
<tr>
<td>205.1 s</td>
<td>222.2 s</td>
<td>17.2 s</td>
<td>6</td>
<td>WalkStep</td>
</tr>
<tr>
<td>223.5 s</td>
<td>240.1 s</td>
<td>16.6 s</td>
<td>6</td>
<td>WalkStep</td>
</tr>
<tr>
<td>241.3 s</td>
<td>255.9 s</td>
<td>14.7 s</td>
<td>6</td>
<td>WalkStep</td>
</tr>
<tr>
<td>209.7 s</td>
<td>262.2 s</td>
<td>52.6 s</td>
<td>7</td>
<td>LiftDown</td>
</tr>
<tr>
<td>262.4 s</td>
<td>278.5 s</td>
<td>16.1 s</td>
<td>6</td>
<td>WalkStep</td>
</tr>
<tr>
<td>258.8 s</td>
<td>314.9 s</td>
<td>56.2 s</td>
<td>5</td>
<td>LiftUp</td>
</tr>
<tr>
<td>316.2 s</td>
<td>332.3 s</td>
<td>16.1 s</td>
<td>5</td>
<td>WalkStep</td>
</tr>
<tr>
<td>296.6 s</td>
<td>344.8 s</td>
<td>48.2 s</td>
<td>6</td>
<td>LiftDown</td>
</tr>
<tr>
<td>333.5 s</td>
<td>348.2 s</td>
<td>14.7 s</td>
<td>5</td>
<td>WalkStep</td>
</tr>
<tr>
<td>349.4 s</td>
<td>364.1 s</td>
<td>14.6 s</td>
<td>5</td>
<td>WalkStep</td>
</tr>
<tr>
<td>365.3 s</td>
<td>381.4 s</td>
<td>16.1 s</td>
<td>5</td>
<td>WalkStep</td>
</tr>
<tr>
<td>369.7 s</td>
<td>427.1 s</td>
<td>57.4 s</td>
<td>4</td>
<td>LiftUp</td>
</tr>
<tr>
<td>428.4 s</td>
<td>444.7 s</td>
<td>16.2 s</td>
<td>4</td>
<td>WalkStep</td>
</tr>
</tbody>
</table>

(a) Meta-action statistics: Four walkers were successfully lifted up, but only three of them were lifted down at the front of the scaffold

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>13736</td>
<td>354.6 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>65422</td>
<td>447.2 KB</td>
</tr>
<tr>
<td>Absorb</td>
<td>210</td>
<td>1.4 KB</td>
</tr>
<tr>
<td>Symmetry</td>
<td>54</td>
<td>0.4 KB</td>
</tr>
<tr>
<td>AssignMetaID</td>
<td>39</td>
<td>0.3 KB</td>
</tr>
<tr>
<td>Region</td>
<td>41</td>
<td>0.4 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>26896</td>
<td>306.2 KB</td>
</tr>
<tr>
<td>Gradient</td>
<td>768</td>
<td>6.8 KB</td>
</tr>
<tr>
<td>Total</td>
<td>107166</td>
<td>1117.4 KB</td>
</tr>
</tbody>
</table>

(b) Packet statistics: 50.8 KB is sent on average per module (avg 1 decimal deviation error)

Table B.4: Christensen’s [29] cluster-flow on a 15x3 scaffold (22 modules)
### B.3 Cluster-flow devised by Brandt et al.

<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Meta id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>94.2 s</td>
<td>152.2 s</td>
<td>58.0 s</td>
<td>5</td>
<td>FlipThrough(BOTTOM_RIGHT)</td>
</tr>
<tr>
<td>263.1 s</td>
<td>354.3 s</td>
<td>91.2 s</td>
<td>5</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>465.2 s</td>
<td>556.8 s</td>
<td>91.7 s</td>
<td>9</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>617.2 s</td>
<td>705.4 s</td>
<td>88.2 s</td>
<td>5</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>651.1 s</td>
<td>706.7 s</td>
<td>55.7 s</td>
<td>6</td>
<td>FlipThrough(BOTTOM_RIGHT)</td>
</tr>
<tr>
<td>765.8 s</td>
<td>841.6 s</td>
<td>75.7 s</td>
<td>9</td>
<td>FlipThrough(LEFT_BOTTOM)</td>
</tr>
<tr>
<td>818.2 s</td>
<td>922.0 s</td>
<td>103.8 s</td>
<td>6</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1032.8 s</td>
<td>1108.7 s</td>
<td>75.9 s</td>
<td>13</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1168.3 s</td>
<td>1224.1 s</td>
<td>55.8 s</td>
<td>10</td>
<td>FlipThrough(BOTTOM_RIGHT)</td>
</tr>
<tr>
<td>1219.6 s</td>
<td>1313.1 s</td>
<td>93.5 s</td>
<td>6</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1374.4 s</td>
<td>1447.4 s</td>
<td>73.0 s</td>
<td>13</td>
<td>FlipThrough(LEFT_BOTTOM)</td>
</tr>
<tr>
<td>1403.0 s</td>
<td>1498.0 s</td>
<td>95.0 s</td>
<td>10</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1575.2 s</td>
<td>1688.6 s</td>
<td>113.3 s</td>
<td>17</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1743.6 s</td>
<td>1813.9 s</td>
<td>70.2 s</td>
<td>14</td>
<td>FlipThrough(BOTTOM_RIGHT)</td>
</tr>
<tr>
<td>1765.8 s</td>
<td>1867.3 s</td>
<td>101.5 s</td>
<td>10</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>1944.5 s</td>
<td>2000.6 s</td>
<td>56.1 s</td>
<td>17</td>
<td>FlipThrough(LEFT_BOTTOM)</td>
</tr>
<tr>
<td>1942.2 s</td>
<td>2029.1 s</td>
<td>86.9 s</td>
<td>14</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>2140.1 s</td>
<td>2362.2 s</td>
<td>222.2 s</td>
<td>21</td>
<td>FlipAlong(RIGHT_BOTTOM)</td>
</tr>
<tr>
<td>2409.5 s</td>
<td>2490.2 s</td>
<td>80.7 s</td>
<td>18</td>
<td>FlipThrough(BOTTOM_RIGHT)</td>
</tr>
</tbody>
</table>

(a) Meta-action statistics: Nineteen meta-actions were successfully performed

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>45112</td>
<td>1574.3 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>336529</td>
<td>2300.5 KB</td>
</tr>
<tr>
<td>Symmetry</td>
<td>576</td>
<td>3.9 KB</td>
</tr>
<tr>
<td>AssignMetaID</td>
<td>353</td>
<td>3.1 KB</td>
</tr>
<tr>
<td>AddNeighbor</td>
<td>2244</td>
<td>19.7 KB</td>
</tr>
<tr>
<td>Region</td>
<td>573</td>
<td>6.2 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>135120</td>
<td>1669.8 KB</td>
</tr>
<tr>
<td>Gradient</td>
<td>7694</td>
<td>67.6 KB</td>
</tr>
<tr>
<td>Total</td>
<td>528201</td>
<td>5645.0 KB</td>
</tr>
</tbody>
</table>

(b) Packet statistics: 434.2 KB is sent on average per module (avg 1 decimal deviation error)

Table B.5: Moving an ensemble of twenty modules forward by using Brandt’s cluster-flow approach, during a simulation of 2889.9 seconds
B. RESULTS OF EXPERIMENTS

<table>
<thead>
<tr>
<th>Start time</th>
<th>End time</th>
<th>Duration</th>
<th>Meta id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>63.9 s</td>
<td>133.9 s</td>
<td>70.0 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>264.5 s</td>
<td>368.4 s</td>
<td>104.0 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>414.8 s</td>
<td>501.0 s</td>
<td>86.2 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
<tr>
<td>562.0 s</td>
<td>649.0 s</td>
<td>87.1 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>728.7 s</td>
<td>818.7 s</td>
<td>90.0 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>864.7 s</td>
<td>934.1 s</td>
<td>69.4 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
<tr>
<td>995.4 s</td>
<td>1081.6 s</td>
<td>86.2 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>1160.4 s</td>
<td>1235.8 s</td>
<td>75.4 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>1295.1 s</td>
<td>1350.4 s</td>
<td>55.3 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
<tr>
<td>1411.1 s</td>
<td>1497.8 s</td>
<td>86.7 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>1558.5 s</td>
<td>1616.7 s</td>
<td>58.2 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>1678.9 s</td>
<td>1759.4 s</td>
<td>80.5 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5743.2 s</td>
<td>5813.5 s</td>
<td>70.3 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>5934.2 s</td>
<td>6001.8 s</td>
<td>67.7 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>6060.7 s</td>
<td>6117.0 s</td>
<td>56.3 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
<tr>
<td>6195.8 s</td>
<td>6266.1 s</td>
<td>70.3 s</td>
<td>5</td>
<td>FlipOver(TOP_RIGHT)</td>
</tr>
<tr>
<td>6345.4 s</td>
<td>6419.4 s</td>
<td>74.1 s</td>
<td>9</td>
<td>FlipThrough(RIGHT_TOP)</td>
</tr>
<tr>
<td>6480.2 s</td>
<td>6535.5 s</td>
<td>55.3 s</td>
<td>1</td>
<td>FlipThrough(TOP_RIGHT)</td>
</tr>
</tbody>
</table>

(a) Fragment of the (18 out of total 48) successfully performed meta-actions.

<table>
<thead>
<tr>
<th>Packet</th>
<th>Total ensemble</th>
<th>Single module avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packets</td>
<td>Bytes</td>
</tr>
<tr>
<td>MetaVarSync</td>
<td>100792</td>
<td>3534.4 KB</td>
</tr>
<tr>
<td>Discover</td>
<td>734247</td>
<td>5019.3 KB</td>
</tr>
<tr>
<td>Symmetry</td>
<td>1508</td>
<td>10.3 KB</td>
</tr>
<tr>
<td>AssignMetaID</td>
<td>834</td>
<td>7.3 KB</td>
</tr>
<tr>
<td>AddNeighbor</td>
<td>5145</td>
<td>45.2 KB</td>
</tr>
<tr>
<td>Region</td>
<td>1167</td>
<td>12.5 KB</td>
</tr>
<tr>
<td>Consensus</td>
<td>291424</td>
<td>3516.6 KB</td>
</tr>
<tr>
<td>Gradient</td>
<td>17271</td>
<td>151.8 KB</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1152388</td>
<td>12297.4 KB</td>
</tr>
</tbody>
</table>

(b) Packet statistics: 1024.8 KB is sent on average per module (avg 1 decimal deviation error)

Table B.6: Moving an ensemble of twelve modules in backwards-left direction by using Brandt’s cluster-flow approach during a simulation of 6545.3 seconds
Appendix C

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

**AST** : Abstract Syntax Tree, a program represented in a tree structure to enable traversal

**ATRON** : The ATRON self-reconfigurable robot

**Cluster-flow** : Locomotion by self-reconfiguration \[72, p47\]

**DSL** : Domain-Specific Language

**DCD-VM** : Distributed-Control Virtual Machine \[12\]

**Desugar** : Remove syntactic sugar from an AST

**DynaRole** : DYNAmically update-able ROLE-based language \[12\]

**Ensemble** : A group of modules that is either directly or indirectly connected

**Flip-Through, Flip-Over, Flip-Along** : Meta-module moves from Brandt \[16\], see Figure 4.12

**GPL** : General-Purpose Language

**Gradient** : An algorithm based on hop-counting of the topological distance between any given module and the hormone emitting module \[57\]

**Locomotion** : The ability to move from one place to another \[72, p146\]

**M-TRAN** : Modular-TRANsformer \[47\] - self-reconfigurable robot

**NBL** : Name-Binding Language - A declarative language that facilitates binding references to their appropriate declarations \[44\]

**Scaffolding** : Modules are divided into two classes, wandering and scaffold modules, where scaffold modules provide a uniform configuration in which the wanering modules can move around easily \[72, p107\]
C. Glossary

**Self-reconfigurable robot** : A robot that can change the way modules are connected by itself [72, p7]

**Self-reconfiguration** : A topology change performed completely by the robot itself from an initial shape to a target shape

**USSR** : Unified Simulator for Self-Reconfigurable Robots [25]