Hardware and Software Codesign

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Marcus Pirron

Date of Defense:	05.03.2024
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Rupak Majumdar
Reviewer:	Dr. Anne-Kathrin Schmuck

DE-386

HARDWARE AND SOFTWARE CODESIGN

MARCUS PIRRON

For my family.

ABSTRACT

Modern robotic applications consist of a variety of robotic systems that work together to achieve complex tasks. Programming these applications draws from multiple fields of knowledge and typically involves low-level imperative programming languages that provide little to no support for abstraction or reasoning. We present a unifying programming model, ranging from automated controller synthesis for individual robots to a compositional reasoning framework for inter-robot coordination. We provide novel methods on the topics of control and planning of modular robots, making contributions in three main areas: controller synthesis, concurrent systems, and verification. Our method synthesizes control code for serial and parallel manipulators and leverages physical properties to synthesize sensing abilities. This allows us to determine parts of the system's state that previously remained unmeasured. Our synthesized controllers are robust; we are able to detect and isolate faulty parts of the system, find alternatives, and ensure continued operation. On the concurrent systems side, we deal with dynamic controllers affecting the physical state, geometric constraints on components, and synchronization between processes. We provide a programming model for robotics applications that consists of assemblies of robotic components together with a run-time and a verifier. Our model combines message-passing concurrent processes with motion primitives and explicit modeling of geometric frame shifts, allowing us to create composite robotic systems for performing tasks that are unachievable for individual systems. We provide a verification algorithm based on model checking and SMT solvers that statically verifies concurrency-related properties (e.g. absence of deadlocks) and geometric invariants (e.g. collision-free motions). Our method ensures that jointly executed actions at end-points are communication-safe and deadlock-free, providing a compositional verification methodology for assemblies of robotic components with respect to concurrency and dynamic invariants. Our results indicate the efficiency of our novel approach and provide the foundation of compositional reasoning of robotic systems.

ZUSAMMENFASSUNG

Moderne Roboteranwendungen bestehen aus einer Vielzahl von Robotersystemen, die zur Bewältigung komplexer Aufgaben zusammenarbeiten. Die Programmierung dieser Anwendungen bedient sich einer Vielzahl von Wissensgebieten und stützt sich üblicherweise auf einfache imperative Programmiersprachen, die wenig bis keine Unterstützung für Abstraktion oder Beweisbarkeit bieten. Wir stellen ein vereinheitlichendes Programmiermodell vor, welches von der automatisierten Steuerungssynthese für einzelne Roboter bis hin zu einem kompositorischen Argumentationsrahmen für die Koordination zwischen Robotern reicht. Unter Verwendung modularer Roboter entwickeln wir neuartige Methoden in drei Bereichen: Controllersynthese, nebenläufige Systeme und Verifikation. Unsere Methode synthetisiert Steuerungscode für serielle und parallele Roboter und nutzt physikalische Eigenschaften, um durch intelligente Sensorlösungen Fähigkeiten und Flexibilität dieser Systeme zu verbessern. Unsere synthetisierten Steuerungen sind robust: Fehlerhafte Teile des Systems können erkannt und isoliert sowie Alternativen zu diesen gefunden werden, um den weiteren Betrieb sicherzustellen; ebenso können wir zuvor nicht bestimmbare Teile des Systemzustands messen. Unser Modell kombiniert nebenläufige Prozesse mit atomaren Bewegungseinheiten und der expliziten Überführung jeweiliger Ortskoordinatensysteme. Die auf diese Weise erzeugten Robotersysteme eignen sich für die Ausführung von Aufgaben, die für Einzelsysteme unausführbar sind. Wir stellen einen auf Modellprüfung und SMT-Solvern basierenden Verifikationsalgorithmus bereit, der nebenläufigkeitsbezogene Eigenschaften (z. B. das Fehlen von Deadlocks) und geometrische Invarianten (z. B. kollisionsfreie Bewegungen) statisch verifiziert. Dadurch ist gewährleistet, daß gemeinsam ausgeführte Aktionen an Endpunkten kommunikationssicher und deadlock-frei sind. Unsere Resultate belegen die Effizienz unseres neuartigen Ansatzes und bilden die Grundlage für die kompositorische Beweisbarkeit von Robotersystemen.

PUBLICATIONS

Part of this work is based on these publications:

[111] **MPERL: Hardware and Software Co-design for Robotic Manipulators**
Marcus Pirron and Damien Zufferey
Proceedings of the International Conference on Intelligent Robots and Systems (IROS), 2019
[112] **Automated Controller and Sensor Configuration Synthesis Using Dimensional Analysis**
Marcus Pirron, Damien Zufferey, and Phillip Stanley-Marbell
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2020.
[7] **PGCD: Robot Programming and Verification with Geometry, Concurrency, and Dynamics**
Gregor B. Banusic, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey
Proceedings of the 10th International Conference on Cyber-Physical Systems (ICCPS), 2019

Furthermore, the subsequent publication advances the concepts presented in [7], but is not incorporated into the present treatise:

[84] **Motion Session Types for Robotic Interactions**
Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey
Proceedings of 33rd European Conference on Object-Oriented Programming (ECOOP), 2019

Chapter 3 is based on work done in [111], Chapter 4 on work done in [112], Chapter 5 on work in [7].

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisors, Damien Zufferey and Rupak Majumdar, for their teaching, guidance and support.
To my family, thank you — you made all of this possible.
To Heinz Dabrock and Albert Damm, who sparked the flame so many years ago.

CONTENTS

1 INTRODUCTION 1

2 PRELIMINARIES 5
- 2.1 Robotic Systems 6
- 2.2 Position and Orientation 7
- 2.3 Connectivity 11
- 2.4 Kinematic Equations 14
- 2.5 Representation as First Order Logic 17
- 2.6 Dynamical System 17

3 CONTROLLER SYNTHESIS FOR ROBOTIC MANIPULATORS 23
- 3.1 The Mperl Language 25
  - 3.1.1 Composing and Manipulating Elements 27
  - 3.1.2 Actuating the system 28
- 3.2 Graph Representation 28
  - 3.2.1 Single SCARA system 29
  - 3.2.2 Dual SCARA system 30
- 3.3 Controller Synthesis 34
  - 3.3.1 Forward and Inverse Kinematics 35
  - 3.3.2 Soundness 39
  - 3.3.3 Workspace Mapping and Singularities 40
  - 3.3.4 Trajectory Generation 43
- 3.4 Sensors and Feedback 44
- 3.5 Example: SCARA System 46
- 3.6 Example: SCARA System as a Component 49
- 3.7 Evaluation 51
- 3.8 Conclusion 54

4 SENSOR CONFIGURATION SYNTHESIS 57
- 4.1 Extensions to Mperl 60
- 4.2 Preliminaries 62
  - 4.2.1 Quantities, Units and Dimensions 62
  - 4.2.2 Buckingham Π Theorem 63
  - 4.2.3 Reach Avoid Specification 64
  - 4.2.4 Relevance to State Estimation 65
- 4.3 Synthesis of Sensor Configurations 66
  - 4.3.1 Search algorithm. 68
  - 4.3.2 Calibration and Run-time 71
  - 4.3.3 Limitations and Extensions of the Method. 73
- 4.4 Evaluation 73
  - 4.4.1 Quantitative Evaluation 74
  - 4.4.2 Qualitative Evaluation 76
- 4.5 Conclusion 81

5 MULTI-ROBOT APPLICATION PROGRAMMING 85

5.1 PGCD Programs 88
5.1.1 Syntax 88
5.1.2 Attached Composition 90
5.1.3 Programs 92
5.1.4 Semantics 93
5.2 Verification 97
5.2.1 Communication safety 98
5.2.2 Trajectories and footprints 99
5.2.3 Specifications and Annotations 100
5.2.4 Extensions 101
5.3 Implementation and Evaluation 101
5.3.1 Run-time System 102
5.3.2 Verifier 102
5.3.3 Evaluation 103
5.3.4 Setup 104
5.3.5 Experiments 104
5.4 Conclusion 107
6 RELATED WORK 111
6.1 Prototyping tools 111
6.2 Physics Integration 114
6.3 Robot Interaction 115
7 FUTURE WORK 117
7.1 Progressing Sensor Integration and Material Science . 118
7.2 Verification of physical properties 119

Appendix

A Implementation Details of State Estimation 123
A.1 State Estimation 123
A.2 Error Detection 125
A.3 Limitations 127
A.4 Evaluation 127
B Technical Implementation Details of Robotic Systems . 128
C Additional PGCD Ressources 132
C.1 Motion Primitive Example 132
C.2 PGCD Code for Fetch 134
C.3 Promela Model Example 134

BIBLIOGRAPHY 137

LIST OF FIGURES

Figure 1.1 Fetch experiment 1
Figure 1.2 Robotic systems used in the running example 2
Figure 1.3 Decomposition of handover motion primitives 3
Figure 1.4 Handover example. Variations of this example are used throughout the thesis 3
Figure 2.1 Typical example of robotic systems 5
Figure 2.2 Serial chain system 6
Figure 2.3 Robotic Systems, Environment, and their interactions 7
Figure 2.4 Global and local reference frames 11
Figure 2.5 Schematic view of a closed loop system 13
Figure 2.6 Schematic view of an open chain system . . . 16
Figure 2.7 Dynamical System 19
Figure 3.1 Mperl workflow 24
Figure 3.2 Single SCARA robot 29
Figure 3.3 Graph of single SCARA system 30
Figure 3.4 Dual SCARA robot, as generated by Mperl . . 32
Figure 3.5 Graph of dual SCARA system 32
Figure 3.6 Example of a rigid chain 33
Figure 3.7 Graph representation of Fig. 3.6 33
Figure 3.8 Mperl controller and its contents 34
Figure 3.9 Chain decomposition 37
Figure 3.10 Types of Singularities 42
Figure 3.11 Dual scara arm 44
Figure 3.12 Embedding sensors directly into the structure of a component 45
Figure 3.13 Mperl code for a single scara arm with flex sensor 45
Figure 3.14 Load induced deflection d 45
Figure 3.15 SCARA system 46
Figure 3.16 High level Mperl Controller 52
Figure 3.17 Systems tested with Mperl 53
Figure 4.1 Range restriction caused by payload 58
Figure 4.2 Quantities of the example cart-and-arm system 59
Figure 4.3 System overview 60
Figure 4.4 Sensor synthesis workflow 66
Figure 4.5 Workspace representation for the armsystem . 78
Figure 4.6 Arm pushing against a spring scale to simulate load on the end-effector 82
Figure 5.1 PGCD Architecture 86
Figure 5.2 Schematic and actual representation of the cart-and-arm system 92

Figure 5.3 Schematic representation of Fetch PGCD process interaction 94
Figure 5.4 Reduction rules for communication 96
Figure 5.5 Reduction rules for motion 97
Figure 5.6 Reduction rules for control flow. 98
Figure 5.7 Verification Process for PGCD programs 99
Figure 5.8 Fetch Experiment 106
Figure 5.9 Handover Experiment 107
Figure 5.10 Twist and Turn Experiment 108
Figure 5.11 Underpass Experiment 109
Figure .1 State Estimation in Mperl 123
Figure .2 Franka Emika Panda robot arm with 7 DoF used in the evaluation of the error detection . 127
Figure .3 Input trajectory to test detection of impact and sensor drift 128
Figure .4 Entropy measurements (Eq. .13) while traversing the trajectory from Fig. .3 128
Figure .5 Schematic of the global controller 129
Figure .6 Schematic of individual components 130
Figure .7 Sensors and electro-mechanical components of the SCARA system 131

LIST OF TABLES

Table 2.1 Degree of freedom of various joint types 14
Table 3.1 Selection of lower kinematic pairs. 27
Table 3.2 Overview of examined robotic systems 53
Table 3.3 Evaluation of the kinematics solvers 54
Table 3.4 Computation of the singularity map 54
Table 4.1 Base dimensions and units of the SI 62
Table 4.2 Parameters and Units for the armsystem . . . 75
Table 4.3 Components and the total number of quantities in the manipulators. 75
Table 4.4 Dimensional analysis summary for the armsystem 76
Table 4.5 Dimensional analysis summary for the dual SCARA system 77
Table 4.6 Dimensional analysis summary for the CoreXY platform 77
Table 4.7 Deflection measurements and calibration for reference weights. 80
Table 4.8 Deriving force by different invariants 81
Table 4.9 Deriving deflection by different invariants . . . 81

Table 4.10 Results for autocalibration. 82
Table 5.1 Motion Primitives Implementation and Specification 105
Table 5.2 Programs, Annotations, and Checks 106

LISTINGS

Listing 3.1 Mperl Language 25
Listing 3.2 Input Description of SCARA system 29
Listing 3.3 Mperl code for a dual SCARA system 30
Listing 3.4 Input Description of SCARA system 47
Listing 3.5 SMT v2 Code for SCARA system 48
Listing 3.6 Results of running the code from Listing 3.5 . 49
Listing 3.7 Changes in SMT v2 Code for SCARA component as opposed to the stand-alone system . . 50
Listing 3.8 Input description of the SCARA system mounted on a cart 50
Listing 4.1 Sensor synthesis algorithm 69
Listing 5.1 PGCD Grammar 89
Listing 5.2 Process Cart 93
Listing 5.3 Process Arm 93
Listing 1 Fold motion primitive implementation 133
Listing 2 PGCD program for Fetch 134
Listing 3 Promela Code of the cart 135
Listing 4 Promela Code of the Arm 135
Listing 5 Promela code of the Scheduler 136

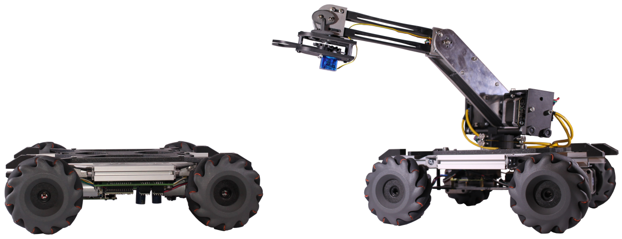
1

INTRODUCTION

Modern robotic applications consist of a plethora of robotic systems that work together to achieve complex tasks. Individual robotic systems are not in a fixed configuration, but can be (field-) modified, depending on the task at hand. Tasks can be arbitrarily complicated, resulting in complex, polyvalent motions that need to be coordinated between multiple robots. As tasks may change, or the environment in which they are performed, robotic systems need to be reconfigurable, thus able to adapt ad hoc. Reconfiguration also implies that robotic systems can be composed from other robotic systems; e.g., an otherwise independent robot arm can be placed on top of a cart. A seemingly simple task like fetching an object from a known position (Fig. 1.1) requires, for example, knowledge about the abilities of the participating robots, including the maximum reach of the arm, feasible load, or even whether the arm is able to grab the object at all. Communication between the cart and the arm is needed, particularly if the object is out of reach for the arm so that the cart has to relocate the arm. Other points to consider include sensing the environment and identifying obstacles to the robots, or even other robots working on adjacent tasks. Solving such scenarios requires knowledge in multiple domains such as control theory, physics, or concurrent systems.

One common denominator are motion primitives, which are abstractions of dynamic controllers. Using a motion primitive allows us to use, e.g., a GRAB command that causes the arm to extend to a specific position and grab an object, encapsulating the lower level specifics such as dynamics or path planning. Motion primitives can be combined to achieve more complex behaviors, e.g., the FETCH command. Contrary to GRAB, which is arm-specific and only able to grab an object within arm's reach, FETCH also includes behavior of the cart. It takes care of the dynamics of moving the cart to a suitable location, from which GRAB can be executed successfully, and it ensures that the cart and the arm are in a configuration that allows subsequent operations. As both cart and arm are two separate robotic systems, the motion

Figure 1.1: A seemingly simple task. The cart-and-arm assembly has to fetch an object in its vicinity.



(a) Carrier system (b) cart-and-arm system

Figure 1.2: Robotic systems used in the running example

primitive FETCH also has to take care of any communication between these two systems, as well as any necessary geometric transformations.

Throughout the thesis, we use a running example called *handover* to illustrate how our system works. We need to examine two sides in this running example: The technical side, concerned with the robotic systems, and the task-specific side, which focuses on the actions and interactions of the involved robotic systems.

On the technical side, we use a *carrier* and a cart-and-arm assembly (See Fig. 1.2), both of which provide a set of functionality. Both carrier and cart are identical in their functionality. They are equipped with meccanum wheels [29], which allow them three degrees of freedom (horizontal, vertical, rotation around z-axis and any combination thereof), and are able to move between two points. Moving between points is done via the MOVE motion primitive. Attached to the cart is a robot arm, with a (detachable) end effector, allowing it to grab objects in its vicinity by means of the motion primitive GRAB. In combination with the cart, the robot arm is able to grab objects at arbitrary positions, provided the cart can move close enough.

On the task-specific side, we use these robotic systems to accomplish the goal of the handover scenario. The carrier transports an object and meets with the cart, which has a robotic arm attached to it (cart-and-arm assembly). Both the carrier and the cart-and-arm assembly use MOVE to meet at some previously defined location. This assumes that the arm assembly is folded safely away (HOME). After meeting at the arranged location, the arm GRAB the object from the carrier, folds itself back into transport safe pose and both cart and carrier move to some new location. Figures 1.3 and 1.4 describe the interactions between those systems in the handover scenario.

Besides the full version, we also consider other variations, e.g., with a stationary robot arm, which can only use GRAB, requiring an object to be placed into its work space (we call this variation FETCH).

In the following chapters, we develop the theory of both the technical side and the task-specific side. All robotic systems and all examples and their variants will be accompanied by code examples and summarized in situ or placed in the appendix.

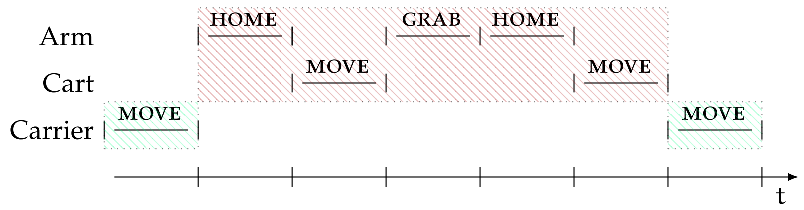
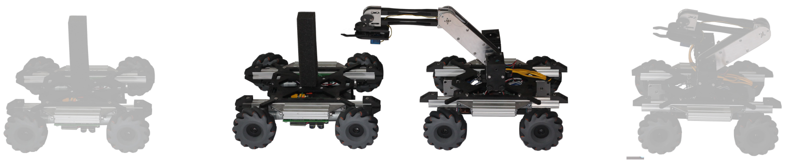
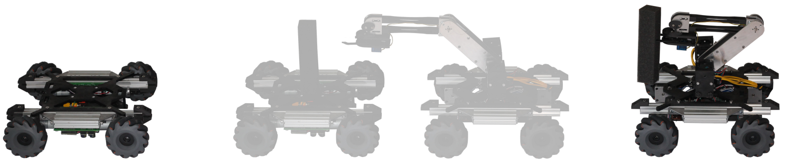


Figure 1.3: Decomposition of the motion primitives in the handover scenario: MOVE motion primitive of the carrier in blue, FETCH motion primitive of the cart-and-arm system in red

(a) Like the fetch example, but the object is placed on a carrier system



(b) Cart and carrier meet at a suitable position and exchange the object



(c) Cart and Carrier return to their initial positions

Figure 1.4: Handover example. Variations of this example are used throughout the thesis

This dissertation contributes to the fields of controller synthesis, concurrent systems, and verification. Controller synthesis automatically generates control software for a cyber physical system, given a model of the environment and a system goal. The synthesized controller is consistent in its behavior to the system goal (i.e., it satisfies the specification), when executed in an environment coherent to the assumed one. Concurrent systems are distinguished by their parallel execution model, along with the necessary synchronization. Processes are executed simultaneously, and their time periods can overlap.

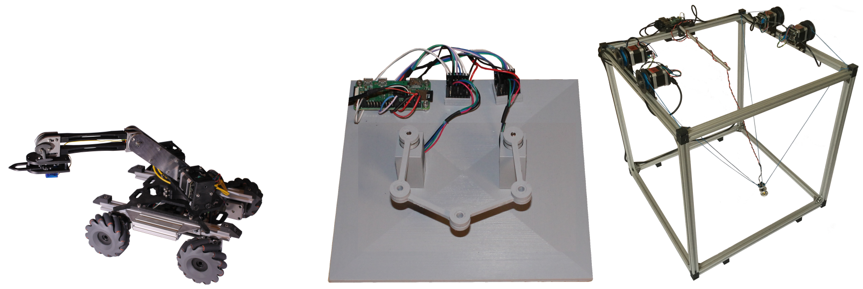
An example of a contribution in the field of controller synthesis is Mperl, which provides a unified approach to controller synthesis of serial and parallel robotic systems (e.g., single and dual SCARA systems); a contribution in the field of verification would be PGCD, which describes robotic systems as message-passing concurrent processes that execute motion primitives.

This dissertation is structured as follows: Chapter 2 introduces prerequisites and the running example. Chapters 3, 4 describe the work concerning individual robotic systems, in particular Mperl and its extension, the synthesis configuration synthesis (SCS) that enhances the synthesized controllers with physical properties, allowing the derivation of additional functional dependencies. We use these results in chapter 5 to illustrate how robotic systems interact with each other, including message passing and verification. Chapter 6 discusses related work, and chapter 7 summarizes the dissertation and highlights selected future work directions. An appendix provides supplementary material.

2 PRELIMINARIES

Robotic systems, as considered in this thesis, are systems of rigid bodies, connected by joints, which allow relative motion between those bodies, governed by a controller. Properties of these rigid systems include the placement of links and joints relative to each other, the constraints these joints impose upon the motion of the robot (e.g. rotation or displacement) or their location in their environment. Other properties include mass and inertia of the systems components, and resulting from that, velocity and acceleration of individual components as well as the system as a whole. Each component of a robotic system is expressed in reference to a coordinate reference frame (*frames*), so a pose can not only be expressed by one component relative to another, but also by their corresponding frames. Figure 2.1 shows typical examples of robotic systems.

Each of these systems is placed either stationary or dynamically (e.g. on top of a cart) in their environment, and each system is at some point equipped with the possibility to attach different tools to interact with their environment, e.g. grippers or sensing equipment like cameras or probes. Fig. 2.2 shows an example of a typical industry robot, whose components are arranged in a single serial chain, with the base of the robot attached to the ground and the end of the robot carrying the end effector. Other possible configurations are kinematic trees, which are e.g. found in the model of a robotic hand, or parallel structures, e.g. Gough-Stewart platforms. To facilitate the development of robotic systems, controllers and trajectories are encapsulated into motion primitives. The purpose of this chapter is to offer a concise overview of the key concepts employed throughout the thesis, drawing


(a) (Serial) SCARA system, mounted on top of a cart

(b) Parallel or Dual SCARA system

(c) Parallel cable driven robot

Figure 2.1: Typical example of robotic systems. The two SCARA systems act as running examples throughout the thesis.

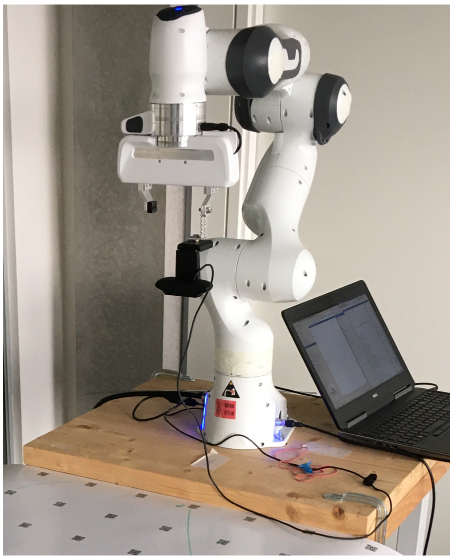


Figure 2.2: Example of a serial chain robotic systems. Pictured is a Franka Emica Panda robot arm [28].

inspiration from the works of Siciliano et al. [126] and Lynch et al. [81].

2.1 ROBOTIC SYSTEMS

Multiple definitions of robots or robotic systems exists; for example, in 2021, the ISO [59] defines robot as a *programmed actuated mechanism with a degree of autonomy to perform locomotion, manipulation or positioning*; another definition outlines robots as *an autonomous machine capable of sensing its environment, carrying out computations to make decisions, and performing actions in the real world* [137]. Other attempts to define a robot follow similar trajectories, e.g. *[Industrial robots] are programmable multi-functional mechanical devices designed to move material, parts, tools, or specialized devices through variable programmed motions to perform a variety of tasks* [119].

Closely related to a robot is a manipulator, a *machine or robotic mechanism of which usually consists of a series of segments [...] for the purpose of grasping and/or moving objects [...]* [59].

Common parts to these definitions are the integration of mechanical assemblies, computer control and the modularity of both, the mechanical aspect and the controlling aspect. We can thus define robotic systems as systems typically consisting of one or more computer controlled, potentially modular assemblies – such as manipulators – that operate as part of the system within an environment.

The modularity of a robot, e.g. a manipulator in the form of a robot arm, can be as simple as attachable tools at the end effector, or as complex as modifying the actual structure. Consequently, we have

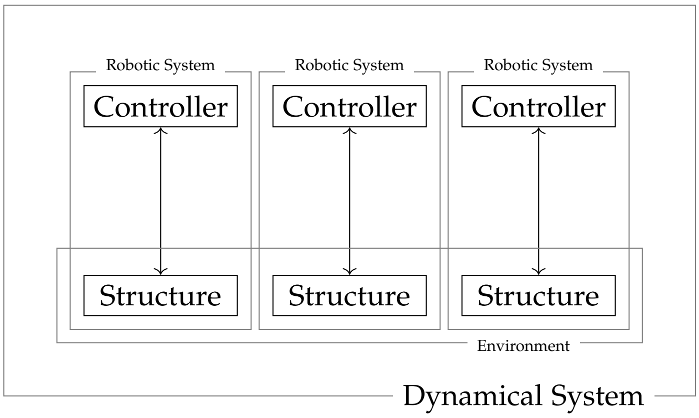


Figure 2.3: Robotic systems consist of a mechanical structure, capable of manipulating within an environment, and a controller. Within the environment, multiple robotic systems may exist and mutually influence each other. Together, robotic systems and their environment form a dynamical system.

to consider the actual physical structure of the robot, its modularity, and how it can be controlled. Control includes calculating the desired position, velocity or other relevant variables, comparing them to their actual, measured values, and outputting the corresponding drive signals to actuate the robot. The path the robot follows between the current position and the desired position, along with the time is called the *trajectory*. The *control problem* thus asks whether a controller (and by extension, the structure of the robot) exists that can follow a given trajectory.

The environment serves a dual purpose; firstly, in a tangible view, it represents the physical space in which a robotic system manipulates objects, encounters obstacles or engages with other robotic systems. Secondly, in a more abstract way, the environment is coupled to the controller via an input/output cycle; i.e. based on inputs received from the controller, the environment generates signals which serve as input to the controller. The individual controller usually can only perceive a subset of these output signals. In Figure 2.3, a visual representation is presented to illustrate the interaction between the individual components. We use the term *robotic system* synonymously to robot, to accentuate their multifaceted nature and the interplay between multiple components.

2.2 POSITION AND ORIENTATION

Each robotic system and each of their components have their own coordinate reference frame $\mathcal{L}$ and must be expressed relative to a frame. A coordinate reference frame, or for short *frame*, $\mathcal{L}$ consists of an origin p, and three mutually orthogonal unit vectors $\mathbf{u}_x$, $\mathbf{u}_y$, and

$\mathbf{u}_z$ for a 3—dimensional euclidean space. The physical world, in which our robots operate, is represented by a three dimensional euclidean space, called $\mathcal{W}$, the world frame. The position of the origin of a frame $\mathcal{L}_i$ relative to a frame $\mathcal{L}_j$ with the same orientation can be expressed by a vector ${}^jp_i$. Between frames with the same orientation, translations and displacements can be calculated using vector addition. We require translations to be affine transformations, i.e. transformations which preserve collinearity and the ratio of distances on a line.

$$
\begin{aligned}
{}^j v &= {}^i v + {}^j p_i && (2.1)\\
{}^n p_i &= {}^n p_{n-1} + {}^{n-1} p_{n-2} + \ldots + {}^{i+1} p_i && (2.2)
\end{aligned}
$$

for some vector v and consecutive frames $i, \ldots, n$.

Orientation of a frame $\mathcal{L}_i$ relative to a frame $\mathcal{L}_j$ is given by the dot product of the basis vectors of the two frames:

$$
{}^jR_i = \begin{bmatrix} \mathbf{u}_{x_i} \cdot \mathbf{u}_{x_j} & \mathbf{u}_{y_i} \cdot \mathbf{u}_{x_j} & \mathbf{u}_{z_i} \cdot \mathbf{u}_{x_j} \\ \mathbf{u}_{x_i} \cdot \mathbf{u}_{y_j} & \mathbf{u}_{y_i} \cdot \mathbf{u}_{y_j} & \mathbf{u}_{z_i} \cdot \mathbf{u}_{y_j} \\ \mathbf{u}_{x_i} \cdot \mathbf{u}_{z_j} & \mathbf{u}_{y_i} \cdot \mathbf{u}_{y_j} & \mathbf{u}_{z_i} \cdot \mathbf{u}_{z_j} \end{bmatrix} \qquad (2.3)
$$

The set of all rotational matrices R, along with matrix multiplication, forms the special group $SO(3)$,

$$
SO(3) = \{R : R \in \mathbb{R}^{3\times 3}, RR^T = 1, \det(R) = 1\} \qquad (2.4)
$$

The set of all homogeneous transformation matrices T form the special Euclidean group $SE(3)$,

$$
SE(3) = \left\{ \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} : R \in SO(3), p \in \mathbb{R}^3 \right\} \qquad (2.5)
$$

As $SE(3)$ is a group of isometries of an euclidean space, transformation of that space preserves the euclidean distance between any two points. In the case of the euclidean space, distance between two points $u = (u_1, u_2, u_3), v = (v_1, v_2, v_3)$ is given by a the length of the line segment between those two points, and we use the euclidean distance given by

$$
d(u, v) = \sqrt{\sum_{i=1}^{3} (u_i - v_i)^2} \qquad (2.6)
$$

and analogously, we define the norm of some vector $v = (v_1, v_2, v_3)$ as usual

$$
\|v\| = \sqrt{\sum_{i=1}^{3} (v_i^2)} \qquad (2.7)
$$

Elementary rotations of a frame $\mathcal{L}$ around the x,y, or z axis are given by the rotation matrices

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\theta & -s_\theta \\ 0 & s_\theta & c_\theta \end{bmatrix} \tag{2.8}$$

$$R_y(\theta) = \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \tag{2.9}$$

$$R_z(\theta) = \begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

where for brevity of notation, we write s_θ and c_θ for $sin(\theta)$ and $cos(\theta)$, resp., and R_x, R_y, R_z to indicate the rotation matrices of angle $\theta_x, \theta_y, \theta_z$ around their respective axis.

We can calculate rotations between frames by matrix multiplication of rotation matrices:

$${}^{n}R_i = {}^{n}R_{n-1}{}^{n-1}R_{n-2}...{}^{n-i+1}R_i \tag{2.11}$$

$$\tag{2.12}$$

for a sequence of frames $i, ..., n$.

We represent the orientation of one frame relative to another frame as a vector of three fixed angles $[\psi, \theta, \phi]^T$, which can be directly derived from the the rotation matrix ${}^jR_i$:

$$\theta = Atan2(-R_{31}, \sqrt{R_{11}^2 + R_{21}^2}) \tag{2.13}$$

$$\phi = Atan2(\frac{R_{32}}{c_\theta}, \frac{R_{33}}{c_\theta}) \tag{2.14}$$

$$\psi = Atan2(\frac{R_{21}}{c_\theta}, \frac{R_{11}}{c_\theta}) \tag{2.15}$$

where R_{ij} represents the entry at the $i-$th row and the $j-$th column. This vector represents the orientation of a coordinate frame i relative to a coordinate frame j. ψ is the *yaw* rotation of frame i around the fixed $\mathbf{u}_{x_j}$ axis of frame j, θ the *pitch* rotation around fixed $\mathbf{u}_{y_j}$ axis, and ϕ represents the *roll* rotation around fixed $\mathbf{u}_{z_j}$ axis

So far, we treated translations and rotations separately. In this thesis, we use homogeneous transformations, which allows us to combine rotational and translational movement. For combined orientation and translation of frame i relative to frame frame j, we can write

$${}^jr = {}^jR_i\,{}^ir + {}^jp_i \tag{2.16}$$

$$\begin{bmatrix} {}^jr \\ 1 \end{bmatrix} = \begin{bmatrix} {}^jR_i & {}^jp_i \\ O & 1 \end{bmatrix} \begin{bmatrix} {}^ir \\ 1 \end{bmatrix} \tag{2.17}$$

${}^{j}T_i = \begin{bmatrix} {}^{j}R_i & {}^{j}p_i \\ O & 1 \end{bmatrix}$ is called the 4x4 homogeneous transformation matrix and it transforms vectors from frame i to frame j. Its inverse, $({}^{j}T_i)^{-1}$, transforms vectors from frame j to frame i.

$$
{}^{j}T_i^{-1} = \begin{bmatrix} {}^{j}R_i^T & -{}^{j}R_i^T {}^{j}p_i \\ O & 1 \end{bmatrix} \tag{2.18}
$$

Multiple rotations between frames are computed in the same manner as the 3×3 rotation matrices, and, since matrix multiplications are not commutative, the order in which the rotations are computed is important. Homogeneous transformations of rotations around an axis are then given by

$$
R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\theta_x} & -s_{\theta_x} & 0 \\ 0 & s_{\theta_x} & c_{\theta_x} & 0 \\ 0 & & 0 & 1 \end{bmatrix} \tag{2.19}
$$

$$
R_y(\theta_y) = \begin{bmatrix} c_{\theta_y} & 0 & s_{\theta_y} & 0 \\ 0 & 1 & 0 & 0 \\ -s_{\theta_y} & 0 & c_{\theta_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.20}
$$

$$
R_z(\theta_z) = \begin{bmatrix} c_{\theta_z} & -s_{\theta_z} & 0 & 0 \\ s_{\theta_z} & c_{\theta_z} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.21}
$$

Rotations around all three axis are always conducted in $z - y - x$ order:

$$
R(\theta_x, \theta_y, \theta_z) = R_z(\theta_z) \times R_y(\theta_y) \times R_x(\theta_x) \tag{2.22}
$$

which is equivalent to using the yaw, pitch and roll angles of the $X - Y - Z$ fixed angles ($\theta_x = \phi, \theta_y = \theta, \theta_z = \psi$).

Displacements are denoted by

$$
D(p) = \begin{bmatrix} \mathbb{1}^{3\times 3} & \lambda \\ 0^{1\times 3} & 1 \end{bmatrix} \tag{2.23}
$$

Both homogeneous transformation matrices (rotation and displacement) can be composed by matrix multiplication, and we can write

$$
{}^{n}T_i = {}^{n}T_{n-1} \, {}^{n-1}T_{n-2} \ldots {}^{i+1}T_i \tag{2.24}
$$

for some frames $i, \ldots, n$.

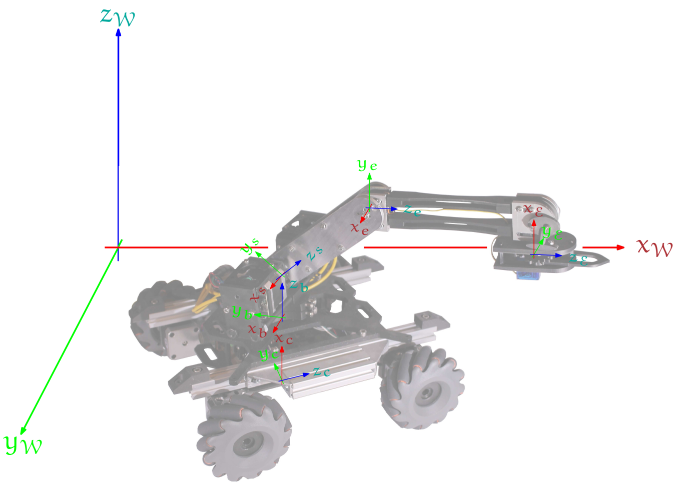


Figure 2.4: Components of the robot arm on top of a cart (cart-and-arm system), along with their corresponding coordinate frames. The orientation of these frames as shown are relative to the world frame $\mathcal{W}$, which represents a model of the physical world. The default orientation of parts are upright, roll axis along the z-axis. From $\mathcal{W}$, frames represent the cart, rotating base of the robot arm, shoulder and elbow part and the end effector.

In two frames, which differ in origin and in orientation, the translation is always preceded by the rotation:

$$^{j}u = {}^{j}R_{i}\,{}^{i}u + {}^{j}p_{i} \tag{2.25}$$

Robotic systems, which consists of rigid bodies connected by joints most often use joints from the group of *lower kinematic pairs*. This group encompasses, among others, revolute, prismatic, helical, cylindrical, spherical and planar joints, which correspond to subgroups of $SE(3)$. For example, prismatic joints are represented by displacements $d \in \mathbb{R}$, revolute joints by an angle $\theta \in [0, 2\pi)$.

We associate two particular frames to robotic systems: An anchor frame ${}^{\mathcal{W}}\mathcal{A}$, which anchors the robotic system to the world frame $\mathcal{W}$ and is stationary with respect to the first component, and a tool- or end effector frame $\mathcal{E}$, which is attached to the system component representing the end effector. $\mathcal{E}$ is not stationary and its position and orientation depends on the configuration of the system.

2.3 CONNECTIVITY

Robotic systems like the arm of the cart-and-arm system are modeled as a sequence of components, linked together by joints. These systems can be categorized into open chains (*serial mechanisms*), kinematic trees and closed chains (*parallel mechanisms*).

One way of categorizing specific systems is by analyzing the sequence of frames between $\mathcal{W}$ and $\mathcal{E}$. An open chain, for example our robot arm from the cart-and-arm system, consists of exactly one anchor frame and one end effector frame, and for each frame in the sequence of frames inbetween it holds, that each of those frames has exactly one successor and exactly one predecessor. An open serial chain with n links is thus given by the following sequence of frames:

$$\mathcal{W}\ {}^{\mathcal{W}}\mathcal{A}\ {}^{\mathcal{W}\mathcal{A}}T_1\ {}^{1}T_2 \dots {}^{n-2}T_{n-1}\ {}^{n-1}\mathcal{E} \tag{2.26}$$

Kinematic trees are systems comprised of multiple open chains, which stem from at most one anchor frame and end in multiple end effectors frames. A typical example would be the human hand.

Closed chains occur if the chain includes one or more loops, which means that a (sub-) sequence of frames links back to a frame other than $\mathcal{E}$. We can characterize a closed loop of length k by the following sequence of frames, which occurs at position i of an open serial chain:

$${}^{i}T_{i+1}\ {}^{i+1}T_{i+2} \dots {}^{i+k-2}T_{i+k-1}\ {}^{i+k-1}T_{i} \tag{2.27}$$

In other words, the transformation matrix between frame i to $i+k$, which links back to frame i is equal to the identity matrix. An example of a closed chain is given in Fig. 2.5 Contrary to open chains, not all joints on closed chains need to be actuated, but can be *implicitly* set by the other (actuated) joints in the system. This may lead to additional constraints the system has to satisfy, but also to a larger variety of design and capabilities.

GRAPH REPRESENTATION An alternative way to describe the composition of a robotic system is by means of a *connectivity* graph, which models components of a system and their connections with the help of graph theory. This is addressed in Chapter 3.

MOBILITY Closely linked to the connectivity of a robotic system is the property of degrees of freedom. Degree of freedom is a measure of the number of independent parameters which are needed to fully specify a robotic system; it is directly dependent on the number of joints in the system and is a measure of the dimension of the configuration space.

A component in 3D-space has $K = 6$ degrees of freedom (three directional, three orientational), and a system of n components has $K = 6n$ degrees of freedom. This number can be reduced if constraints are introduced to these components, e.g. by restricting components either to only rotation or only position (e.g. planar movements), the mobility of each of these components would be $K = 3$. One of the most common type of joints, the revolute joint, imposes five constraints on the motion between the two connected bodies and allows only one degree of freedom in the form of rotation along its rotation axis. Table

2.1 gives an overview over the degrees of freedom and the imposed constraints of various joint types.

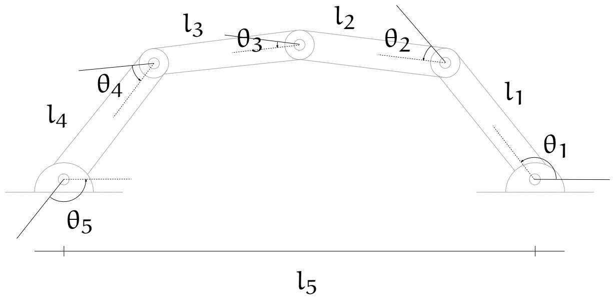


Figure 2.5: Schematic view of a closed loop system. $\theta_1, ..., \theta_5$ are the opening angles of the joints, $l_1, ..., l_4$ the length of the beams, and l_5 is the distance between both anchors.

Only joints whose constraints reduce the dimensions of the configuration space are taken into account (*holonomic constraints*), but no constraints, which do not reduce the configuration space dimension, but instead impose restrictions on the trajectory (*non-holonomic constraints*). An example for a joint with holonomic constraints is the revolute joint; it restricts the movement of its two connecting rigid bodies to be on an arc, and it is not possible for both bodies to move in a straight line relative to each other. An example for non-holonomic constraints are those imposed by car Ackermann steering. While steering (generally) allows movement in all directions with arbitrary orientation, thus not reducing any degree of freedom, constraints are imposed on the trajectory, e.g. by not allowing sideways movement or turning on the spot.

Thus, the *mobility* of a system can be stated by the well known formula (e.g., [58] [132], or [46] for a discussion of potential issues)

$$F = K(n-1) - \sum_{i=1}^{j} (K - f_i) \tag{2.28}$$

$$= K(n-j-1) + \sum_{i=1}^{j} f_i \tag{2.29}$$

where n is the number of links in the system, j is the number of joints in the system, f_i is the number of constraint on K by the i-th joint, provided that the constraints are independent.

In 3D-space, we set $K = 6$ to allow orientation and position and we get the

$$F = 6(n-j-1) + \sum_{i=1}^{j} f_i \tag{2.30}$$

which is known as the Kutzbach-Grübler formula [50]. By convention, n includes a reference frame, e.g. $\mathcal{W}$, relative to which the system moves.

Joint	Degree of freedom	Constraints
Prismatic	1	5 (2)
Revolute	1	5 (2)
Cylindrical	2	4
Helical	1	5
Spherical	3	3
Screw	1	5
Universal	2	4
Planar	3	3 (0)

Table 2.1: Planar and spatial joint constraints. Number of constraints are for the spatial joint types, in parentheses are for the planar version

The mobility formula does provide information about the inability of a system to move. *Overconstrained* systems are systems with mobility $F \leqslant 0$ that can still move; well known examples include the Sarrus linkage or Bennetts linkage. Likewise, *underconstraint* systems are systems with mobility $F > 0$ which can still move, but they have the disadvantage that they cannot follow any arbitrary trajectory.

For example, the planar dual SCARA (Fig. 2.5) consists of five revolute joints, each allowing one degree of freedom and five links inbetween. The mobility of the mechanism is thus $3(5-1-5)+5=2$.

Only systems with at least six degrees of freedom are able to arbitrarily position and orient its end effector in their workspace.

2.4 KINEMATIC EQUATIONS

Kinematic equations establish the link between the *configuration* of a robot and the position and orientation of its end effector, possibly as a sequence (*path*) or as a function over time (*trajectory*). In the case of the serial scara system, the end effector can be described as the vector $[x, y, z, \theta_e]^T$, describing the position in space and its rotation, which is directly linked to the angular values of its joints $\theta_1, \theta_2, \theta_3$. The *configuration* of a robot specifies the position of every point of the robotic system. In the case of rigid body systems, this usually entails only the value of the parameters of its joints, e.g. the opening angles or displacement values; also, the number of degrees of freedom of such a system is the smallest number needed to represent its configuration. The *configuration space* $\mathcal{C}$ of a robot contains all possible configurations of the robot, and a configuration in $\mathcal{C}$ is represented by a n-dimensional vector $c \in \mathcal{C} \subseteq \mathbb{R}^n$, where n denotes the number of parameters of the system.

The workspace of a robot is a subset of $\mathcal{W}$, in which it is physically located and where it performs its tasks. We can thus define the workspace as

$$\mathcal{W} = \{x \in \mathcal{W} : x = f(q), \text{for some } q \in \mathcal{C}\} \tag{2.31}$$

for some function $f : \mathcal{C} \mapsto \mathcal{W}$, which maps a configuration q to a point in the robot's workspace.

$\mathcal{W}$ describes the set of all reachable positions of the robot. The workspace can be divided into a *reachable* workspace, which the robotic system can reach with any orientation, and a dexterous workspace, where both, position and orientation, must coincide. [72].

The relationship between some point x in the workspace and some configuration q in the configuration space is established by the kinematic equations of a robotic system. The *forward kinematics*

$$f : \mathcal{C} \mapsto \mathcal{W} \tag{2.32}$$

calculates the position and orientation of the end effector of the robotic system from its configuration; the *inverse kinematics*

$$f^{-1} : \mathcal{W} \mapsto \mathcal{C} \tag{2.33}$$

calculates the configuration from the position and orientation of the robotic system.

To illustrate the forward- and inverse kinematics, consider the robot arm in Figure 2.6, which is also known as SCARA (Selective Compliance Articulated Robot Arm). It is compliant in $x - y$ plane and rigid along the $z-$axis. Geometrically, the position of the end effector in the x-y plane and its orientation for given values of θ_1, θ_2 can be given by:

$$x = l_1 cos(\theta_1) + l_2 cos(\theta_1 + \theta_2) \tag{2.34}$$

$$y = l_1 sin(\theta_1) + l_2 sin(\theta_1 + \theta_2) \tag{2.35}$$

$$\theta = \theta_1 + \theta_2 \tag{2.36}$$

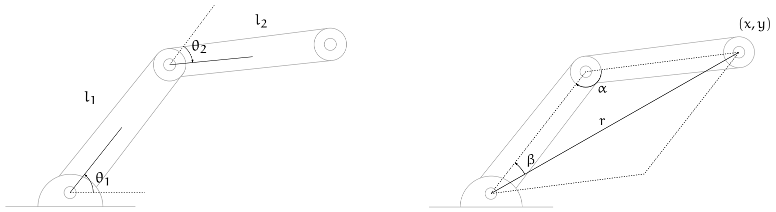
where l_1, l_2 denote the known and fixed length of the robots beams.

By using the reference frames of the robot, we calculate the product of the homogeneous transformation matrices

$$S = {}^{\mathcal{W}}T_1 \, {}^{1}T_2 \ldots {}^{n-1}T_{\mathcal{E}} \tag{2.37}$$

The x and y coordinates and the orientation are then given by the components ${}^{\mathcal{W}}p_{\mathcal{E}}$ and ${}^{\mathcal{W}}R_{\mathcal{E}}$ of S, respectively.

The inverse kinematic equations calculates the configuration θ_1, θ_2 for a specific position of the end effector, and its derivation is generally much harder. In addition, solutions to a specific problem might not exist (e.g., if the position is outside of the reachable workspace, or an orientation is required which does not lie in the dexterous workspace), or multiple solutions might exists (e.g., a mirrored configuration if



(a) θ_1, θ_2 are the opening angles of the joints, l_1 and l_2 the length of the beams

(b) The constructed triangles used for calculation of the inverse kinematics

Figure 2.6: Schematic view of an open chain system; its end effector moves in the $x-y$ plane.

the orientation is not specified, or, generally, in the case of redundant manipulators). In simple cases like an open chain as in this example, the inverse kinematic equation can be stated as a closed form expression. In other cases, e.g., closed chains, this is usually not the case, and solving the inverse kinematics amount to using numerical or heuristic approaches.

In this case, we can derive the inverse kinematics analytically, resulting in a closed form. We use the law of cosine and construct a triangle as shown in fig. 2.6; from that it follows that

$$\alpha = \cos^{-1}\left(\frac{l_1^2 - l_2^2 + x^2 + y^2}{2 * l_1 r}\right) \tag{2.38}$$

$$\beta = \cos^{-1}\left(\frac{l_1^2 + l_2^2 - x^2 - y^2}{2 l_1 l_2}\right) \tag{2.39}$$

$$\gamma = \mathrm{atan2}(y, x) \tag{2.40}$$

for $r = \sqrt{x^2 + y^2}$, which results in up to two possible solutions:

$$\theta_1 = \gamma - \alpha, \theta_2 = \pi - \beta \tag{2.41}$$

$$\theta_1 = \gamma + \alpha, \theta_2 = -\pi + \beta \tag{2.42}$$

If $r \notin [l_1 - l_2, l_1 + l_2]$, then no solution exists. In chapter 3, we will show an alternative way on how to find values for θ_0, θ_1 for a specific location of the end effector. In this example, we do not consider the orientation of the end effector, which results in what is commonly known as an *elbow-up* and *elbow-down* solution, i.e. a mirrored configuration of the arm along the axis anchor - end-effector. For systems with more than six degrees of freedom (*redundant manipulators*), e.g., a human arm, an infinite number of solutions to the kinematic equations exist.

Depending on the number of degrees of freedom, the resulting equations can be non-linear and solutions, if they exist, can not always be expressed in closed form. In general, closed form solutions are preferable, because they quickly return all solutions, but have the disadvantages that they are system-specific and are possibly hard to calculate. On the other side of the spectrum reside numerical methods, which are not tied to a specific robotic system, but can be slower and are not guaranteed to find all solutions.

2.5 REPRESENTATION AS FIRST ORDER LOGIC

Throughout the thesis, any equation or constraint is treated as a formula in first order logic. This allows us to use logical connectives like and $\wedge$, or $\vee$, negation $\neg$, but also the use of the existential quantifier $\exists$ and the universal quantifier $\forall$. The problems we encounter throughout this thesis consists for the most part of kinematic equations, along with constraints and *invariants*, i.e. facts about the system which must always hold true. Formulating the problem in this matter allows us to use automated reasoning tools, in particular those which are able to support the theory of reals with non-linear real functions. These tools solve these formulas by either finding an assignment of parameters, for which the formula evaluates to true (the formula is SAT, or satisfiable), or they provide a minimal set of clauses that are unsatisfiable. A proper introduction to first order logic can be found in many textbooks, e.g., [14].

2.6 DYNAMICAL SYSTEM

A dynamical system models the interaction between a controller and its environment over time as a mathematical model. The controller perceives the state of the environment (or parts thereof) and calculates its control input according to its control goal.

Figure 2.7 shows an overview of the individual parts of a dynamical system, and their interaction.

ENVIRONMENT The environment imposes restrictions upon the controller, for example in the form of obstacles, but also, on a more abstract level, it has full knowledge of the state of the robotic system. For example, if the robotic system is an underactuated system, i.e. if it has fewer actuated joints then degree of freedom, the non-actuated joints can be considered as set by the environment. This can lead to the robot not being able to follow certain trajectories, even if it would be otherwise capable. A dynamical system evolves over time T, and at any point of time $t \in T$, the environment can be in any state $x(t) \in X$.

In general, the controller cannot sense the entire state of the environment X. We call the state which the controller perceives Y; often, the perceived state is some subset $Y \subset X$, but that is no necessary always true, and can even be different. The output function models the input from the environment to the controller, and it is limited by the sensors and the appropriate rules available to the controller. It maps states to the output of the controller over time.

$$g : X \times U \mapsto Y \tag{2.43}$$

Only in theoretical settings it holds that $X = Y$; in practice, even with a rich model of the environment, the controller still depends on sensor readings, which are prone to noise and uncertainty.

CONTROLLER Based on the perceived state of the environment, the controller of a robotic system manipulates the mechanical part of the robotic system, and by its actions influences the environment. Denote by Y the set of inputs of the environment to the controller, and by U the output of the controller to the environment. The actions of the controller, which affect the environment, is given by the set of input functions

$$\Sigma = \{\sigma : Y \mapsto U\} \tag{2.44}$$

which return the control input based on the perceived state Y. Here, the controller uses Y for computing the control inputs, e.g., by means of using sensors like positional sensors or force sensors. This is called a closed loop controller. If the controller has no knowledge of Y, it is considered an open loop controller, setting its control input without any knowledge about the current state. In this case, the set of input functions is given as

$$\Sigma = \{\sigma : \emptyset \mapsto U\} \tag{2.45}$$

Typically, for a robot arm like the SCARA system, control inputs are forces or accelerations, and X consists of the configuration q, along with the velocities, or, if the robot can control the velocities directly, X consists of q. The representation of the controller, in particular the choice of σ generating the control inputs, is simplified. Because the controller only partially perceives the state, and is subject to noisy, uncertain or missing sensor readings, it tries to reconstruct an estimated state $\hat{x}(t)$ from $y(t)$, and based on the estimated state, determines a control input $u(t)$ which satisfies its control goal.

DYNAMICAL SYSTEM We can now define formally a dynamic system. Figure 2.7 summarizes the interactions between the controller and the environment within a dynamical system. Given a set of inputs U, a set of states X, a set of outputs Y, and a set of input functions Σ, the state transition function f

$$f : X \times U \mapsto X \tag{2.46}$$

calculates the state $f(x(t), u(t)) = x(t)$ at time $t \in T$ subject to the input signal σ. The controller perceives $y(t) = g(x(t), u(t))$, based on that chooses the next control input $u(t)$.

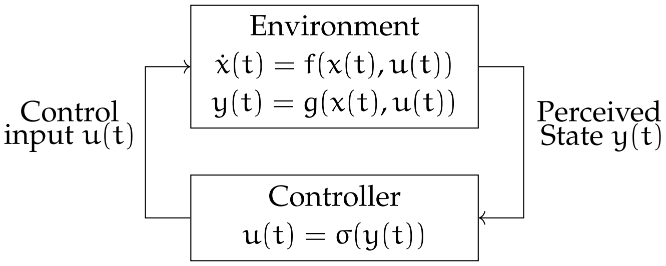


Figure 2.7: Interactions between controller and environment in a dynamical system. Note that f denotes the state transition function, which is different from the kinematic mapping of Eq. 2.32

Then the continuous time-invariant dynamical system is given by

$$\dot{x}(t) = f(x(t), u(t)) \tag{2.47}$$
$$y(t) = g(x(t), u(t)) \tag{2.48}$$
$$u(t) = \sigma(u(t)) \tag{2.49}$$

As outlined in Figure 2.3, the environment of a dynamical system can accommodate multiple robotic systems. The global state X is partitioned into a set of X_i for each individual controller i, and from the point of view of controller i, the environment is given by

$$W_i = X \setminus X_i \tag{2.50}$$

making it explicit that other controllers interact alongside controller i. We extend the state transition function f and the output function g to reflect the influence of the environment by

$$f : X_i \times W_i \times U_i \mapsto X_i \tag{2.51}$$
$$g : X_i \times W_i \times U_i \mapsto Y_i \tag{2.52}$$

We can then generalize the continuous time-invariant dynamical system for controller i by

$$\dot{x}_i(t) = f_i(x_i(t), w_i(t), u_i(t)) \tag{2.53}$$
$$y_i(t) = g_i(x_i(t), w_i(t), u_i(t)) \tag{2.54}$$
$$u_i(t) = \sigma_i(y_i(t)) \tag{2.55}$$

TRAJECTORY A trajectory ξ represents the configuration of the robotic systems as a function of time. Usually, this includes constraints which are not strictly necessary for geometric reasoning, but also constraints on joint velocities or torques. One commonly used approach is to first define a *path* that is decoupled from time. It serves as a representation of a curve in the robot's configuration space, ranging from the starting configuration to the target configuration. Subsequently, time is introduced, regulating the rate at which each configuration in the path is adopted by the robot. By adding time, further constraints such as as those governing joint velocities, torques, or the desired smoothness of the trajectory, can be formulated.

More formally, a path π is a mapping $\pi : [0,1] \mapsto X$, which maps an interval, usually ranging between 0 and 1 to specific configurations of the robotic system. $\pi(0)$ represents the configuration at the start of the path, $\pi(1)$ the configuration at the end of the path. In particular, we assume that the state space X is path connected, i.e. for any $x_0, x_1 \in X$, there exists a continuous path connecting both points:

$$\forall x_0, x_1 \in X \, \exists \pi : \pi(0) = x_0 \wedge \pi(1) = x_1. \tag{2.56}$$

A path is time-independent and is only concerned with the geometrical position of the robotic system. In contrast, a trajectory is time-dependent and considers, for example, speed and acceleration. By adding a continuous, monotonic time scaling

$$s : [0,T] \mapsto [0,1] \tag{2.57}$$

we assign a time scaling s to each $t \in [0,T]$. Thus, a trajectory ξ is given by the tuple (π, s). If clear from the context, we write $\xi(t)$ for $(\pi, s(t))$.

Speed and direction directly follow from the derivatives of ξ:

$$\dot{\xi} = \frac{d\xi}{ds}\dot{s} \tag{2.58}$$

$$\ddot{\xi} = \frac{d\xi}{ds}\ddot{s} + \frac{d^2\xi}{ds^2}\dot{s}^2 \tag{2.59}$$

CONTROL GOAL The control goal describes the task the controller tries to achieve. For example, the controllers in chapter 3 and 4 try to follow an input trajectory $\xi = (\pi, s)$, and their control goal can be stated follows: Given an initial state x_0, find a set of controls $\sigma : [0,T] \mapsto U$ for each π_i at time t_i, such that $\forall t \in [0,T], x(s(t)) = \pi(t)$. Often related to this goal is the idea of robot learning (e.g. [122]), where the input trajectory is given by a human directly physically manipulating the robot. The robot then tries to reconstruct and follow the learned trajectory.

Trajectories can be subject to various constraints which can affect the states at the beginning, end and in between. For example, moving the cart-and-arm system after grabbing an object requires the arm to be positioned in such a way that the center of gravity is low enough for the system to not get out of balance during movements. Ideally, the final state $x(T)$ of the arm trajectory should reflect the appropriate configuration of the arm; likewise, as a prerequisite for $x(0)$, the arm should not already have grabbed an object. In addition, it is easy to imagine additional constraints which have to hold for all points of time $0, ..., T$; for example, if the weight of the lifted object is close to the load limit of the arm, an additional constraint could state that the arm moves in such a way that the load on the arm never rises, e.g., by not extending the arm, but only retracting it. Complex motions, e.g., fetching an object, consist of several composite trajectories, each with their own constraints. These parts of the trajectory, along with their requirements and their underlying dynamical systems can be encapsulated into *motion primitives*.

MOTION PRIMITIVE Motion primitives can be considered as abstractions of dynamic controllers and the resulting trajectories. They provide encapsulated high-level capabilities, e.g. the HOME motion primitive, that brings the robot arm into a predefined configuration. The abstraction facilitates reasoning about dynamical systems by specifying conditions under which the motion primitive can be applied, conditions which hold during execution, and the conditions which are established at the end of the motion primitive's execution.

Formally, a motion primitive m is a tuple

$$m = (T, \mathrm{Pre}, \mathrm{Inv}, \mathrm{Post}) \tag{2.60}$$

consisting of a duration T , a pre-condition

$$\mathrm{Pre} \subset X \times W \tag{2.61}$$

an invariant

$$\mathrm{Inv} \subset ([0, T] \mapsto X) \times ([0, T] \mapsto W), \tag{2.62}$$

and a post-condition

$$\mathrm{Post} \subset X \times W \tag{2.63}$$

From the point of view of a dynamical system, motion primitives encapsulate the control input u and the underlying perceived state y for its duration, leaving only the changes in the state of the system x_i and the environment w_i visible.

Thus, we can formalize a valid trajectory ξ of duration T of a motion primitive m by requiring that at time $t = 0$ of the motion primitive, the state of the system must satisfy the constraints in Pre, and at the end at time T, the resulting state $x(T)$ must be in $Post$. Similar, any invariants must hold during the complete trajectory.

$$\xi(0) \in Pre \tag{2.64}$$

$$\wedge \xi(T) \in Post \tag{2.65}$$

$$\wedge \forall t \in T, \xi(t) \in Inv \tag{2.66}$$

Consider the motion primitive HOME, used in the Fetch example (Fig. 1.3. It is used to retract the arm to a safe position, which enables the cart to drive without tipping over. No preconditions are imposed, i.e., $Pre = \{\}$, which means that at the beginning of HOME, the motion primitive can be executed from any configuration the robot has. At the end of the motion primitive, the arm is required to be folded, thus after executing MOVE, $Post$ consists of the single state in which the arm is folded: $Post = \{(\theta_1 = \pi/6), (\theta_2 = 5/6\pi)\}$. In this case, no invariants need to be formulated.

3 CONTROLLER SYNTHESIS FOR ROBOTIC MANIPULATORS

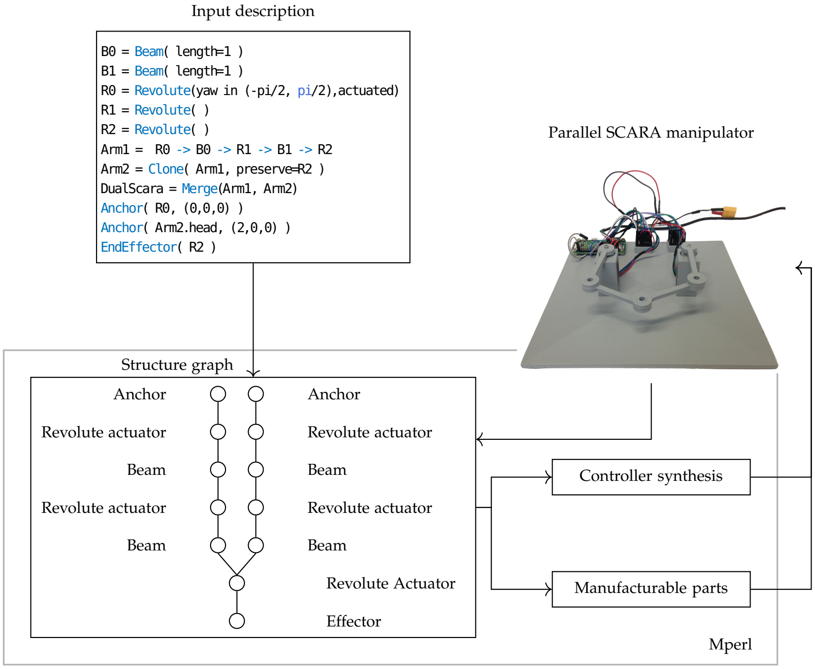
Advances in rapid prototyping and manufacturing technology have made building custom robots more accessible and affordable. Low-cost rapid manufacturing tools such as 3D printers and other CNC tools have simplified the creation of mechanical structures, and inexpensive micro controllers, sensors, and actuators can easily be added to produce functional robots. Custom robotic systems are not anymore limited to industrial settings; lower entry costs allow motivated tinkerers to build robotic systems on a limited budget. However, as designing and constructing robotic systems draw from many different areas, domain specific knowledge for each of these steps is required: designing the electromechanical components, figuring out how to control the robot, and implementing the code.

Mperl (Multipurpose Parallel End effector Robotics Language) allows non-expert users to create and test robotic manipulators programmatically. These manipulators can subsequently be produced utilizing 3D printing technology. Our primary focus lies in streamlining the programming process by automatically generating high-level motion primitives. These primitives facilitate the movement of specific components of a robotic manipulator, typically the end effector, to specific target locations. To demonstrate the functionality of our approach, we showcase its application in a SCARA system (Selective Compliance Assembly Robot Arm) using the Mperl language.

Mperl integrates elements from computational design and fabrication, inverse kinematic solvers, and feedback control. Mperl's input is an abstract view of the general structure of the robot, i.e., its different components and their connections to each other. Mperl supports any element which generates a rigid motion and comes with default geometries which can be used for fabrication. On the software side, we first perform an analysis to estimate the workspace of the structure and find the singularities within or at the boundary of the workspace. We then generate constraints corresponding to the structure in a form that can be handled by an inverse kinematic solver. Figure 3.1 summarizes the functionality of Mperl.

A structure in Mperl starts with anchors to which other components are attached. Anchors have fixed coordinates and cannot move. The other components' positions are constrained by the rigid transformations between the components and the anchors. The components roughly fill the following roles:

- *actuators* can be controlled, e.g. motors;



```
B0 = Beam( length=1 )
B1 = Beam( length=1 )
R0 = Revolute(yaw in (-pi/2, pi/2),actuated)
R1 = Revolute( )
R2 = Revolute( )
Arm1 =  R0 -> B0 -> R1 -> B1 -> R2
Arm2 = Clone( Arm1, preserve=R2 )
DualScara = Merge(Arm1, Arm2)
Anchor( R0, (0,0,0) )
Anchor( Arm2.head, (2,0,0) )
EndEffector( R2 )
```

Figure 3.1: Mperl workflow

- *sensors* are controlled by the environment but we can read the components' state;
- *passive* components are controlled by the environment and we cannot read the component's state.

The relative position of components is exposed with parameters which can be static or dynamic. For instance, the angle of an actuated revolute joint is a dynamic parameter while a beam is a prismatic joint with a static parameter.

The sensing elements can give feedback on the motors and also on the robot's overall structure. 3D printed structures are typically made of polymers and can be quite flexible. Origami style structures [90, 123] which have also been used to quickly build customized robots share the same problem. Mperl can embed deflection sensors in the structure at manufacturing time and when a load is applied to the system the sensor reports the deflection back to the controller. The controller can then account for the deflection and adapt the actuation to reach the current end effector target position.

CO-DESIGN. The term co-design is used to underline the equal consideration of hardware and software components in the controller synthesis of Mperl. Control software can be synthesized from an abstract description, along with any missing values, but it can also be synthesized by actually building the robot and analyzing and querying the physical structure. Physical components are embedded with electronics, which provide an uniform interface and the necessary communication protocols. Virtual and physical components can be mixed, which allows e.g. simulations of robotic systems incorporating real-world sensor data. Furthermore, it is possible to generate 3D-manufacturable parts from the abstract description.

In the following, we first introduce the graph representation and the language constructs of Mperl. We describe the underlying controller of Mperl, along with the elements which are synthesized, and we round off the chapter by giving a detailed example and an evaluation.

3.1 THE MPERL LANGUAGE

In the following section, we describe the language constructs of Mperl and their properties. Section 3.5 shows by means of the SCARA system how Mperl is used, along with the resulting equations.

SCOPE. In the scope of this work, we consider only kinematic models. Mechanical components are modeled as rigid bodies and the relation between them are rigid motions. We use the special euclidean group SE(3) for rigid movement; rigid transformations preserve distance and orientation.

```
RS         ::= Primitive Component
             | RS -> RS
             | Clone RS
             | Decompose RS
             | Merge RS RS
Modifiers  ::= EndEffector RS
             | Anchor RS Coordinate
Action     ::= Move RS Coordinate
             | Actuate RS Parameter Value
```

Listing 3.1: Mperl Language

Robots are created by composing small building blocks (*Primitive Components*) together to form larger structures, that can also be part of robotic systems, too. Operations like *clone, decompose* or *merge* help with the composition of robotic systems. The Mperl language (List. 3.1) supports the assignment of anchors and end-effectors to specific parts of the robotic system, and it is able to provide the *move* and *actuate* action. In the following, we explain in more detail each element.

Forming fundamental building blocks, primitive components fall roughly into the following categories: anchors, beams, joints, and actuators.

Anchors are used to connect a robotic system to the world frame $\mathcal{W}$. They specify an absolute position and orientation, cannot move and do not have any parameter.

Beams are static connections between components and have static parameters, for instance, the length. Varying static parameters result in different robots and they need to be fixed before the controller is synthesized.

Joints are connections with dynamic parameters. In the simplest form, the parameters are neither visible nor controlled. The position of the joint is constrained by the other elements in the system and, if under-constrained, the environment picks the joint configuration. If a motor is connected to the joint then it becomes an *actuator* and the controller can set the actuator's configuration. Every component can be equipped with *sensors*, which are able to read the actual configuration of a component and relay it back to the controller.

Abstractly, components are described by their signature, a triple

$$(P, C, S) \tag{3.1}$$

where $P \subseteq \mathcal{C}$ is the set of all parameters of a component (e.g., a primitive component, or several components bundled together, forming a larger component), $C \subseteq \mathcal{C}$ is the set of constraints imposed upon a parameter, and $S : P \mapsto \mathcal{W}$ maps parameters to their state.

Constraints model the limits on an element's motion, e.g. how much a revolute joint can turn, or how large the displacement is of a linear actuator. In general, constraints are predicates over the set of parameters. In particular, we assume that every parameter has an explicit lower and upper bound. These bounds are required to reason about robotic systems using automated solvers.

The set of parameters P consists of active parameters P_a and passive parameters P_p, i.e. $P = P_a \uplus P_p$. Active parameters can be set explicitly, e.g. the angular value of an actuator, and passive parameters are set by the environment. Components with passive parameters are encountered, for example, in underactuated components; the degree of freedom of these components is larger than the number of their actuated (active) parameters. This does not always has to be the case, for example, the Gough-Stewart platform includes passive revolute joints, but by design is not underactuated [31]. The value of passive parameters is implicitly existentially quantified.

Given a component (P, C, S) and a target state X_T, finding whether the component can induce this motion reduces to solving:

$$\exists p \in P : S(p) = X_T \wedge C(p) \tag{3.2}$$

We provide an initial set of components including revolute, prismatic, and spherical joints, and Mperl can also be extended with user defined components. Table 3.1 gives an overview of some of the sup-

Joint	Rotation Matrix ${}^{j}R_i$	Position vector ${}^{j}p_i$	Parameter
Revolute R	$\begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$	θ
Prismatic P	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} d \\ 0 \\ 0 \end{bmatrix}$	d
Helical H	$\begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} d\theta \\ 0 \\ 0 \end{bmatrix}$	$d\theta$
Cylindrical C	$\begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} d \\ 0 \\ 0 \end{bmatrix}$	θ, d
Spherical			
Planar P	$\begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} c_\theta d_x - s_\theta d_y \\ s_\theta d_x + c_\theta d_y \\ 0 \end{bmatrix}$	θ, d_x, d_y

Table 3.1: Selection of lower kinematic pairs.

ported components in Mperl, along with their states, parameters and constraints. The state matrix of component c is given by

$$S(c) = \begin{bmatrix} {}^{j}R_i & {}^{j}p_i \\ 0 & 1 \end{bmatrix} \tag{3.3}$$

3.1.1 Composing and Manipulating Elements

Primitive components are connected together to form a robotic system, resulting in a graph of interconnected components. A robotic system can contain other robotic systems; thus, Mperl can directly manipulate and compose graphs of individual robotic systems. The primitives are any joint natively supported by Mperl or provided by the user. The `->` operator concatenates components and is used to describe the general structure of the robotic system. Cloning is used to duplicate an existing structure; for example, modeling a system can be done by explicitly modeling every part of the system, or, if the system is symmetric in its structure, it is often sufficient to model one half, which is then cloned. To simplify the creation of more complex structures, we support the composition and decomposition of graphs into serial chains, and the cloning/merging of robotic systems. Section 3.3.1 provides a more detailed explanation, while Listing 3.3 offers a concrete example in form of the dual SCARA system.

Once a system is build, we can give specific roles to certain elements: anchors and end-effectors, which correspond to their counterparts ${}^{\mathcal{W}}\mathcal{A}$ and $\mathcal{W}$. Frames of components which are marked with *anchor* are ${}^{\mathcal{W}}\mathcal{A}$, and frames of *end-effector* components are treated as $\mathcal{E}$ frames. Assigning these attributes imposes a direction on the otherwise undirected graph structure and has direct influence on the resulting system. For example, the dual SCARA system has two anchors, attached at R_0 and R'_0, resp., and R_2 is used as an end effector. Swapping the position of anchors and end effectors results in R_0 and R'_0 being connected to end effectors, and R_2 being attached to an anchor. This reverses the direction of the graph, thus transforming the closed chain to a kinematic tree and the resulting mechanism resembles two finger.

3.1.2 Actuating the system

Actions include commands for manipulating robotic systems, most notably actuating parts (forward kinematics) and moving the end effector (inverse kinematic). We discuss how the inverse kinematic is handled in the next section. It is worth mentioning that we can use the action commands during the development to simulate the robotic systems; in combination with the properties commands we can inspect the robot's state in specific configurations.

In the next section, we discuss the graph structure on which Mperl operates and give concrete examples.

3.2 GRAPH REPRESENTATION

Denote by $G = (V, E)$ a graph G with vertices V and edges E. Vertices represent components, and an edge (v_i, v_j) between two components v_i, v_j represents the connection between them. A labeling function maps the vertices to the type of the component (revolute, prismatic, etc.) and its properties (anchors, effector). A *walk* of a graph is a sequence of alternating vertices and edges, starting and ending with a vertex; a *path* is a walk whose vertices are distinct. A path is called a *loop*, if the beginning and ending vertex is the same and each other vertex inbetween appears exactly once. The direction of the graph is given by assigning specific vertices the roles of anchor or end effector. Two vertices $v_i, v_j \in V$ are *adjacent* if they are directly connected by an edge $(v_i, v_j) \in E$, and the adjacency matrix A of a graph representing a robotic system with n components is a $n \times n$ matrix, where $A_{ij} = 1$ if v_i is adjacent to v_j. The in-degree of a vertex is the number of incoming edges, or, in terms of the adjacency matrix, the sum of the corresponding column entries. Similar, the out-degree of a vertex is the number of outgoing edges or the sum of the corresponding row entries of the adjacency matrix.

Using Mperl, we model a serial robotic manipulator in the form of a single SCARA system, and parallel robotic manipulator in the form of a dual SCARA robot and show the resulting systems and graphs.

3.2.1 Single SCARA system

Fig. 3.2 shows the single SCARA system, whose control code is generated by the Mperl code in Listing 3.2. The SCARA arm consists of two beams (shoulder and elbow), each 100mm long, which are connected to each other with a revolute actuator. The shoulder is attached to an anchor via a revolute actuator, and the end effector is attached to the revolute actuator at the end of the elbow beam.

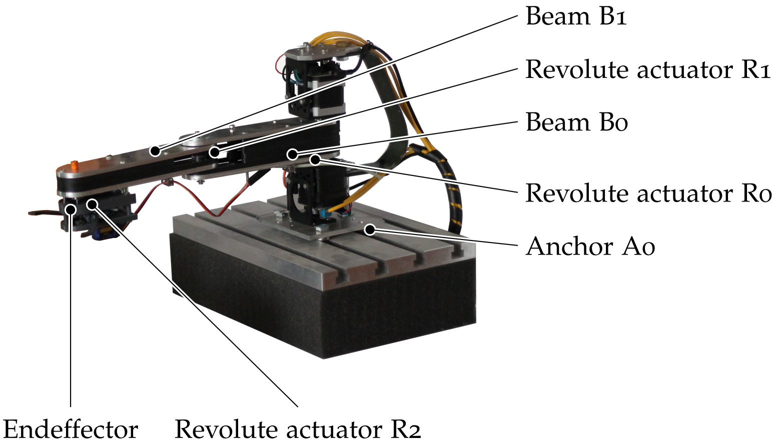


Figure 3.2: Single SCARA robot

In Figure 3.2, lines 1-5 initialize the building blocks of the robot — two beams with length 100 (mm), and three revolute actuators (i.e. joints which are actuated). The angular motion of each actuators is restricted to prevent self intersection, i.e. crashing into itself. If no specific constraints on the length or angle are given, default bounds are used. Line 6 uses the → operator to connect these elements into a chain. The remaining two lines add anchor and endeffector to the system.

```
B0 = Beam( length=100 )                                  1
B1 = Beam( length=100 )                                  2
R0 = Revolute(yaw in (-2pi/3, 2pi/3), actuated)          3
R1 = Revolute(yaw in (-2pi/3, 2pi/3), actuated)          4
R2 = Revolute(yaw in -pi, pi)                            5
Scara =  R0 -> B0 -> R1 -> B1 -> R2                      6
Anchor( R0, (0,0,0) )                                    7
EndEffector( R2 )                                        8
```

Listing 3.2: Input Description of SCARA system

From this description, Mperl builds the graph structure of the system in Fig. 3.3. For each primitive component, a node is inserted into the graph, and for each -> operator, an edge between its arguments is inserted. The direction of the graph is given by the connectiveness of its anchors and end effectors; with generally the direction being from anchor to end effector. An exception are closed loops, which are discussed in section 3.3.1.

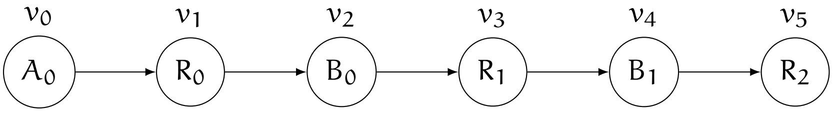


Figure 3.3: Graph representation of single SCARA system

The adjacency matrix of the single SCARA system is given by

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.4}$$

Each row consists of the elements $v_1, v_2, ..$, in order from left to right, and each column likewise from top to bottom. An entry in the i-th row and $j-$th column represents an edge from v_i to v_j. For example, the entry in the first row, second column is 1 and indicates a connection from vertex v_0, which corresponds to the anchor A_0, to vertex v_1, which corresponds to the revolute actuator R_1.

The direction is given by the position of the anchor (A0, node v_0), and the end effector node v_5, which represents the revolute actuator R_2. v_0 has no incoming edges, and v_5 no outgoing edges, indicating their status as anchor and end effector. In addition, no vertex has an in-degree and out-degree other than 0 or 1, which means that this graph represents a chain.

3.2.2 Dual SCARA system

A slightly more interesting example is the dual SCARA system. Fig. 3.4 shows the dual SCARA system, generated by the Mperl code in Listing 3.3.

```
B0 = Beam( length=50 )                                  1
B1 = Beam( length=50 )                                  2
R0 = Revolute(yaw in (-pi/2, pi/2),actuated)            3
R1 = Revolute( )                                        4
R2 = Revolute( )                                        5
Arm1 =  R0 -> B0 -> R1 -> B1 -> R2                      6
Arm2 = Clone( Arm1, preserve=R2 )                       7
DualScara = Merge(Arm1, Arm2)                           8
Anchor( R0, (0,0,0) )                                   9
Anchor( Arm2.head, (80,0,0) )                           10
EndEffector( R2 )                                       11
```

Listing 3.3: Mperl code for a dual SCARA system

The code is similar to Listing 3.2 and results in the graph structure shown in Fig.3.5. The adjacency matrix is given by

$$\left[\begin{array}{cccccc|ccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}\right] \tag{3.5}$$

The structure mirrors a single SCARA system, with R1 and R2 not being actuated, and the construction in Line 6 is the same. Because of the symmetry to the single SCARA system, Mperl allows us to duplicate (*clone*) this structure, while at the same time preserving R2. This is reflected in the adjacency matrix by extending its size from 6×6 to 11×11 and adding an additional edge between R_2 and the duplicated B_1 node (B'_1). The entries in the upper left block resemble the adjacency matrix of the single SCARA system. The direction of the graph is given by the position of the anchors (vertices v_0, v_{10}), and vertex v_5, which represents the revolute actuator R_2. The in-degree of v_5 is 2, the out-degree is 0, indicating that v_5 joins two chains and is an endeffector; likewise, v_0 and v_{10} have no incoming edges, making them anchor points.

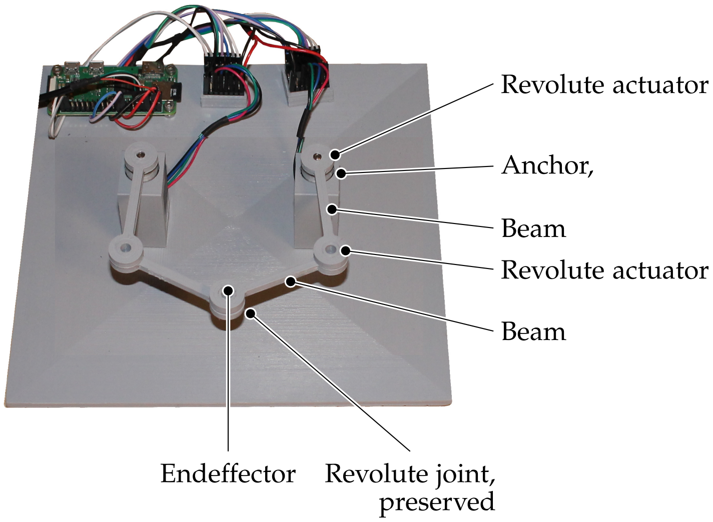


Figure 3.4: Dual SCARA robot, as generated by Mperl

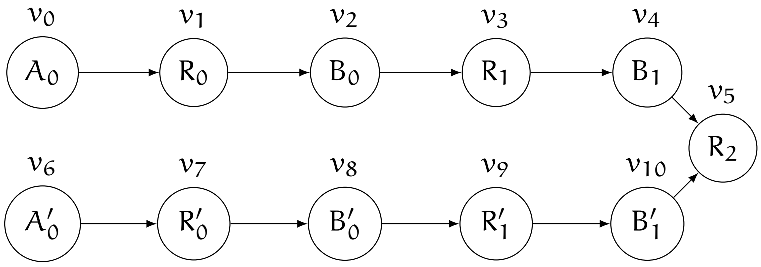


Figure 3.5: Graph representation of dual SCARA system

As previously mentioned, anchors and end effectors lend the graph its direction. Swapping anchors and end effectors in the dual SCARA example yields a kinematic tree anchored at R_2, with end effectors at R_0 and R'_0. If we do not change the order of vertices, the adjacency matrix changes to

$$\left[\begin{array}{cccccc|ccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right] \tag{3.6}$$

The previous anchor elements at v_0 and v_10 each now have an in degree of 1, and the previous end effector at v_5 has no incoming edges anymore, but two outgoing edges (to v_4 and v_10, which represent B_1 and B_1'.

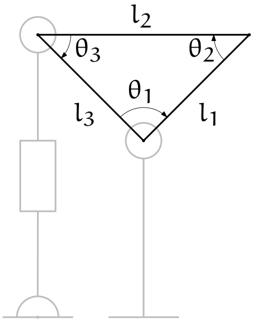


Figure 3.6: The black component depicts a rigid (sub-) chain. It has zero degree of freedom and has constant angles $\theta_1, \theta_2, \theta_3$ and constant lengths l_1, l_2, l_3. The grayed out parts show an example mechanism, in which that component is embedded.

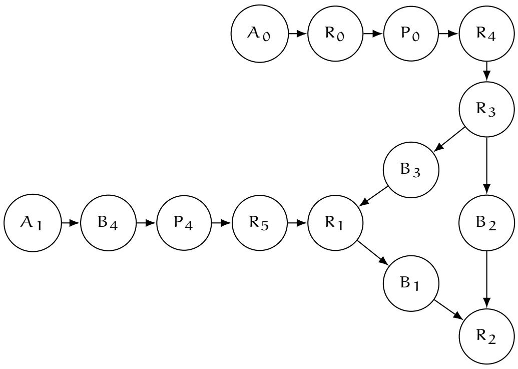


Figure 3.7: Graph representation of Fig. 3.6

The graph structure, along with the appropriate constraints, is also able to model polygonal structures. Polygons can be modeled as constrained rigid chains, i.e. chains with zero degree of freedom. For example, the mechanism in Fig. 3.6 has such a chain – depicted in black – embedded; its graph structure is given in Fig. 3.7. The subchain consists of three revolute joints ($R_1, \ldots, R_3$) and three beams ($B_1, \ldots, B_3$), and the set of constraints of the subchains restrict the corresponding parameters $\theta_1, \ldots, \theta_3, l_1, \ldots, l_3$ to constant values, thus removing any degree of freedom. The subchain itself can manipulated by the rest of the mechanism; in this case, the subchain rotates around the tip by adjusting the displacement of the prismatic actuator to the left, resulting in a circular motion of the right corner at R_2.

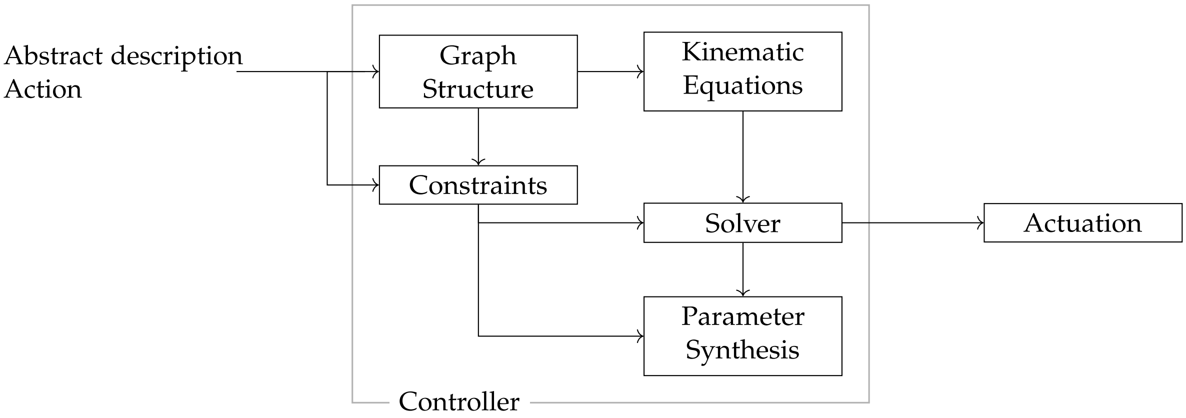


Figure 3.8: The abstract input description, along with the trajectory acts as input, from which the graph is generated. Graph and constraints are used to derive the kinematic equations, from which the actuation command (i.e. the control input) is calculated. The parameter synthesis derives missing values based on the system's model and the input trajectory.

3.3 CONTROLLER SYNTHESIS

The main benefit of using Mperl is not so much for realizing the physical structure but in getting some baseline software functionality implemented out-of-the-box. Not only can we generate commands to set some actuator values but we also provide the more user friendly MOVE command where the user specifies a target position for the end effector and Mperl computes the actuators values to reach the target by solving the inverse kinematics.

The Mperl controller consists of these parts:

1. Deriving and pre-processing the graph structure from the abstract input description
2. (Kinematic) control equations, derived via the system's graph from the abstract description
3. Synthesizing unknown parameters
4. Mapping the workspace and calculating singularities
5. Generating trajectories
6. Adding feedback loops for each sensor

Inputs to the controller are an *action*, i.e. a reference position the robotic system should attain either in configuration space or in workspace, along with the input description of the robotic system. The graph preprocessing evaluates the the graph structure of the robotic system, and, where appropriate, normalizes and reduces the graph. Based on the processed graph, we generate kinematic equations, along with any additional constraints, and depending on the input, we either solve the forward or the inverse version. In chapter 4, we introduce the sensor configuration synthesis, an approach to place sensors and to derive (additional) functional dependencies between components in the system. This approach affects the controller synthesis; it especially has effects on the workspace mapping (*Adaptive Grid*) and the singular regions. In the case that a trajectory is provided, the controller adheres to it accordingly. If only a list of specific configurations is supplied, the controller performs trajectory interpolation, ensuring the inclusion of each designated configuration. This interpolation process takes into account both singular regions and the workspace. If sensors are present in the system, feedback loops are generated for each sensor, such that the system can compare the target configuration with the actual configuration. This is implemented via a PI controller. For each sensor, a Kalman filter ensures signal smoothing (see also appendix).

In particular, two parts are done offline: The parameter synthesis, as it is used if an actual configuration of a component is not known, and the creation of the workspace mapping, as it has high costs upfront. After initialization, the workspace mapping can be updated during operation of the robotic system without incurring additional costs. It is also possible to use the system virtually, i.e. without actually sending actuation commands, for example, during the construction of the robotic system.

Fig.3.8 represents the controller synthesis workflow of Mperl.

3.3.1 Forward and Inverse Kinematics

From the graph structure of a robotic system, we can extract information to solve the forward and inverse kinematics. The forward kinematics compute the position of the structure given actuator values and the inverse kinematics computes the parameter values for a target configuration. The difficulty for these problems depends on the structure of the graph.

Chains, i.e., paths where all the vertices have degree 2 except the start and end with degree 1, are simpler to handle. Chains are also called serial structures. For graphs composed of a single chain, the forward kinematics is easy and boils down to matrix multiplication. On the other hand, the inverse kinematics and both the forward and inverse kinematics for parallel structures are much more difficult. These problems cannot, in general, be solved analytically and we rely on numerical solvers and optimization of Mperl for non-linear systems of equations. Below, we discuss how to find the parameters for a single configuration. For trajectories, we compute the configurations for points along the trajectories and interpolate the configurations between these points.

PREPROCESSING THE GRAPH The first step is to normalize and reduce the graph.

1. *Splitting the anchors.* As the anchors are fixed, having multiple elements connected to a single anchor is semantically equivalent to having these elements connected to their own copy of the anchor. This transformation may increase the number of vertices in the graph but preserves the edges. The goal of this splitting is to increase the serial parts of the graph. Denote by V_a the set of all vertices $v \in V$, for which an edge (a, v) between anchor a and v exists. Then, for any $v \in V_a$, we add a duplicate a' of a to V and an edge (a', v) to E.

2. *Collapsing chains.* Sequences of joints compose nicely and can be simplified. Two joints (P_1, C_1, S_1) and (P_2, C_2, S_2) are simplified to $(P_1 \uplus P_2, C_1 \wedge C_2, S_1 S_2)$ removing the intermediate node altogether. In this step, the graphs for parallel structures may become multigraphs.

OPEN CHAINS An open chain c of length n exists if exactly one node $v_a \in V$ labeled as anchor, exactly one node v_e labeled as end effector, and the vertices can be arranged such that there exists a path from v_a to v_e s.t. for a sequence of edges $v_1, v_2, ..., v_n$, $(v_i, v_j) \in E$ and $(v_a, v_1), (v_n, v_e) \in E$. For any two non-adjacent nodes v_i, v_j, it must hold that $(v_i, v_j) \notin E$.

The kinematic equations are the state $S(c)$ and are then given by

$$S(c) = \prod_{i=1}^{n} S(v_i) \tag{3.7}$$

The parameters are given by

$$P(c) = \bigcup_{i=1}^{n} P(v_i) \tag{3.8}$$

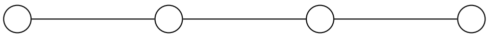
and, analogously, its constraints

$$C(c) = \bigwedge_{i=1}^{n} C(v_i) \tag{3.9}$$

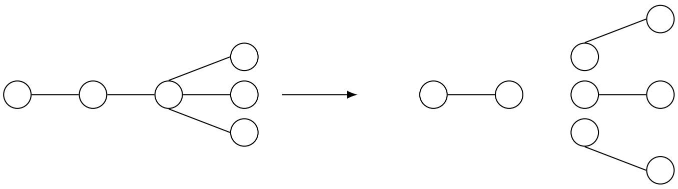
For the inverse kinematics, v_e typically assumes a fixed position (and orientation, if applicable). Denote the state of v_e by X_T, and the inverse kinematic is given by equation 3.2.

PARALLEL STRUCTURES To handle parallel structures, we reduce the problem to finding a common solution for multiple chains at the same time. We distinguish implicit and non-implicit chains (Fig. 3.9b, 3.9c).

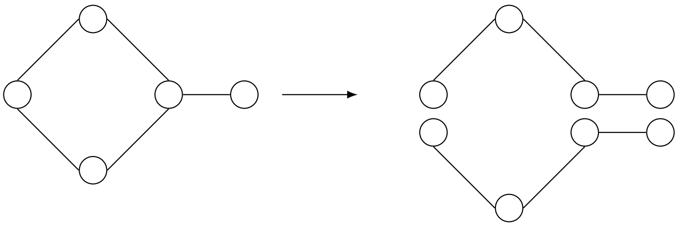
For the inverse kinematics, the end effector is assigned a fixed position assigned and, therefore, it becomes an anchor in the graph.



(a) For serial chains, no processing is needed



(b) Implicit chains are decomposed into a collection of serial chains. No additional constraints need to be generated



(c) Non-implicit chains are decomposed into a minimal set of paths covering all edges. Additional constraints are generated (possible node duplication)

Figure 3.9: Chain decomposition

Splitting that node is often sufficient to turn the graph into a collection of chains. We call these types of architectures *implicit chains*. Implicit chains can be decomposed into serial chains and are characterized by having multiple anchor and end effectors, but no loops, i.e there is no sequence of edges $e_i, e_{i+1}, ..., e_{i+j}$ with a corresponding vertex sequence $v_i, v_{i+1}, ..., v_{i+j}, v_i$. Kinematic trees, for example, are a typical example of implicit chains. They have exactly one anchor, multiple end effectors, and no loop.

Denote by v_s the splitting node, i.e. the node with out degree > 1, and by V_e the set of end effectors. For each $v_e \in V_e$, we build a sequence $v_a, v_1, ..., v_s, ..., v_e$. As v_s is the splitting node and the chain does not include loops, it follows that for each $v_e \in V_e$, the sequence $v_s, ..., v_e$ is itself at least an implicit chain, if not a serial chain. If the resulting sequence forms an implicit chain again, we split again until all resulting chains are serial. We can then proceed to calculate the kinematic equations for each split chain in the same way as for serial chains. The case for out degree of $v_s > 1$ follows analogously. For systems which do not decompose into chains, the first step is to find a minimal set of simple paths which covers all the edges in the graph. We do this using a depth-first search algorithm. Assume that the graph gets decomposed into n simple paths called P_i with $1 \leqslant i \leqslant n$. We use these paths to generate the constraints.

First, for each path P_i which starts and ends with anchors, we generate the same constraints as for chains. Second, for each vertex v in the graph which is not an anchor, we generate constraints to make sure all the paths going through v agree on the position of that vertex. More precisely, for each pair of paths P_i, P_j going through v, we compute the partial path constraints from one of the anchor to v. Let us call these positions $P_i \downarrow v$ and $P_j \downarrow v$. The additional constraint $P_i \downarrow v = P_j \downarrow v$ makes sure the solutions for individual chains are consistent in the global structure.

Solving the inverse kinematics requires finding a solution to the conjunction of all the constraints generated in the two steps described above. Implicit chains are easier to handle for the inverse kinematics solver as each chain can be solved independently. Otherwise, we need to solve the constraints over the entire graph. While we generate a number of constraints polynomial in the size of the graph constraints solvers are exponential in the number of parameters.

PARAMETER SYNTHESIS If the abstract input description is missing some values of parameters, which should be set prior to running the system, e.g., the length of a beam, we can use the kinematic equations, along with any constraints, to compute values for missing static parameters. This is especially useful if a target configuration / position or an input trajectory is provided. We can solve for values of the static parameters such that the system can reach all the target points with the same static parameters values.

Consider the single SCARA system from the running example, a target point of $(\sqrt{2}/2, \sqrt{2}/2, 0)$, and unspecified lengths of both the beams and the joints. Mperl returns a solution which contains possible values for the length of B0 and B1:

$$\begin{array}{ll} \mathtt{B0}_{length} = 0.765; & \mathtt{B1}_{length} = 0.999; \\ \mathtt{R0}_{yaw} = 1.963; & \mathtt{R1}_{yaw} = -1.964; \\ \mathtt{R2}_{yaw} = 0 & \end{array}$$

LIMITATIONS The kinematic equations include parameters which we cannot actuate (underconstraint mechanism); actuating the system may not give the expected result. To warn the user of potential problems we compute the degrees of freedom using the Grübler-Kutzbach criterion [50] and report an error if it does not match the number of active parameters. This approach can lead to false positives, as indicated in Chapter 2. For example, by partitioning the dual SCARA system into two single SCARA systems, each of these single SCARA systems yields a set of parameters $P = \theta$, but by the Grübler-Kutzbach criterion, the system has two degrees of freedom. In this case, one of the revolute joints is not actuated and can not be controlled by Mperl. To resolve this, the joint can either be restricted to a fixed value, thus removing one degree of freedom, or the joint can be actuated, thus increasing the parameter set.

At this point, Mperl does not check for self-intersection, i.e. the robotic system crashing into itself, as we allow the user to provide a custom geometry for the components. In Chapter 5, we show how robot interactions are verified for intersections.

3.3.2 Soundness

Mperl uses dReal [39], a SMT solver with theory of reals to solve its non-linear constraints. SMT (Satisfiability modulo theories) generalizes the Boolean satisfiability problem (SAT), which asks if a propositional boolean formula $f(x_1, x_2, .., x_n)$ evaluates to true. It produces a witness in the form of an assignment to the formula's arguments if the formula is satisfiable, or it produces a counter example if the formula evaluates to false. While SAT only targets propositional logic, SMT is able to check the satisfiability of formulas in decidable first order theories by providing additional information in the form of theories (e.g. theory of reals, or the theory of arrays). In our case, generated equations involve trigonometrics over real numbers, for which the standard decision problem is undecidable, and thus, the SMT approach is not directly usable. Instead, we use a relaxed formulation called $\delta-$decision problem [39], [40]. δ is a user specific numerical error bound, and for any formula ϕ we can decide whether that formula is unsatisfiable, i.e. no solution exists, or δ-satisfiable, i.e. a solution is guaranteed to exists up to some numerical perturbation of at most δ.

All variables x_i in ϕ can take any value from within a bounded interval I_i and we use a bounded quantifier $\exists^I x_i.\phi = \exists x_i.(x_i \in I \land \phi)$. Formally, the delta weakening is defined as: Let $\delta \in \mathbb{Q}_0^+$ be constant and ϕ be a Σ_1 sentence in standard form,

$$\phi = \exists^I x(\land_{i=1}^{m}(\lor_{j=1}^{k_i} f_{ij}(x) = 0)),\ x = x_1, x_2, .., x_n \tag{3.10}$$

The $\delta - weakening$ of ϕ is defined as

$$\phi_\delta = \exists^I x(\land_{i=1}^{m}(\lor_{j=1}^{k_i} |f_{ij}(x)| \leqslant \delta)),\ x = x_1, x_2, .., x_n \tag{3.11}$$

A typical formula ϕ in our setting consists of the conjunction of kinematic equations, along with constraints on the opening angles of the revolute actuators and constraints on the load bearing of each component. We use dReal [40], which implements a δ-complete decision procedure. In addition to determining the (un-) satisfiability of input formulas, it also produces a witness (i.e. a solution, if applicable), or a proof tree in case of unsatisfiability. In particular, this means that if dReal classifies an input formula ϕ as $\delta-$sat, it provides a solution $a \in \mathbb{R}^n$ s.t. $\phi_\delta(a)$ is true. The δ weakening can lead to spurious solutions; we address this issue in the relevant chapters.

While dReal returning unsat guarantees that no solution exists, a $\delta-$sat only guarantees solutions up to some numerical error bound δ, i.e. solutions which satisfy a δ-perturbation of the input. In practice, the reduction in accuracy does not carry much weight; physical robotic systems are not arbitrarily exact, and installed sensors and actuators have limited resolution. Instead, limiting the accuracy can lead to a much faster solving time. A highly accurate solution may not only be useless, but can potentially lead to time-outs or take so much time, that the produced solution is obsolete when found. This implies a possible soundness issue in our synthesis process, especially for larger values of δ. Therefore, any synthesized solution is cross-checked against the model, and if the tentative solution cannot be verified, it is dismissed. In addition, if a trajectory is provided, Mperl uses the discretized workspace (section 3.3.3) to check that every (discrete) state of the trajectory is path connected and that this path does not cross any singularity region. If the trajectory crosses a cell, we assume that the robot can be anywhere within that cell (over-approximation); if the cell is close to a singularity, we try to find an exact partial path in that cell without crossing the singularity. This idea is similar to interval abstraction [6, 102]. In praxis, we did not encounter any problems arising from this problem, especially considering the nature of the (additive) manufacturing process and that Mperl is not intended for safety critical systems, but for simplifying creation and programming of robotic systems by combining programming language and robotic design.

3.3.3 Workspace Mapping and Singularities

We follow trajectories by computing the joints configuration for points along the trajectories and interpolate between these configurations. This only works if the robotic system is not in a singular configuration, i.e. a configuration in which the system looses some degree of freedom and cannot operate as expected. These singularities happen at the boundary of the workspace, e.g. when a serial arm is fully extended (*boundary singularity*), or inside the work space, e.g. if two or more axis of motion are aligned (*internal singularities*); singularities can also lead to sudden occurrences of very high input joint velocities.

SERIAL STRUCTURES The (forward) kinematic mapping $S : \mathcal{C} \mapsto \mathcal{W}$, maps a robots configuration to the position of the end effector x, and for $x = f(q)$, the first order differential equation is given by differentiating x after time.

$$\dot{x} = J(q)\dot{q} \tag{3.12}$$

where $\dot{x}$ is the vector denoting positional velocity of the end effector, $\dot{q}$ the vector denoting the change in velocity of the configuration, and $J(q)$ is the $m \times n$ analytic Jacobian $\partial p / \partial q$ and q is a $n \times 1$ vector.

For a general purpose manipulator, which is able to freely position and orient its end effector in its dexterous workspace, $m = 6$, and x

contains a (minimal) representation of position and orientation, e.g $[x, y, z, \psi, \theta, \phi]$, and J has full rank if rank $J(\theta) = min(m, n)$.

J becomes singular at configuration θ_s, if $rankJ(\theta_s) < max_\theta rank(\theta)$; in other words, J looses degrees of freedom at configuration θ_s, compared to other configurations, and the robotic system is not able to move in at least one direction. a The mobility of a robotic system is also reflected in J; if J is in a non-singular configuration, if $n = m$, the system has full mobility, and if $n < m$, or $n > m$, J reflects kinematically deficient systems, which are not able of all m degree of freedom, and redundant systems, which can attain multiple solutions for a given end effector position and orientation, respectively.

PARALLEL STRUCTURES We follow the approach in [47] to guide the singularity analysis for parallel structures. Reformulating the kinematic mapping from Eq. 2.32 as an implicit function, we can write the relation between x and q as

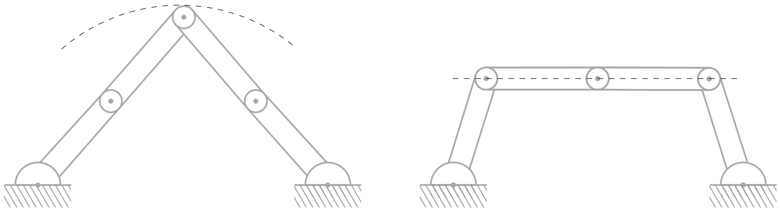
$$F(q, x) = 0 \tag{3.13}$$

where 0 is the n-dimensional zero vector. By taking the derivative of F with respect to time yields

$$\begin{bmatrix} \frac{\partial F}{\partial q} \cdots \frac{\partial F}{\partial x} \end{bmatrix} \begin{bmatrix} \dot{q} \\ \dot{x} \end{bmatrix} = J_q(\dot{q}) + J_x(\dot{x}) = 0 \tag{3.14}$$

Depending on which Jacobian (J_q or J_x) looses rank, three different types of singularities can be distinguished for general closed loop chain. Type 1 singularities occur if $det(J_q) = 0 \wedge det(J_x) \neq 0$ and is comparable to a kinematic singularity of a serial mechanism. This implies that $\exists \dot{\theta} \neq 0$ s.t. $\dot{x} = 0$, i.e. the mechanism looses degree of freedom. While the position and orientation of the end effector does not change, at least one joint can travel an infinitesimal distance; in particular. If the input trajectory to a system consists of points in the configuration space, this type of singularity is not critical and the system can operate in their vicinity. In the case of the input being points in the workspace, generated joint velocities can reach infinity. These type of singularities are found at the edge of the workspace, and involve collinearity of at least two components (Fig. 3.10a) or if links of a chain are folded back on to themselves.

If $det(J_x) = 0 \wedge det(J_q) \neq 0$, this leads to $\dot{x} \neq 0$ for $\dot{q} = 0$, which occurs, e.g., if two joints are locked (Type 2 singularity). Even though joints are locked up, the end effector can still be moved. Multiple solutions of the forward kinematics may be found for one configuration, and the system gains one or more degrees of freedom and gets uncontrollable. The system can get stuck in these configurations and cannot proceed by itself; therefore, parallel systems are typically not operated close to these singular regions [25]. Type 2 singularities can be found inside the workspace and represent a border, where



(a) Singularity type 1. The dashed line represents the edge of the workspace, where the singularity occurs.

(b) Singularity type 2. The dashed line represents an inner workspace boundary; it forms a transition from one configuration space to another.

Figure 3.10: Examples of type 1 and type 2 singularities of a dual scara arm

two configuration spaces met. [16]. In Fig. 3.10b, such singularity is shown. The dual SCARA system workspace is separated at this border and special precautions have to be undertaken if that border is to be passed, e.g. by temporarily underactuating the system.

A type 3 singularity occurs if both Jacobians, J_q and J_x, are rank deficient. A typical example for this type of singularity is when the output variable is constant for any value of the input variable.

A robotic system can also loose degrees of freedom if parameters are constrained and the current value(s) of that parameter are at the border of the constraint, effectively prohibiting any further movement of the end effector.

CLOSENESS TO SINGULARITIES For practical purposes, it is not only interesting to see if a configuration is singular, but also how close to a singularity it is. The determinant itself does not give any information about the closeness to a singularity; instead, we turn to the singular values as a measure. Mperl uses Singular Value Decomposition to find the direction towards which the manipulator is most difficult to move, and therefore, to get the closeness to a singularity [34, 83]. More precisely, we compute the Minimum Singular Value (MSV) [66, 67] as follows.

The Jacobian of the robotic system can be written as

$$J = U\Sigma V^T \tag{3.15}$$

where U is a $m \times m$ diagonal matrix, V a $n \times n$ diagonal matrix, and Σ the $m \times n$ diagonal matrix consisting of the singular values σ_i:

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \sigma_m \end{pmatrix} \tag{3.16}$$

The MSV is then given by $\min(\sigma_1, \dots, \sigma_m)$, and if the robot is in a singular configuration, then $\min(\sigma_1, \dots, \sigma_m) = 0$.

To find Σ, we need to derive U and V. In a first step, we find the orthonormal eigenvectors of $J^T J$, which define the matrix V. Denote by $\lambda_1, \lambda_2, ...$ the eigenvalues of $J^T J$. Then, $v_1, v_2, .., v_i$ are the corresponding orthonormal eigenvectors to λ_i, and V is given by

$$V = \begin{bmatrix} v_1 & v_2 & ... \end{bmatrix} \tag{3.17}$$

Σ has the same dimensions as J, and its diagonal entries Σ_{ii} consist of λ_i^2; all other entries are 0.

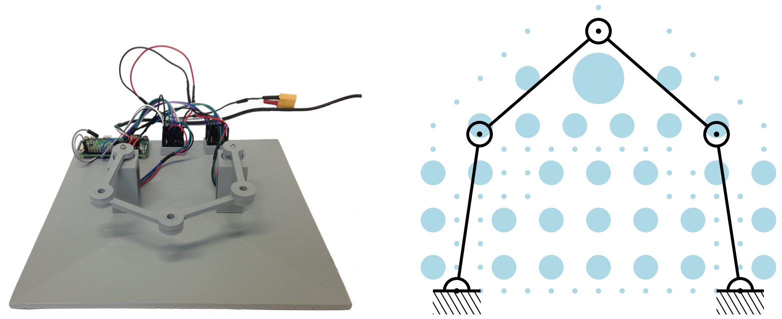
If $\lambda_i \neq 0$, the i-th column if U is given by $\lambda_i^{-1} J v_i$; otherwise, the remaining columns of U can be arbitrarily extended to form an orthonormal basis in $\mathbb{R}^m$.

ADAPTIVE GRID During initial setup of the robotic system, we discretize the manipulator's workspace into cells, marking each cell with its distance to a singularity. When moving the robotic system near a possible singularity, the grid can be refined only in this area, and if a trajectory moves towards or through a singular region, an error is reported. Figure 3.11 shows the singularity map for the dual scara arm. The workspace is discretized evenly into k cells of size s, each cell labeled with the minimum singular value the mechanism attains in the center of each cell. If a trajectory passes through a singular cell, this cell again is split into k cells of size s/k, each of which is again marked with its minimum singular value.

Mperl is able to adjust the precision of it calculations and thereby, of the resulting motion of the system. If there are no additional constraints on ξ or parts thereof, Mperl allows the user to use the cell size s as an upper bound on the numerical error δ. In this case, the actual target position is automatically adjusted to be the midpoint of the cell containing the actual target and δ is set to $s/2$. Thus, for large parts of ξ, the calculation of the control input can be sped up by allowing less precise solutions, and in parts, where the trajectory is near singular regions or close to the final state, the precision can be increased (at the cost of computation time).

3.3.4 Trajectory Generation

The final part of the controller synthesis is the trajectory generation, which takes as input a list of configurations or coordinates. While it is preferable to generate trajectories in the configuration space, Mperl also allows the trajectory to consist of coordinates in the workspace. Depending on the user's preference, Mperl offers simple linear interpolation and spline curve generation. In particular, using splines allows for smoother trajectories, as continuous acceleration can be emphasized. Generated trajectories respect the adaptive grid and the calculated singular regions. If a path crosses a singular region, the



(a) Synthesized dual SCARA system

(b) Singularity Map for the dual SCARA system. The smaller the dots, the closer the end effector is to a singularity.

Figure 3.11: Dual scara arm

adaptive grid can be refined in that area, or the system can try to find an alternative route which does not lead close to a singularity.

The synthesized controller so far consists of the generation of the kinematic equations, based on the graph structure, along with the singularity mapping and the trajectory generation. Control inputs depend solely on the input trajectory, and the controller cannot sense its state. In this case, we have an open-loop controller; its control inputs to the environment are done without any knowledge about the environment. In the next section, we add a feedback loop to the system in form of two types of sensors, which enables the system to sense its current state. Adding this loop effectively allows the controller to read (part of) the environment, and thus, the resulting controller can be considered a closed loop controller.

3.4 SENSORS AND FEEDBACK

Sensors can be embedded in beams (see Figure 3.12) or in actuators and become extra nodes in the graph. A sensor reports the error between the expected configuration and the current state. In the case of an actuator, this is the distance to the set point and, for structural elements, this is the deflection. For example, if the sensed position is equal to the target position, the sensor's configuration matrix is the identity matrix I; otherwise it reflects the error. At run-time the sensors are polled, the corresponding transformation matrices are updated, and the inverse kinematics solver uses this information to update the actuation values. We currently use a simple proportional feedback controller as more complex controllers such as PID controllers are often unstable in parallel structures.

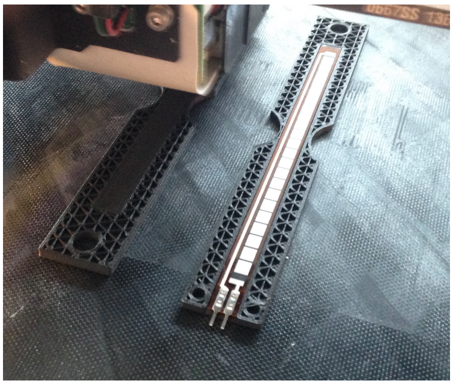


Figure 3.12: Embedding sensors directly into the structure of a component

Listing 3.13 gives the source code for a single scara arm, which automatically adjusts load induced deflection. The *FlexBeam* consists of a beam embedding a load cell. As soon as the system polls the sensor and notices a deviation, it automatically engages the actuated joint in R1 to counter the difference in position (Fig. 3.14). As soon as the weight is attached or removed from the arm, the position of the arm will change. The load cell senses the change in position and the system corrects for it by actuating the revolute actuator located at the elbow of the arm.

```
S0 = FlexBeam( length=150mm )
B0 = Beam( length=150mm )
R0 = Revolute(yaw in (-pi/2, pi/2),actuated)
R1 = Revolute(yaw in (-pi/2, pi/2),actuated)

Arm =  R0 -> S0 -> R1 -> B0
Anchor( R0, (0,0,0) )
Anchor( Arm.head, (2,0,0) )
EndEffector( B0 )
```

Figure 3.13: Mperl code for a single scara arm with flex sensor

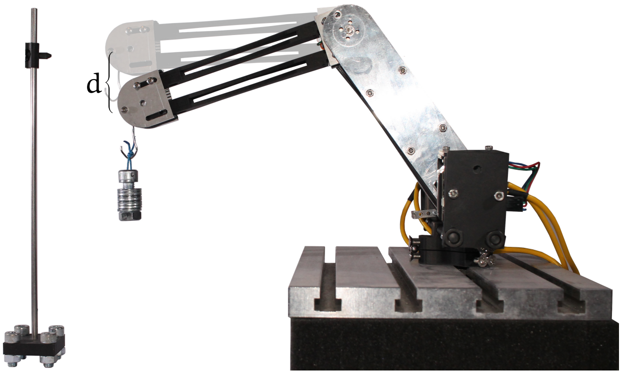


Figure 3.14: Load induced deflection d

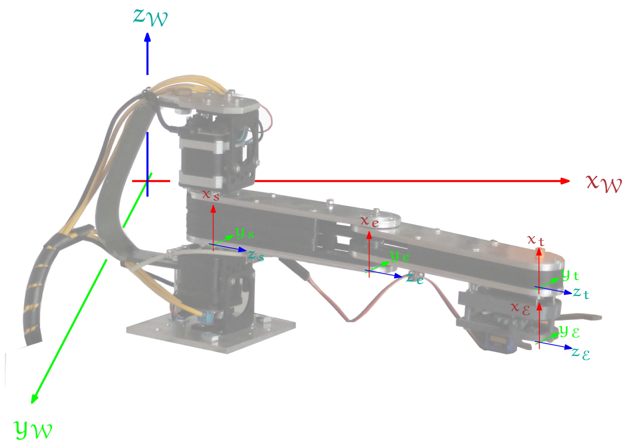


Figure 3.15: SCARA system

By embedding flex sensors into the beams and positional sensors into revolute joints and actuators, feedback is provided to the controller about the actual state of the system. The control input u now depends no longer only on time t, but also depends on the current (perceived) state Y the system is in. The synthesized controller is thus a closed loop controller.

3.5 EXAMPLE: SCARA SYSTEM

The SCARA system (Fig. 3.15) is a serial kinematic chain, consists of alternating revolute actuators (`R0, R1, R2`) and beams (`B0, B1`). The abstract description in Listing 3.4 acts as input to Mperl. Both beams are $100mm$ long, and all three revolute actuators are actuated, i.e. their angle can be explicitly set, their opening angle being restricted to be in the interval $[-2\pi/3, 2\pi/3]$ for the elbow and shoulder beams, and $[-\pi, \pi]$ for the end effector (Line 1-5). In line 6, the general structure of the system is determined, the following two lines set anchor and end effector of the structure. The anchor `R0` is located at position $(0, 0, 0)$ in the world frame.

```
B0 = Beam( length=100 )
B1 = Beam( length=100 )
R0 = Revolute(yaw in (-2pi/3, 2pi/3), actuated)
R1 = Revolute(yaw in (-2pi/3, 2pi/3), actuated)
R2 = Revolute(yaw in -pi, pi)
Scara =  R0 -> B0 -> R1 -> B1 -> R2
Anchor( R0, (0,0,0) )
EndEffector( R2 )
Move Scara (170,170,0)
```

Listing 3.4: Input Description of SCARA system

From this description, Mperl builds the graph structure of the system. As the system is a serial chain, there is no graph preprocessing to do.

From the graph structure, Mperl generates the orientation and position formulas of the end effector R2, that are then translated into the SMT v2 format, ready to be used with a SMT solver, e.g. dReal. Listing 3.5 shows the generated first-order formula in smt v2 format. For clarity, the orientatin formulas are omitted. Lines 4-11 declare all parameters which occur in the system, directly corresponding to the set P. The relevant constraints for each parameter are declared in Lines 12-23. For this example, we restrict the opening angles of the revolute actuators and set a fixed length for the beams. Lines 24-26 set the target position. Lines 27 - 29 represents the position of the end effector. These lines directly correspond to Eq. 3.2.

```
(set-logic QF_NRA)
(declare-const pi Real)
(assert (= pi 3.1415926535 ) )
(declare-fun theta0_yaw () Real)
(declare-fun Target_x () Real)
(declare-fun theta1_yaw () Real)
(declare-fun l1_length () Real)
(declare-fun theta2_yaw () Real)
(declare-fun l0_length () Real)
(declare-fun Target_y () Real)
(declare-fun Target_z () Real)
(assert (>= theta0_yaw (- (* (/ 2 3) pi)) ))
(assert (<= theta0_yaw (* (/ 2 3) pi)) )
(assert (>= theta1_yaw (- (* (/ 2 3) pi)) ))
(assert (<= theta1_yaw (* (/ 2 3) pi)) )
(assert (>= theta2_yaw (- pi) ))
(assert (<= theta2_yaw pi ))
(assert (<= l0_length 100 ))
(assert (>= l0_length 100 ))
(assert (>= theta1_yaw (- pi) ))
(assert (<= theta1_yaw pi ))
(assert (<= l1_length 100 ))
(assert (>= l1_length 100 ))
(assert (= (+ 100 (* 50 (^ 2 (/ 1 2)))) Target_x ))
(assert (= (* 50 (^ 2 (/ 1 2))) Target_y ))
(assert (= 0 Target_z ))
(assert (= (+ (* 100 (cos theta0_yaw)) (- (* 100 (sin theta0_yaw) (sin
    theta1_yaw))) (* 100 (cos theta0_yaw) (cos theta1_yaw))) Target_x ))

(assert (= (+ (* 100 (sin theta0_yaw)) (* 100 (cos theta0_yaw) (sin
    theta1_yaw)) (* 100 (cos theta1_yaw) (sin theta0_yaw))) Target_y ))
(assert (= 0 Target_z ))

(check-sat)
(exit)
```

Listing 3.5: SMT v2 Code for SCARA system

After running dreal with the input from Listing 3.5, the solver either returns *delta-unsat*, if no solution exists, or *delta-sat*, if a solution exists, along with a set of parameter as witness. See also 3.3.2 for a discussion on why a solution is found if one exists. In our example, we provided a target position the arm can reach and found an appropriate configuration (Listing 3.6, Lines 8-10). We can use a smaller or larger delta to speed up the solution finding, depending on how accurate our solution has to be.

```
< delta-sat with delta = 0.001
pi : [3.14159265349999961, 3.141592653500000054]
l1_length : [100, 100]
l0_length : [100, 100]
Target_x : [170.7106781186547551, 170.7106781186547551]
Target_y : [70.71067811865475505, 70.71067811865475505]
Target_z : [0, 0]
theta0_yaw : [-0.7858981633749999585, -0.7848981633750000686]
theta1_yaw : [0.7848981633750000686, 0.7858981633749999585]
theta2_yaw : [0,0]
```

Listing 3.6: Results of running the code from Listing 3.5

In the input description (Listing 3.4, Lines 3-5), we saw that the revolute actuators had the keyword *actuated*, which assigned their parameters to the set of active parameters P_a. For each of these parameters, Mperl sends the numerical value to the corresponding component, which actuates the system.

The SCARA system itself can be considered a component itself, similar to, e.g., a revolute joint, and can be reused. Thus, we can consider the SCARA system as a component with parameters $\theta_0, \theta_1, \theta_2, l_0, l_1$, which represent the angles of the revolute actuators and the length of the beams. The rotation matrix ${}^jR_i$ and position vector ${}^jp_i$ follow directly from the state matrix of the end effector.

3.6 EXAMPLE: SCARA SYSTEM AS A COMPONENT

If we change line 7 in the description of the SCARA system (Listing 3.4) to

```
Anchor( R0 )
```

the anchor ist not set to a fixed position (e.g., at position (0,0,0)), but the position becomes a parameter. Thus, the set of parameters consists of

$$P = \{anchor_x, anchor_y, anchor_z, \theta_1, \theta_2\} \tag{3.18}$$

and the equations change subsequently to reflect the parameterized position of the arm (Listing 3.7).

```
(declare-fun anchor_x () Real)
(declare-fun anchor_x () Real)
(declare-fun anchor_z () Real)
(assert (= (+ Anchor0_x (* 100 (cos theta0_yaw)) (- (* 100 (sin theta0_yaw)
     (sin theta1_yaw))) (* 100 (cos theta0_yaw) (cos theta1_yaw)))
    Target_x ))
(assert (= (+ Anchor0_y (* 100 (sin theta0_yaw)) (* 100 (cos theta0_yaw) (
    sin theta1_yaw)) (* 100 (cos theta1_yaw) (sin theta0_yaw))) Target_y )
    )
(assert (= Anchor0_z Target_z ))
```

Listing 3.7: Changes in SMT v2 Code for SCARA component as opposed to the stand-alone system

The SCARA component can be reused or combined with other components; for example, the whole system can be rotated $\pi/2\,\mathrm{rad}$ along the y-axis and put onto a cart. The cart's position and orientation can be represented, e.g., by a planar joint, which results in a system similar to the cart-and-arm system (Fig. 1.2a). The arm's anchor position is then set relative to the position on the cart and the absolut position of the arm in $\mathcal{W}$ would then be dependent on the location of the cart. Listing 3.8 shows the Mperl code for this example.

```
scara = Scara()                 1
cart = Planar()                 2
cart_arm = cart -> scara        3
Move cart_arm (0,0,0)           4
Move Scara (170,170,0)          5
```

Listing 3.8: Input description of the SCARA system mounted on a cart

LIMITATIONS Using the compositional aspects of Mperl allows to quickly prototype systems like the cart-and-arm system. In this setting, the cart-and-arm system is considered to be one system, with no clear distinction between the cart and the arm. While it is possible to set individual parameters of the arm, and to only re-calculate the arm, there is no higher level functionality like communications. For instance, if moving the scara arm (e.g., Line 5 in Listing 3.8) to a non-reachable point of the arm, the cart does not automatically update its position to allow the arm that movement. Chapter 5 addresses this issue by modelling the cart as well as the arm as *processes*, which control the physical parts individually, have the ability to communicate and execute motion primitives.

3.7 EVALUATION

We implemented Mperl in Python. The equations and constraints are manipulated using the Sympy library [94] and, to solve kinematics equations, we provide support for the dReal SMT solver [41], least squares optimization[1], and our own implementation of cyclic coordinate descent [134].

Nodes in the graph can be either actual physical devices or virtual nodes (loop devices). As an actual physical device, each actuated or sensing building block consists of a standardized interface, running on an Atmega micro processor and the component which does the acting or sensing, e.g. a motor or a rotary sensor. Communication between nodes is done using UART or SPI. As a virtual device, we provide an implementation as Python classes. It is also possible to mix virtual and physical classes, e.g. to have a simulation of a robotic system, which incorporates real-world sensor data. We support all lower kinematic pairs and also provide some support for higher kinematic pairs and wrapping pairs.

To generate physical parts from the structural description, we provide parameterized OpenSCAD[2] descriptions, which can be user modified, e.g. different beam designs. These parts are then 3D printed (in the case of the dual scara system), or manufactured by both additive (3D printer) and subtractive process (milling machine).

CONTROLLER Each robotic system's central controller is running on a raspberry pi, that takes as input the graph of the system, from which the kinematic equations are derived. Each component in the system has a micro controller attached (*component controller*), which is able to interface with the electro-mechanical component (either directly, or via a *component driver*) of the component, or is able to read out and convert sensor readings. Each component controller implements a standardized interface and is connected to a common message bus. The advantage in using component controllers (instead of directly interfacing with the electro mechanical parts) over a message bus is in an additional abstraction. In place of translating the results of the calculation, e.g., into steps or torque / current values, the controller just sends the numerical values of the parameters as control input to the associated component controller. This keeps the controller itself simple, allowing even a casual user to modify and extend the system. The component controller takes care of translating the numerical values into a signal the component or its driver understands, and, if sensors are present, implements a local feedback loop. The controller can also query component controllers for their current state, as well as

---

[1] We use scipy.optimize.least_squares.

[2] `https://www.openscad.org/`

query additional properties like mass, material, etc. (See also chapter 4)

Fig. 3.16 shows the high level block diagram of the controller which underlies Mperl. Additional information about the implementation can be found in the appendix b.

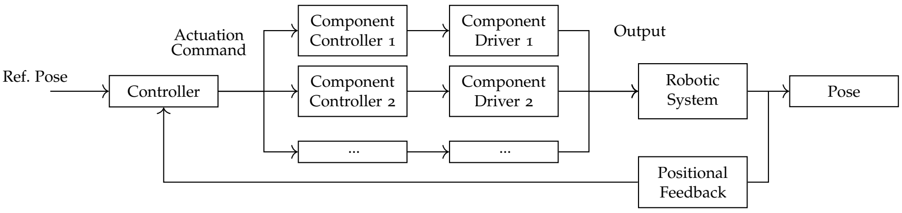
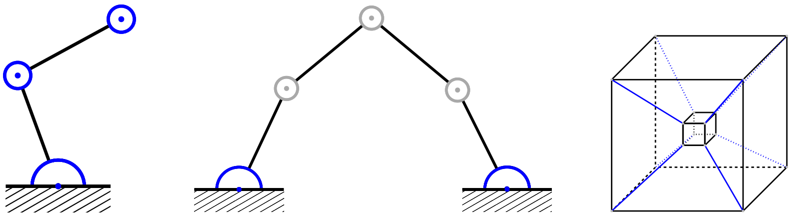


Figure 3.16: High level block diagram of the Mperl Controller. Inputs to the controller are the reference pose and the input description. Based on the input description, the controller generates the structure graph and the corresponding kinematic equations, which are solved for the reference position. Based on these calculations, an actuation command is send to each compont controller, which takes care of interacting with the electro mechanical parts.

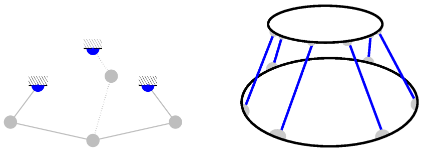
Depending on the reference position, either the forward kinematic equations are solved (if component parameters are provided), or the inverse kinematic equations (if an absolute position is given). Relating thereto, the workspace of the robotic system is mapped with regards to workspace boundaries and singularities. After solving these equations, the controller sends actuation commands to each component controller, provided that the target position is reachable and no singular region is crossed. These commands take the form of a high level command, e.g. a value in radians for revolute actuators, or a distance value for prismatic actuators. The component controller converts the actuation command into a signal, which is fed into the component driver and then into the component itself. After each component is actuated, the robotic system attains the pose resulting from the reference pose.

We test Mperl on the single and dual version of the SCARA system [85], a redundantly constraint Cable Robot [2] with 8 cables, a Delta robot [21], and a Gough-Stewart platform [130]. Figure 3.17 shows the overall structure of these examples.

Table 3.2 gives statistics about these examples. We report the size of the graph representation of the robotic systems and the number of parameters as a measure of the complexity of the examples. Table 3.2 also reports the setup time of the example in Mperl, which is the time for the system to initialize a new system: building the graph, traversing and checking for constraints, and generating the kinematic equations (but not solving them). The setup is needed once or when the structure of the robot changes. We run Mperl on a single core of an Intel i7-6920HQ (2.9GHz) with Debian Linux.



(a) Single SCARA system **(b)** Dual SCARA system **(c)** Cable robot



(d) Delta robot **(e)** Gough Stewart platform

Figure 3.17: Systems tested with Mperl

Robotic System	Type	Size of graph	$\lvert P_p \rvert$	$\lvert P_a \rvert$	Setup [ms]
Single SCARA Arm	Serial	7 Nodes	2	3	316
Dual SCARA Arm	Parallel	13 Nodes	8	2	641
Cable Robot	Parallel	40 Nodes	48	8	917
Delta Robot	Parallel	15 Nodes	4	3	612
Gough Stewart	Parallel	30 Nodes	36	6	1366

Table 3.2: Overview of examined robotic systems. $\lvert P_p \rvert$ and $\lvert P_a \rvert$ denote the number of passive and active parameters

Table 3.3 shows the time to calculate forward and inverse kinematics for different robotic systems. We take the average for 20 different arbitrary but valid configurations (forward kinematics) and for arbitrary but reachable points in the working space (inverse kinematics). For the inverse kinematics, we compare the three solvers but we use only dReal for the forward kinematics.

Table 3.4 summarizes the time needed to compute the singularities. Starting from the center, the work space is evenly divided in each direction. For the single and dual SCARA arm, this results in an initial 81 cells, for the Cable Robot, Delta Robot and the Gough Stewart Platform, we sample 541 cells. Figure 3.11 shows the initial grid for the dual scara system. For each point, we calculate the singular value decomposition of the robotic system (serial chains), which gives the closeness to singularities. If a target point is near a singularity, the

Robotic System	Inverse kinematics [ms]			Forward kinematics [ms]
	dReal	LS	CCD	
Single SCARA Arm	52	124	439	44
Dual SCARA Arm	156	448	918	138
Cable Robot	594	955	825	217
Delta Robot	367	822	1287	177
Gough Stewart	516	755	640	273

Table 3.3: Evaluation of the kinematics solvers

cell containing the singularity can be further divided and the map gets more refined. Refinement steps add to the computation time but increase the usable work space with a better approximation of the singular regions.

Robotic System	Initialization [s]	Refinement step [s]
Single SCARA Arm	67	64
Dual SCARA Arm	34	30
Cable Robot	621	615
Delta Robot	432	402
Gough Stewart	489	463

Table 3.4: Computation of the singularity map

3.8 CONCLUSION

In this chapter, we have introduced a controller synthesis approach for serial and parallel manipulators and evaluated it on several common manipulator architectures. As the generated controllers are quite simple, we introduce *sensor configuration synthesis* in the next chapter and enhance the model to consider dynamic effects in addition to the hitherto kinematic-only model. This allows for more sophisticated motion primitives, which find their use in chapter 5, which deals with the coordination of actions of multiple robots. In the future, we plan to address the aforementioned limitations and to extend Mperl by connecting it to a wider ecosystem of components, including legs and wheels taking inspiration from the work of Schulz et al. [123] and Ha et al. [52].

4 SENSOR CONFIGURATION SYNTHESIS

The previous chapter introduced Mperl, a way to build robotic systems and to synthesize controllers to provide some basic functionality. Based on the rigid body model, simple controllers could be synthesized, able to solve forward and inverse kinematics, which provides a first foundation for motion primitives. By 3D printing parts, not only is the manufacturing process sped up, but it is possible to integrate computation, actuation, and especially, sensing, directly into materials [13, 57, 60, 96]. This has a direct impact on the complexity of synthesized controllers, and in particular, on the amount of available information [77, 88, 101, 129, 140]. Instead of first building a robotic system, and then adding the sensing infrastructure in a second step, information provided by sensors are available at build time, and functional dependencies between relevant parts have to be found. These functional dependencies can be used to describe parts of the state, which are otherwise not measurable, and it extends the capabilities of the motion primitives. For example, consider the SCARA system (Fig. 1.3), which is tasked with lifting an object with hitherto unknown weight. Its upper beam is 3D-printed in Nylon, and it includes a deflection sensor, which measures the amount of flex of the beam it is embedded into; the lower beam is made from aluminum. The robotic system is capable of safely lifting objects weighing around $78\mathrm{g}$ within its entire workspace without encountering a motion error that could result in the payload being dropped or the system crashing. If the load is increased to $100\mathrm{g}$, the robotic system can still function, although its operation is contingent upon not fully extending the arm to its maximum range of $300\mathrm{mm}$. This can be achieved by restricting the opening angles of all joints in the system, which ensures that the system operates within safe and functional parameters while handling heavier payloads. The range has to be restricted to $200\mathrm{mm}$ to have similar safety margins as when lifting $78\mathrm{g}$ (Fig. 4.1).

Without sufficiently accurate information about the mass of the object, the robotic system may not be able to lift the object at all, may not traverse the entirety of its workspace, or it may even overshoot the desired target position, leading to an oscillating movement. This can lead to the system destroying itself, e.g., by exerting too much torque and burning out the motors, or crashing down. To this effect, the controller ideally should dynamically adjust the arm's working envelope depending on the load attached to the effector. Thus, the motion primitive MOVE would be extended by the ability to determine

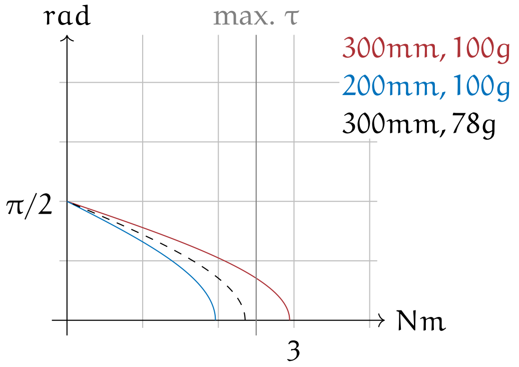


Figure 4.1: The arm system is capable of safely lifting 78g in its complete range without exceeding the maximum torque ratings (dashed line). In this configuration, the arm can be fully extended (300mm). If the weight is 100g, the arm can only operate safely in a limited range (max. extension 200mm, blue line) or is unsafe to operate (red line).

the attached weight and would restrict its movement based on the load.

Instead of treating the structure of the robot as black box, and trying to reconstruct the missing information, we make use of the detailed knowledge of its components and their properties. Fig. 4.2 shows the example SCARA system as used in this chapter, along with physical quantities associated with each component. We annotate properties of robotic systems with their physical dimensions, thus treating them as physical *quantities*. A property, in this context, is considered any information, e.g., a (constant) property like length or material, a functional dependency like stalling current, or the output of a sensor. This opens up the system to adhere to physical rules, in particular rules regarding the composition of dimension.

In this chapter, we introduce the *sensor configuration synthesis*, a method to explore possible instrumentations which allow the system to keep track of a specific property – in the example, the weight of the object. Figure 4.3 illustrates the connections between the components, inputs, and stages of the method. Using dimensional analysis [1], the method derives multiple candidate invariants from these properties and then filters these candidates in order to find a possible equation which can be used to derive the target quantity. These candidate invariant are then calibrated to the system they are used in.

For the example system, an obvious solution would be to add a sensor at the end effector, which directly measures the weight of the object. Expanding the search, we include the upper beam as the next-closest element, which explores the properties of the beam including its length, height, width, flexural rigidity and deflection. These components together yield the Euler-Bernoulli beam equation, which

---

[1] Automated by Newton [79]

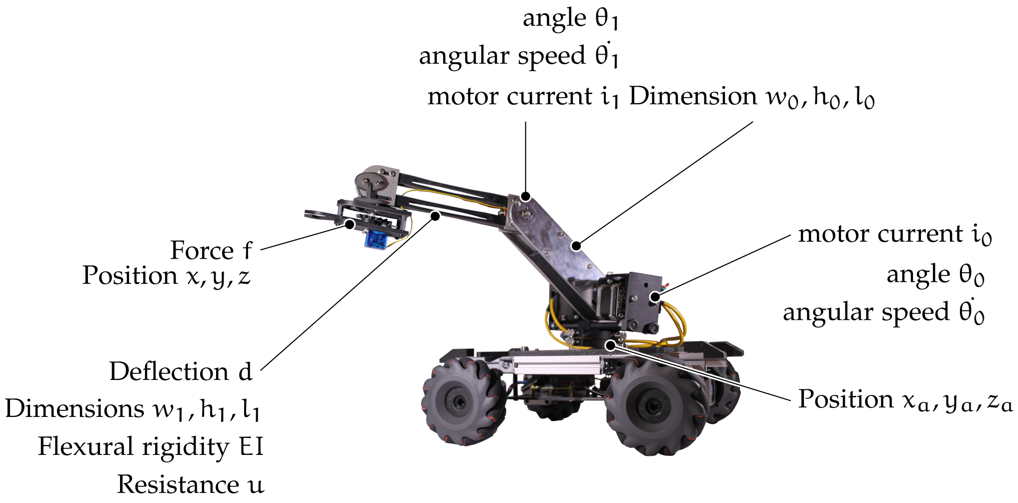
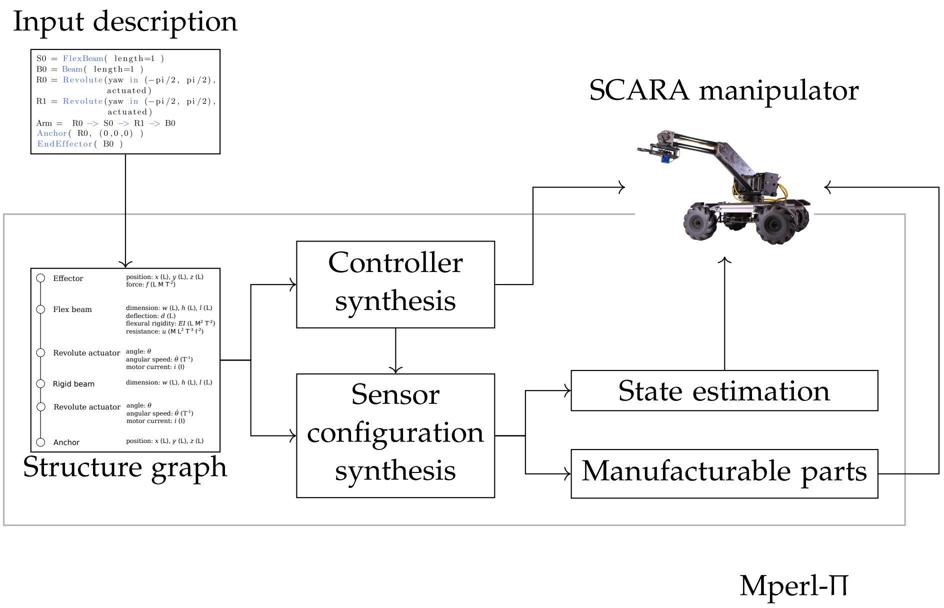


Figure 4.2: All properties of a robotic system are treated as physical quantities, i.e. a pair consisting of numerical value and physical dimension. This allows us to use dimensional analysis to (automatically) detail functional relationships between them

estimates the payload by measuring the deflection in the beam. Continuing the search will reach the revolute actuator, along with its current sensor; these properties, along with the length of the beam, can be used to compute the load by looking at the force exerted by the actuator. These different possibilities are presented to the user, who can select the most appropriate one. The sensor configuration synthesis can also suggest adding specific sensors. For example, assume the revolute actuator has no sensor attached. By considering properties of its actuators, within the motor specification is a table provided which specifies the torque produced by the motor in relation to the rotational speed and current consumption. To use the latter found equation, the sensor configuration synthesis considers quantities with dimension T^{-1} (rotational speed) and I (current). Because the actuator contains a rotary encoder, the rotational speed can be determined. The current consumption, on the other hand, is unknown; our method then suggests the addition of a current sensor to the motor. While usually some notion of cost dictates the type of sensor used, multiple, redundant ways of determining the same quantity can prove beneficial; see also section 4.3.2. In this chapter, we describe how we can use dimensional analysis to automatically search over the space of physically well-typed expressions in order to derive a combination of sensors to measure and estimate the system's state (*sensor configuration synthesis*). Our method leverages the knowledge of physical units in the model of a system to find ways of directly or indirectly measuring parts of the system's state which cannot be measured directly.

We present a physics-based search strategy to find configurations of sensors to directly or indirectly estimate the overall state of the system.



```
S0 = FlexBeam( length=1 )
B0 = Beam( length=1 )
R0 = Revolute(yaw in (-pi/2, pi/2),
              actuated)
R1 = Revolute(yaw in (-pi/2, pi/2),
              actuated)
Arm =  R0 -> S0 -> R1 -> B0
Anchor( R0, (0,0,0) )
EndEffector( B0 )
```

Figure 4.3: System overview

We have implemented a prototype tool into Mperl (Fig. 4.3), and we evaluate our approach with both simulated and real experiments.

In the following, we show how to use dimensional analysis to automatically search over the space of physically well-typed expressions. We can thereby derive a combination of sensors to measure and estimate the system's state. Our technique bridges these robotic design tools with more advanced controller synthesis techniques.

4.1 EXTENSIONS TO MPERL

Every value in Mperl is annotated with with is associated physical unit, from which the dimension is derived. For example, a beam whose length parameter $l \in \mathbb{R}$ is extended to $l \in \mathbb{R} \times \mathbb{D}$, where $\mathbb{D}$ represents the set of physical dimensions. For example, the length of a beam is no longer just a numerical value of 100, but now $100\mathrm{cm}$, with dimension L (length). The underlying control structures also reflect these changes, e.g. the state space ist no longer some $X \subset \mathbb{R}^n$, but $X \subset (\mathbb{R} \times \mathbb{D})^n$. We use $\mathrm{dim} : \mathbb{R} \times \mathbb{D} \mapsto \mathbb{D}$, to map quantities to their respective dimensions.

Similar to a type system, which classifies expressions of a programming language in terms of computed values, and governs rules on composition on these terms, adding physics, and in particular, dimensional analysis, results in additional rules for composition of quantities. Common conventions / rules encompass

- Comparing, adding and subtracting two quantities is only possible if both have the same dimension. This implies, that both quantities have the same unit, up to some scaling factor (e.g. km and cm).

- Multiplying two quantities v_1, v_2 changes their dimensions to $\mathrm{dim}(v_1)\mathrm{dim}(v_2)$.
- Differentiation and derivation of a physical equation changes the dimensions, but preserve the *dimensional homogeneity*.

We go into greater detail in section 4.2.1.

We extended Mperl to compute certain dynamic effects on its structure, in particular the forces at different points of the structure. For each component, additional information is provided; for example, torque ratings and stalling currents for revolute actuators, or flexural rigidity and deflection for beams. These properties originate from datasheets, from the abstract input description, or from sensor readings. While the conception and elaboration is generally applicable, we focus the technique to Mperl to gain a larger understanding of our end-to-end tool chain. The sensor synthesis approach presented here is a natural extension to Mperl and its high-level description of robotic manipulators. In the first step, our method considers the model of the dynamical system as a white box and allows the controller to read from the whole system's state. In a second step, the controller is then inspected to extract elements within the overall system's state that are used by the controller. Thirdly, the method runs a search procedure to find the required sensors to measure the signals which the controller uses. This search procedure tries to match each dimension of the state with a sensor, or otherwise expands the search by generating dimensionally-plausible candidate invariants that can be used for indirect measurement.

By first finding a local solution, and only then extending the search, we take into account the modular concept of Mperl. Robotic systems themselves can be composed from other robotic systems; in the exemplary single SCARA system, we can partition it into the individual systems gripper (end effector), arm, and cart platform. A general search boundary is given by the graph of an individual, and the user can decide if that boundary is extended to adjoining systems. By adding a weight sensor directly at the end effector, the resulting invariant can be reused even if the remaining systems are changed; on the other hand, if the weight at the endeffector is determined by placing load cells at the wheels derived via measuring the load distribution, this invariant depends on the structure of the whole system. In this case, a good middle course is to restrict the search to the end effector and the arm, thus ensuring that the cart and the arm system remain independent.

Base dimension	Symbol	Base unit	Symbol
Time	T	Second	s
Length	L	Meter	m
Mass	M	Kilogramm	kg
Electric current	I	Ampere	A
Absolute temperature	θ	Kelvin	K
Amount of substance	N	Mole	mol
Luminous intensity	J	Candela	cd

Table 4.1: Base dimensions and units of the SI

4.2 PRELIMINARIES

In the following sections, we provide some fundamentals concerning physical dimension and units as well as dimensional analysis, which leads to dimensional analysis and the formulation of the Buckingham Π−theorem, a fundamental theorem which forms the basis of the sensor configuration synthesis presented in this chapter. The section is rounded of by providing a brief definition of reach-avoid specifications, which are used to guide the synthesis procedure.

4.2.1 Quantities, Units and Dimensions

A quantity $(v, d) \in (\mathbb{R} \times \mathbb{D})$ consists of a numerical value $v \in \mathbb{R}$ and a physical dimension $d \in \mathbb{D}$. To obtain the dimension of a quantity, we use $\mathrm{dim} : (\mathbb{R} \times \mathbb{D}) \mapsto \mathbb{D}$.

Let $\mathbb{D}$ be the set of physical dimensions, and let $\mathbb{D}_B$ be the set of base dimensions. Base dimensions are indepedendent and mutally exclusive; each base dimension cannot be expressed in terms of other dimensions in $\mathbb{D}_B$. $\mathbb{D}$ is formed by building the monomial consisting of base dimensions $d_i \in \mathbb{D}_B$ and dimensional exponents r_i

$$\mathbb{D} = \left\{ \prod_{i=1}^{n} d_i^{r_i}, r_i \in \mathbb{Z} \right\} \tag{4.1}$$

For a dimensionally independent set of dimensions,

$$\mathrm{dim}(\prod_{i=1}^{n} d_i^{r_i}) = 1 \tag{4.2}$$

holds only for $r_i = 0, i = 1, ..., n$ (linear independence of $\mathbb{D}$). The *International System of Units* (SI) defines seven base dimensions, along with a commonly associated standard unit of measure (Table 4.1). These seven base units form a dimensionally independent set.

All other dimensions are derived from these seven base dimensions, and we can thus formulate $\mathbb{D}_B$ as

$$\mathbb{D}_B = \{T, L, M, I, \theta, N, J\} \tag{4.3}$$

Dimensions and units are different, in that dimensions describe the nature, or the qualitative properties, of the physical quantity, and units describe a measurement, i.e. a quantitative property. Quantities can only be compared if their dimensions are the same, and multiplying dimensions follows the rules of multiplying monomials

$$d_1 d_2 = \prod_{i=1}^{n} d_i^{r_i} \prod_{i=1}^{n} d_i^{s_i} \tag{4.4}$$

$$= \prod_{i=1}^{n} d_i^{r_i + s_i}, d_1, d_2 \in \mathbb{D} \tag{4.5}$$

and forms new physical dimensions. For example, by dividing the physical dimensions of length L by time T, we get the dimension of velocity $T^{-1}L$. The dimension of a quantity can be 1 if each dimensional exponent is 0; the quantity is then called dimensionless. This can happen if the dimensional exponents cancel each other out; for example, the unit *radian* is defined as the coefficient of arc length and radius, which results in a dimension of $\frac{L}{L} = 1$.

A physical law $f : (\mathbb{R} \times \mathbb{D}) \times ... \times (\mathbb{R} \times \mathbb{D}) \mapsto (\mathbb{R} \times \mathbb{D})$ establishs the relationship between quantities. For this law to be *physically meaningful*, it must fulfill the property of dimensional homogeneity, which means, that both sides of the law must have the same dimension. By rewriting $f(v_1, v_2, ..., v_{n-1}) = v_n$ implicitly as $f'(v_1, v_2, ..., v_n) = c$, $c \in \mathbb{R}$ being a dimensionless constant, it must hold that $\dim(f') = 1$. We call $\dim(f')$ a *dimensionless product*, and all variables in f are captured in a set $S = \{v_1, v_2, ..., v_n\}$.

4.2.2 Buckingham Π Theorem

The idea of the Buckingham Π theorem states that any physical law establishing a relationship between physical quantities, can be equivalently expressed by as a relation between dimensionless quantities.

The theorem bounds the number of variables that have to be grouped in order to find dimensionless groups. Denote by n the number of quantities in a physical law, i.e. $n = |S|$, and $k = |\mathbb{D}^*|$ the number of basic dimensions forming a dimensionally independent basis with which the quantities in S can be expressed. From the parameters in S, $n - k$ dimensionless products π_i can be formed

$$\pi_i = q_1^{a_1} q_2^{a_2} ....q_n^{a_n} \tag{4.6}$$

These $n - k$ dimensionless products form the root of some function ϕ with

$$\phi(\pi_1, \pi_2, ..., \pi_{n-k}) = 0 \tag{4.7}$$

ϕ can be rearranged into ϕ' such that

$$\phi'(\pi_1, \pi_2, ..., \pi_{i-1}, \pi_{i+1}, ..., \pi_{n-k}) = \pi_i \tag{4.8}$$

The Buckingham Π Theorem is more commonly used to form an unknown physical law in terms of dimensionless variables, and to reparameterize physical laws in terms of smaller number of parameters. Recently, Wang et al. [136] showed that this theorem can be used to efficiently generate candidate invariants of physical systems.

For example, we want to establish a physical law connecting quantities force F, mass m and acceleration a, with $\dim(F) = ML/T^{-2}$, $\dim(m) = M$, and $\dim(a) = L/T^{-2}$.

We have $n = 3$ variables and $\mathbb{D}^* = \{M, LT^{-2}\}$ consists of two independent dimensions. This means that we have one resulting dimensionless $\Pi-$group.

We thus form the dimensionless physical law

$$f(F, m, a) = 0 \tag{4.9}$$

The $\Pi-$group given by

$$\dim(F)^\alpha \dim(m)^\beta \dim(a)^\gamma = 0 \tag{4.10}$$

must be dimensionless for a choice of dimensional exponents are α, β, γ. Solving these constraints yields $\alpha = -1, \beta = \gamma = 1$, which results in

$$C = F^{-1}ma \tag{4.11}$$

where C is a dimensionless constant; in this case, $C = 1$. Generally, the value of C is not known a priori and must be found experimentally. While invariants of physical systems are dimensionally correct, the converse is not true and further analysis is required to identify actual invariants within the space of dimensionless groups.

4.2.3 Reach Avoid Specification

We use reach avoid specifications to guide the controller governing the robotic system to goal states, while avoiding fail states. A reach avoid specification consists of a target state set X_t, and any trajectory $\xi(t)$ starting from an initial state $\xi(0) = x_0$ should end in X_t. Equation 2.56 asserts that all states on a trajectory are connected by a path. Therefore, if a trajectory starts at x_0 and ends at some state $x \in X_t$, it is possible to find a control input that allows the system to move from x_0 to x.

Let X_p denote the set of permissible states the system is allowed to assume during operation, and let X_f denote the set of states the system is not allowed to assume. X_f can be seen as the complement of X_p.

We can thus formalize a safe set A as

$$A(X_f) = \{x \in X : \exists t \geqslant 0 : \xi(t) \notin X_t \land \xi(0) = x\} \tag{4.12}$$

which denotes the set of states from which trajectories exist whose states are never fail states.

Likewise, we can define the set R of all states from which trajectories exist that reach states in X_t

$$R(X_t) = \{x \in X : \exists t \geqslant 0 : \xi(t) \in X_t \land \xi(0) = x\} \tag{4.13}$$

We can then instruct the controller to reach a goal state while avoiding any fail state as the reach avoid specification RA:

$$\begin{aligned} RA(X_t, X_f) = \{x \in X : t \geqslant 0 : \\ \xi(t) \in X_t \land \forall t_p \in [0, t], \xi(t_p) \notin X_f \land \xi(t), \xi(t_p) = x\} \end{aligned} \tag{4.14}$$

4.2.4 Relevance to State Estimation

In section 2.6, we introduced dynamical systems, which model the interaction between the environment and a controller that generates control inputs u depending on the perceived state y. This is a generalization; in reality, typically the controller tries to estimate a state $\hat{x}$ from y to get a more accurate representation of the current state, and the control input u is then bases on $\hat{x}$ instead of y. Such a closed loop controller requires sensors to provide feedback on the system's state. For each sensor, a functional relationship is needed to map its output to the desired target quantity.

One of the most common approaches is to use (extended) Kalman filters [62, 87, 128] or one of its varieties. Kalman filters use a state transition function that describes the behavior of the system, and an observation function h, which creates a functional dependency between measurements taken in the system and the output of the state transition function. We are interested in finding these functional dependencies.

The observation function h is a mathematical function that maps sensor readings to values in Y. h could be a linear or non-linear function, depending on the relationship between the sensor readings and the state variables. In order to enable the estimation of the state of a system, it is necessary to identify a suitable combination of Y and the observation function h, which not only has a feasible physical implementation but also provides sufficient information.

We refer to the problem of identifying the optimal combination of sensors that can provide accurate and relevant information about the state of a system as the *sensor configuration synthesis* problem. This problem is related to the observation function in state estimation, as the choice of sensor configuration affects the accuracy and structure of the observation function. An appropriate sensor configuration should be capable of measuring the relevant quantities of interest while minimizing certain pre-defined factors like redundancy, cost, and complexity.

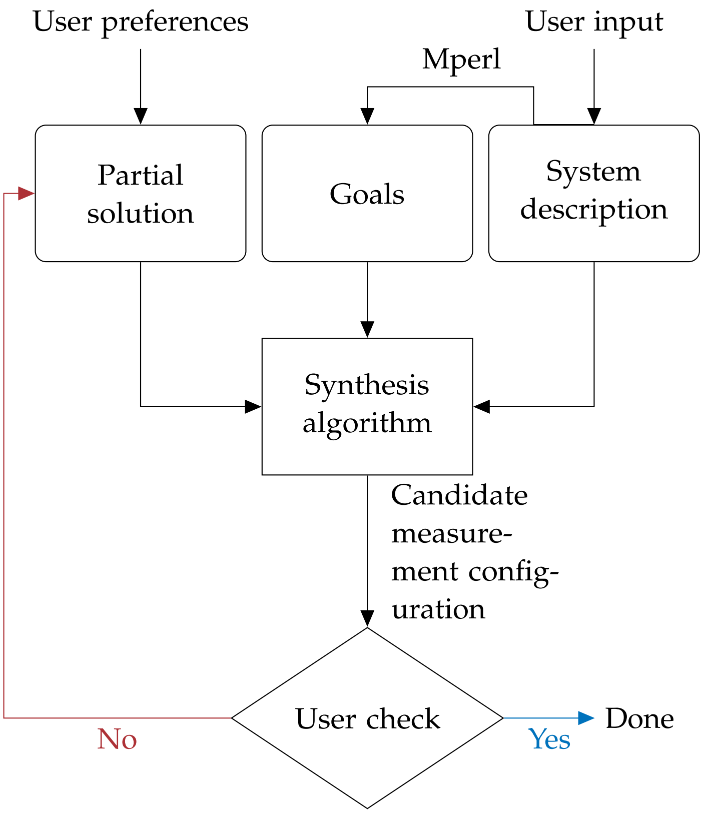


Figure 4.4: Sensor synthesis workflow

4.3 SYNTHESIS OF SENSOR CONFIGURATIONS

Figure 4.4 shows the different parts of the sensor synthesis workflow and their interaction. Inputs to the sensor synthesis algorithm are the input description of the system, from which Mperl generates the control code for the system (Chapter 3), and unobservable parts from the controller, along with any user specified preference. The synthesis algorithm leverages dimensional analysis to find candidate invariants, that establish a functional dependency between the unobservable target quantity and sensors or properties present in the system. These candidate equations are presented to the user, who is responsible for checking the sensor configuration proposed by the synthesis algorithm. The algorithm is also able to exclude specific sensors. This proves useful in finding alternative ways of determining a quantity, for example, if a sensor malfunctions.

GOAL STATE-SPACE The goal state space consists of elements in Y which are not observable. We assume the controller in the dynamical system to be a state-feedback controller, i.e., all state variables are available for observation. In particular, we assume that the controller keeps a history of the mapping between the system trajectory and the corresponding control input. In this step, which depends on the structure of C, we find out which part of the state the controller actually needs. For instance, if C is a linear function of the form $u(t) = -K(\tau(t))$, we select all columns of K with a non-zero entry. If the internal structure of the controller is not available, we can over-approximate necessary parts of the state by taking the full state. In the rest of this section, we denote by $Y^* \subseteq Y$ the part of the state we need to reconstruct.

SENSORS While multiple sensors with different ranges, precision, and accuracy, can be present in the robotic system, for the sake of simplicity, we assume at most one sensor per dimension. The available sensors are represented by the function *sensor* which maps elements of $\mathbb{D}$ to sensors or $\perp$ when no sensor exists

$$sensor : \mathbb{D} \mapsto \begin{cases} \mathbb{D}, |\mathbb{D}| > 0 \\ \perp, otherwise \end{cases} \tag{4.15}$$

Furthermore, the *cost* function maps the sensor to an additive cost. Cost in this context is to be understood as general cost; for example, it can be understood in terms of monetary cost, e.g. when building a preferably cost-efficient system, or in terms of power consumption, for when the goal is to maximize the autonomy of a battery-powered system. To simplify the presentation of the algorithm, we assume that *cost*$(\perp) = \infty$. Our synthesis procedure minimizes the overall cost of the sensors.

EXTENDED SYSTEM DESCRIPTION The sensor configuration synthesis greatly benefits from an abstract model with detailed physical descriptions of its components. A highly detailed model allows for finding a wide range of sensing possibilities. This includes not only finding a way of measuring or deducing a hitherto unknown quantity in the system, but also finding alternative ways of measuring the same quantity. Consider the controller of the SCARA arm from Figure 4.2 needs to limit the range of the arm depending on the payload weight; otherwise, the maximal torque of the motors could be exceeded. For instance, in a minimal model consisting of speed and acceleration of the payload and the torque exerted by the motor. our algorithm would suggest the use of a torque sensor. With a more general model of the system, our algorithm can explore some details in more detail. For instance, the motor, being an electric motor, turns electric energy into torque, and the datasheet of an electric motor provides information about the relationship torque - speed - current consumption, which is (ideally) reflected in the description of that component. Using these elements, the system would suggest measuring the current consumption going into the motor, along with its speed, to derive the applied torque.

DISTANCE BETWEEN COMPONENTS The downside of having access to all this information is the great increase in size of the search space and the number of Π groups. Owing to the modular nature of our systems, quantities and their functional relationships are often in closer proximity, which allows us to prioritize elements which are close together, and then successively widening the search space. Therefore, we assume that we have a function $dist_S$ which returns the distance, for some notion of distance in S, between two quantities in the system. When generating the Π groups to measure a quantity q, we restrict the search by considering only elements below some distance from q. Mperl represents the structure of the robotic system by means of a graph, and the distance measurement follows naturally by choosing the shortest path between two elements in the graph. Because the graph must be connected, i.e., it is not allowed to have components in a system which are disjoined, for any two nodes in the graph, such a path exists. If a physically meaningful relationship between two nodes in the graph exists, the search algorithm will eventually discover a Π−group between them, irrespective of a potential cost function. This follows from the fact, that the sensor configuration synthesis successively widens the search, eventually covering the whole graph, and from the Buckingham Π−theorem.

4.3.1 Search algorithm.

Our search procedure, shown in Algorithm 4.1, is a backtracking search which progressively adds more sensors to the system while keeping the current best found solution to prune the search.

We start the search with the available sensors, their cost, the known constant quantities, and the variables $x_0 \dots x_n$ we need to measure. In our running example, known constant quantities are elements like the length of the beam, or motor characteristics like current consumption, which can be used without incurring any cost. For each variable, we try to find out if a sensor exists or expand the search by generating a candidate invariant involving that variable. The cost function drives the search into generating candidate invariants even if we have a sensor or cut the search if a branch becomes too expensive.

The expansion uses dimensional analysis to find candidate invariants involving the variable to measure. When we find a candidate invariant, we remove the variable to measure from the goal and add all other expressions occurring in the invariant to the goals. In the algorithm, this search is done by calling FindPiGroups(q, Q). Q is the set of all the quantities that can be used to find the Π groups and q is a quantity which must be part of the Π groups, i.e., the exponent for q must be nonzero. FindPiGroups is implemented using the algorithm by Wang et al. [136] and filtering the groups where q is not present. As the number of candidate invariants grows exponentially with the

```
Require: $y_0 \dots y_n \in Y^*$, the quantities to measure
Require: initStatus, all the known constant quantities of $S$
Require: $\Delta$ ($\geqslant 0$), the scope of the search
 1: function SEARCH($q, NA, status, c_{curr}, c_{max}$)
 2:     if $(q, \_) \in status$ then
 3:         return $(status, c_{curr})$
 4:     $(status_{best}, c_{best}) \leftarrow (\bot, c_{max})$
 5:     if $cost(sensor(dim(q))) + c_{curr} \leqslant c_{max}$ then
 6:         $status_{best} \leftarrow status \cup \{(q, sensor(dim(q)))\}$
 7:         $c_{best} \leftarrow c_{curr} + cost(sensor(dim(q)))$
 8:     $Q \leftarrow \{q' \mid dist_S(q, q') \leqslant \Delta\} \setminus NA$
 9:     $\mathcal{G} \leftarrow$ FINDPIGROUPS($q, Q$)
10:     for $G \in \mathcal{G}$ do
11:         $(s, c) \leftarrow (status, c_{curr})$
12:         for $q' \in G \wedge c \neq \infty$ do
13:             $(s, c) \leftarrow$ SEARCH($q', NA \cup \{q\}, s, c, c_{best}$)
14:         if $c \leqslant c_{best}$ then
15:             $(status_{best}, c_{best}) \leftarrow (s \cup \{(q, G)\}, c)$
16:     if $status_{best} \neq \bot$ then
17:         return $(status_{best}, c_{best})$
18:     else
19:         return $(Failure, \infty)$
20:                                         ▷ Apply the search to all the $y_i$
21: $s \leftarrow initStatus$
22: $c \leftarrow 0$
23: for $y \in y_0 \dots y_n \wedge s \neq Failure$ do
24:     $(s, c) \leftarrow$ SEARCH($y_i, \emptyset, s, c, \infty$)
25: return $s$
```

Listing 4.1: Sensor synthesis algorithm

available number of variables, and many can be spurious, we use the distance function to search only for candidate invariants involving quantities close to each other.

The algorithm search procedure (line 1–19) keeps track of (1) q the quantity that needs to be measured, (2) *NA* the quantities not available, (3) *status* a partial solution, (4) c_{curr} the current cost, and (5) c_{max} the maximal cost. In the set *NA*, we keep track of the elements below the current branch of the search tree. These elements cannot be used as it would introduce circular dependencies in the result. *status* is a set containing pairs of quantities and how they are measured: either directly with a sensor or derived indirectly through an invariant. The maximal cost c_{max} cuts the search when a better solution is already known.

In the search, we first try to check if a quantity is already known (line 2), in which case there is nothing to do. Then, we try to find a sensor which matches the quantity's dimension (line 5); if that fails, we consider indirect measurements. We gather the elements in the system in the neighborhood of q (line 8), find the dimensionless groups over these elements, and search recursively on them (line 9–15). The search is applied to all the elements we need to measure (line 21–25).

Restricting the search according to a distance has two goals: 1. improving the accuracy of the solution and 2. limiting the complexity of the search. Currently, we rely on the state estimation to deal with measurement errors. Measuring a quantity over a large distance means multiplying multiple terms between the measurements and the goals. This compound errors and further measurements are less likely to yield good data. In Section 4.3.3, we discuss methods to improve error handling. The second aspect is the scalability of the method. The number of Π groups is exponential in the size of the overall system, and a naive application of the search only works on small systems. On the other hand, the size of neighborhoods within a system should contain roughly the same number of elements independently of the overall system, which makes it possible to apply the search to larger systems.

USER CONSTRAINTS ON THE SEARCH Algorithm 4.1's presentation is minimal. Minor modifications of the starting state allows the algorithm to not only return one solution, but to explore other solutions. For instance, by adding a specific sensor to *initStatus*, it is possible to enforce the usage of that sensor; conversely, the use of a specific sensor can be prevented by adding the corresponding quantity to *NA* at the beginning of the search. If the algorithm fails to find a solution, the scop δ can be increased successively. Searching for different solutions can also be used to generate multiple ways of measuring the same quantity. Using multiple sensors can improve the quality of the state estimation if the elements used are independent across measurements. This mechanism can also be utilized to search for alternative measurements with reduced cost, or in the case of a sensor malfunctioning. In the case of determining the weight of the load at the end effector, if the torque sensor is not working, or its measurements are only intermittend available, the derivation via bending sensor or current consumption can be substituted in. In the context of filtering, combining multiple alternative measurements can significantly reduce uncertainty and improve the accuracy of the information obtained. In particular alternative measurements from (physically) disparate sources lead to smoother signals [141].

CHECKING CANDIDATE SOLUTIONS The sensing configurations stem from dimensionally-correct equations, but these equations may not correspond to actual physical processes in the system. We can distinguish between two sources of spurious solutions: (1) the Π group does not correspond to any physical law and (2) the Π group corresponds to a physical law, but associates the (right) dimension to the wrong element. The first case requires checking if there exists any law of physics corresponding to what the algorithm suggested. The sensing configuration can also be tested against a small number of known configurations to check if they hold. The second case happens because the search does not differentiate quantities with the same dimension. Such errors are harmless when they stem from constant terms, e.g. using the width instead of the length of a beam, as the Π groups are an equality up to a constant. When the error is not about a constant term, then the process to discard such solutions is similar to the first case.

Checking candidate solutions still requires some knowledge from the user, but as checking a given solution is easier than finding a solution, our method still lowers the expertise requirement to build CPS. A non-expert may be overwhelmed and might not know how to solve the problem; the algorithm gives them a "place to start." For an expert user, the system can help them to accomplish the task faster.

4.3.2 Calibration and Run-time

After the generation and selection of meaningful candidate sensor configurations, the output space Y and the observation functions H are generated. The observation function corresponds to extracting the equations stored in *status* and reordering them such that their value is a function of the state. The proportionality constants that stem from the Π−groups are still contained in the observation functions and a specific numerical value has to be found. This can only be done experimentally and is done in a system-specific calibration process. In Section 4.4.1 we explain how we did this in our evaluation. If no other invariant is present, this requires user involvement. Currently, we only use the Π groups vetted by the user but we can also take advantage of this phase to reduce the burden on the user. As a mitigation technique, once the user has selected a sensor configuration, we gather all Π groups for that configuration and later, during the calibration phase, use regression to find the relevant ones automatically. Furthermore, if the sensors and their placement get cheap enough [77, 88, 101, 129, 140], we can embed more sensors for Π groups which have not been checked by the user. The additional data is then used during calibration to learn which Π groups correspond to actual invariants of the system. If multiple (tentative) invariants with conjunct parameter sets are available, or the system is sufficiently equipped with sensors, the calibration can be done automatically.

In fact, in some cases, it may be more efficient to use previously calibrated invariants as part of the sensor configuration synthesis process to calibrate a newly found invariant. By doing so, the number of direct measurements can be reduced, and the accuracy of the new invariant may be improved by leveraging existing knowledge.

For example, we have a calibrated invariant which determines the load on the end effector by means of beam deflection and beam properties. We use the sensor configuration synthesis to find an other invariant based on the current consumption of the revolute actuator, the angular speed of the motor and the beam length. Each quantity in this new found invariant can be traced back to a direct measurement (current sensor at the revolute actuator), or an indirect measuremt (angular speed derived from previous states, beam length a known constant); the target quantity determining the load is given by the already calibrated invariant. Thus, the newly found invariant can be calibrated automatically.

Denote by i_c an already calibrated invariant, and by $param(i_c)$ the set of quantities in i_c; analogoulsy, denote by i_{nc} the invariant which is not calibrated, and its parameter set $param(i_{nc})$. The target quantity must then be in the intersection of both parameter sets. Then we require that each parameter can be traced back to either a direct measurement, or an invariant:

$$\forall p \in param(i_c), p \in sensor \vee \exists i, q : i(q) = p \tag{4.16}$$

where i is a calibrated invariant which determines p from its input quantities q, and $p \in sensor$ means that a sensor exists which measures p. Assuming there is only one sensor, this means that that sensor measures $dim(p)$.This ensures that i_c determines q_t, which is then used to determine the dimensionless constant C.

In many cases, the calculation of a specific invariant may depend on the values of other input variables, which themselves are calculated by a sequence of other invariants. In this case, we can use a directed acyclic graph, whose nodes nodes represent invariants and directed edges represent dependencies between them. To evaluate the invariants in this grpah, a topological sort can be used to find a sequence of nodes such that all dependencies are satisfied, resulting in an order in which the invariants should be evaluated.

The strength of the automatic calibration process lies in providing multiple ways of determining the same quantities. The controller can either switch between them, depending on their cost, or, in case of errors or filter divergence, multiple measurements can be aggregated, e.g. by consensus. Due to the abstraction provided by the dimensional analysis, invariants can be found which measure the same target quantity in physically different ways (depending on the equipped sensors), which allows significantly better results in sensor fusion [53, 141], consenus [106, 138] or general state estimation [27, 38].

In chapter a.1, we show how Mperl and the sensor configuration synthesis is integrated in the state estimation and how the state estimator handles measurement errors.

4.3.3 Limitations and Extensions of the Method.

The sensor configuration synthesis as outlined in this chapter has two major drawbacks. The first stems directly from the Buckingham Π-theorem, in that dimensionless groups – while phyically well typed – not necessarily correspond to physically meaningful equations. Especially in new systems without existing invariants, we rely on the user's judgment for checking the meaningfulness of a Π group. In the context of providing (non-expert) users a starting point from which to use and extend a system, this is a reasonable limitation; however, in light of a more robust system, which uses these invariants for sensor fusion or as alternative ways of determining the state of a component in case of a failure, a more automated method is needed. During the calibration, we need to get accurate measurements, which implies the possibility of measuring the entirety of the system's state. For some systems, this might not be possible.

The second drawback is that while running the system, we need to take measurement errors and the model adequacy into account. We use an extended Kalman filter for the potentially non-linear dynamics of the system. Unlike the (linear) Kalman filter, the extended Kalman filter is not optimal, and thus cannot yield any guarantees on the quality of the estimation. The filter also assumes Gaussian noise on the measurements, and therefore, we need to ensure that the sensors have this noise profile. Regarding sensor selection and sensor placement, our model can be extended to provide better results. Sensors can sense over different ranges, accuracies, and sampling frequencies. The range requirements for sensors could be established using interval analysis [3]. We currently use a sample-and-hold controller, and therefore, the controller-loop frequency determines the required sampling speed of the sensors. This method could be used with event-triggered control [26] for sensors which support programmable interrupts. Sensor placement can also affect the quality of their output. Some sensors are simple to place, such as a current sensor which can be placed anywhere on a wire, but sensors related to material properties are much more sensitive to location. For instance, optimizing the placement of a deflection sensor requires finite element analysis [22]. The model adequacy is obviously outside of our control.

4.4 EVALUATION

In the following, we show by means of two examples, a quantitative one, and a qualitative one, the validity of our method. In the quantitative evaluation, we search for common quantities in three different robotic systems (single SCARA, dual SCARA, CoreXY) and report the performance of the sensor configuration synthesis. In the qualitative evaluation, we elaborate on our running example by providing concrete, experimentally verified data and we show how the calibration process is executed. For this, we run Mperl, combined with the sensor configuration synthesis, on real hardware and show, that accurate measurements are provided to the controller.

4.4.1 Quantitative Evaluation

We evaluate Algorithm 4.1 on three different robotic manipulator architectures. These experiments only evaluate the search, without the physical implementation of the robots. We show that sensor configurations can be efficiently generated and that our search heuristic effectively reduces the number of Π-groups to a manageable number. In this section, we use the term quantities and parameters interchangeably as the quantities are parameters of the design space for Mperl. Constants like physical dimensions are only fixed at manufacturing time and they can "vary" during the exploration of the design space.

SEARCH FOR MEASUREMENTS. We show that our synthesis algorithm can effectively generate possible ways of measuring a goal quantity that is given by its dimension and and the location where we want to measure it. We evaluate the search on three manipulator architectures: 1. The single SCARA manipulator as shown in Figure 4.2, 2. a dual SCARA manipulator [85], and 3. a CoreXY platform [98]. We report the number of quantities in the search space in Table 4.2 and 4.3. Quantities include constant properties, e.g. the size of structural elements like beams, or variables like current consumption of an electric motor. Anchors are mounting points, used to attach the manipulator to the ground, the end effector is the element which effects the world; in this case, the part which carries an object. Beams are divided into rigid aluminum beams which do not deform under load, and 3D printed beams made of plastic which slightly deform under load. The flexible beams have a cavity where a flex sensor is inserted during printing; the sensor is fully enclosed and acts as a variable resistor, along with a Wheatstone bridge, which makes resistance and voltage drop parameters of the flexible beam.

For each manipulator, we show that our method can synthesize physically meaningful ways of measuring quantities of the system's state. We gather a set of 9 dimensions as goals to measure, including acceleration, velocity, momentum, their equivalents for rotation, torque, flexural modulus, and second area moment. For each goal, we try to take measurements at various points in the system, and for each manipulator and each goal, we search for physically valid Π groups which contain the goal, i.e., Π groups which can be used to derive the goal.

Element (# Parameters)	Parameters
Anchor (3)	position (x, y, z)
End effector (4)	position (x, y, z), force f
Revolute joint (1)	angle θ
Revolute actuator (2)	angle θ, current i
Revolute actuator with pulley (3)	angle θ, current i, radius r
Rigid beam (3)	width w, height h, length l
Flex beam (8)	width w, height h, length l, flexural rigidity EI, density ρ, deflection d, resistance u, voltage drop v
Pulley (4)	position (x, y, z), radius r

Table 4.2: Parameters for the robot's components. Each parameter has the standard dimension associated with the parameter's name.

	Anchor	End effector	Revolute joint	Revolute actuator	Revolute actuator with pulley	Rigid beam	Flex beam	Pulley	$\sum$ Parameters
Single SCARA	1	1		2		1	1		22
Dual SCARA	2	1	3	2		2	2		41
CoreXY	2	1			2			8	48

Table 4.3: Components and the total number of quantities in the manipulators.

In Tables 4.4, 4.5, and 4.6, we report statistics about the search. We provide the starting node, from which the search originates, along with the range needed to find suitable Π–groups. We direct the search in such a way that no direct measurement, i.e., via sensor, is allowed; thus, a range of 0 is not possible. As starting node for the single SCARA system, we use the following numbering: (0) end effector, (1) wrist, (2) lower arm, (3) elbow actuator, (4) upper arm, (5) shoulder actuator. For the dual SCARA system, we use the following numbering of the starting node: (0) end effector, (1) revolute joint middle, (2) beams (distal), (3) revolute joint, (4) beams (proximal), (5) revolute actuators to anchors. Due to the system's symmetry, the starting nodes are only in one half of the system.

As starting node for the CoreXY system, we use the following numbering: (0) revolute actuators, (1-4) pulleys. Similar to the dual SCARA system, the starting nodes are only in one half of the system due to the symmetry of the system.

Goal	Starting node	Range needed	Π groups searched	Time [s]
Velocity	0	1	93	0.049
	2	1	63	0.023
	4	1	54	0.024
Momentum	1	1	19	0.045
	3	3	278	0.398
	5	5	575	1.020
Acceleration	0	1	93	0.032
	2	1	63	0.063
	4	1	54	0.032
Angular momentum	1	1	75	0.482
	3	3	152	0.403
	5	5	385	1.202
Flexural modulus	2	2	148	0.243
Torque	3	2	261	0.560
	5	5	586	1.403

Table 4.4: Single SCARA arm, dimensional analysis summary.

The search uses iterative deepening and we report the value of the distance Δ in Algorithm 4.1, at which we find a valid Π group. Goals which can be found directly on the node where the search starts are omitted from the tables. As number of Π group searched we report the number of Π groups which are generated until a valid one is found. We only count unique Π groups by filtering out equivalent Π groups, e.g. groups where all the exponents are multiplied by -1. The times reported in the tables correspond to Mperl-Π running on a single core of an Intel i7-6920HQ (2.9Ghz) with Debian Linux.

We observe that the synthesis algorithm efficiently explores the search space of our manipulator designs even though the designs contain at least 20 and up to 48 parameters. The running time is low even when the search range spans large portions of the systems. For larger distances, the number of Π groups considered can grow above 1000 but the local search usually keeps the number low.

4.4.2 Qualitative Evaluation

We now present a more detailed end-to-end case study, including a physical prototype of the robot arm, as seen in the cart-and-arm assembly. The arm's role is to perform pick-and-place tasks. While

Goal	Starting node	Range needed	Π groups searched	Time [s]
Velocity	0	1	93	0.083
	2	1	63	0.054
	4	1	54	0.053
Momentum	1	1	102	0.032
	3	3	531	0.036
	5	5	1667	1.356
Acceleration	0	1	93	0.045
	2	1	63	0.036
	4	1	54	0.060
Angular momentum	1	1	74	0.041
	3	3	361	0.683
	5	5	1054	1.210
Flexural modulus	2	2	242	0.398
Torque	3	2	504	0.705
	5	5	1382	1.225

Table 4.5: Dual SCARA arm, dimensional analysis summary.

Goal	Starting node	Range needed	Π groups searched	Time [s]
Velocity	0	4	1464	1.302
	1	3	662	0.892
	2	2	452	0.714
	3	1	272	0.352
Acceleration	0	4	1464	1.321
	1	3	662	0.912
	2	2	452	0.716
	3	1	272	0.420
Torque	0	4	1328	1.453
	1	3	568	1.032
	2	2	398	0.786
	3	1	244	0.596

Table 4.6: CoreXY platform, dimensional analysis summary.

performing these tasks, it has to move different payloads, each with their own weight, while the controller has to ensure that the forces on the actuators stay within the operating ranges specified by the motors' respective data sheets. Exceeding these values would result in the arm collapsing in an uncontrolled way; therefore, the controller needs a way of estimating the payload weight before deciding if it can reach the target location. In Figure 4.5, we give a visual representation of the arm's workspace and how it is affected by the load.

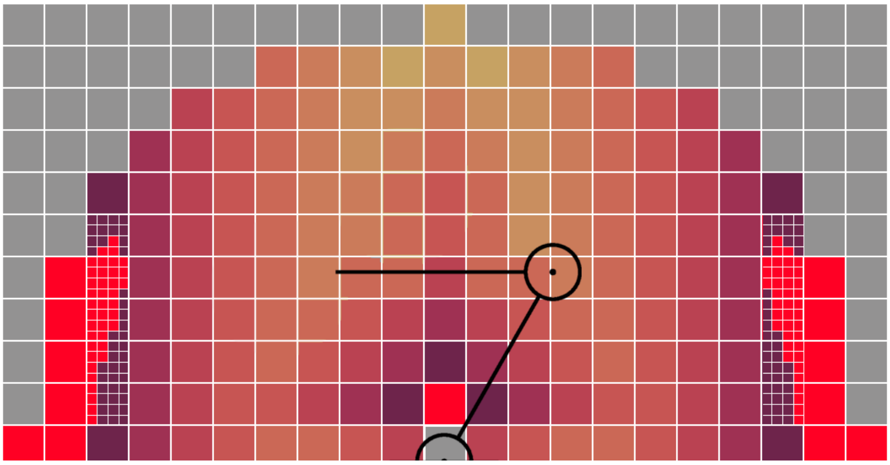


Figure 4.5: Representation of the SCARA arm workspace while torque is applied to the actuators. Grey regions are outside of the reachable workspace. Regions with low torque are represented by lighter colors, darker regions represent a higher load. Red regions are unsafe regions of the workspace, where the motors' maximal torque is exceeded.

We describe how to use Mperl-Π from a user's perspective and we test two different sensor configurations found by our algorithm. First, we explain how the sensor configurations are derived and how the system is calibrated. Ensuing, we evaluate the measurement errors for the two configurations and show, that we can estimate the payload's mass with a reasonable accuracy. The root-mean-square error is around 5g for payloads ranging from 40g to 200g.

The arm consists of two beams (shoulder and elbow), and two revolute actuators. The shoulder beam is solid, the elbow beam is allowed to flex and has a load cell embedded. The load cell is a resistor that changes with its deflection and is connected to a Wheatstone bridge for measuring the change in resistance (and therefore, the deflection). Both revolute actuators are brushed dc motors, equipped with 48 CPR quadrature encoders and gearboxes; in addition, their electric current and power are monitored by two INA219B sensors. Each actuator has a PID controller that follows the trajectory. The actuators and sensors are build as self contained units communicating with a central Mperl-Π controller via UART. The central controller computes trajectories for the overall system and dispatches commands to the actuator's controllers.

MEASUREMENT WITH THE EULER-BERNOULLI BEAM MODEL. We describe how to use the flexural rigidity of the elbow beam by polling the system for its embedded sensors, creating the appropriate Π groups and calibrating the resulting equations by means of reference weights. We start with a description of the SCARA system which already contains a flex sensor. In that work, the controller was simply adjusting the actuator's angle to compensate for the deflection, but this was not tied to any force. Now we use Mperl-Π to find how to use this sensor to compute the weight attached to the beam. As we do not know the equation, we use dimensional function analysis to synthesize suitable candidate equations. Starting with a local search of distance 1 from our beam, we get the following building parts: the beam itself, the effector, and the actuator. The beam has geometric properties (length, width, height), the sensor input (an angle) and sensor output (a resistance), and fabrication dependent parameters like the flexural rigidity and density. The effector applies a downward force which is proportional to the attached weight. Finally, the actuator's properties are its positions (angle), current, and torque.

From this generated description, we generate the Π groups; among them, we get the candidate

$$F^1(EI)^{-1}L^3d^{-1} \tag{4.17}$$

where F is the force, L the length of the beam, EI the flexural rigidity, and d the deflection. The Π group results in the tentative equation

$$C = \frac{FL^3}{EId} \tag{4.18}$$

It now remains to determine a value for C by calibration. We move the systems to a known pose where the two actuators have the angles $\theta_0 = \pi/4$ and $\theta_1 = -\pi/4$. These angles correspond to the pose where the shoulder beam is vertical and the elbow beam is horizontal. We attach known reference weights to the end effector and measure the deflection. To average out any noise or outliers, we repeat this step 10 times. Table 4.7 summarizes the parameters and their values subject to different weights. Using these values we estimate that the constant C has the value 4.042.

MEASUREMENT WITH THE MOTOR CURRENT. Instead of relying on the flex sensor, we also have the possibility of deducing the weight by combining the current consumption of the revolute actuators, the length of the beams, and the known functional dependency between torque and power consumption from the motor's datasheet. Compared to the previous experiment, we add to the actuator specification

Reference weight [g]	deflection [mm]	C
0	0.00	–
100	3.58	4.045
140	5.03	4.032
200	7.18	4.044
Average	–	4.042

Table 4.7: Deflection measurements and calibration for reference weights.

a torque vs current diagram and perform the Π-group generation and calibration again. We now evaluate the accuracy of the sensor's predictions. Our model is idealized as it does not take the friction between the elements into account, and therefore, the system can stabilize at the same position but with different current consumption as long as the difference in torque output is smaller than the gearbox friction. Instead of relying on fixed, known weights, we put the robot into a pose were it is possible for it to pull a precision spring scale. By pulling with varying intensity, the current consumption changes and can be used to draw conclusion about the force exerted (weight equivalent). In order to measure the error, the actuator pulls on the spring scale with similar intensity as when using reference weights and some intermediate values. Table 4.8 reports the respective numbers. The first row reports the force as measured with the precision spring scale. The next two rows report the measured current (via current sensor) and the derived force by means of the above mentioned invariant. The last two rows repeat the deflection experiment, but the force is derived via the torque/current invariant instead of being measured. We observe that the model's simplification and change of setting results in small measuring errors. Deriving force via torque/current diagram and current sensor, we report a RMSE of 4.768g, as compared to the actual values. Using this invariant to determine the force by means of measuring the deflection results in a RMSE of 6.770g.

AUTOCALIBRATION. We now show how the auto calibration performs in our running SCARA system if redundant invariants are available. In this case, we assume a calibrated invariant which calculates force based on the deflection, and a non-calibrated invariant which used the torque/current approach. The experiment consists of two steps. Firstly, the arm pulls against the spring scale with varying intensity, during which the calibrated invariant records the calculated force (Fig. 4.6). This serves as a reference for the uncalibrated invariant, which can then be evaluated with this calculated value, along with the timely reading of the current sensor; as all quantities in the invariant are known, the dimensionless constant can be calculated. During calibration, the dimensionless constant is averaged as long as

Force [N] (Spring)	Current [A] (Sensor)	Force [N] (Invariant 1)	Deflection [mm](calc.)	Force [N] (Invariant 2)
0.39 (40)	0.20 A	0.33 (33.9)	1.4223	0.49 (50)
0.59 (60)	0.31 A	0.52 (52.7)	2.1442	0.69 (70)
0.78 (80)	0.46 A	0.77 (78.16)	2.8262	0.83 (85)
0.98 (100)	0.60 A	0.99 (101.94)	3.5401	1.03 (105)
1.18 (120)	0.69 A	1.15 (117.24)	4.2498	1.18 (120)
1.47 (150)	0.85 A	1.42 (144.42)	5.2716	1.42 (145)

Table 4.8: We calculate Force via invariant 1 (torque/current curve) and invariant 2 (deflection and invariant 1). The RMSE of invariant 1 compared to the actual force values is 4.768g, the RMSE of invariant 2 is 6.770g. Numbers in parenthesis report weight equivalent of force in [g].

Deflection [mm] (calculated)	Force [N] (calculated)	Force [N] (measured)
1.4223	0.39 (40)	0.49 (50)
2.1442	0.59 (60)	0.69 (70)
2.8262	0.78 (80)	0.83 (85)
3.5401	0.98 (100)	1.03 (105)
4.2498	1.18 (120)	1.18 (120)
5.2716	1.47 (150)	1.42 (145)

Table 4.9: We measure the deflection when lifting weights, derive force from the previously calibrated curve, and compare it to actual force (measured via scale). The force columns has the weight equivalent of the force [g] in parenthesis. The RMSE is 6.770 w.r.t to weight in grams.

the arm pulls on the scale. Secondly, we move the robot arm again to a pose where the shoulder beam is vertical and the elbow beam is horizontal; we then pull at the spring scale to simulate different weights and report the values as calculated by the manually calibrated invariant and the automatically calibrated one. Table 4.10 summarizes the result from the second step. The RMSE of the manually calibrated invariant is 5.2414, the RMSE of the automatically calibrated invariant is 7.8149.

4.5 CONCLUSION

In light of recent advancements in controller synthesis and computational fabrication techniques for robotic systems, we have presented a

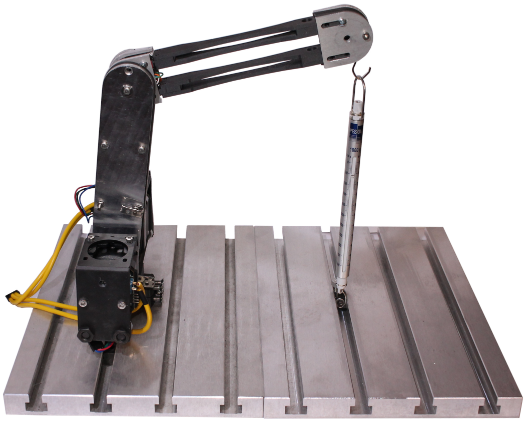


Figure 4.6: Arm pushing against a spring scale to simulate load on the end-effector

Force [N] Spring	Force [N] Cal. Invariant	Force [N] Autocal. Invariant
0.39 (40)	0.44 (45.11)	0.42 (43.55)
0.59 (60)	0.68 (69.35)	0.52 (53.17)
0.78 (80)	0.84 (85.66)	0.71 (73.28)
0.98 (100)	0.97 (98.52)	0.91 (93.42)
1.18 (120)	1.19 (122.02)	1.29 (131.92)
1.47 (150)	1.43 (146.39)	1.55 (158.75)

Table 4.10: Results for autocalibration.

novel application of dimensional analysis. Specifically, we have applied this approach to the design of a robotic arm and have demonstrated the ability to identify two distinct sensor configurations that enable accurate estimation of the system's state.

One key factor of these types of methods is scalability. Moving forward, the sensor configuration synthesis will benefit from enhanced filtering techniques that minimize the number of irrelevant $\Pi-$ groups generated during the dimensional analysis process. This will enhance scalability, particularly for larger systems, and will eventually allow the integration of a more realistic physics model. While we adopt a correct by construction strategy for the controller, our current methodology does not account for measurement precision during sensor selection, and the state estimation lacks any guarantees. To address these limitations, end-to-end correctness assurances for the system should be obtained, and confidence bounds on the measurements should be derived. A second key factor is the tighter integration into the controller synthesis step. The algorithm itself should be enhanced to provide a feed back loop that allows to return information about the available sensors to the controller synthesis of Mperl. For instance, if the sensor configuration synthesis fails to derive direct or indirect measurements for a specific quantity, the controller synthesis should backtrack and try to find a controller which does not depend on that quantity.

In the previous chapters, we outlined the concepts and language of Mperl and how individual robotic systems can be created. While Mperl is able to create simple controllers for non-expert users, the Sensor Configuration Synthesis makes it possible to synthesize considerably more sophisticated controllers by deriving physical invariants that capture physical properties of the system. Motion primitives like GRAB, which allows the arm to reach for an object can be enhanced by adding ways to derive the weight of the object, which allows for richer abilities of the robot, but also lends, in the case of redundant invariants, a certain robustness to the controller.

5 MULTI-ROBOT APPLICATION PROGRAMMING

The techniques discussed in the previous chapters, aimed at synthesizing controllers and configuring sensors, have provided valuable tools for developing robotic systems. However, developing multi-robot applications still presents significant challenges due to the complex interplay between dynamic controllers, geometric constraints, and concurrency and synchronization requirements. Motion primitives involve continuous physical processes that can be coupled, and executing one may affect the physical state of a concurrently executing motion primitive. Additionally, since the robotic components exist in 3D space, the software must ensure that the range of motions executed by the motion primitives are compatible with the geometry of the components and that there are no collisions. This complexity is illustrated in the example of a robotic assembly consisting of a mobile cart with an arm attached to it.

To address these challenges, we present a new programming model called *PGCD* in this chapter. PGCD consists of assemblies of robotic components, along with a run time and a verifier, which combine message-passing concurrent processes with motion primitives and explicit modeling of geometric frame shifts. These features allow for the continuous evolution of trajectories in geometric space under the action of dynamic controllers and relative coordinate transformations between components evolving in space. In addition, a verification algorithm has been developed that can statically verify concurrency-related properties and geometric invariants, providing a high level of confidence in the correctness of PGCD programs. The PGCD programming model, along with the methods developed in the previous chapters, offer a powerful set of techniques for developing robotic systems. While the previous chapters were focused on providing accessible techniques for non-experts, the PGCD model is applicable to experts and non-experts alike. The next chapter will delve into the details of the PGCD programming model, exploring how it can be used to develop collaborative robotic systems that are reliable, efficient, and effective. Illustrating the challenges faced by robotics programmers, let us consider the example of the task FETCH, that requires a robotic system consisting of a mobile cart with an attached arm. The objective of this task is to execute a sequence of actions involving the grabbing of an object by the arm. If the target object is out of the arm's reach, the cart to which the arm is attached must be moved to a suitable location to facilitate the object's retrieval. Both the cart and the arm are equipped with specific capabilities, such as the ability to move and to grab objects within its reach. Despite the seemingly simple nature of the task, implementing and verifying it is not straightforward due to various reasons.

Firstly, controllers of the components are dynamically coupled. For instance, the weight and center of mass of the arm affect the movement of the cart. A light and small arm may allow for a faster motion planner compared to a heavy or bulky arm. Secondly, since the cart and the arm exist in a 3D space, their range of actions is influenced by their surroundings. For example, whether the cart can navigate through a passage may depend on the state of the arm. An extended arm may collide with an obstacle and invalidate a path that the cart, by itself, could traverse. Conversely, the base of the cart restricts the range of motion of the arm. Thirdly, the motion primitives of the cart and the arm refer to coordinates in their local frame. Therefore, when the cart moves, the arm moves along, and any communication of the geometric space between the cart and the arm requires a transformation of information between their coordinate systems. Finally, an natural approach to break down the FETCH task involves the cart moving towards the target while the arm is retracted, followed by the arm grabbing the object, and then the cart moving back to the starting point. Achieving this sequence necessitates coordination between the code controlling the cart and the arm. The processes must signal which step is currently in progress, the guarantees provided by each process to the other, and manage the intricacies of concurrent programming.

Current state-of-the-art programming languages and tools do not provide a comprehensive solution for simultaneously dealing with these complexities, leading to challenges in developing robust and efficient robotics applications. While messaging systems like the robot operating system (ROS) [116] exist to support communication between different components of a robotics system, there is a significant gap in the availability of programming models and tools that enable reasoning about the interaction between these domains. As a result, robotics application developers are often left to navigate these complexities by themselves, using low-level robotics languages provided by robot manufacturers or imperative languages like C++ or Python.

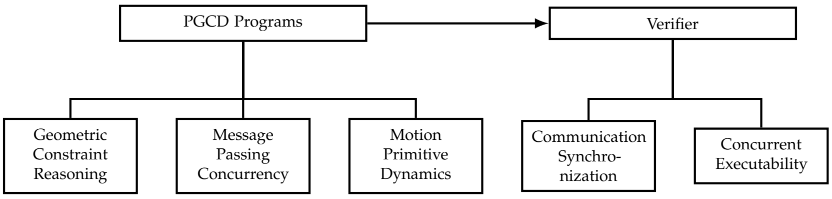

PGCD Programs
Verifier
Geometric Constraint Reasoning
Message Passing Concurrency
Motion Primitive Dynamics
Communication Synchro-nization
Concurrent Executability

Figure 5.1: PGCD Architecture

In this chapter, we present PGCD, a concurrent programming model (Fig. 5.1), which combines geometric constraint reasoning of the robotic components with message passing concurrency and motion primitives for dynamics. A program in our model consists of a set of *processes* —each process represents a logical portion of a robotic assembly ("cart" or "arm"). Processes run sequential code and send or receive messages as well as execute a set of *motion primitives* on the underlying physical state. A program is structurally determined by assembling processes through *attachments*: an attachment couples the physical state of two components and also determines the relative coordinate transformations between their geometries. For instance, for the *Fetch* example, there is a process for the cart and a process for the arm. The program is obtained by attaching the arm process to the cart process. The semantics of PGCD programs define a transition system in which communication occurs in logical "zero time" and motion primitives execute in real time. The semantics include geometric transformations between processes. Thus, the content of messages between the cart and the arm processes is transformed to the recipient's coordinate system.

Our work involves the development of a verifier for PGCD programs, which takes as input a collection of robotic components, a PGCD program, a specification of the motion primitives, and a description of the environment. The specification of a motion primitive includes both constraints on a robot's state and its *footprint*, i.e., the region of space used by the robot when executing that motion primitive. The verification process involves two steps. Firstly, we verify communication and synchronization correctness, such as the absence of deadlock. Secondly, we ensure concurrent executability of motion primitives through assume-guarantee reasoning and separation of geometric resources. By enforcing this separation, it is possible to maintain the important invariant that different components of a system do not collide with each other.

The programmer writes constraints and footprints of each process in its local frame, while the run time and the verification engine automatically performs frame shifts. Our implementation is in PYTHON, and we provide a run-time system to execute our programs on top of ROS. We have used our implementation to verify actual multi-robot co-ordination implementations and execute verified programs on real robotic hardware. Our evaluation demonstrates that our framework for programming of multi-robot co-ordination and manipulation can lead to statically verified implementations that run on off-the-shelf and custom-built robotics hardware platforms. Overall, our programming model, run-time system, and verifier form the basis for correct design and static verification of complex robotic applications that interact dynamically in geometric space.

5.1 PGCD PROGRAMS

The proposed concurrent programming model, PGCD, is specifically designed to handle cyber-physical components that operate within 3D geometric space. In this model, a program is comprised of multiple processes that execute concurrently, each responsible for controlling a set of physical variables by executing motion primitives. These processes must also communicate and synchronize with one another to achieve their objectives. As the dynamics of physical variables are coupled, the program structure reflects this coupling, with the composition of processes reflecting the couplings and frame shifts between physical variables. Therefore, concurrent processes can operate in different reference frames relative to one another. For example, an arm attached to a cart may describe its motion in its local frame, but its local frame may move relative to the world frame if the cart moves. The PGCD programming model includes constructs for hierarchical frame shifting, and its semantics ensure that values exchanged between processes are transformed into the world frame by evaluating values under sequences of frame shift operations. This programming model is designed to provide a formal approach to programming cyber-physical systems operating in 3D space.

The correctness of a program is determined by its syntax and its semantics. Syntax refers to the formal rules that specify the structure of the program and how the elements of the language can be combined to form valid expressions, statements, and programs. Semantics, on the other hand, defines the meaning and behavior of those expressions, statements, and programs, and based on that, creates a model of computation.

In this chapter, we will discuss both syntax and semantics of the PGCD language.

5.1.1 Syntax

We consider a fixed finite set $\mathbb{C}$ of *processes*. Each *process* $P \in \mathbb{C}$ is a tuple

$$(Var, \mathcal{M}, S, \rho, rsrc) \tag{5.1}$$

where *Var* is a set of variables, with two distinguished disjoint subsets X and W of *physical state* and *external input* variables, $\mathcal{M}$ is a set of motion primitives, $\rho : Var \to [\![Var]\!]$ is a *store* mapping variables to values, $rsrc : \rho \to 2^{\mathbb{R}^3}$ is a *resource function*, and S is a *statement* generated by the grammar provided in Listing 5.1.

Within this grammar, $a \in \mathbb{C}$ represents a (different) component, $\mathbf{m} \in \mathcal{M}$ is a motion primitive, $x \in Var$ ranges over variables, l is a label from a fixed finite set of labels, and *expr* comes from an (unspecified) effectively computable language of arithmetic expressions.

```
S ::= x := expr
   | m
   | send(a, l, expr)
   | receive(m){(l, x, S)+}
   | S; S
   | if expr then S else S
   | skip
   | while expr do S
```

Listing 5.1: PGCD Grammar

To ensure ease of understanding and clarity, we have excluded type annotations for variables in our implementation and we assume that all processes and motion primitives used in the model are well-typed.

A process represents a unit of a program which controls a continuous physical system through the application of motion primitives and, additionally, communicates with other concurrently executing processes. The *store* ρ gives values to the local process variables *Var*; this includes valuations to the physical state variables. The resource function gives an upper bound of the geometric space used by a process. Part of the statements form a core imperative programming language, with skip, assignments, sequential composition, conditionals, and loops. In addition, a process can execute motion primitives and send and receive *labeled* messages for synchronization. The message labels come from a finite set known to all processes. As shorthand, we often write labeled messages as $l(v)$ where l is the label and v a value. When the message does not carry a value, we simply write l. Receiving messages operates on a list of triples of the form (l, x, S) where l is a label, x is a name to which the received value is assigned, and S is the continuation. Since receive is a blocking operation it also takes as argument a motion primitive which is executed while the process waits for a message. We consider messages with a single payload value, but this can be generalized to tuples.

The resource function *rsrc* takes as input the state of the process and returns an over-approximation of the space used by the robot (a subset of $\mathbb{R}^3$). The resource function does not need to be precise but only to over approximate the robot. Typically, we use 3D hyper-rectangles, called *bounding boxes*, because checking collision between boxes is efficient.

To illustrate this, consider the example of the cart of the cart-and-arm system. The process that corresponds to the cart encapsulates the motion primitives and implements a control program that decides when to execute these motion primitives. The set

$$Var = \{\mathbf{p}_{\text{cart}}, \mathbf{r}_{\text{cart}}, s_{\text{cart}}\} \cup \{m_{\mathbf{obj}}\} \tag{5.2}$$

gives the physical state and external input variables, respectively. The program

```
receive(idle(ρ(pcart))){step, _, forward(ρ(pcart,ρ(rcart)))}
```

specifies that the cart remains idle at its current position $\rho(\mathbf{x})$ until it receives a *step* message (without other values), and then steps one unit using the motion primitive **forward**.

Let d_l, d_w, d_h be bounds on the cart's length, width, and height, s.t. their center point coincides with the cart's center. The resource function *rsrc* gives a bounding box around the cart's position:

$$\left\{ p' \in \mathbb{R}^3 \middle| \begin{array}{l} -d_l/2 \leqslant (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_x) \leqslant d_l/2 \\ \wedge -d_w/2 \leqslant (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_y) \leqslant d_w/2 \\ \wedge 0 \leqslant (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_z) \leqslant d_h \end{array} \right\} \tag{5.3}$$

We write the resource in this style, rather than the more obvious predicate

$$\{x \mid |x_1 - p_{\text{cart}}.x| \leqslant \frac{d_l}{2}, |x_2 - p_{\text{cart}}.y| \leqslant \frac{d_w}{2}, |x_3 - p_{\text{cart}}.z| \leqslant \frac{d_h}{2}\} \tag{5.4}$$

to make frame reasoning about resources easier; in Eq. (5.3), vectors are defined as abstract elements that can be directly transformed across frame shifts. □

5.1.2 Attached Composition

Assuming that two processes, P_1 and P_2, have separate variable sets, the most straightforward approach to combining them is by linking certain physical variables of one process to the external variables of the other, and vice versa. This connection results in the coupling of the motion primitives of the two processes. A *connection* θ between P_1 and P_2 is a finite set of pairs of variables, $\theta = \{(x_i, w_i) \mid i = 1, \ldots, m\}$, such that:

1. for each $(x, w) \in \theta$, we have $x \in P_1.X$ and $w \in P_2.W$ or $x \in P_2.X$ and $w \in P_1.W$, and
2. there does not exist $(x, w), (x', w) \in \theta$ such that x and x' are distinct.

Two connections θ_1 and θ_2 are *compatible* if $\theta_1 \cup \theta_2$ is a connection. Given a connection θ, we write

$$\theta(x) = \{w \mid (x, w) \in \theta\} \tag{5.5}$$

and

$$rng(\theta) = \{w \mid \exists x.(x, w) \in \theta\} \tag{5.6}$$

In a connection, it is possible for a physical state variable of a process to be connected to several external variables. However, each external variable of a process can only be connected to one state variable at most. A connection between components couples the variables of these processes, but they may be interpreted in distinct frames. As such, communicating geometric objects through a connection between components may necessitate a frame shift. This need for frame shifting is what led to the following definition of attached compositions.

Let θ be a connection between P_1 and P_2 and let M be a term over the variables of P_1. We assume M evaluates to a frame transformer in $SE(3)$. We define the *attached composition* operation $P_1 \lhd_{\theta,M} P_2$ which connects variables through a connection θ and applies a frame shift M for any communication of geometric objects (points or vectors) from P_2 to P_1 and a reverse shift $P_1.\rho(M^{-1})$ for any communication from P_1 to P_2. The semantic rules in section 5.1.4 apply the necessary frame shifts automatically.

A connection introduces the following constraint on stores:

$$P_1.\rho(w) = P_1.\rho(M)(P_2.\rho(y)) \tag{5.7}$$

whenever

$$(y, w) \in \theta, \text{with } y \in P_2.X \wedge w \in P_1.W \tag{5.8}$$

and

$$P_2.\rho(w) = P_1.\rho(M^{-1})(P_1.\rho(y)) \tag{5.9}$$

whenever

$$(y, w) \in \theta, \text{with } y \in P_1.X \wedge w \in P_2.W \tag{5.10}$$

After the frame shift, the values in $P_1.W$ connected to $P_2.X$ are "set" by the corresponding values in P_2's store, and the same applies in reverse.

Let P_1, P_2, and P_3 be processes, θ_{12} be a connection between P_1 and P_2 and θ_{13} be a connection between P_1 and P_3. Let M_{12} and M_{13} be relative frame shifts between P_1 and P_2 and P_1 and P_3, respectively. The operation

$$P_1 \lhd_{\theta,M} P_2 \tag{5.11}$$

is considered left associative and we write

$$P_1 \lhd_{\theta_{12},M_{12}} P_2 \lhd_{\theta_{13},M_{13}} P_3 \tag{5.12}$$

for

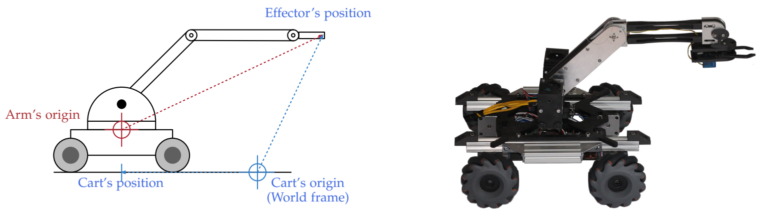
$$(P_1 \lhd_{\theta_{12},M_{12}} P_2) \lhd_{\theta_{13},M_{13}} P_3 \tag{5.13}$$

In this expression, both P_2 and P_3 are children of P_1. Therefore, we have

$$(P_1 \lhd_{\theta_{12},M_{12}} P_2) \lhd_{\theta_{13},M_{13}} P_3 \tag{5.14}$$

$$=(P_1 \lhd_{\theta_{13},M_{13}} P_3) \lhd_{\theta_{12},M_{12}} P_2 \tag{5.15}$$

Unlike the convention parallel composition of processes, attached composition is not commutative. For example, the frame of the attached composition $P_1 \lhd_{\theta,M} P_2$ is the frame of P_1. Swapping P_1 and



(a) Schematic representation. The blue line shows the position of the cart and the end effector relative to the origin of $\mathcal{W}$; the red line the relative position of the end effector to the origin of the arm's reference frame

(b) Actual cart-and-arm system

Figure 5.2: Schematic and actual representation of the cart-and-arm system

P_2 results in a change of frame to P_2, unless $M = I$. Therefore, the attached composition is only of a restricted form of commutativity, i.e.

$$P_1 \lhd_{\theta,I} P_2 = P_2 \lhd_{\theta,I} P_1. \tag{5.16}$$

To provide an example that illustrates the preceding results, consider a second process of an arm mounted on the cart. The cart's variables are the three angles α, β, γ, representing the joints between each of its parts, and m_{arm} which represents the overall mass of the arm (motion platform, gripper, carried object). The frame of the arm is the center of its base as shown in Figure 5.2a. We model this using the attached composition operation $C \lhd_{\theta,M} A$ with $M = \begin{bmatrix} \mathbf{r}_{\text{cart}} & \mathbf{p}_{\text{cart}} \\ 0_{1\times 3} & 1 \end{bmatrix}$ which shifts the frame according to the cart's position and heading. $\theta = \{(m_{\text{arm}}, m_{\mathbf{obj}})\}$ connects the arm's mass to the cart's payload. ☐

5.1.3 Programs

A (concurrent) *program* Π connects processes using the attached composition operator

$$\Pi ::= P \mid \Pi \lhd_{\theta,M} \Pi \tag{5.17}$$

where θ is a connection, and M is a term representing a frame transformer. Since attached composition is left associative, a program arranges processes in a tree structure which induces a parent-child relationship between processes.

For simplicity of notation, we assume there is a *virtual world process* $\mathcal{W}$ defined as

$$\mathcal{W} = (\_, \emptyset, C_V, \{\mathsf{idle}\}, \mathsf{while}(\mathsf{true})\mathsf{idle}) \tag{5.18}$$

with

$$C_V = (\emptyset, \emptyset, \emptyset, \_ \to \emptyset, \_ \to \emptyset) \tag{5.19}$$

This allows us to represent two components P and P' which are not "physically attached" together by attaching both of them to $\mathcal{W}$, i.e.,

$$\mathcal{W} \lhd_{\emptyset,M} P \lhd_{\emptyset,M'} P' \tag{5.20}$$

$\mathcal{W}$ may also be used to model features of the environment, e.g., attaching obstacles to it. When $M = M' = I$, we just write $P|P'$.

For instance, by mounting an arm onto the cart, it becomes feasible to accomplish more sophisticated tasks that neither component could perform on its own. To illustrate this, we developed a concurrent program that controls both the arm and the cart, with the objective of retrieving an object located at a remote site.

Two code fragments are presented in Algorithms 5.2 and 5.3 to emphasize their communication, and the interaction is visually summarized in Figure 5.3. To simplify the presentation, additional details of the computation are omitted, and some notation is simplified for readability. However, these programs can still be readily compiled into our core grammar for statements.

```
send(arm, fold);
receive(idle)
    folded ⇒ skip
while (|target − p| > reach) do
    moveToward(target)
send(arm, grab(target));
receive(idle)
    grabbed ⇒ skip
send(arm, fold);
receive(idle)
    folded ⇒ skip
while (p ∉ homeRegion) do
    moveToward(homeRegion)
send(arm, done);
```

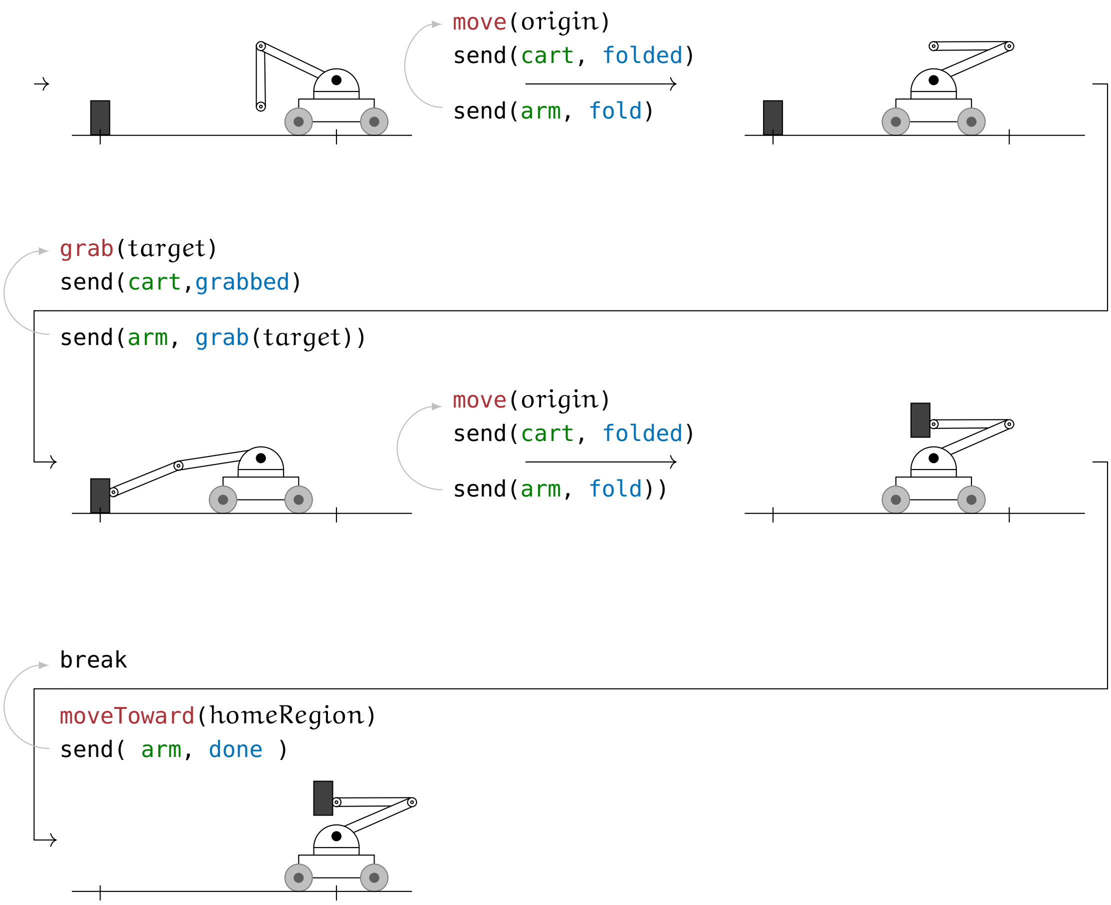
Listing 5.2: Process Cart

```
while true do
    receive(idle)
        fold ⇒
            move(origin)
            send(cart, folded)
        grab(loc) ⇒
            grab(loc)
            send(cart, grabbed)
        done ⇒
            break
```

Listing 5.3: Process Arm

5.1.4 Semantics

The execution of PGCD programs can be intuitively divided into rounds, each of which comprises two sub-rounds. During the first sub-round, components exchange messages in logical zero time. In the second sub-round, each component executes a motion primitive for a duration of T. The real-time execution of these motion primitives synchronizes all components. The semantics of our programming model can be formally defined as labeled transition rules between program states. A program state consists of a program Π and the stores of each process in Π. Given a store ρ of a process and a variable



```
move(origin)
send(cart, folded)
send(arm, fold)

grab(target)
send(cart,grabbed)
send(arm, grab(target))

move(origin)
send(cart, folded)
send(arm, fold))

break
moveToward(homeRegion)
send( arm, done )
```

Figure 5.3: Abbreviated schematic representation of Fetch PGCD process interaction from the arm's perspective. Motion primitives in red, messages in blue, participants in green; actions of the cart are below the arrow, actions of the arm above. Receive statements are omitted.

$x \in \mathit{Var}$ of that process, we write $\rho(x)$ for the value of variable x and lift this to expressions: $\rho(e)$ is the evaluation of e in environment ρ. We write $\rho(x) \leftarrow \nu$ for the store which maps variable x to ν but agrees with ρ on all other variables.

The transitions $\rightarrow$ between program states are of the form:

- $\xrightarrow{\tau}$: an internal step to a process.
- $\xrightarrow{p!l(\nu)}$: sending label l with value ν to component p.
- $\xrightarrow{p?l(\nu)}$: p receiving a message with label l and value ν.
- $\xrightarrow{\xi,\nu,T}$: the system follows the output trajectory ξ, with external inputs ν for the duration T.

5.1.4.1 *Contexts.*

We define the semantics in a contextual style. A *statement context* Σ extracts the next statement to be executed in a sequence

$$\Sigma ::= [] \mid \Sigma; S \tag{5.21}$$

and $\Sigma[S]$ is obtained by replacing $[]$ by a statement S in Σ.

The semantic rules presented below rely on several auxiliary functions and predicates. For instance, the *leftH* function is used to return the output of the leftmost process in a program and is recursively computed using

$$\mathit{leftH}(P) = P.\rho \tag{5.22}$$

$$\mathit{leftH}(\Pi \lhd_{\theta,M} \Pi') = \mathit{leftH}(\Pi) \tag{5.23}$$

We use *leftH* for the frame shift in a composition that is a function of the output of the leftmost process. For example, we write $\mathit{leftH}(\Pi_1 \lhd_{\theta,M} \Pi_2)(M)$ for the transformation in $\mathit{SE}(3)$ obtained by evaluating the term M in the store of the leftmost process in the program. $\mathit{disjoint}(P, Q)$, with $P \lhd_{\theta,M} Q$, is a shorthand for

$$P.\mathit{rsrc}(P.\rho) \cap \mathit{leftH}(P)(M)(Q.\mathit{rsrc}(Q.\rho)) = \emptyset \tag{5.24}$$

This predicate verifies the absence of overlap in the footprint of the left and right components. Whenever a process undergoes a state transition, the disjointness constraint must be preserved.

5.1.4.2 *Transitions*

The main transition rules (inter-process communication) are presented in Figures 5.4 and 5.5; Figure 5.6 summarizes the reduction rules for the control-flow. Inter-process communication happens by rendezvous on a shared channel: a sender process sends a value v on a channel a; simultaneously, a receiver process receives the value, shifted to its own frame, and continues executing. The semantics take care of the frame shift of value v from the sender to the receiver. This approach of implicitly taking care of the frame shift is used in existing systems like the TF2 library [33]. The (CommSend) and (CommRecv) rules describe how processes send and receive messages. When sending a message, the expression e is evaluated to a value v in the store of P ($\rho(e) \Downarrow v$). Sending does not change the local state, it only consumes the send instruction. Receiving a message binds the value v carried by the message to the receiving variable x in the store ($\rho(x) \leftarrow v$) and continues with the appropriate statement S determined by the label.

The rules (CompL) and (CompR) *propagate* communication and silent steps. More specifically, if a component of a propagate can make a transition labeled a, then the entire program can also make a transition labeled a. During the propagation, the store is updated to

(CommSend)

$$\frac{\rho(e) \Downarrow v}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{send}(a, l, e)], \rho, \cdot) \xrightarrow{a!l(v)} (\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{skip}], \rho, \cdot)}$$

(CommRecv)

$$\frac{\rho' = \rho(x) \leftarrow v}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{receive}(\mathbf{m})\{\dots(l, x, S)\dots\}], \rho, \cdot) \xrightarrow{P?l(v)} (\mathit{Var}, \mathcal{M}, \Sigma[S], \rho', \cdot)}$$

(CompL)

$$\frac{P \xrightarrow{\alpha} P' \qquad \mathit{disjoint}(P', Q)}{P \lhd_{\theta,M} Q \xrightarrow{\alpha} P' \lhd_{\theta,M} Q}$$

(CompR)

$$\frac{\begin{array}{c} Q \xrightarrow{\alpha} Q' \qquad \mathit{disjoint}(P, Q') \\ \alpha = \alpha' = \tau \lor (\alpha = a\#l(v) \land \mathit{leftH}(P)(M)(v) \Downarrow v' \land \alpha' = a\#l(v') \land \# \in \{!, ?\}) \end{array}}{P \lhd_{\theta,M} Q \xrightarrow{\alpha'} P \lhd_{\theta,M} Q'}$$

(CommSyncLR)

$$\frac{P \xrightarrow{a!v} P' \qquad Q \xrightarrow{a?v'} Q' \qquad \mathit{leftH}(P)(M^{-1})(v) \Downarrow v' \qquad \mathit{disjoint}(P', Q')}{P \lhd_{\theta,M} Q \xrightarrow{\tau} P' \lhd_{\theta,M} Q'}$$

(CommSyncRL)

$$\frac{Q \xrightarrow{a!v} Q' \qquad P \xrightarrow{a?v'} P' \qquad \mathit{leftH}(P)(M)(v) \Downarrow v' \qquad \mathit{disjoint}(P', Q')}{P \lhd_{\theta,M} Q \xrightarrow{\tau} P' \lhd_{\theta,M} Q'}$$

Figure 5.4: Reduction rules for communication

reflect the frame and connections. If the transition carries a message from the right side of the composition, the value in the message is shifted to match the overall frame inherited from the left process ($\mathit{leftH}(P)(M)(v) \Downarrow v'$). Furthermore, the transition may update the local state which changes the resources used by the processes, and it is necessary to ensure that the two components are disjoint to avoid conflicts between their resource usage.

The (CommSyncLR) and (CommSyncRL) rules match the send and receive statements. Their structure is similar to the (CompL) and (CompR) rules with both sides changing. Once the send and receive are matched, the label of the action is not propagated further and the action becomes an internal action (τ) of the composed processes.

The execution of motion primitives is illustrated in Figure 5.5. Transitions are labeled with the trajectory (ξ) of the local states, the external inputs (ν), and the total time of the transition (T). The (Motion) rule checks the conditions of the motion primitives and makes sure the store matches the initial state and the final states of the trajectory. The (Time) rule combines trajectories of individual processes; the trajectory of one process is projected to become part of the disturbances of the other process. Specifically, ν_P receives the external inputs projected

(Motion)

$$\frac{\begin{array}{c} \bigvee \begin{cases} S{=}\Sigma[\mathbf{m}] \wedge S'{=}\Sigma[\mathbf{skip}] \\ S{=}\Sigma[\mathbf{receive}(\mathbf{m})\{\dots\}] \wedge S'{=}S \end{cases} \\ \xi(0) = \rho|_X \qquad \rho' = \rho|_{\mathit{Var}} \cup \xi(T) \\ \mathbf{m}.\mathrm{Pre}(\xi(0), \nu(0)) \qquad \mathbf{m}.\mathrm{Post}((\xi(T), \nu(T))) \\ \forall t \in [0, T].\ \mathbf{m}.\mathrm{Inv}((\xi(t), \nu(t))) \end{array}}{(\mathit{Var}, \mathcal{M}, S, \rho, \mathit{rsrc}) \xrightarrow{\xi,\nu,T} (\mathit{Var}, \mathcal{M}, S', \rho', \mathit{rsrc})}$$

(Time)

$$\frac{\begin{array}{c} P \xrightarrow{\xi_P,\nu_P,T} P' \qquad \nu_P = \nu|_P \cup \xi(M)(\theta(\xi_Q)) \\ Q \xrightarrow{\xi_Q,\nu_Q,T} Q' \qquad \nu_Q = \nu|_Q \cup \xi(M^{-1})(\theta(\xi_P)) \\ \xi = (\xi_P \cup \xi_P(M)(\xi_Q)) \\ \forall t \in [0, T].\ P.\mathit{rsrc}(\xi_P(t)) \cap \xi_P(t)(M)(Q.\mathit{rsrc}(\xi_Q(t))) = \emptyset \end{array}}{P \lhd_{\theta,M} Q \xrightarrow{\xi,\nu,T} P' \lhd_{\theta,M} Q'}$$

Figure 5.5: Reduction rules for motion

on P ($\nu|_P$), along with the output of Q through the connection ($\theta(\xi_Q)$) to which we apply the frame shift ($\xi(M^{-1})$). Finally, we also check that the resources used by the two processes stay disjoint during the execution of the motion primitives.

The semantics of the internal control flow of the processes are presented in Figure 5.6. These rules are typical for an imperative programming language and involve silent transitions that follow the usual semantics of such languages.

At the root of the $\lhd_{\theta,M}$ tree, only $\xrightarrow{\tau}$ and $\xrightarrow{\xi,\nu,T}$ are allowed. Messages **send** and **receive** must be matched inside the system (closed world hypothesis). Once **send** and **receive** are matched the label becomes τ.

5.2 VERIFICATION

There are various ways in which a PGCD program may encounter issues during execution and become unresponsive. One such scenario arises when the message passing mechanism gets deadlocked because the send operations are blocking. Another possibility is when a process attempts to execute a motion primitive while its precondition is not met. Finally, there can be a resource conflict between two processes executing their motion primitives concurrently. In the following section, we present an algorithm for verifying PGCD programs to address these issues.

The verification of PGCD programs is based on the separation of the verification problem into two types of periods: logical zero-time message-passing periods and real-time periods when time elapses following the trajectories defined by motion primitives (Fig. 5.7). Our verification algorithm uses a combination of model-checking

(Seq)

$$\frac{}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{skip}; S], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[S], \rho, \mathit{rsrc})}$$

(Assign)

$$\frac{\rho(e) \Downarrow v \qquad \rho' = \rho(x) \leftarrow v}{(\mathit{Var}, \mathcal{M}, \Sigma[x := e], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{skip}], \rho', \mathit{rsrc})}$$

(WhileT)

$$\frac{\rho(e) \Downarrow \mathsf{true}}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{while}\ e\ \mathbf{do}\ S], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[S; \mathbf{while}\ e\ \mathbf{do}\ S], \rho, \mathit{rsrc})}$$

(WhileF)

$$\frac{\rho(e) \Downarrow \mathsf{false}}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{while}\ e\ \mathbf{do}\ S], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{skip}], \rho, \mathit{rsrc})}$$

(IteT)

$$\frac{\rho(e) \Downarrow \mathsf{true}}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S'], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[S], \rho, \mathit{rsrc})}$$

(IteF)

$$\frac{\rho(e) \Downarrow \mathsf{false}}{(\mathit{Var}, \mathcal{M}, \Sigma[\mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S'], \rho, \mathit{rsrc}) \xrightarrow{\tau} (\mathit{Var}, \mathcal{M}, \Sigma[S'], \rho, \mathit{rsrc})}$$

Figure 5.6: Reduction rules for control flow.

and constraint-solving. The model-checker checks the correctness of message communication between processes. A numerical solver for non-linear constraints over the reals checks the correctness of motion primitives.

5.2.1 Communication safety

For the messages, we want to show that the communication between components is well-formed. In particular, we verify that the program does not get into a state where some process is forever blocked on a send operation and that there is no unbounded execution solely with message passing (making time "stop").

Our verification algorithm converts a program into concurrently executing control flow automata (CFA) [54]. We abstract the code of each CFA, preserving only the send, receive, and motion primitives. Additionally, we abstract local computation and treat internal choices (**if** **then** **else**) as non-deterministic. We then verify that there are no loops (cycles) in each CFA without any motion primitives, which is sufficient to prevent potentially infinite "0-time" computation.

Next, we take the synchronized product of all the CFAs, synchronizing matching **send** and **receive** statements based on labels. Motion primitives synchronize *all* processes globally in real-time. For **receive** statements, a motion primitive executes only when no more communication is possible. Finally, we check for deadlock, which occurs when

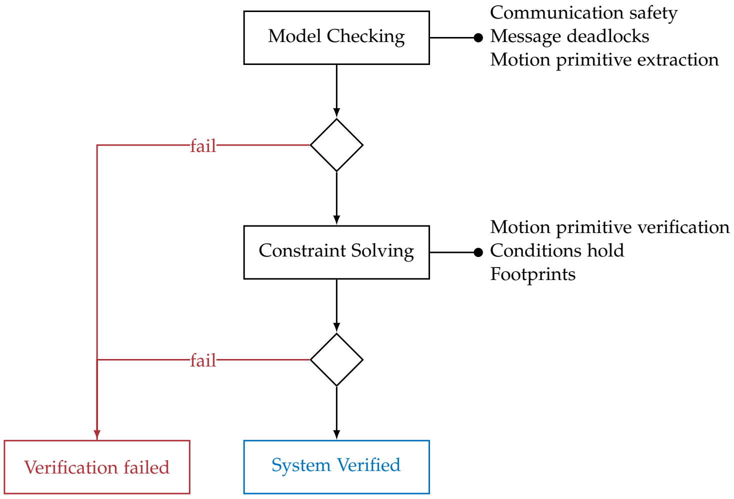


Figure 5.7: Verification is done in two steps, logical zero-time message passing and real-time motion primitive periods. The model-checker ensures correct message communication, while the constraint solver checks the correctness of motion primitives.

a non-final state has no successor, by exploring the state space of the synchronized product using a model checker. In our implementation, we encode the CFAs as Promela processes and perform the exploration using the Spin model checker [55], instead of building the product explicitly. It is worth noting that the construction presented earlier results in a single final state where all processes have finished. As a result, the deadlock check ensures that the processes are either communicating, executing a motion primitive, or have terminated. Additionally, we extract the set of motion primitives executed concurrently during the state space exploration, which is used in the second part of the verification process.

5.2.2 Trajectories and footprints

To verify the correct execution of the abstract motion primitives, we perform two checks, based on the combination of motion primitives recorded during model checking. The first check involves motion primitives from different processes executing concurrently. We ensure that a trajectory satisfying the rules for motion (shown in Figure 5.5) exists. The second check involves motion primitives executed sequentially by the same process. We verify that the post-condition of the first motion primitive implies the pre-condition of the following one. Currently we rely on the user giving state invariants in the form of program annotations to help with the verification process. These annotations associate predicates with program locations. To verify that motion primitives executing concurrently have a joint trajectory, we employ an assume-guarantee style of reasoning. When two processes are attached, one process depends on the invariants of the other's output (which can be an external input) to satisfy its own invariant, and vice versa. To obtain the assumption and guarantee for a predicate $\mathcal{P}$ of process P (e.g., the invariant of a motion primitive), we project $\mathcal{P}$ onto P's external inputs to obtain the assumption, and onto P's physical state variables to obtain the guarantee. That is, $A_{\mathcal{P}} \Leftrightarrow (\exists x \in P.X.\ \mathcal{P})$ and $G_{\mathcal{P}} \Leftrightarrow (\forall w \in P.W.\ \mathcal{P})$.

For assume-guarantee reasoning, we follow the method presented by Nuzzo [104]. Given a program Π and an invariant I_P for each process P, we first traverse Π starting from the leaves to generate assume/guarantee contracts for each attached composition. Each $P \lhd_{\theta,M} Q$ gets a contract based on the contract of their children:

- $A \Leftrightarrow (A_P \vee M(A_Q) \vee \neg(G_P \wedge M(G_Q))) \wedge \forall(x,y) \in \theta.\, x = y$
- $G \Leftrightarrow G_P \wedge M(G_Q) \wedge \forall(x,y) \in \theta.\, x = y$

These rules are the composition rules from [104, Section 2.3.2] to which we have added the frame shifts. At each step of this process we need to check that the well-formedness of the composed contracts:

- *compatibility*: A is satisfiable,
- *consistency*: G is satisfiable, and
- *spatial separation*: $G \Rightarrow P.\mathit{rsrc} \cap M(Q.\mathit{rsrc}) = \emptyset$ is valid.

The *spatial separation* check is new in our system and states that under the guarantees provided by both P and Q, their respective resources must be disjoint. Furthermore, when the root of Π is reached, the final A must be implied by the environment assumption and the final G is the overall behavior of the system.

5.2.3 Specifications and Annotations

The verifier for PGCD is not fully automatic; to help the verifier, the programmer needs to provide some annotations. As mentioned earlier, a motion primitive **m** is specified by $(\mathrm{Pre}, \mathrm{Inv}, \mathrm{Post})$. Further, a resource function for a robot with a complicated geometry can be complex; for example, for a robotic arm, it can be the geometry of the arm itself. In practice, the check for spatial separation uses programmer-specified *abstract footprint* predicates which over-approximate $P.\mathit{rsrc}$ and $Q.\mathit{rsrc}$ but for which the separation check is efficient.

For instance, in the GRAB motion primitive of an arm, we can over-approximate the arm's working envelope (which can be a complex and non-convex set) by a half sphere around its base with a radius corresponding to the arm's maximal extension. This formula is simpler than the arm's resource function as it does not depend on the arm's state but may be sufficient to handle some scenarios. With the extra footprint specification, it becomes possible to divide the collision check in two parts:

1. we check that each component stays within the footprint of its motion primitive and

2. we check that the footprint of concurrently executing motion primitives do not intersect.

The second type of user annotations are invariants given as constraints over the system's state at particular program locations. For instance, in Algorithms 5.2 and 5.3 when the cart is at line 6 and the arm at line 2, we have $|target - C.p| \leqslant target \wedge A.\gamma = 0 \wedge A.\beta = minLowerAngle \wedge A.\alpha = maxUpperAngle$. This is the conjunction of the exiting the loop (Algorithms 5.2 line 4) and the arm in a folded state. Similar to loop invariants, these invariants play a crucial role in our reasoning process and enable us to break down the verification into a finite number of checks.

5.2.4 Extensions

The following extension pertains specifically to the IDLE motion primitive. Currently, our implementation only supports state formulas as specifications and not relational formulas. However, the IDLE motion primitive is a typical example of a relational property, where the state after is the same as the state before. In our current implementation, we also specify which part of the state each motion primitive modifies. The part of the state not modified is preserved like a frame rule. Prior to performing the compatibility check, we perform a data flow analysis. For each x_i in the choreography, we associate a set of predicates that hold at that point in the choreography. Predicates that only involve unmodified variables are automatically transferred across motion primitives. Another extension related to the IDLE motion primitive is the ability to have variable durations, also known as preemptible motion primitives. If a motion primitive is followed by a message rather than another motion primitive, we can infer its duration from the other concurrent motion primitives. This enables us to extend motion primitives to support variable durations.

5.3 IMPLEMENTATION AND EVALUATION

We have implemented PGCD, along with the various analyses, as an interpreter, library, run-time system, and verifier in PYTHON. The code is publicly available at `https://github.com/MPI-SWS/pgcd`. To execute a program, each robot runs a copy of the interpreter and the run-time system. Each interpreter instance executes its respective processes on the actual robot hardware, directly interacting with the hardware and communicating through the run-time system. The run-time system manages the messages by using an additional server that acts as a central broker. This server keeps track of up-to-date frame shifts between the different robots to ensure consistency across the system. The verifier takes a program, along with its invariant annotations, and specifications for the motion primitives and the environment as input. It then decomposes the program into a list of processes, their connections, and respective specifications. After this decomposition, the verifier performs the checks described in Section 5.2. If any of these checks fail, the verifier reports an error and provides detailed information on the nature of the failure.

5.3.1 Run-time System

We use ROS for the message-passing layer [116]. ROS is a publish-subscribe system where processes advertise topics, publish messages or subscribe on specific topics. Topics can be hierarchically arranged in namespaces. A ROS master node manages the topics. Each process in our run-time system is assigned a namespace based on its identity, and the labels of the messages correspond to topics in the ROS system.

To accurately model our semantics, we have implemented a synchronous rendezvous communication on top of ROS, which by default implements asynchronous message-passing. In our implementation, when a process encounters a **receive** statement, it subscribes to the topics corresponding to the labels occurring in the receive block, and after receiving a message, it unsubscribes from the topics. This allows the sender to query the ROS framework to check the presence of a receiver and to block until a receiver is ready. Therefore, the send operation blocks, accurately implementing our synchronous semantics. The use of ROS lets us reuse a lot of robotics infrastructure available in the ROS ecosystem. We build on ROS's TF2 library [33] to deal with frame shifts. In our implementation, every component periodically publishes its frame shift relative to its children, as frame shifts are dynamic and based on the current state. The receive operation for a process queries TF2 for the frame shift from the sender's frame to the frame of the recipient process and transforms the content of the message appropriately. Motion primitives are hardware specific and each motion primitive is currently implemented directly for the corresponding robot based on its Mperl description. Mperl (and its extension, the sensor configuration synthesis) take care of interfacing with the hardware of each individual robot and provide implementations of the appropriate motion primitives.

5.3.2 Verifier

In our implementation, we use Promela models to check the communication between the different processes in the program. We generate the PROMELA models from the program and its annotations and then analyze them with SPIN. PROMELA (PROcess MEta LAnguage) [56] is a process modeling language used to model concurrent systems. SPIN [55] is a tool that checks PROMELA models for correctness using a model-checking algorithm. This allows us to detect communication errors, such as deadlock or livelock, and ensure that the program satisfies its communication specifications. For the geometric reasoning we use SymPy [95], a symbolic manipulation package which contains a module for 3D Cartesian coordinate systems.[1] In our implementation, it is used to express formulas in their component's frame and to construct frames on top of their parent. When the frame shifts are given, a formula can be automatically translated to a specific coordinate system using symbolic manipulation in SymPy. As the frame shift and motion primitives involve reasoning about non-linear arithmetic, e.g., trigonometric functions arising from rotations, we use the dReal solver [41] for non-linear theories over the real to discharge the verification conditions.

Each motion primitive needs to have a specification in the form of a $(\text{Pre}, \text{Inv}, \text{Post})$ triple. The specifications are written directly as Python functions by extending the appropriate base classes provided by the tool. The specification of a motion primitive describes *frame conditions* by explicitly describing the variables modified (using a **modifies** clause similar to ESC/Java [32]). Additionally, we allow the programmer to separately specify footprints (regions of space used by the motion primitive) for Pre, Inv, and Post. For example, the resource function of an arm consists of a number of cylinders with spherical joints, but the footprint can simply return a half-sphere bounding the arm.

The verifier checks that the names of messages and motion primitives coincide, and the values passed to the motion primitives syntactically match those in the specification. Thus, correct programs, which may perform local computations to provide values to the motion primitives, may be rejected. Verifying programs in the presence of local computations would require implementing a symbolic execution engine. The stronger syntactic check was sufficient for our examples.

5.3.3 Evaluation

We have implemented several examples involving multi-robot coordination in our system. First, we describe our experimental setup, both for the hardware and software. Then, we explain the experiments. Finally, we report on the size of the programs, specifications, and verification time. [2]

---

[1] `http://docs.sympy.org/latest/modules/vector/index.html`

[2] A short video of our experiments can be seen at `https://people.mpi-sws.org/~mpirron/files/All_Experiments.mp4`. Note that it uses older prototype versions of the robots shown in this thesis.

5.3.4 Setup

Our system for testing and evaluation is based on a carrier and the cart-and-arm system. Both cart and carrier consists of two omnidirectional driving platforms that allow for three degrees of freedom (two in translation and one in rotation) when moving on a flat surface. This wheel configuration enables the easy control of all three degrees of freedom and does not require complex planning. The cart as a robot arm mounted on top that is capable of grabbing objects.

The carts and the arm use stepper motors to accurately control their motion. However, the carts do not have any global feedback mechanism to determine their position and instead rely on a technique called *dead reckoning* (e.g., [63]). This involves knowing the initial position of the carts and subsequently updating their position by counting the number of steps the motors make. If slippage is controlled and the maximum torque of the motors is not exceeded, there is relatively little error accumulation as long as the initial position is accurately known. Furthermore, using stepper motors allows us to know the time it takes to execute a given motion primitive by fixing the rate of steps. Each robot is equipped with a Raspberry Pi 3 model B, which is used to run the process and the run-time system. The messaging services are provided by a separate Raspberry Pi, which runs the ROS master node to which all the processes connect. The Raspberry Pi runs Raspbian OS (based on Debian Jessie) with ROS Kinetic Kame, along with the Mperl software, which provides the motion primitive abstractions.

Table 5.1 estimates the implementation and specification effort for motion primitives in terms of lines of code (LOC). Cart and Carrier each have five motion primitives, the arm has eight. The implementation of the motion primitives for the carts require 92 and 97 lines of code, while the implementation for the arm requires 154 lines. Additionally, there are 151 lines of shared code. The specification consists of 174 and 75 LOC for the carts and 221 LOC for the arm. Note that the two carts share much of the same specification, but their motion primitives differ due to their specific hardware configurations.

5.3.5 Experiments

In the following, we provide a summary of four experiments on which we evaluated our tools.

In the first experiment, called *Fetch* (Fig. 5.8), the arm, which is mounted on top of the cart, is tasked with grabbing an object. If the object is not in the vicinity of the arm, the cart has to relocate to a suitable position to allow the arm to reach the object. After successfully grabbing the object, the arm folds back into a transport pose, and the cart moves back to its original position.

Robot	Motion Primitives	Implementation (LoC)	Specification (LoC)
Arm	SetTurntable SetLowerArm SetUpperArm, SetPose, Grab, Grip, Fold, Idle	154	221
Cart, Carrier	Move, Strafe, Rotate, Sweep. Idle	92 + 97	174 + 75
Shared	–	151	–

Table 5.1: Motion Primitives Implementation and Specification

The second experiment, *Handover*, is an extension of Fetch, where an extra cart, i.e., the carrier, is added to the experiment that transports the object. Both carts must meet before the arm takes the object placed on top of the carrier. After the exchange, both carts go back to their initial position (Fig. 5.9).

In the *Twist and Turn* experiment (Fig. 5.10), the carrier initially starts in front of the cart, and the arm takes an object from it. Then, all three robots move simultaneously: the cart rotates in place, the carrier describes a curve around the cart, and the arm moves from one side of the cart to the other. Finally, the arm places the object back on the carrier.

In the last experiment, called *Underpass* (Fig. 5.11), the carrier cart delivers an object to the arm which then picks it up. The carrier cart moves around the arm, passing through an obstacle that is too low to accommodate the object on top. Finally, the arm places the object back onto the carrier on the other side of the obstacle.

The motion primitives used by the carts are explicitly implemented and include moving straight, strafing, rotating, and sweeping. During the underpass experiment, for example, the carrier cart executed a sequence of motion primitives consisting of rotate, move straight, rotate, and strafe to go around the arm. As gripping an object is considered a collision, we exclude the gripper from the arm's footprint and do not model the objects it grips. Obstacles in the environment are modeled as regions in three-dimensional space, and collision detection is performed against these regions.

Table 5.2 shows the size of the programs in the core language of Section 5.1 (summed across for all the robots) as well as the size of the specifications. The program includes the statements for each process and the connections between processes. As part of the program we include the *world* description: the world is a virtual process which is

(a) The cart and arm assembly is tasked to fetch an object

(b) Cart moves to the object's position, arm grabs the object

(c) Arm retracts into its home position, cart returns to initial position

Figure 5.8: Fetch Experiment

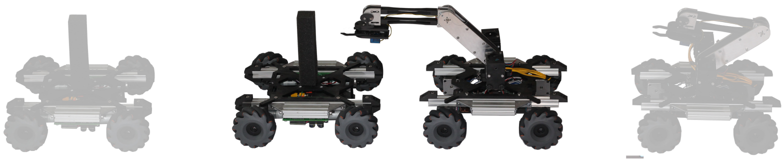
Table 5.2: Programs, Annotations, and Checks

Experiment	Program (LoC)	Annotations (LoC)	#VCs	Time (sec.)
Fetch	35	12	82	16
Handover	29	18	183	86
Twist and Turn	38	18	93	79
Underpass	52	40	393	103

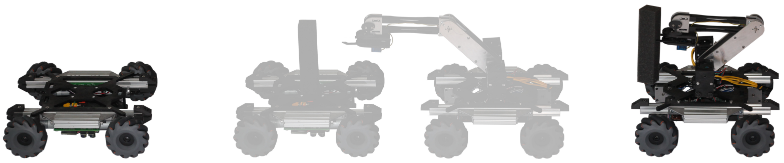
the root of the parent-child attached composition. Additionally, the world contains obstacles used for additional collision checks.

Finally, we show the number of verification conditions (#VCs) generated when checking the motion primitives and the total running time. The communication protocols are simple and can be verified by Spin in less than a second. However, verifying the motion primitives requires more complex reasoning, which dominates the run time. The total number of verification conditions is significant, as ensuring the absence of collision is a quadratic problem in the number of components. Overall, the experiments show that PGCD is expressive enough for complex coordination tasks and, at the same time, the verifier can scale to statically verify concurrency and geometric properties of these tasks.

(a) Like the fetch example, but the object is placed on a carrier system



(b) Cart and carrier meet at a suitable position and handover the object

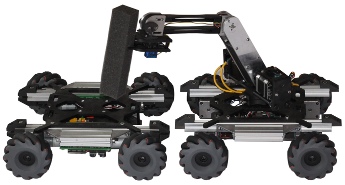


(c) Cart and Carrier return to their initial position

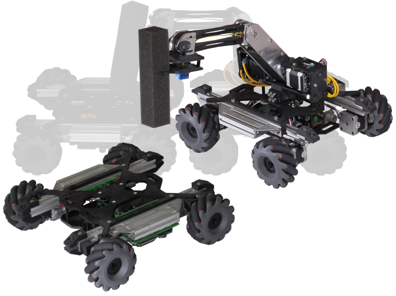
Figure 5.9: Handover Experiment

5.4 CONCLUSION

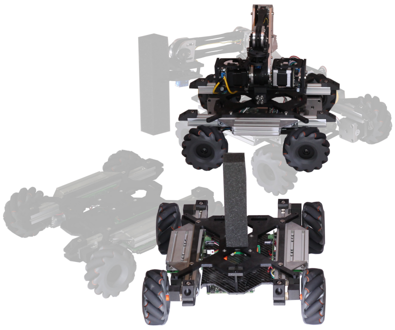
In conclusion, our language and verification system provide a comprehensive solution for specifying, verifying, and executing concurrent, communicating components in a physical and geometric environment. Motion primitives provide a powerful means of specifying and controlling capabilities and behaviors of robots by allowing us to to define a repertoire of fundamental actions that can be composed and orchestrated to accomplish complex tasks. Due to their encapsulation and compositionality, motion primitives enable robots to perform a wide range of actions and adapt to different environments and scenarios. By taking into account relative frames of reference and transforming geometric data appropriately, our system enables precise coordination and interaction among multiple robots. The verifier, a key component of our system, ensures the absence of message passing deadlocks, validates the properties of executed motion primitives, and detects resource conflicts in concurrent execution. Through our evaluation, we have demonstrated the effectiveness of our language and run time in handling complex coordination tasks, while providing the necessary verification. Overall, our approach offers a powerful framework for developing and validating robot systems, facilitating the creation of robust and reliable robotic applications.



(a) Carrier (with object) and the cart and arm assembly face each other, the arm grabs the object

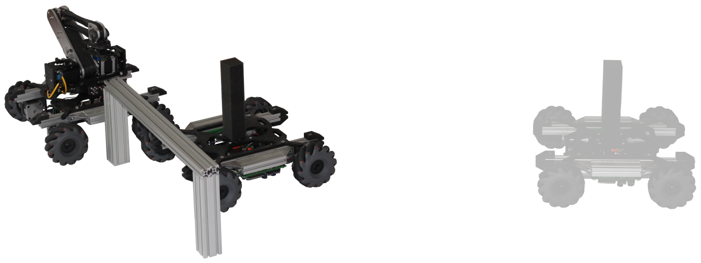


(b) All three systems move: the carrier moves and rotates counterclockwise, the cart rotates clockwise, the arm follows the movement of the carrier

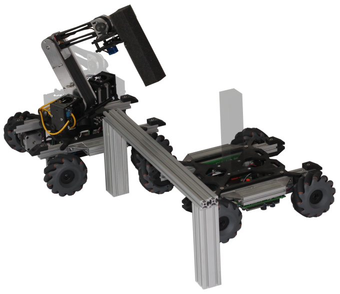


(c) After moving, the arm returns the object to the carrier

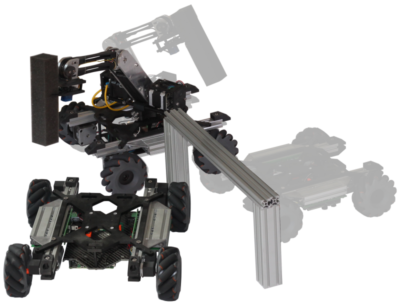
Figure 5.10: Twist and Turn Experiment



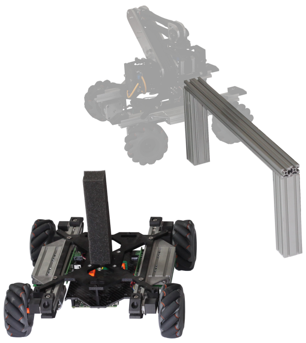
(a) The carrier with object is too high to clear the underpass



(b) The arm removes the object from the carrier, which is now able to drive through the underpass



(c) After clearing the obstacle, the object is put back



(d) The carrier continues on its path

Figure 5.11: Underpass Experiment

6 RELATED WORK

In recent years, rapid prototyping tools and personal manufacturing tools became readily available, leading not only to a democratization of manufacturing, but also of innovation [42]. This trend has even been compared to the way books have democratized information and the way the internet has democratized information [48]. Machines like 3D printers or mills allow production even at home of mechanical and electronical parts, and microcontrollers, along with a vast array of easily obtainable driver boards, sensoric and actuatoric elements lower the bar of entry intro developing and manufacturing robotics significantly. This development has lead, and indeed, is still leading a new generation of interactive tools and techniques that help non-expert users to create, simulate, operate and optimize robotic systems. Areas which have to be taken into account encompass the mechanical structure, along with the appropriate actuation, which directly affects kinematics and – by extension – dynamics of the robot, the sensing infrastructure, which deals with perceiving and estimating the environment and the state of the system, and motion planning. In light of increasingly complex tasks, interaction and cooperation of multiple robotic systems have also the be accounted for.

In the following, we take a look at various tools and techniques in this context and see, how the work presented in this thesis is situated.

6.1 PROTOTYPING TOOLS

Similar to Mperl are the robot compiler by Mehta et al. [89, 90] and the work by Ha et al. [52].

The robot compiler [89, 90] uses a high level structural description, along with a library of components, to generate manufacturable outputs. While targeting a much wider class of robots, only method stubs to communicate with the hardware are generated, while the user has to implement the actual control logic. The robot compiler has been extended [92] to use reactive synthesis to generate a finite state machine software controller. This allows the robot compiler to generate complex temporal behaviors, but it does not support complex calculations like inverse kinematics or finding the singularities of the system. On the contrary, the work presented in this thesis targets only robotic manipulators, but it generates higher level software functionalities.

Mehta et al. work on robot creation from functional specification [91] is the closest to our work, as presented in chapter 3. The authors synthesize both a robot and its controller from a structured english specification of the robot's task. The Structured English description is turned into Linear Temporal Logic (LTL); atomic propositions are actions of the robot. On the controller side, the authors use reactive synthesis to find a controller, and on the physical side, the authors start with a common platform and add elements according to the actions: moving requires wheels or legs, grabbing an object requires a gripper, etc. Their method relies on a database mapping words to components for both, actuators and sensors.

Ha et al. [52] use a library of components consisting of joints and links which, together with a user specified input trajectory, generate robotic systems following the input trajectory. Rather than trying to design a structure that achieves a given motion, their tool generates the software infrastructure to achieve any motion allowed by the structure.

Giusti et al. [44, 45] developed a framework for the generation of modular manipulators which starts from a human demonstration of the task to be accomplished. From this demonstration, a manipulator is generated by searching over all possible robotic assemblies. Their search method optimizes the structure, based on kinematic constraints, and the software controller, based on the speed of accomplishing the task.

Campos et al. [18] work on designing modular manipulators to accomplish a given task. Given a task description with locations to reach and regions to avoid, their algorithm searches the space of physical designs to produce a robot which accomplishes the task. At the same time they also produce the control configuration to move the manipulator. Their controller is based on local feedback similar to [111], and therefore, could also benefit from Mperl to handle a wider range of tasks' specification.

Related areas we have to consider are modular robotics, origami-style robots, and domain specific languages (DSL).

MODULAR ROBOTICS Modular robotics [142] – or modular (self-) reconfigurable robotics – share similar goals in automating the design and programming of modular robots. These systems are composed of multiple independent, but often identical modules; e.g., a 2-modular system contains only two types of modules, regardless of the number of modules used in the system. Common to these system is their ability to change their shape (variable morphology) and functionality by rearranging their modules [68], or even by changing the modules themselves [121]. The versatility and usefulness stem from the combination of many modules, and less so from individual modules. For example, the PolyBot **??** consists of two modules with at most one degree of freedom. By combining a sufficient amount of nodes, rolling track locomotion and sinusoidal locomotion can be achieved by this system.

Automating the design and the programming of modular robots, along with selecting the most appropriate configuration is an ongoing problem [61, 143]. Recent works in the area of modular robotics [44, 45, 61, 108] have a similar goal of automating the design and programming of modular robots by integrating modular and reusable control manipulators and software, and possibly a task-based planning software.

For example, the work of Jing et al. [61] proposes a system for modular self reconfiguring robots, consisting of a high level mission planner, a design library with configurations and behaviors, and a design and simulation tool. Their robots are made of many copies of a single universal module; different behaviors are achieved through dynamic reconfiguration. Given a specification, their motion planner searches the space of actions and configurations for suitable configurations.

While inchworm like locomotion (e.g. [69]) is possible in Mperl, the work presented in here is located on a lower level. It focuses on composing electromechanical components to build robotic manipulators and synthesizing the sensing infrastructure and is aimed mainly at non-expert users, enabling them to design and manufacture a robotic system with minimal domain specific knowledge.

ORIGAMI STYLE ROBOTS Origami robots are a different approach of rapid prototyping of robots. These origami parts are designed in a two dimensional plane, manufactured from flat sheets, and then folded in to their three dimensional, final form. Typical examples are, for example, deformable wheels [75], or foldable robot arms [65]. It is not uncommon of origami robots to make heavy use of shape memory alloys, which allow parts or even complete robots to be self-folding (self-deploying) [30, 135], and to some extend, even actuation [10, 115]. Folded structures exhibit many material properties that robotics that advantage of, for example nonlinear stiffness, multistability or impact absorption [78]. Individual, origami style structures can be combined with more rigid actuated structures [124]. Interactive robogami [123] provides an end-to-end toolchain for manufacturing and simulating foldable ground robots based on a library of pre-defined parts.

These larger origami-style structures lend themselves well for integration into the work presented in this thesis. These parts share similar properties to 3D-printed parts (e.g., the inherent flexibility due to the manufacturing process), and sensors can be embedded in a similar fashion like the deflection sensor into the 3D-printed robot arm. Smaller origami-styled parts or robots are not the target of this work, but these devices often share similar goals like rapid prototyping, reconfigurability, and also, to some extend, compositionality [139].

DOMAIN SPECIFIC LANGUAGES In contrast to general purpose programming languages, which have no specialised feature pertaining a specifica are, domain specific languages are specialised to specific areas and provide corresponding (language) constructs, concepts and notations, but typically no end-to-end capabilites as presented in this work. In their area, however, they provide abstractions and tools which more closely associate the programming, and, to some extend, reasoning, to the intended domain. For a more detailed overview, we refer to the Robotics DSL Zoo [103].

Several domain specific languages for rigid body kinematics and dynamics exists, i.a., [36, 37, 73], usually providing support for geometric or dynamical relationsips, including position, orientation, velocities and accelerations by standardizing terminology and notation and providing semantic checks on these representations.

Similar in goal, Reckhaus et al. [118] presents a graphical programming environment that eases the develops robot control programs, focusing on parallel sequences of actions (similar to motion primitives). It does not, however, generate control code of the motion primitives, and is limited to a single robot, not considering multi robot interaction or communication.

6.2 PHYSICS INTEGRATION

Dimensional analysis, and the concept of dimensional analysis was first proposed in 1822 by Fourier [35] and refined 1877 by Rayleigh [117] and formalized by Buckingham [15]. Dimensional analysis has long been used to analyse and develop mathematical models describing physical phenomena, e.g., in drip irrigation systems [144], weirs [12], and other non-linear systems [109]. Recently, dimensional analysis has even been applied to machine learning [107].

Integrating physical dimensions, units, and dimensional analysis into type systems have a long history, e.g., the AMPERE programming language [8], the strongly typed programming language presented by Kennedy [64], the fully static dimensional analysis for C++ [133], which verifies dimensional integrity and handles unit conversions, or the work presented in [76], which applies dimensional analysis to statistical modelling.

Often, these approaches use dimensional analysis to ensure dimensional homogeneity of equations, thus making sure that equations are physically well types and that quantities are used in a dimensionally correct way.

Type systems and type information in the context of computer science is not only used to ensure the well typedness of expression, but it can also be used for code completion and code synthesis. Examples encompass the work in [86], which synthesizes code fragments given input and output types, or [51], which generates a list of expressions with a specified type. Richer typing systems, e.g., inductive data types and refinement types [114], allow even for synthesizing complete methods like sorting algorithms.

In combining both aspects, Newton [136] is the only work we know that uses physical information in the type system for synthesis. Taking this approach further one step further, we apply dimensional analysis to robotics to synthesize sensor configurations by providing a goal directed search over the system's components. Goals directly stem from the controller and its unmeasurable parts.

6.3 ROBOT INTERACTION

Even more so than the development and manufacturing of individual robots, robotic applications are typically programmed in low level imperative programming languages, that provide little to no abstractions tailored to modelling the interactions between robotic systems, or even reasoning about them. While programming robotics applications, the programmer still has to deal with dynamic controllers and their influence on the physical state, geometric constraints imposed upon the components, as well as concurrency and synchronization. Writing these programs draws from a multitude of domain specific knowledge, and the prevailing programming languages such as C++ or Python, or even more specialised, but still low level robotics languages (e.g., Rapid [49], KRL [99]) provide hardly any support in navigating these complexities.

Many computational approaches have been developed for concurrent and real-time communication and computation, but few cover the combination of communication, complex geometry, and dynamic control of physical state. Modeling paradigms for hybrid systems such as hybrid automata and its extensions [4, 5, 82] allow expressive dynamics, but little support for compositional programming and reasoning about communication. Timed extensions to process algebras [9, 11, 97], Petri nets [93, 100, 127], or other concurrency models allow the mixing of message passing and time, but do not combine geometric reasoning and resource accounting. For the most part, analysis algorithms for these models are intractable. In principle, logics such as differential dynamic logic [113] and hybrid process algebras [11, 17, 74, 120] enable reasoning about arbitrarily complex concurrent and hybrid programs, but their primary goal is the interactive verification of models.

Cardelli and Gardner [19] define a process algebra for geometric interaction, which combines communication and frame shifting, but they do not consider dynamic flows of geometric objects over time, which is crucial in a robotics context. Moreover, the objective in their process algebra is an abstraction theorem, and not reasoning about programs.

Another approach is spatial logics, that have also been explored from a topological perspective [20]. Here, modal operators describe neighborhood relations. While such a framework can express and check properties about arrangements, it cannot deal with temporal evolutions.

From the perspective of DSLs for distributed robotic systems, recent projects like StarL [80], Drona [24], and Koord [43] integrate a DSL, specification, and verification support in the same framework. StarL programs are composed of coordination and motion primitives which have been specified in an interactive theorem prover which can be used to verify the programs. Drona is built on top of a state-machine based programming language, integrates a motion planner, and uses a model-checker to test the programs. Koord is event based where events trigger global actions which perform computations and call motion primitives for different robots. The verification uses a bounded model checker or user provided inductive invariants. None of these systems integrate programming and reasoning with concurrency, dynamics, and geometry.

To our knowledge, the work presented in Chapter 5 is the first to simultaneously reason about the interaction between concurrency, geometry, and dynamics, and it lays the the foundation for correct design and static verification of complex robotic applications interacting dynamically in geometric space.

7 FUTURE WORK

This dissertation explored controller synthesis for reconfigurable robotic systems, encompassing both serial and parallel configurations. This broadened the scope to include diverse manipulators, ranging from basic SCARA systems to intricate ones like cable and delta robots. In particular, we laid our emphasis on the co-design of hardware and software, putting both on equal terms. Controllers were synthesized through two approaches: an abstract description or by constructing the hardware and refining it based on the physical structure. Furthermore, a combination of physical and virtual components was permitted, offering valuable testing capabilities for robotic systems.

Using motion primitives provided a powerful means of specifying, controlling and encapsulating basic capabilities of the robotic systems. Motion primitives can be composed and orchestrated, enabling robots to perform a wide range of actions and adapt to different environments and scenarios.

In our co-design approach, we extended our focus to include sensor selection and placement in form of the sensor configuration synthesis. By carefully configuring and placing sensors on robots, we enable these systems to perceive and understand their surroundings more effectively. In addition, by providing physical dimensions and units for each component, physical invariants that describe the functional dependencies between specific parts of the system could be generated. These invariants were used to describe certain behaviours of the system, and they found also use the the context of state estimation and filtering. Additionally, by leveraging invariants, the system was able to discover alternative approaches for coping with erroneous parts, enabling the potential continuation of system operation. This further contributed to the generation of motion primitives, which represented encapsulated actions that our robotic systems were capable of executing.

With the ability to describe motion primitives in hand, we shifted our focus to a higher level of robotics, namely the interactions between multiple robotic systems. We examined the interactions among collaborative robotic systems that aimed to accomplish a shared objective, such as a handover between two robotic systems. Robotic programs were modeled as processes with associated motion primitives, thereby addressing the concurrent nature of multiple robotic systems working in unison towards a common goal. Each program was constructed from processes that represented a logical segment of a robotic assembly.

The structure of the system was established through the attachment of processes, which interconnected the physical state of two components and facilitated the necessary coordinate transformations. The PGCD processes had the capability to send and receive messages and execute motion primitives, albeit limited to sequential code execution. To ensure the correctness of communication and synchronization, such as the prevention of deadlocks and the concurrent executability of motion primitives, we developed a verifier specifically designed for these PGCD programs. This verification process involved assume-guarantee reasoning and the separation of geometric resources.

Throughout the dissertation, each aspect, including controller synthesis, concurrent systems, and verification, was accompanied by physical implementations and thorough evaluations to demonstrate the effectiveness of our approach. These implementations provided tangible demonstrations of the proposed methodologies and allowed for comprehensive assessments of their performance and capabilities. Our work is distinctly situated at the intersection of physics and verification, combining principles from both domains. This unique approach opens up possibilities for further exploration and development.

In the following sections, we outline two potential directions in which our work can be extended and expanded upon.

7.1 PROGRESSING SENSOR INTEGRATION AND MATERIAL SCIENCE

Multiple materials are used in the creation of robotic systems, such as aluminum and plastics, each possessing distinct characteristics like strength, hardness, or elasticity. Material properties can vary over time and are influenced by the environment in which the robot operates, directly impacting its functionality and safety. For instance, consider a structural part, such as a beam, within a robotic system. If the beam is made from brittle material, it will fail more rapidly compared to ductile materials that can withstand a higher degree of plastic deformation. Moreover, ductility and brittleness also depend on environmental factors like temperature (e.g., glass transition temperature) or humidity (e.g., decreasing surface ductility). Thus, ductile materials are potentially preferable in this case, as they allow the system to return to a safe state, e.g., by controlled unloading. On the contrary, other parts in the system may benefit from materials that exhibit low plastic deformation. In an ideal scenario, the robotic system should possess awareness of the material properties associated with its components. This awareness can be achieved through direct measurements, such as embedding sensors directly into the materials, or indirect measurements, which involve utilizing invariants or deducing information from other measurements. By leveraging these measurements, the robotic system can make informed statements or even assertions about the present and future state of the system as well as its individual parts.

To tie in with the modular nature of Mperl, ideally the system is able to figure out certain key properties of a component, even if the material is unspecified or unknown, and make a projection of the expected behavior and failure mode. In some preliminary tests, we used a robot running Mperl, but modified material properties of the robot, e.g., exchanged an arm made of PLA for one made of PETG. In these first test, we were able to automatically derive the flexural modulus and thus to relate observable properties (e.g. load induced deflection when lifting known weights) to specific materials.

7.2 VERIFICATION OF PHYSICAL PROPERTIES

The verification process in PGCD primarily focuses on the higher-level composition of motion primitives, overlooking the lower-level code implementation. However, in line with the earlier discussion on physics integration, Mperl, together with its physical invariants, presents an opportunity to not only semantically verify program code but also physically verify robotic systems. This expanded approach to verification encompasses various aspects, such as incorporating support for quantities and dimensions, accommodating uncertainties in variables (stochastic variables), and checking invariants and assertions that may be influenced by system configuration or environmental factors. Additionally, with the ability to facilitate ad-hoc modifications and reconfigurations of robotic systems, properties may dynamically change during runtime, thereby necessitating verification to become a continuous process throughout operation.

The adoption of motion primitives further reinforces this approach, as motion primitives demonstrate varying behaviors contingent upon their configurations. For instance, consider the scenario of the carrier operating independently, where minimal constraints are imposed on its motion. In contrast, when the carrier is coupled, e.g., with a trailer and possesses a significant mass or high center of gravity, restrictions on its degree of freedom are imposed, as well as boundaries for acceleration and deceleration are set, and limitations on its turning radius. A verification process ideally should account for these variations, to ensure the system's compliance.

Both, the verification of low level robotics and the inclusion of material science aspects lead to richer and more varied robotic systems. As an example, consider an autonomous research drone used in a remote area, where human interference is limited or not possible and where the impact on the environment has to be minimized. We could easily imagine the drone to be biodegradable, with assertions that the propulsion system can safely reach the intended position then disintegrate, and the research appliance could report its measurements with guarantees on its precision during its intended operation time.

Verifying these programs is challenging and adds a lot of complexity to already existing verification approaches, but results in systems that are verified not only on a code level or motion level, but on a physical level, thus making them far safer and more capable.

APPENDIX

In this chapter, we describe the technical implementation in more detail. In particular, we describe how state estimation and filtering is practically integrated into Mperl and the sensor configuration synthesis and how certain types of failures can be detected.

A IMPLEMENTATION DETAILS OF STATE ESTIMATION

In this section, we tie the findings of the previous chapters together and show, how Mperl and the sensor configuration synthesis supplement state estimation, and, in turn, how the state estimation provides insight in potentially faulty sensors.

Fig. summarizes our approach. From Mperl, we derive the model necessary for the state estimation, and by means of the sensor configuraiton synthesis, unkown observation functions are derived, if not provided otherwise. Estimated states and available observations are analyzed by means of the Kullback-Leibler Divergence, while not without its shortcomings, provides an intuitive measurement. If the measured state and the predicted deviate, this indicates a potentially faulty quantitiy, e.g., a sensor or an invariant. To continue using the robotic system, an alternative way of determining the faulty quantity has to be found. This can be by using an alternative invariant, or by deriving a new one by means of the sensor configuration synthesis.

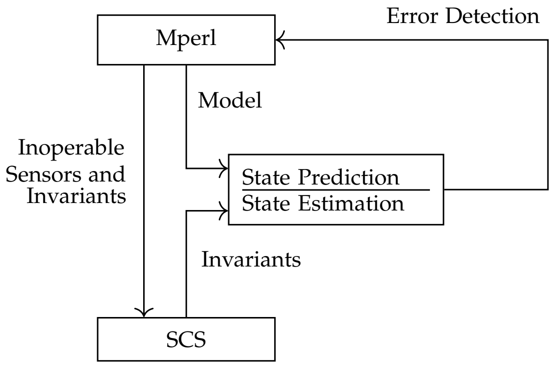


Figure .1: State estimation integrates the model from Mperl and invariants from the SCS. Potentially faulty quantities are identified by analyzing predicted and measured states, and the SCS is used to derived alternative measurements.

A.1 State Estimation

In section 2.6, we introduced dynamical systems, which model the interaction between the environment and a controller that generates control inputs u depending on the perceived state y. This is a generalization; in reality, typically the controller tries to estimate a state $\hat{x}$ from y to get a more accurate representation of the current state, and the control input u is then bases on $\hat{x}$ instead of y. Multiple reasons exist why x may differ from y; often, inherent noisy and perhaps unstable sensor readings are the cause, as are missing or intermittend observations, which can occur if information is only shared between mobile robots if they are in close proximity to each other. These observations or measurements have to be preprocessed before they can be put to use. It is often assumed that the initial state X_0 is known, and every subsequent state is estimated. Kalman filters have long been used as state estimators; for linear dynamical systems, the kalman filter is an optimal estimator [62], and for non-linear systems, the extended Kalman filter [87, 128] is widely used, but it may no longer be optimal.

The standard extended Kalman filter is given by

$$\hat{x}_{t|t-1} = \phi(x_{t-1|t-1}, u_t) \tag{.1}$$

$$P_{t|t-1} = \Phi_t P_{t-1|t-1} \Phi_t^T + Q_t \tag{.2}$$

$$K_t = P_{t|t-1} H_t^T (H_t P_{t|t-1} H_t^T + R_t)^{-1} \tag{.3}$$

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t (z_t - h(\hat{x}_{t|t-1})) \tag{.4}$$

$$P_{t|t} = (I - K_t H_t) P_{t|t-1} \tag{.5}$$

where Φ is the Jacobian of $\phi(x_{t-1|t-1}, u_t)$, H_t the Jacobian of $h(x_{t|t-1})$, and Q_t and R_t are the process and measurement uncertainty. We use t as the discrete time index, $\hat{x}_{t|t-1}$ represents the predicted state for time index t, given the data up to and including $t-1$; similar, $P_{t|t-1}$ denotes the predicted estimate uncertainty and K_t the Kalman gain at time t, possibly based on the data including $t-1$.

The kalman filter uses $\phi : Y \times U \mapsto X$ to reconstruct $\hat{x}$ from the perceived state y (i.e., the *observations*). ϕ differs from f, the actual state transition equation of the environment; contrary to f, ϕ is often an idealization of the actual system.

Using Mperl, we gain a comprehensive understanding of the system's behavior and trajectory. The model, ϕ, represents the rigid body structure of the robotic system based on its input description.

For a robotic system with feedback from node v, meaning there is a sensor measurement associated with the component, we calculate the transition matrix from the anchor point to v. If a component, such as a robot arm, is not attached directly to an anchor point, the resulting model may have unbound variables, which become bound once the component is attached to either another component, or to anchor points.

Any alterations to the mechanical structure are thus reflected in ϕ, which maintains the modularity and compositionality of Mperl. In the case of parallel robots, multiple transition matrices exist due to the partitioning into chains, each with potential additional sensors. Sensor fusion is then used to achieve more accurate estimates.

The observation function, h, describes the functional relationship between a measurement and ϕ output, whether it's direct or indirect. If there is no known function mapping measurement space to state space, sensor configuration synthesis is employed to derive the function.

A.2 Error Detection

We implement an error detection by means of measuring the relative entropy between the observed values of the sensors and the corresponding values of the Mperl model. For both we keep a history of values. The compositionality of Mperl's robotic systems lend itself well to an error detection which stays model free and does not assume any prior knowledge of the underlying distributions. Under the assumption, that errors occur rarely, we can use relative entropy (*Kullback Leibler Divergence*) to compare sequences of sensor readings and their (theoretical) counterparts. If both sequences have little difference, the relative entropy between these two are is close to 0. To estimate the distributions, we bin values using $\lceil\sqrt{k}\rceil$ bins, where k can be considered the size of a sliding window of values. We write $\lfloor Y \rfloor$ for the bins created out of Y.

Denote by X a discrete random variable, e.g. a sensor reading which is updated at specific time intervals, and $\mathtt{Prob}(P)$ the probability mass function over P, then the (Shannon) entropy [125] is given by

$$SH(X) = -\sum_{x \in X} \mathtt{Prob}(x) \log \mathtt{Prob}(x) \tag{.6}$$

We can then define the relative entropy (i.e. the Kullback-Leibler divergence) [23, 71] between two probability distributions P, Q

$$D(P, Q) = \sum_{i=1}^{n} P(x_i) \log(P(x_i)/Q(x_i)) \tag{.7}$$

$D(P, Q)$ is not a metric, i.e. $D_K L(P, Q) \neq D_K L(Q, P)$; it is a measure of information entropy of one distribution relative to a reference distribution. The reference distribution comes from the theoretical values of Mperl, which generates these values from the abstract description (and the corresponding synthesized controller), along with the input trajectory. We measure the difference in entropy between the measurements and the predicted state, and if observations and estimated state are close together, we observe low entropy between them. In the case of high entropy, anomal values are expected, which hints at a potential sensor malfunctioning. From the history, we can compute the Krichevsky-Trofimov estimator [70] of the observations; for each $z \in \mathrm{hist} Z$ (Z is in $\lceil \sqrt{k} \rceil$ bins), we calculate

$$\hat{Z}_x(z) = \frac{|\{i \mid t-k < i \leqslant t \wedge h_i(x_i) \in z\}| + 1/\lceil \sqrt{k} \rceil}{\lceil \sqrt{k} \rceil + 1} \tag{.8}$$

$$\hat{Z}_o(z) = \frac{|\{i \mid t-k < i \leqslant t \wedge z_i \in z\}| + 1/\lceil \sqrt{k} \rceil}{\lceil \sqrt{k} \rceil + 1} \tag{.9}$$

We look at the given observations ($\hat{Z}_o$) and the observations matching the estimated states ($\hat{Z}_x$). From that, we compute the Kullback-Leibler divergence of the two distributions by

$$D_t(\hat{Z}_x, \hat{Z}_o) = \sum_{z \in \lfloor Z \rfloor} \hat{Z}_x(z) \log(\hat{Z}_x(z)/\hat{Z}_o(z)) \tag{.10}$$

If both distributions $\hat{Z}_x$ and $\hat{Z}_o$ are the same, the Kullback-Leibler Divergence D_t is zero, and if both distributions are similar, but not the same (e.g., two different sensor measuring the same quantity), the divergence is constant. For each new measurement, we calculate

$$g_{t-1} = D_{t-1}(\hat{x}_{t-1|t-1}\ , \ldots\ o_{t-1}) \tag{.11}$$

$$g_t = \frac{1}{|O_t|} \sum_{o \in O_t} D_t(\ldots\ \hat{x}_{t-1|t-1}\ \hat{x}_{t|t-1}\ , \ldots\ o_{t-1} o) \tag{.12}$$

and calculate the gradient

$$g = g_t / g_{t-1} \tag{.13}$$

The gradient g is expected to be close to 1 during normal operation.

TYPES OF ERRORS While a complete fault diagnostic is outside of the scope of this dissertation, Eq. .13 holds some interesting properties which can be used to narrow down the type of error.

If $1 < g_t/g_{t-1} \leqslant \Delta$, i.e. the entropy is continuously growing, but not suddenly spiking, a possible sensor drift is detected. This means, that the measured value from the sensor isslowly deviating from the measured value. The measurement noise, instead of being distributed along $\mathcal{N}(0, R)$ for some R, it comes from a distribution $\mathcal{N}(d(t), R)$ where $d(t)$ is a monotonic function. This can occur over time, if a sensor ages, or due to environmental factors, e.g. a sensor which produces waste heat while measuring temperature sensitive quantities. As the sensor warms during use, the readings drift.

Large spikes, i.e. $\Delta < g_t/g_{t-1}$, indicate external events not accounted for by the system design which influence its dynamic in a sudden way, e.g. a crash, or the robot bumping into an object which causes it to deviate from its path. Depending on the severity of the external event, the robot may be able to resume operation and returns to its original trajectory.

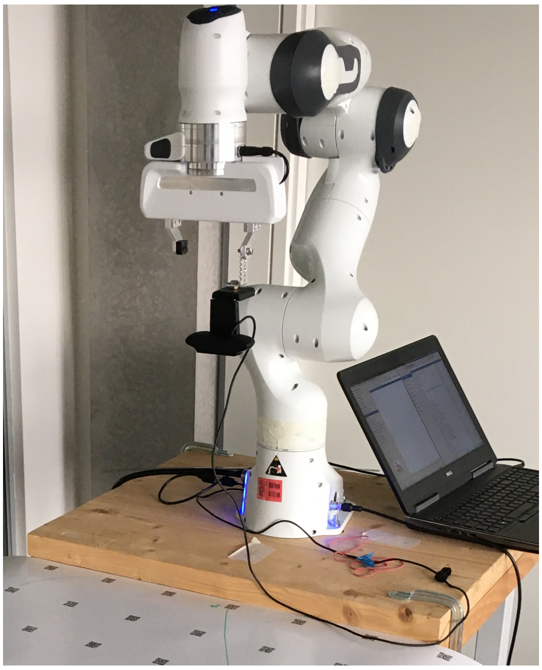


Figure .2: Franka Emika Panda robot arm with 7 DoF used in the evaluation of the error detection

A.3 Limitations

Using entropy in this way is not without its problems; in particular, bounding Δ is often more an engineering problem and depends on the robotic system and the task the system should be solving, and there exists a well-known problem with the Kullback-Leibler divergence that can occur if $\hat{Z}_x(z)$ approaches 0. Due to numerical issues, $\hat{Z}_o(z)$ can become very large, thus rendering the Kullback-Leibler divergence unsuitable in these cases. This can happen especially when learning continuous distributions from observations, e.g. when using Kalman filters, and the distribution of $\hat{Z}_o(z)$ is dominated by these anomalies. If anomalies occur seldom, $\hat{Z}_o(z)$ is fully characterized by the uncertainty around the expected measurement with no empty bins. This is a reasonable assumption in our case, as the reference trajectory ties observations to the set value and thus, anomalies get detected early on.

A.4 Evaluation

To give a concrete example, we run Mperl on a commercial robotic manipulator (Fig. .2) with seven degrees of freedom.

It is equipped with 14bit position encoders, its pose repeatability is $\pm 0.1mm$ and it can follow a path within $\pm 1.25mm$. The task of the robot arm is to move from its home position P_H to position P_1, pick up an object, place it at P_2, and return to P_H (Fig. .3). Between P_H and P_1,

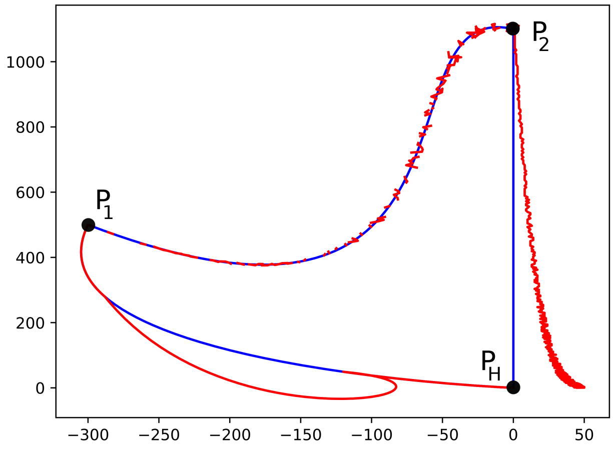
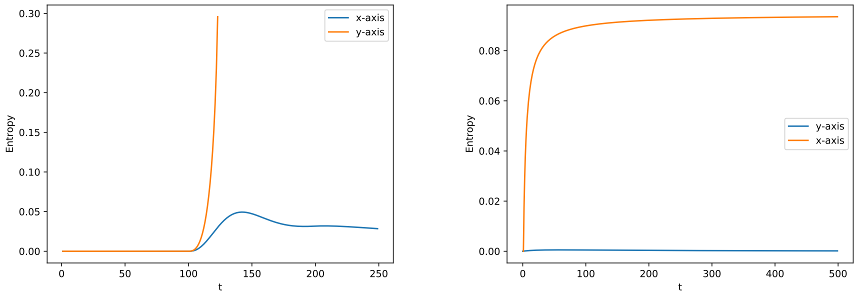


Figure .3: Input trajectory to test detection of impact and sensor drift



(a) During impact, $P_H - P_1$

(b) During drift, $P_2 - P_H$

Figure .4: Entropy measurements (Eq. .13) while traversing the trajectory from Fig. .3

the robot arm is physically subjected to a bump, which throws it off its position; the robot is able to resume operation afterwards. During its return from P_2 to P_H, we introduce drift to the position encoders. Both, the impact and the sensor drift, is successfully detected.

Figure .4 shows the entropy measurement during these phases.

B TECHNICAL IMPLEMENTATION DETAILS OF ROBOTIC SYSTEMS

Components in Mperl that actuate or sense are equipped with *component controllers* that provide an interface between high level commands from the main controller and the low level interaction with the attached electro-mechanical components.

The main controller receives as input the Mperl model, from which it creates the graph, and, subsequently, the kinematics and dynamics. It relays the relevant information as input to the component controllers (e.g., the system model ϕ_i for node v_i for use in the state estimation).

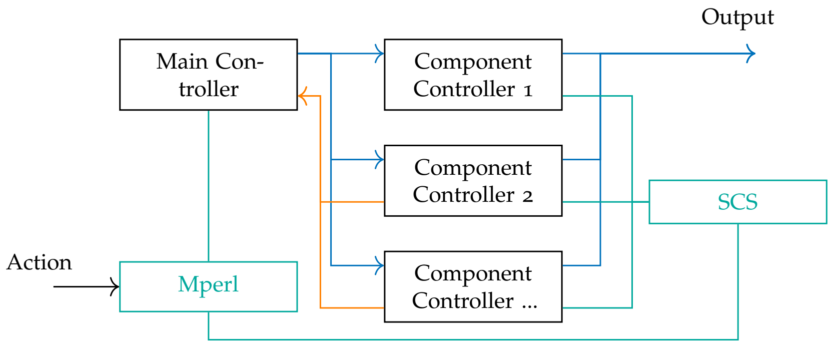


Figure .5: Schematic view of the general control scheme. Black denotes an input, e.g., a user provided trajectory; green depicts links between the controllers and the synthesis architecture (Mperl and Sensor Configuration Synthesis, SCS). Orange denotes feedback from the component controllers to the main controller. Blue colored lines indicate communication between controllers.

Based on the action that is to be performed, it calculates the target configuration of each node, and sends a high level command via UART or SPI, e.g. the angle in radians to revolute actuators, or displacement in mm to linear actuators. In turn, the component controllers translate these commands into signals which are fed into the electro-mechanical component, e.g., a PWM signal in case of a servo, or pulses in the case of a stepper motor. If a sensor is present, it can take two forms. Firstly, it can provide local feedback only. For example, in the case of a stepper motor, a hall effect sensor may only interface locally with the component controller to ensure that no steps are missed. Secondly, it can provide global feedback, which is fed back into the main controller. The measurements that are taken typically need to be translated from there measured value (often a voltage, or in the case of a quadrature encoder, two square waves) to a high level observation that can be used in the main controller (e.g., the deflection in mm, or the absolute position in radians).

In case of a sensor only component, after filtering the measurements and translating them into observations which match the expected high level type, these observations are always fed back into the main controller.

Multiple sensors can be embedded into one block; e.g., in the aforementioned figure, we have a local feedback loop, which feeds its data back to the local controller, which corrects the output locally, without interaction wiht the main controller. A second feedback loop is provided which relays its sensor readings directly back to the main controller.

Figure . .6 provides a view of the controller and the relevant links to Mperl and the sensor configuration synthesis. The motor driver is optional, but many electro-mechanical components require additional circuitry to be useful. Their complexity can be as simple as switching on or off power, e.g., in the case of a solenoid used as a simple end effector, or they can be highly integrated and provide diagnostics and additional sensoric [131]. The component controllers ensure that these complexities are shielded from the main controller, and by extension, from the non-expert user, but are available if so desired.

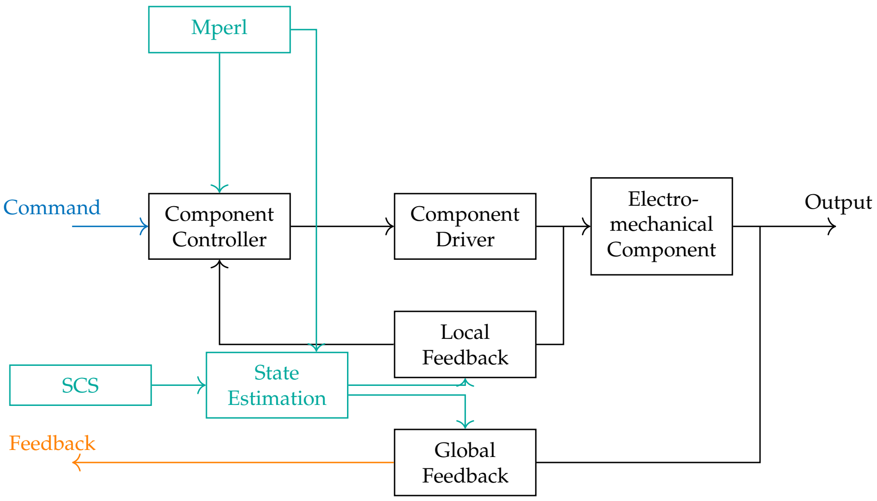


Figure .6: Schematic view of an individual component. The component controller receives high level commands from the main controller, and translates them so that the electro mechanical component, or an attached component driver, can be driven. Blue lines depict the interactions with Mperl, SCS and the state estimation. Gobal feedback is provided to the main controller, while local feedback is only relayed to the local controller.

To provide a more specific example, consider the example SCARA system. On the technical side, the system consists of one main controller, and two local controllers. The local controllers control the motion of the shoulder and the elbow. The system can be equipped with two different motors: stepper motors (standard Nema17), and brushed DC motors.

Depicted in Fig. .7 is the configuration with two stepper motor. In this case, the sensors used for each local controller are hall effect sensors (Allegro A1324), which provide feedback on the angular rotation of the respective axis', and a current sensor (via TMC2209) which is used to solely detect if the steppers are stalling. In this case, a continuous monitoring of the current consumption is not possible. The motor itself is driven by an TMC2209 driver; thus, the component driver consists of the TMC2209 and the A1324 ICs, along with an Raspberry Pi Pico, which assumes the role of the local controller. In lieu of stepper motors, brushed DC motors can be used. They are equipped with quadrature encoders, that are used get feedback on the actual position, and they can be continuously monitored for their current consumption (IN219). Here, the component driver consists of the A1324, the IN219 and the H-Bridge for the motor. Regardless

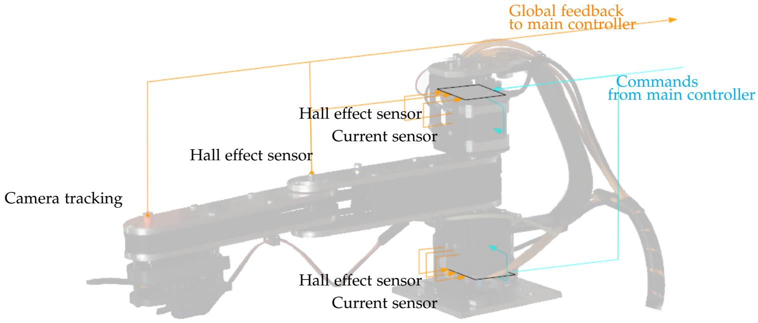


Figure .7: Sensors and electro-mechanical components of the SCARA system. Depicted are the types of sensor used in the system, along with the local and global feedback (orange) and the input (blue) from the main controller. Camera tracking is global feedback only, the hall effect sensors on the joint axis' are both, local and global. The current sensors and the hall effect sensor of the upper motor is local feedback only.

of the motor choise, in case of the elbow joint, a belt transmission is used to offset the motor, and by that, to allow a lighter, more nimble arm. To make sure that the input rotation is correctly transferred, a hall effect sensor monitors the (remote) joint axis. The upper motor, which drives the elbow joint, has an additional hall effect (or quadrature encoder, resp.). This sensor is local feedback only and is used to check if the input shaft rotation corresponds to the output shaft revolution. In addition, a global feedback only sensor is available in the form of a camera which tracks the orange marker at the end of robot arm. The feedback from this sensor fed back directly into the main controller. The feedback of the hall effect sensors on the elbow axis and on the shoulder axis are also reported to the main controller; each sensor (with the exception of the camera) provides local feedback to the component controllers. The main controller is a Raspberry Pi [110] which runs the Mperl software. All communication is done via SPI and UART.

On the controlling side, we provide Mperl with an input description of the system. For each local controller, we calculate the partial model from ${}^{\nu}\mathcal{A}$, where ν is the node in the graph at which that controller's component is located. This partial model is used in the state estimator; the observation functions that establish the relationship between the output of the hall effect sensor and the system model are user provided, and the the current sensor (in the case of the brushed dc motors) are integrated via the sensor configuration synthesis. For uncertainties, which are not known or are not user provided, an automatic derivation is done by means of autocovariance least squares [1, 105]. The main controller provides specific motion primitives, which control the arm in a specific way. Apart from primitives concerning forward and inverse kinematics, the arm supports, i.a., FOLD, which folds up the arm so that it can safely be driven on top of the cart, or RetractArm, which retracts the arm while minimizing the load to the system.

C ADDITIONAL PGCD RESSOURCES

In the following, we offer supplementary resources.

C.1 Motion Primitive Example

To provide a clearer understanding of the intricacies involved in specifying motion primitives, listing 1 demonstrates the fold motion primitive specification for the arm.

```
class Fold(MotionPrimitive):                                           1
    def __init__(self, name, component):                               2
        super().__init__(name, component)                              3
                                                                       4
    def duration(self):                                                5
        return Int(10) # takes 10 seconds                              6
                                                                       7
    def modifies(self):                                                8
        # changes all the arm's state variables                        9
        return self.component.variables()                             10
                                                                      11
    def pre(self):                                                    12
        return true # true as a Sympy formula                         13
                                                                      14
    def inv(self):                                                    15
        return true                                                   16
                                                                      17
    def post(self):                                                   18
        # lower arm: 130 degrees                                      19
        a = Eq(self.component.alpha(), self.component.maxAngle        20
            )
        # upper arm: -130 degrees                                     21
        b = Eq(self.component.beta(), self.component.minAngle)        22
        # turntable at 0                                              23
        c = Eq(self.component.gamma(), 0)                             24
        return And(a, b, c)                                           25
                                                                      26
    def preFP(self, point):                                           27
        # reuse cmpt resources                                        28
        return self.component.resources(point)                        29
                                                                      30
    def invFP(self, point):                                           31
        i = self.component.resources(point)                           32
        # lift formula to trajectories                                33
        # (variables as function of time)                             34
        return self.timify(i)                                         35
                                                                      36
    def postFP(self, point):                                          37
        return self.component.resources(point                        38
```

Listing 1: Fold motion primitive implementation

The `MotionPrimitive` base class is inherited by the given class. The base class defines the interface for specifying motion primitives. The constructor, which is denoted by the `__init__` method, takes two inputs: a `name` used by processes to call the class, and a specification of the components on which it executes (Line 31) The duration of the motion primitive is specified by the `duration` parameter (Line 5). The class also includes the methods `pre`, `inv`, and `post`, which correspond to the precondition, invariant, and postcondition of the motion primitive. The `modifies` method specifies which variables are modified by the motion primitive (Lines 12, 15, 18). In this case, the precondition and invariant are trivial, and the postcondition specifies specific angles for the upper/lower arm and the turntable. The `preFP`, `invFP`, and `postFP` methods (Lines 27, 31, 37) are the footprints of the precondition, invariant, and postcondition, respectively. They correspond to the resource function of the arm to which the motion primitive is attached. The function takes a `point` as input and returns a formula that evaluates to true if the `point` is in the footprint. While all the methods in the `MotionPrimitive` interface are redefined in the given class, in most cases, motion primitives retain the default behaviors. For example, the `modifies` method defaults to returning all the component's variables, which makes the specification shorter.

C.2 PGCD Code for Fetch

Listing 2 shows the PGCD source for the Fetch example.[1]

```
// Process arm
while true do
    receive(idle)
        fold ⇒
            move(origin)
            send(cart, folded)
        grab(loc) ⇒
            grab(loc)
            send(cart, grabbed)
        done ⇒
            break
```

```
// Process cart
send(arm, fold)
receive(idle)
    folded ⇒ skip
    moveToward(target)
    send(arm, grab(target))
    receive(idle)
        grabbed ⇒ skip
    send(arm, fold)
    receive(idle)
        folded ⇒ skip
    while (p ∉ homeRegion) do
        moveToward(homeRegion)
    send(arm, done)
```

Listing 2: PGCD program for Fetch

C.3 Promela Model Example

In Listings 3 - 5, we provide the Promela model for the Fetch example. For comparison, Listing 2 show the corresponding PGCD code, from which the promela code is generated.

The synchronization is achieved using standard SPIN constructs such as channels, and we employ an additional process called the `scheduler` to oversee time management. Listing 3 shows the code of the cart, along the the combination of concurrently executed motion primitives, that are extracted during the state space exploration. The

---

[1] Modified taken from `https://github.com/MPI-SWS/pgcd/blob/master/pgcd/nodes/verification/test/fetch_setup.py`.

code for the arm is displayed in Listing 4, while the code for the scheduler can be found in Listing 5. There exists only one final state in which all processes have completed their execution. Consequently, processes are either engaged in communication (e.g., Line 14 of listing 3), executing a motion primitive (line 18), or in a terminated state (line 36).

```
1   // Message motion primitives
2   mtype = { fold, folded, grab, grabbed, done, idle, moveToward, move,
        grabbing }
3   chan channel[2] = [0] of { mtype }
4   int busy_for[2] = 0
5   mtype doing[2]
6   bool terminated[2] = false
7
8   #define set_mp(id, mp, time) { doing[id] = mp; \
9   busy_for[id] = time; busy_for[id] == 0 }
10
11  active proctype cart() {
12      channel[1]!fold;
13      do
14      :: channel[0]?folded -> break
15      :: timeout -> set_mp(0, idle, 1)
16      od;
17      do
18      :: set_mp(0, moveToward, 1)
19      :: break
20      od;
21      channel[1]!grab;
22      do
23      :: channel[0]?grabbed -> break
24      :: timeout -> set_mp(0, idle, 1)
25      od;
26      channel[1]!fold;
27      do
28      :: channel[0]?folded -> break
29      :: timeout -> set_mp(0, idle, 1)
30      od;
31      do
32      :: set_mp(0, moveToward, 1)
33      :: break
34      od;
35      channel[1]!done;
36      EXIT_0: terminated[0] = true
37  }
```

Listing 3: Promela Code of the cart

```
active proctype arm() {
    do
    :: channel[1]?fold ->
    set_mp(1, move, 1);
    channel[0]!folded
    :: channel[1]?grab ->
    set_mp(1, grabbing, 1);
    channel[0]!grabbed
    :: channel[1]?done ->
    goto EXIT_1
    :: timeout ->
    set_mp(1, idle, 1)
    od;
    EXIT_1: terminated[1] = true
}
```

Listing 4: Promela Code of the Arm

```
active proctype scheduler() {
    do
    :: busy_for[0] > 0 }\&\& busy_for[1] > 0 -> d_step{
        # code for  printing time + motion primitives
        if
            :: busy_for[0] > busy_for[1] ->
               busy_for[0] = busy_for[0] - busy_for[1];
               busy_for[1] = 0
            :: else ->
               busy_for[1] = busy_for[1] - busy_for[0];
               busy_for[0] = 0
        fi;
    }

    :: terminated[0] \&\& terminated[1] ->
    break
    od;
}
```

Listing 5: Promela code of the Scheduler

The code responsible for printing the time and motion primitive state has been omitted due to its length. Printing the `m_type` variable directly is not possible because of the way it is compiled in SPIN. As a workaround, we need to conduct a case analysis on the potential values of `m_type` and then print the corresponding literal. Unfortunately, SPIN lacks support for functions, resulting in a lengthy expanded code. Furthermore, the model must be compiled with the `-DPRINTF` flag to ensure that the printing takes place during the model checking process.

BIBLIOGRAPHY

[1] Bernt M. Akesson, John Bagterp Jorgensen, and Sten Bay Jorgensen. “A Generalized Autocovariance Least-Squares Method for Covariance Estimation.” In: *2007 American Control Conference*. 2007, pp. 3713–3714. DOI: 10.1109/ACC.2007.4282878.

[2] James Albus, Roger Bostelman, and Nicholas Dagalakis. “The NIST robocrane.” In: *Journal of Robotic Systems* 10 (1993). DOI: 10.1002/rob.4620100509.

[3] Götz Alefeld and Günter Mayer. “Interval analysis: theory and applications.” In: *Journal of Computational and Applied Mathematics* 121.1 (2000), pp. 421–464. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00342-3.

[4] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. “Compositional modeling and refinement for hierarchical hybrid systems.” In: *J. Log. Algebr. Program.* 68.1-2 (2006), pp. 105–128.

[5] Rajeev Alur and Thomas A. Henzinger. “Modularity for Timed and Hybrid Systems.” In: *CONCUR*. Vol. 1243. LNCS. Springer, 1997, pp. 74–88.

[6] Eugene Asarin, Olivier Bournez, and Thao Dang. “Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems.” In: *Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems* (May 2000).

[7] Gregor B. Banušić, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. “PGCD: Robot Programming and Verification with Geometry, Concurrency, and Dynamics.” In: *ICCPS*. ACM/IEEE, 2019.

[8] M Babout, H Sidhoum, and L Frecon. “AMPERE: a programming language for physics.” In: *European Journal of Physics* 11.3 (May 1990), pp. 163–171. DOI: 10.1088/0143-0807/11/3/007.

[9] Jos CM Baeten and Willem Paul Weijland. *Process algebra*. Cambridge university press, 1991.

[10] Amine Benouhiba, Kanty Rabenorosoa, Patrick Rougeot, Morvan Ouisse, and Nicolas Andreff. “A Multisegment Electro-Active Polymer Based Milli-Continuum Soft Robots.” In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2018), pp. 7500–7506.

[11] J.A. Bergstra and C.A. Middelburg. “Process algebra for hybrid systems.” In: *Theoretical Computer Science* 335.2 (2005). Process Algebra, pp. 215–280. ISSN: 0304-3975.

[12] Mohammad Bijankhan and Vito Ferro. "Dimensional analysis and stage-discharge relationship for weirs: a review." In: *Journal of Agricultural Engineering* 48.1 (Feb. 2017), pp. 1–11. DOI: `10.4081/jae.2017.575`. URL: `https://agroengineering.org/index.php/jae/article/view/575`.

[13] Sampada Bodkhe, Clara Noonan, Frederick P. Gosselin, and Daniel Therriault. "Coextrusion of Multifunctional Smart Sensors." In: *Advanced Engineering Materials* 20.10 (2018), p. 1800206. DOI: `https://doi.org/10.1002/adem.201800206`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/adem.201800206`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/adem.201800206`.

[14] Aaron Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification*. Jan. 2007. DOI: `10.1007/978-3-540-74113-8`.

[15] E. Buckingham. "On Physically Similar Systems; Illustrations of the Use of Dimensional Equations." In: *Phys. Rev.* 4 (4 Oct. 1914), pp. 345–376. DOI: `10.1103/PhysRev.4.345`. URL: `https://link.aps.org/doi/10.1103/PhysRev.4.345`.

[16] Christoph Budde, Manfred Helm, Philipp Last, Annika Raatz, and Jürgen Hesselbach. "Configuration Switching for Workspace Enlargement." In: *Robotic Systems for Handling and Assembly*. Ed. by Daniel Schütz and Friedrich M. Wahl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-16785-0. DOI: `10.1007/978-3-642-16785-0_11`. URL: `https://doi.org/10.1007/978-3-642-16785-0_11`.

[17] Joseph Campbell, Cumhur Erkan Tuncali, Peng Liu, Theodore P. Pavlic, Ümit Özgüner, and Georgios E. Fainekos. "Modeling concurrency and reconfiguration in vehicular systems: A π-calculus approach." In: *CASE*. IEEE, 2016, pp. 523–530.

[18] Thais Campos, Jeevana Priya Inala, Armando Solar-Lezama, and Hadas Kress-Gazit. "Task-Based Design of Ad-hoc Modular Manipulators." In: *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*. IEEE, 2019, pp. 6058–6064. DOI: `10.1109/ICRA.2019.8794171`.

[19] Luca Cardelli and Philippa Gardner. "Processes in space." In: *Theor. Comp. Sci.* 431 (2012), pp. 40–55.

[20] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. "Model Checking Spatial Logics for Closure Spaces." In: *Logical Methods in Computer Science* 12.4 (Apr. 2016). DOI: `10.2168/LMCS-12(4:2)2016`. URL: `https://lmcs.episciences.org/2067`.

[21] Reymond Clavel. "Device for the movement and positioning of an element in space." Patent US4976582A (US). 1985.

[22] Robert Davis Cook. *Finite Element Modeling for Stress Analysis*. 1st. USA: John Wiley & Sons, Inc., 1994. ISBN: 0471107743.

[23] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 2012. ISBN: 9781118585771. URL: https://books.google.de/books?id=VWq5GG6ycxMC.

[24] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. "DRONA: a framework for safe distributed mobile robotics." In: *ICCPS 17*. ACM, 2017, pp. 239–248.

[25] Franz Dietrich, Jochen Maass, Carlos Cezar Bier, Ingo T. Pietsch, Annika Raatz, and Jürgen Hesselbach. "Detection and Avoidance of Singularities in Parallel Kinematic Machines." In: *Robotic Systems for Handling and Assembly*. 2011.

[26] M. C. F. Donkers and W. P. M. H. Heemels. "Output-Based Event-Triggered Control With Guaranteed $\mathcal{L}_\infty$-Gain and Improved and Decentralized Event-Triggering." In: *IEEE Transactions on Automatic Control* 57.6 (2012), pp. 1362–1376.

[27] Wilfried Elmenreich. "An introduction to sensor fusion." In: *Vienna University of Technology, Austria* 502 (2002), pp. 1–28.

[28] Franka Emika. *Datasheet Robot Arm and Control*. https://download.franka.de/Datasheet-EN.pdf. [Online; accessed 25-Apr-2023]. 2020.

[29] Ilon Bengt Erland. "Wheels For a course stable selfpropelling vehicle movable in any desired direction on the ground or some other base." Patent US3876255A (US). Nov. 1972.

[30] Samuel M. Felton, Michael Thomas Tolley, ByungHyun Shin, Cagdas D. Onal, Erik D. Demaine, Daniela Rus, and Robert J. Wood. "Self-folding with shape memory composites†." In: *Soft Matter* 9 (2013), pp. 7688–7694.

[31] Eugene F Fichter. "A Stewart platform-based manipulator: general theory and practical construction." In: *The international journal of robotics research* 5.2 (1986), pp. 157–182.

[32] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java." In: *PLDI*. ACM, 2002, pp. 234–245.

[33] Tully Foote. "tf: The transform library." In: *(TePRA) Technologies for Practical Robot Applications*. Open-Source Software workshop. 2013, pp. 1–6.

[34] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice Hall Professional Technical Reference, 1977. ISBN: 0131653326.

[35] J.B.J. Fourier. *Théorie analytique de la chaleur*. Manuscripta; History of science, 18th and 19th century. Chez Firmin Didot, père et fils, 1822. URL: https://books.google.de/books?id=TDQJAAAAIAAJ.

[36] Marco Frigerio, Jonas Buchli, and Darwin G Caldwell. "Model based code generation for kinematics and dynamics computations in robot controllers." In: *Workshop on Software Development and Integration in Robotics, St. Paul, Minnesota, USA*. Vol. 3. 1. 2012, p. 6.

[37] Marco Frigerio, Jonas Buchli, Darwin G Caldwell, and Claudio Semini. "RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages." In: *Journal of Software Engineering for Robotics (JOSER)* 7.1 (2016), pp. 36–54.

[38] Man Lok Fung, Michael Z. Q. Chen, and Yong Hua Chen. "Sensor fusion: A review of methods and applications." In: *2017 29th Chinese Control And Decision Conference (CCDC)*. 2017, pp. 3853–3860. DOI: 10.1109/CCDC.2017.7979175.

[39] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. "Delta-Complete Decision Procedures for Satisfiability over the Reals." In: *CoRR* abs/1204.3513 (2012). arXiv: 1204.3513. URL: http://arxiv.org/abs/1204.3513.

[40] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals." In: *CADE-24*. Vol. 7898. LNCS. Springer, 2013, pp. 208–214.

[41] Sicun Gao, Soonho Kong, and Edmund M. Clarke. "dReal: An SMT Solver for Nonlinear Theories over the Reals." In: *Automated Deduction - CADE-24*. Vol. 7898. Springer, 2013.

[42] Neil Gershenfeld. *Fab: The Coming Revolution on Your Desktop–from Personal Computers to Personal Fabrication*. USA: Basic Books, Inc., 2007. ISBN: 0465027466.

[43] Ritwika Ghosh, Sasa Misailovic, and Sayan Mitra. "Language Semantics Driven Design and Formal Analysis for Distributed Cyber-Physical Systems: [Extended Abstract]." In: *ApPLIED@PODC 2018*. ACM, 2018, pp. 41–44.

[44] A. Giusti and M. Althoff. "On-the-Fly Control Design of Modular Robot Manipulators." In: *IEEE Transactions on Control Systems Technology* 26.4 (July 2018), pp. 1484–1491. ISSN: 1063-6536. DOI: 10.1109/TCST.2017.2707336.

[45] Andrea Giusti, Martijn Zeestraten, Esra İçer, Aaron Pereira, Darwin G. Caldwell, Sylvain Calinon, and Matthias Althoff. "Flexible Automation Driven by Demonstration: Leveraging Strategies that Simplify Robotics." In: *IEEE Robotics & Automa-*

tion Magazine PP (May 2018), pp. 1–1. DOI: 10.1109/MRA.2018.2810543.

[46] Grigore Gogu. "Mobility of mechanisms: a critical review." In: *Mechanism and machine Theory* 40.9 (2005), pp. 1068–1097.

[47] C. Gosselin and J. Angeles. "Singularity analysis of closed-loop kinematic chains." In: *IEEE Transactions on Robotics and Automation* 6.3 (1990). ISSN: 1042-296X. DOI: 10.1109/70.56660.

[48] Jason Griffey. "Absolutely Fab-Ulous." In: *Library technology reports* 48 (2012), p. 21.

[49] ABB Group. *Operating manual – Introduction to RAPID*. http://rovart.cimr.pub.ro/docs/OpIntroRAPID.pdf. [Online; accessed 25-Apr-2023]. 2012.

[50] Martin Fürchtegott Grübler. *Getriebelehre: eine Theorie des Zwanglaufes und der ebenen Mechanismen.* Springer, 1917.

[51] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. "Complete completion using types and weights." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 27–38. DOI: 10.1145/2491956.2462192.

[52] S. Ha, S. Coros, A. Alspach, J. M. Bern, J. Kim, and K. Yamane. "Computational Design of Robotic Devices From High-Level Motion Specifications." In: *IEEE Transactions on Robotics* 34.5 (2018). ISSN: 1552-3098.

[53] J.K. Hackett and M. Shah. "Multi-sensor fusion: a perspective." In: *Proceedings., IEEE International Conference on Robotics and Automation*. 1990, 1324–1330 vol.2. DOI: 10.1109/ROBOT.1990.126184.

[54] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy abstraction." In: *POPL*. Ed. by John Launchbury and John C. Mitchell. ACM, 2002, pp. 58–70.

[55] G.J. Holzmann. "The Model Checker SPIN." In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[56] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. USA: Prentice-Hall, Inc., 1990. ISBN: 0135399254.

[57] Robert Hoyt, Jesse Cushing, and Jeffrey Slostad. *SpiderFab: Process Apertures for On-Orbit Construction of Kilometer-Scale*. Tech. rep. NNX12AR13G. Bothell, WA 98011: Tethers Unlimited, Inc., July 2013.

[58] K.H. Hunt and K.H. Hunt. *Kinematic Geometry of Mechanisms*. Oxford engineering science series. Clarendon Press, 1990. ISBN: 9780198562337. URL: https://books.google.de/books?id=0rlqQgAACAAJ.

[59] *Robotics — Vocabulary*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2021.

[60] R. Janeliukstis and D. Mironovs. "Smart Composite Structures with Embedded Sensors for Load and Damage Monitoring – A Review." English. In: *Mechanics of Composite Materials* 57 (2021). Russian translation published in Mekhanika Kompozitnykh Materialov, Vol. 57, No. 2, pp. 189-222, March-April, 2021., pp. 131–152. ISSN: 0191-5665. DOI: `10.1007/s11029-021-09941-6`.

[61] Gangyuan Jing, Tarik Tosun, Mark Yim, and Hadas Kress-Gazit. "Accomplishing high-level tasks with modular robots." In: *Auton. Robots* 42.7 (Oct. 2018), pp. 1337–1354. ISSN: 0929-5593. DOI: `10.1007/s10514-018-9738-1`. URL: `https://doi.org/10.1007/s10514-018-9738-1`.

[62] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: *Journal of Basic Engineering* 82 (1960). ISSN: 0021-9223. DOI: `10.1115/1.3662552`. URL: `https://doi.org/10.1115/1.3662552`.

[63] Y. Kanayama, D. MacPherson, and G. Krahn. "Two dimensional transformations and its application to vehicle motion control and analysis." In: *[1993] Proceedings IEEE International Conference on Robotics and Automation*. 1993, 13–18 vol.3. DOI: `10.1109/ROBOT.1993.291928`.

[64] Andrew Kennedy. "Dimension Types." In: *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*. Ed. by Donald Sannella. Vol. 788. Lecture Notes in Computer Science. Springer, 1994, pp. 348–362. DOI: `10.1007/3-540-57880-3\_23`.

[65] Sukjun Kim, Dae-Young Lee, Gwang-Pil Jung, and Kyu-Jin Cho. "An origami-inspired, self-locking robotic arm that can be folded flat." In: *Science Robotics* 3 (Mar. 2018), eaar2915. DOI: `10.1126/scirobotics.aar2915`.

[66] Charles A. Klein and Bruce E. Blaho. "Dexterity Measures for the Design and Control of Kinematically Redundant Manipulators." In: *The International Journal of Robotics Research* 6.2 (1987), pp. 72–83.

[67] V. Klema and A. Laub. "The singular value decomposition: Its computation and some applications." In: *IEEE Transactions on Automatic Control* 25.2 (1980), pp. 164–176.

[68] Keith Kotay and Daniela Rus. "Locomotion versatility through self-reconfiguration." In: *Robotics Auton. Syst.* 26 (1999), pp. 217–232.

[69] Keith Kotay and Daniela Rus. “The Inchworm Robot: A Multi-Functional System.” In: *Autonomous Robots* 8 (2000), pp. 53–69.

[70] R. Krichevsky and V. Trofimov. “The performance of universal encoding.” In: *IEEE Transactions on Information Theory* 27.2 (1981), pp. 199–207. DOI: 10.1109/TIT.1981.1056331.

[71] S. Kullback and R. A. Leibler. “On Information and Sufficiency.” In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. ISSN: 00034851. URL: http://www.jstor.org/stable/2236703 (visited on 04/25/2023).

[72] A. Kumar and K. J. Waldron. “The Workspaces of a Mechanical Manipulator.” In: *Journal of Mechanical Design* 103.3 (July 1981), pp. 665–672. ISSN: 0161-8458. DOI: 10.1115/1.3254968. eprint: https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/103/3/665/5662752/665_1.pdf. URL: https://doi.org/10.1115/1.3254968.

[73] Tinne De Laet, Wouter Schaekers, Jonas de Greef, and Herman Bruyninckx. *Domain Specific Language for Geometric Relations between Rigid Bodies targeted to robotic applications*. 2013. arXiv: 1304.1346 [cs.RO].

[74] Ruggero Lanotte and Massimo Merro. “A Calculus of Cyber-Physical Systems.” In: *LATA*. Springer, 2017, pp. 115–127. ISBN: 978-3-319-53733-7.

[75] Dae-Young Lee, Ji-Suk Kim, Sa-Reum Kim, Je-Sung Koh, and Kyu-Jin Cho. “The deformable wheel robot using magic-ball origami structure.” In: *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 55942. American Society of Mechanical Engineers. 2013, V06BT07A040.

[76] Tae Yoon Lee, James V. Zidek, and Nancy Heckman. *Dimensional Analysis in Statistical Modelling*. 2021. arXiv: 2002.11259 [math.ST].

[77] Jennifer A. Lewis and Bok Y. Ahn. “Three-dimensional printed electronics.” In: *Nature* 518.7537 (Feb. 2015), pp. 42–43. ISSN: 1476-4687. DOI: 10.1038/518042a.

[78] Suyi Li, Hongbin Fang, Sahand Sadeghi, Priyanka Bhovad, and Kon-Well Wang. “Architected Origami Materials: How Folding Creates Sophisticated Mechanical Properties.” In: *Advanced Materials* 31 (Feb. 2019), p. 1805282. DOI: 10.1002/adma.201805282.

[79] Jonathan Lim and Phillip Stanley-Marbell. “Newton: A Language for Describing Physics.” In: *CoRR* abs/1811.04626 (2018). arXiv: 1811.04626. URL: http://arxiv.org/abs/1811.04626.

[80] Yixiao Lin and Sayan Mitra. "StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems." In: *LCTES 15*. ACM, 2015, 9:1–9:10.

[81] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. 1st. USA: Cambridge University Press, 2017. ISBN: 1107156300.

[82] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. "Hybrid I/O automata." In: *Inf. Comput.* 185.1 (2003), pp. 105–157.

[83] Anthony A. Maciejewski and Charles A. Klein. "The Singular Value Decomposition: Computation and Applications to Robotics." In: *The International Journal of Robotics Research* 8.6 (1989), pp. 63–79. DOI: `10.1177/027836498900800605`. URL: `https://doi.org/10.1177/027836498900800605`.

[84] Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. "Motion Session Types for Robotic Interactions (Brave New Idea Paper)." In: *European Conference on Object-Oriented Programming, ECOOP 2019*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. 2019, 28:1–28:27.

[85] Hiroshi Makino. "Assembly Robot." Patent US4341502A (US). 1979.

[86] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. "Jungloid mining: helping to navigate the API jungle." In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 48–61. DOI: `10.1145/1065010.1065018`.

[87] B. A. McElhoe. "An Assessment of the Navigation and Course Corrections for a Manned Flyby of Mars or Venus." In: *IEEE Transactions on Aerospace and Electronic Systems* AES-2.4 (1966), pp. 613–623. DOI: `10.1109/TAES.1966.4501892`.

[88] M. A. McEvoy and N. Correll. "Materials that couple sensing, actuation, computation, and communication." In: *Science* 347.6228 (2015). ISSN: 0036-8075. DOI: `10.1126/science.1261689`. eprint: `https://science.sciencemag.org/content/347/6228/1261689.full.pdf`.

[89] Ankur M. Mehta, Joseph DelPreto, and Daniela Rus. "Integrated Codesign of Printable Robots." In: *J. Mechanisms Robotics* 7.2 (2015).

[90] Ankur M. Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. "Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications." In: *IROS*. 2014.

[91] Ankur M. Mehta, Joseph DelPreto, Kai Weng Wong, Scott Hamill, Hadas Kress-Gazit, and Daniela Rus. "Robot Creation from Functional Specifications." In: *Robotics Research, Proceedings of the 17th International Symposium of Robotics Research, ISRR 2015, Sestri Levante, Italy, September 12-15, 2015, Volume 2*. Ed. by Antonio Bicchi and Wolfram Burgard. Vol. 3. Springer Proceedings in Advanced Robotics. Springer, 2015, pp. 631–648. DOI: `10.1007/978-3-319-60916-4\_36`.

[92] Ankur M. Mehta, Joseph DelPreto, Kai Weng Wong, Hadas Kress-Gazit, and Daniela Rus. "Robot Creation from Functional Specifications." In: *Springer Proceedings in Advanced Robotics*. 2017.

[93] Philip M Merlin. "The Time-Petri-Net and the Recoverability of Processes." In: (1974).

[94] Aaron Meurer et al. "SymPy: symbolic computing in Python." In: *PeerJ Computer Science* 3 (2017). ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`.

[95] Aaron Meurer et al. "SymPy: symbolic computing in Python." In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`.

[96] Aleksey Mironov, Alexandrs Priklonskiy, Deniss Mironovs, and Pavel Doronkin. "Application of Deformation Sensors for Structural Health Monitoring of Transport Vehicles." In: *Reliability and Statistics in Transportation and Communication*. Ed. by Igor Kabashkin, Irina Yatskiv, and Olegas Prentkovskis. Cham: Springer International Publishing, 2020, pp. 162–175. ISBN: 978-3-030-44610-9.

[97] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. "RoboChart: modelling and verification of the functional behaviour of robotic applications." In: *Software & Systems Modeling* 18 (2019), pp. 3097–3149.

[98] Ilan E. Moyer. *CoreXY Cartesian Motion Platform*. `https://corexy.com/index.html`. [Online; accessed 25-Apr-2023]. 2012.

[99] Henrik Mühe, Andreas Angerer, Alwin Hoffmann, and Wolfgang Reif. *On reverse-engineering the KUKA Robot Language*. 2010. arXiv: `1009.5004 [cs.RO]`.

[100] BB Muminov and Eshankulov Kh. "Modelling asynchronous parallel process with Petri net." In: *International Journal of Engineering and Advanced Technology (IJEAT)* 8 (2019), pp. 400–405.

[101] Joseph T. Muth, Daniel M. Vogt, Ryan L. Truby, Yiğit Mengüç, David B. Kolesky, Robert J. Wood, and Jennifer A. Lewis. "Embedded 3D Printing of Strain Sensors within Highly Stretchable Elastomers." In: *Advanced Materials* 26.36 (2014), pp. 6307–6312. DOI: `10.1002/adma.201400334`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/adma.201400334`.

[102] Venkatesh Mysore and Bud Mishra. "Algorithmic Algebraic Model Checking III: Approximate Methods." In: *Electronic Notes in Theoretical Computer Science* 149.1 (2006). Proceedings of the 7th International Workshop on Verification of Infinite-State Systems (INFINITY 2005), pp. 61–77. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2005.11.017`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066106000545`.

[103] Arne Nordmann, Nico Hochgeschwender, Dennis Leroy Wigand, and Sebastian Wrede. "A Survey on Domain-Specific Modeling and Languages in Robotics." In: *Journal of Software Engineering in Robotics (JOSER)* 7.1 (2016), pp. 75–99.

[104] Pierluigi Nuzzo. "Compositional Design of Cyber-Physical Systems Using Contracts." PhD thesis. EECS Department, University of California, Berkeley, Aug. 2015. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html`.

[105] Brian J. Odelson, Murali R. Rajamani, and James B. Rawlings. "A new autocovariance least-squares method for estimating noise covariances." In: *Automatica* 42.2 (2006), pp. 303–308. ISSN: 0005-1098. DOI: `https://doi.org/10.1016/j.automatica.2005.09.006`. URL: `https://www.sciencedirect.com/science/article/pii/S0005109805003262`.

[106] R. Olfati-Saber and J.S. Shamma. "Consensus Filters for Sensor Networks and Distributed Sensor Fusion." In: *Proceedings of the 44th IEEE Conference on Decision and Control*. 2005, pp. 6698–6703. DOI: `10.1109/CDC.2005.1583238`.

[107] Michael W. Oppenheimer, David B. Doman, and Justin D. Merrick. "Multi-scale physics-informed machine learning using the Buckingham Pi theorem." In: *Journal of Computational Physics* 474 (2023), p. 111810. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2022.111810`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999122008737`.

[108] Christiaan J.J. Paredis, H. Benjamin Brown, and Pradeep K. Khosla. "A rapidly deployable manipulator system." In: *Robotics and Autonomous Systems* 21.3 (1997). Critical Issues in Robotics, pp. 289–304. ISSN: 0921-8890. DOI: `https://doi.org/10.1016/S0921-8890(97)00081-X`. URL: `https://www.sciencedirect.com/science/article/pii/S092188909700081X`.

[109] Sushant M. Patil, R.R. Malagi, R.G. Desavale, and Sanjay H. Sawant. "Fault identification in a nonlinear rotating system using Dimensional Analysis (DA) and central composite rotatable design (CCRD)." In: *Measurement* 200 (2022), p. 111610. ISSN: 0263-2241. DOI: `https://doi.org/10.1016/j.measurement.2022.111610`. URL: `https://www.sciencedirect.com/science/article/pii/S0263224122008211`.

[110] Raspberry Pi. *Datasheet Raspberry Pi 4 Model B*. `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf`. [Online; accessed 25-Apr-2023]. 2020.

[111] Marcus Pirron and Damien Zufferey. "MPERL: Hardware and Software Co-design for Robotic Manipulators ©." In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*. IEEE, 2019, pp. 7784–7790. DOI: `10.1109/IROS40897.2019.8968188`. URL: `https://doi.org/10.1109/IROS40897.2019.8968188`.

[112] Marcus Pirron, Damien Zufferey, and Phillip Stanley-Marbell. "Automated Controller and Sensor Configuration Synthesis Using Dimensional Analysis." In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.11 (2020), pp. 3227–3238. DOI: `10.1109/TCAD.2020.3013044`. URL: `https://doi.org/10.1109/TCAD.2020.3013044`.

[113] André Platzer. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer, 2010.

[114] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program synthesis from polymorphic refinement types." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 522–538. DOI: `10.1145/2908080.2908093`.

[115] Johannes Prechtl, Stefan Seelecke, Paul Motzki, and Gianluca Rizzello. "Self-Sensing Control of Antagonistic SMA Actuators Based on Resistance-Displacement Hysteresis Compensation." In: Sept. 2020. DOI: `10.1115/SMASIS2020-2224`.

[116] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. "ROS: an open-source Robot Operating System." In: *ICRA workshop on open source software*. 2009.

[117] J.W.S. Rayleigh. *The Theory of Sound*. The Nineteenth Century. General Collection Bd. 1. Macmillan and Company, 1877. URL: `https://books.google.de/books?id=kvxYAAAAYAAJ`.

[118] Michael Reckhaus, Nico Hochgeschwender, Paul G. Ploeger, and Gerhard K. Kraetzschmar. *A Platform-independent Programming Environment for Robot Control*. 2010. arXiv: `1010.0886 [cs.RO]`.

[119] *Robotics Overview*. `https://www.osha.gov/robotics`. Accessed: 2023-02-12.

[120] William C. Rounds and Hosung Song. "The Phi-Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems." In: *HSCC*. Springer, 2003, pp. 435–449.

[121] Daniela Rus and Marsette Vona. "A physical implementation of the self-reconfiguring crystalline robot." In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)* 2 (2000), 1726–1733 vol.2.

[122] Stefan Schaal and Christopher Atkeson. "Learning Control in Robotics." In: *Robotics & Automation Magazine, IEEE* 17 (July 2010), pp. 20–29. DOI: `10.1109/MRA.2010.936957`.

[123] Adriana Schulz, Cynthia R. Sung, Andrew Spielberg, Wei Zhao, Robin Cheng, Eitan Grinspun, Daniela Rus, and Wojciech Matusik. "Interactive robogami: An end-to-end system for design of robots with ground locomotion." In: *I. J. Robotics Res.* 36.10 (2017).

[124] Sara Seager et al. "The Exo-S probe class starshade mission." In: Sept. 2015, 96050W. DOI: `10.1117/12.2190378`.

[125] C. E. Shannon. "A mathematical theory of communication." In: *The Bell System Technical Journal* 27.4 (1948), pp. 623–656. DOI: `10.1002/j.1538-7305.1948.tb00917.x`.

[126] Bruno Siciliano and Oussama Khatib, eds. *Springer Handbook of Robotics*. 2nd ed. Springer Handbooks. Berlin: Springer, 2016. DOI: `10.1007/978-3-540-30301-5`.

[127] José Reinaldo Silva and Pedro MG Del Foyo. "Timed petri nets." In: *Petri Nets: Manufacturing and Computer Science*. InTech, 2012, pp. 359–378.

[128] Gerald L. Smith, Stanley F. Schmidt, and Leonard A. McGee. *Application of Statistical Filter Theory to the Optimal Estimation of Position and Velocity on Board a Circumlunar Vehicle*. Tech. rep. 19620006857. National Aeronautics and Space Administration, 1962.

[129] Phillip Stanley-Marbell, Diana Marculescu, Radu Marculescu, and Pradeep K Khosla. "Modeling computational, sensing, and actuation surfaces." In: *Low-Power Processors and Systems on Chips* (2005), pp. 16–1.

[130] D. Stewart. “A Platform with Six Degrees of Freedom.” In: *Proceedings of the Institution of Mechanical Engineers* 180.1 (1965). DOI: 10.1243/PIME_PROC_1965_180_029_02.

[131] Hamburg Trinamic. *TMC2100 Datasheet*. https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC2100_datasheet_rev1.13.pdf. [Online; accessed 25-Apr-2023]. 2022.

[132] Lung-Wen Tsai. “Mechanism Design: Enumeration of Kinematic Structures According to Function.” In: *Journal of Mechanical Design* 122.4 (Dec. 2000), pp. 583–583. ISSN: 1050-0472. DOI: 10.1115/1.1334346. eprint: https://asmedigitalcollection.asme.org/mechanicaldesign/article-pdf/122/4/583/5687334/583_1.pdf. URL: https://doi.org/10.1115/1.1334346.

[133] Zerksis D. Umrigar. “Fully static dimensional analysis with C++.” In: *SIGPLAN Notices* 29.9 (1994), pp. 135–139. DOI: 10.1145/185009.185036.

[134] L. -. T. Wang and C. C. Chen. “A combined optimization method for solving the inverse kinematics problems of mechanical manipulators.” In: *IEEE Transactions on Robotics and Automation* 7.4 (1991). ISSN: 1042-296X. DOI: 10.1109/70.86079.

[135] Wei Wang, Nam-Geuk Kim, Hugo Rodrigue, and Sung-Hoon Ahn. “Modular assembly of soft deployable structures and robots.” In: *Mater. Horiz.* 4 (3 2017), pp. 367–376. DOI: 10.1039/C6MH00550K. URL: http://dx.doi.org/10.1039/C6MH00550K.

[136] Youchao Wang, Sam Willis, Vasileios Tsoutsouras, and Phillip Stanley-Marbell. “Deriving Equations from Sensor Data Using Dimensional Function Synthesis.” In: *ACM Trans. Embedded Comput. Syst.* 18.5s (2019), 84:1–84:22. DOI: 10.1145/3358218.

[137] *What Is a Robot?* https://robots.ieee.org/learn/what-is-a-robot/. Accessed: 2023-02-12.

[138] L. Xiao, S. Boyd, and S. Lall. “A scheme for robust distributed sensor fusion based on average consensus.” In: *IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005.* 2005, pp. 63–70. DOI: 10.1109/IPSN.2005.1440896.

[139] Wenzhong Yan and Ankur Mehta. “A Cut-and-Fold Self-Sustained Compliant Oscillator for Autonomous Actuation of Origami-Inspired Robots.” In: *Soft Robotics* 9 (Nov. 2021). DOI: 10.1089/soro.2021.0018.

[140] Wenzhong Yan, Yun-Chen Yu, and Ankur Mehta. “Rapid Design of Mechanical Logic Based on Quasi-Static Electromechanical Modeling.” In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China,*

November 3-8, 2019. IEEE, 2019, pp. 5820–5825. DOI: 10.1109/IROS40897.2019.8967964.

[141] Guang-Zhong Yang, Javier Andreu-Perez, Xiaopeng Hu, and Surapa Thiemjarus. “Multi-sensor Fusion.” In: *Body Sensor Networks*. Ed. by Guang-Zhong Yang. London: Springer London, 2014, pp. 301–354. ISBN: 978-1-4471-6374-9. DOI: 10.1007/978-1-4471-6374-9_8. URL: https://doi.org/10.1007/978-1-4471-6374-9_8.

[142] M. Yim, Ying Zhang, and D. Duff. “Modular robots.” In: *IEEE Spectrum* 39.2 (2002), pp. 30–34. DOI: 10.1109/6.981854.

[143] Mark Yim, Wei-min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. “Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics].” In: *IEEE Robotics and Automation Magazine* 14.1 (2007), pp. 43–52. DOI: 10.1109/MRA.2007.339623.

[144] H. Yurdem, V. Demir, and A. Degirmencioglu. “Development of a mathematical model to predict head losses from disc filters in drip irrigation systems using dimensional analysis.” In: *Biosystems Engineering* 100.1 (2008), pp. 14–23. ISSN: 1537-5110. DOI: https://doi.org/10.1016/j.biosystemseng.2008.01.003. URL: https://www.sciencedirect.com/science/article/pii/S153751100800024X.

CURRICULUM VITAE

RESEARCH INTERESTS

Programming Languages, Analysis and (Formal) verification of dynamical systems, especially under the influence of physics.

EDUCATION

2024, **Doctor of Engineering**, Max Planck Institute for Software Systems.
2016, **Master of Science**, Saarland University
2013, **Bachelor of Science**, Saarland University