Physics Engines for Verification of Robotic Systems

Arjun Singh Badyal

PhD

University of York

Computer Science

January 2026

Abstract

Simulation is widely used to informally test designs, but the evidence it provides is restricted to the scenarios that are simulated. This limitation can be addressed by complementing simulation with proof, which provides mathematical evidence of expected behaviour across all relevant scenarios. Proof-based verification helps meet the trustworthiness demands of modern robotic applications. However, co-verification using simulation and proof requires consistency between the simulation and proof models. Achieving such consistency remains difficult because the physics engines used by most simulators do not expose their underlying algorithms; at best, reliance on source code is required. This thesis introduces a framework that enables transparency, configurability, and extensibility in the use of physics engines. The results not only support consistent co-verification, but also help expose, and therefore address, the reality gap by explicitly recording simulation assumptions.

List of Contents

1 Introduction **1**
- 1.1 Motivation . . . 3
- 1.2 Objectives . . . 7
- 1.3 Contributions . . . 8
- 1.4 Thesis Structure . . . 8
- 1.5 Glossary of Terms and Acronyms . . . 9

2 Background and Physical Modelling **12**
- 2.1 Physical Modelling . . . 12
  - 2.1.1 p-model Overview . . . 12
  - 2.1.2 Compositionality . . . 16
  - 2.1.3 Extending the p-model . . . 19
- 2.2 RoboSim . . . 24
  - 2.2.1 RoboSim Semantics . . . 25
- 2.3 Evaluation of Physics Engines . . . 26
  - 2.3.1 Criteria of Assessment . . . 26
  - 2.3.2 Preliminaries . . . 28
  - 2.3.3 Robot Modelling Languages and File Formats . . . 29
  - 2.3.4 Evaluation of Physics Engines . . . 32
- 2.4 Summary . . . 41

3 Simulation Framework **42**
- 3.1 Overview . . . 42
- 3.2 Technique 3: EquationComposition . . . 54
  - 3.2.1 Overview . . . 55
  - 3.2.2 Step 1: Calculate Formulations . . . 56
  - 3.2.3 Step 2: Localise Reference Components . . . 60
  - 3.2.4 Step 3: Validate Generator Compatibility . . . 63
  - 3.2.5 Step 4: Determine Topology . . . 64
  - 3.2.6 Step 5: Assign Numbering . . . 65
  - 3.2.7 Step 6: Calculate Equations . . . 65
- 3.3 Solution Domain Specific Languages . . . 68
  - 3.3.1 Solution References (slnRef) . . . 68
  - 3.3.2 Solution Description Format (slnDF) . . . 70
- 3.4 Technique 4: Guided Choice of Solutions . . . 73
  - 3.4.1 Overview . . . 74
  - 3.4.2 Step 1: Initialise Sidecar and Add Null Type . . . 75
  - 3.4.3 Step 2: Calculate Formulation . . . 77
  - 3.4.4 Step 3: Resolve Solution (First Pass) . . . 77
  - 3.4.5 Step 4: Reorder Solutions . . . 81
  - 3.4.6 Step 5: Resolve Each Solution in Order . . . 82
- 3.5 Technique 5: Generation for a Platform Engine . . . 85
  - 3.5.1 Overview . . . 85

3.5.2 Step 1: Translate SolutionRef to slnDF . . . 86
3.5.3 Step 2: Translate slnDF to Target Language . . . 90
3.6 Summary . . . 95

4 Tool and Case Studies **96**
4.1 Tool Implementation . . . 96
4.1.1 T3: Equation Composition . . . 96
4.1.2 T4: Guided Choice . . . 106
4.1.3 T5: Code Generation . . . 118
4.1.4 Mapping Generation . . . 127
4.1.5 Proof Generation . . . 131
4.2 Validation of Simulation Framework . . . 134
4.2.1 Framework Coverage Tests . . . 134
4.2.2 Benchmarking . . . 139
4.3 Case Studies . . . 141
4.3.1 Simple Arm Pick-and-Place . . . 141
4.3.2 Acrobot . . . 143
4.3.3 Mobile Robot Obstacle Avoidance . . . 153
4.4 Discussion . . . 159
4.4.1 Extensibility . . . 159
4.4.2 Critical Reflections . . . 160

5 Conclusions **162**
5.1 Summary . . . 162
5.2 Evaluation of Objectives . . . 162
5.3 Future Work . . . 163
5.3.1 Simulation Framework . . . 163
5.3.2 p-model Metamodel and Semantics . . . 165
5.3.3 Verification and Proof . . . 165

A Appendices **166**
A.1 Examples . . . 166
A.1.1 SimpleArm: Pick and Place . . . 166
A.1.2 Acrobot: Swing-up Control . . . 168
A.2 Simulation Framework Appendix . . . 173
A.2.1 Technique 3: Equation Composition . . . 173
A.2.2 Technique 4: Guided Choice of Solutions . . . 188
A.2.3 Technique 5: Generation for a Platform Engine . . . 196
A.3 Pseudocode and Implementation Comparison . . . 200
A.3.1 Technique 3: Equation Composition . . . 201
A.3.2 Technique 4: Guided Choice . . . 216
A.3.3 Technique 5: Generation for a Platform Engine . . . 227
A.4 Physics Engine Evaluation Reference Locations . . . 227
A.5 Robot Modelling Languages and File Formats: Documentation References . . . 243
A.6 Coverage Tests . . . 245
A.6.1 eqnComp Coverage . . . 245
A.6.2 guidedChoice Coverage . . . 254
A.6.3 sourceCodeGen Coverage . . . 256
A.6.4 Metamodel Coverage . . . 257

List of Tables

1.1 Glossary of terms, acronyms, and abbreviations. 9

2.1 Special variables in the p-model and their mathematical semantics. 16
2.2 Coverage and extensibility of selected robot modelling languages and description formats. 31
2.3 Configurability comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a). 39
2.4 Coverage comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a). *Drake uses 6D spatial vectors but not Plücker coordinates specifically. 40
2.5 Extensibility comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a). 40

3.1 The unfolded solution methods for $\boldsymbol{\theta}$ and the method *GeneralisedPosition_method1*. 50

4.1 MetaOperations functions for slnDF to EMF model manipulation. 99
4.2 Mapping between pseudocode notation and Xtend implementation. 100
4.3 Reserved variables generated by all formulation libraries. 106
4.4 Key concepts and artefacts in the solution library workflow. 111
4.5 Ordered solution pipeline for the Simple Arm example. The torque input $\boldsymbol{\tau}$ is supplied by the platform mapping solution; the remaining entries are unfolded from the user-level $\boldsymbol{\theta}$ request. 112
4.6 File generation categories for Technique 5 (C++ target) 120
4.7 Simple Arm generator execution times 141
4.8 Acrobot generator execution times 152
4.9 Mobile Robot generator execution times 157
4.10 Extensibility status of library components in each technique. T4 and T5 support user extension. T3's equation library is now specified in slnDF, but formulation detection still requires generator modification. 159

A.1 Correspondence between semantic translation rules and imperative algorithm sections 198
A.2 MuJoCo Documentation locations for evaluation references. 227
A.3 PyBullet Documentation locations for evaluation references. 230
A.4 Drake Documentation locations for evaluation references. 234
A.5 RBDL Documentation locations for evaluation references. 237
A.6 PhysX Articulations Documentation locations for evaluation references. 238
A.7 Pinocchio Documentation locations for evaluation references. 240
A.8 DQRobotics Documentation locations for evaluation references. 241
A.9 GAFRO Documentation locations for evaluation references. 242
A.10 Test coverage for eqnComp algorithm steps. 246

A.11 eqnComp algorithm to test case traceability matrix. 251
A.12 eqnComp integration tests organised by category and formulation. 253
A.13 Test coverage for guidedChoice algorithm steps. 254
A.14 guidedChoice algorithm to test case traceability matrix. 255
A.15 guidedChoice integration tests organised by category and formulation. 256
A.16 Test coverage for sourceCodeGen algorithm steps. 256
A.17 sourceCodeGen algorithm to test case traceability matrix. 257
A.18 sourceCodeGen integration tests organised by category and formulation. 258
A.19 P-model metamodel component to test mapping. 258

List of Figures

1.1 Simple Arm . . . 1
1.2 System model. . . . 2
1.3 RoboStar Framework . . . 4
1.4 Feedback controller model, based on the presentation in [14]. . . . 5

2.1 The Simple Arm serial-chain p-model. . . . 14
2.2 The joint, sensor, and actuator definitions used by the Simple Arm p-model. . . . 15
2.3 System view of a p-model showing the robotic platform with sensor and actuator blocks. . . . 17
2.4 System view extended with system-level variables and equations (e.g., rigid-body dynamics). . . . 18
2.5 Extended `PModel` and `Definition` class hierarchy. . . . 20
2.6 Extended `DynamicDevice` with InOuts and constraints. . . . 20
2.7 Relation properties on `Link`, `Body`, and `Joint` for system correspondence. . . . 21
2.8 Solutions infrastructure for encoding numerical methods. . . . 21
2.9 New expression types for mathematical constructs. . . . 22
2.10 New type classes for vector and matrix types. . . . 22
2.11 P-model+ content for the Simple Arm. . . . 23
2.12 Examples of Mathematical Structures used in Robotics . . . 27
2.13 Rigid body pipeline in Bullet from left to right. Taken from [83]. . . . 34

3.1 An overview of the RoboSim simulation framework. . . . 43
3.2 Simple Arm Simulation Framework relationship . . . 45
3.3 The Simple Arm system-level block after applying eqnComp. . . . 47
3.4 The relation between Simple Arm system-level variables and `BaseLink`. . . . 48
3.5 Selected Simple Arm p-model+ content after guidedChoice. . . . 51
3.6 Technique 3: Equation Composition. This technique takes a p-model and returns an extended p-model with system-level equations. . . . 55
3.7 Calculating the formulation by checking reference components. . . . 59
3.8 Calculating the formulation by checking variables and equations of local components. . . . 60
3.9 Translating a p-model with reference components to one with localised components. . . . 63
3.10 Assigning numbering to the links and joints of the p-model. . . . 65
3.11 Technique 4 (Guided Choice): from a p-model+ and the solution library obtain a p-model+ with resolved solutions. . . . 74
3.12 Solution Dependency Graph. Arrows show variable dependencies between solutions. . . . 81
3.13 sourceCodeGen: Generation for a Platform Engine. This technique takes a p-model+ with solutions and produces platform engine source-code. . . . 85

4.1 Toolbar buttons for invoking the generators . . . 96
4.2 Xtext generator pattern and eqnCompGenerator comparison . . . 97
4.3 eqnComp generator architecture . . . 98

4.4 eqnComp algorithm and implementation comparison . . . 102
4.5 Side-by-side comparison of EquationLib dispatch mechanism. The library routes equation generation to the appropriate formulation-specific specification based on the detected formulation and topology. . . . 104
4.6 SKO formulation: procedures and four-step equation generation structure. . . . 105
4.7 guidedChoice generator architecture . . . 107
4.8 guidedChoiceGenerator class structure . . . 107
4.9 T4 loop: solution references are iteratively unfolded into resolved `slnRef` entries via the `SolutionRef` class and the shared solution library. . . . 110
4.10 sourceCodeGen generator architecture . . . 118
4.11 Side-by-side comparison of sourceCodeGen pipeline and implementation. . . . 119
4.12 Runtime architecture showing CyPhyCircus semantic correspondence and numbered cycle transitions. The world mapping computes `sensor_output` from world and platform state; the platform mapping is updated via `update FromSensors` and provides input events for `SendToDModel`. . . . 124
4.13 Correspondence between CyPhyCircus semantics and the generated C++ orchestrator implementation. . . . 125
4.14 T5 flow: the generator consumes emitted `slnRef` files, instantiates solution templates from the shared library, and writes the resulting `slnDF` description. . . 126
4.15 Refined Technique 2 mapping generation . . . 128
4.16 Simple Arm platform mapping . . . 129
4.17 Proof generation from the p-model . . . 131
4.18 Acrobot trajectory comparison . . . 140
4.19 Acrobot forward dynamics benchmark . . . 140
4.20 Simple Arm d-model . . . 142
4.21 Acrobot p-model . . . 145
4.22 Acrobot Featherstone joint definition . . . 146
4.23 Acrobot CGA joint definition . . . 146
4.24 Acrobot formulation trajectory comparison . . . 146
4.25 Acrobot d-model . . . 148
4.26 Acrobot simulation . . . 149
4.27 Acrobot swing-up and LQR stabilisation . . . 153
4.28 Mobile robot p-model . . . 154
4.29 Mobile robot sensors . . . 155
4.30 Mobile robot d-model . . . 156
4.31 Mobile robot platform mapping . . . 157
4.32 Mobile robot visualisation . . . 158
4.33 Mobile robot clearance over time . . . 158

A.1 The SimpleArm p-model, and d-model being animated in the Drake simulator. 168
A.2 Side-by-side comparison of the Technique 3 main algorithm and its implementation. . . . 202
A.3 Side-by-side comparison of calculateFormulations algorithm and implementation. 203
A.4 Side-by-side comparison of findDynamicsFormulation algorithm and implementation. . . . 205
A.5 Side-by-side comparison of findSensorFormulations algorithm and implementation. . . . 206
A.6 Side-by-side comparison of findActuatorFormulations algorithm and implementation. . . . 207
A.7 Side-by-side comparison of localiseRefs algorithm and implementation. . . . 208

A.8 Side-by-side comparison of convertReferenceJointsToLocal algorithm and implementation. 208
A.9 Side-by-side comparison of the validate algorithm (SKO formulation) and its implementation. 209
A.10 Side-by-side comparison of calculateTopology algorithm and implementation. . 210
A.11 Side-by-side comparison of assignNumbering algorithm and implementation. . 210
A.12 Side-by-side comparison of orderPmodel algorithm and implementation (reverse ordering case for SKO). 211
A.13 Side-by-side comparison of calculateEquations algorithm and implementation. . 212
A.14 Side-by-side comparison of findSensorByFQN algorithm and implementation. . 213
A.15 Side-by-side comparison of findActuatorByFQN algorithm and implementation. 214
A.16 Side-by-side comparison of getBaseLink algorithm and implementation. 214
A.17 Side-by-side comparison of getChildLinks algorithm and implementation. . . . 215
A.18 Side-by-side comparison of the Technique 4 main algorithm and its implementation. 216
A.19 Side-by-side comparison of initialiseSidecar algorithm and implementation. . . . 217
A.20 Side-by-side comparison of resolveSolutionFirstPass algorithm and implementation. 218
A.21 Side-by-side comparison of updateVariable algorithm and implementation. . . . 219
A.22 Side-by-side comparison of the reorder solutions step and its implementation. . 220
A.23 Side-by-side comparison of resolveSolution algorithm and implementation. . . . 221
A.24 Side-by-side comparison of Step 5 main loop and its implementation. 222
A.25 Side-by-side comparison of calculateInitialConditions algorithm and implementation. 223
A.26 Side-by-side comparison of processAlgebraicConstraints algorithm and implementation. 224
A.27 Side-by-side comparison of evaluateVariable algorithm and implementation. . . 225
A.28 Side-by-side comparison of getEquation algorithm and implementation. 226
A.29 Side-by-side comparison of sourceCodeGen pipeline and implementation. . . . 227

Acknowledgements

First, I would like to thank my supervisors for their patience, guidance, and continued support throughout my PhD. Their insight and encouragement have been invaluable at every stage of this work.

I am deeply grateful to my family, Mum and Dad, Jodh, Mike, Rob, and Gary, for their constant support and understanding. I would especially like to thank Chachaji, who encouraged my early interest in science by giving me physics books when I was younger, an influence that ultimately led me to this work.

I would like to thank my friends during the PhD for their support, in particular Gricel, Scott, Christian, Lex, Holly, Ziggy, Santiago, and Gautham. I am especially grateful to Christian for answering my many questions about Isabelle and programming languages, and to Lex for teaching me about compilers.

I would also like to thank Alvaro and Pedro for the many meetings and discussions in which they generously shared their time and expertise, answering numerous questions about Robotool.

Finally, I would like to thank my grandparents.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for a degree or other qualification at this University or elsewhere. All sources are acknowledged as references.

1 Introduction

"I was born not knowing and have only had a little time to change that here and there."

— Richard Feynman

How can it be ensured that a robot will move where it is expected to move, and generally interact with its environment in the way it is expected? This is the fundamental question motivating this thesis. The modern day roboticist would perhaps answer this question with a simulation of their robotic system. The question then becomes how would one verify the system using other techniques (such as testing or proof) to confirm the behaviour observed in simulation? This requires using the equations of the model of the robotic system and their solutions, adopted for simulation, and then ensuring that there is consistency with the equations used, for example, in testing and proof techniques.

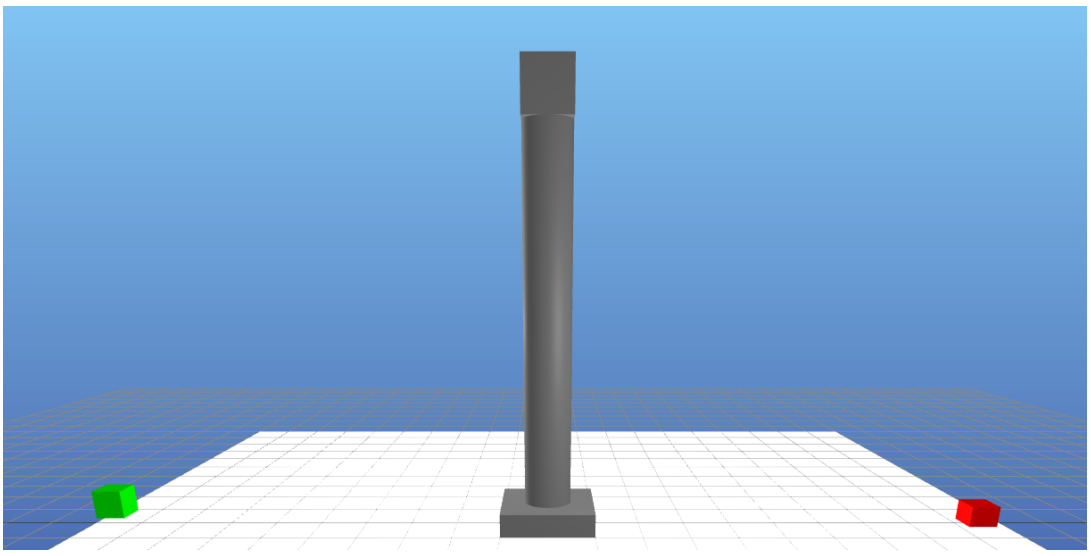


Figure 1.1: Simple Arm

Consider the following example, which is henceforth used to illustrate the majority of points in this thesis: A simple inverted pendulum to pick and place an object between two blocks (Appendix A.1.1). How can the evolution over time of the robotic platform within its environment be precisely modelled? This requires a series of modelling abstractions, the first of which is to distinguish between the controller and the physical. The robot's controllers form the controller portion of the system, communicating with the robot at discrete time-steps. A model can be associated to this controller portion of a robotic system (**controller model**). The complement of this is the physical system, which exists in continuous time. The physical system consists of a model of the robotic platform (**robotic platform model**), and a model of its environment (environment model). Note that even though a physical system exists in continuous time, it may have a discrete model. Discrete time behavioural equations in the robotic platform model are not considered here. Similarly a controller system may have a continuous model, such as a

model of a PID controller [1]. Continuous models of controllers are not considered as part of the controller model in this thesis, however controllers are considered as models of actuators and sensors in the robotic platform model.

Next, the relationship between the controller model and the robotic platform model must be determined. A common abstraction is to consider the inputs and outputs of a robot as sensors and actuators. The output of a sensor reading from a robot corresponds to the input of an event in the controller model. Operations sent from the controller model translate to physical inputs of the sensors and actuators. These relationships sufficiently characterise the relationship between the controller model and the robotic platform model, and this is called the **controller-platform mapping**.

Definition 1. A **robotic system** is completely represented by the combination of the controller model, the robotic platform model, and the controller-platform mapping.

All that remains is to consider the robot's relationship to its environment. The model of the environment, which consists of several robotic platform models (as well as fluid models, static entity models, etc.), is called the **environment model**. The mapping from the environment model to the robotic platform model is called the **environment mapping**. In order to fully describe the relationship between several physical entities, the position of all the entities is required, along with the relationship between important physical properties of the entities and the wider environment. For example, the robotic arm has a sensor on its base which is used to detect whether or not the end of the gripper is obstructing the target blocks. The physical property of interest is distance, however it could also be voltage, or any other physical property which the sensor may take as input. The equation which describes the distance between entities is in the environment model. The environment mapping relates the variable, distance, in this equation, to the physical property of interest in the robotic platform model.

The system model described above is presented in Fig .1.2.

A **simulation** of a robotic system in its environment is the animation of this controller-physical system model as a program run on a computer.

In this thesis, the controller portion of a simulation can be thought of as the execution of the code generated from the controller model. The simulation of the physical system is called a **physics engine**. Note that in general, robotic simulators do not consider the discrete portion of the simulator as a controller. Instead, they are usually more generic discrete-time simulators.

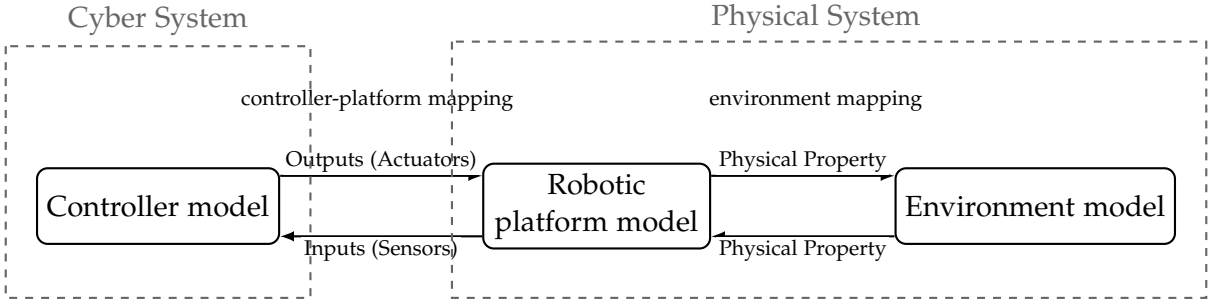


Figure 1.2: System model.

In general, physics engines are components of a simulation that embody, determine, or use equations describing physical properties or behaviour. They are found across many disciplines, including animated films [2], video games [3], and robotics [4]. Indeed, the methods used in

physics engines can be relevant for any piece of software where physics occurs. As a result, physics engines today are often designed to be broad enough to cover multiple domains, rather than focusing on robotic systems alone. For instance, most physics engines favour speed over accuracy when calculating the dynamics of a robotic system [5], something that is not ideal, as it exacerbates the reality gap that occurs when behaviour observed in simulation is not reproduced in the real world. Furthermore, there are currently no convenient means by which the developer of a simulation can determine or even find out the underlying equations used by the physics engine. Without this possibility or knowledge, consistency between simulation and other verification techniques cannot be guaranteed.

These issues amongst others that provide motivation for this work are discussed in the next section. Section 1.2 presents the research objectives guiding the work. Finally, Section 1.4 describes the structure of the thesis.

1.1 Motivation

Besides simulation, **verification** of a robotic system can involve testing and proof (using model checkers and theorem provers such as FDR [6] and Isabelle [7]). Testing and simulation alone are not enough for the verification of safety-critical or mission-critical robotic systems [8]. With complementary proof-based verification, one can ensure that observations in model-based testing and simulation apply for all scenarios. Use of a variety of verification techniques, however, is only effective if any observations in simulation can be expected to be consistent with the proof of any properties relating to the robot and the environment. This is only possible if one can guarantee that the simulation and the proof use the same models, and that the error due to the solution of any behavioural equations of these models used in simulation is accounted for. Ensuring consistency between simulation and proof reduces the amount of testing required in the verification of a robotic system, thereby reducing costs. Examples where proof has been considered as part of the verification of a robotic system include surgical robots [9] and robotic ground vehicles [10].

Consistency is defined more precisely follows.

Definition 2. Consistency: A simulation and a proof of a property of a robotic system are consistent if the equations used in the model of the robotic system and its environment adopted in proof are the same as those used in simulation by the physics engine.

To establish and provide evidence of Consistency, it is necessary that the physics engine adopted satisfies three properties: transparency, configurability and extensibility. These properties are defined as follows:

- **Transparency**: Whether one can determine the equations and their solutions in the physics engine and to what level this is possible.
- **Configurability**: Whether one can modify the equations and their solutions used in a physics engine.
- **Extensibility**: Whether one can extend the equations and their solutions used in a physics engine.

It is useful to clarify the inter-dependency between these properties. Establishing consistency

between proof and simulation is the main goal of this work. Consistency requires configurability and extensibility, otherwise there would be equations that one could specify in the model but would not be able to animate at the simulation level. Transparency is not a necessary condition for consistency, however it is much more difficult to determine consistency without transparency e.g. by inferring the equations through a series of tests or through guarantees from the software developer of the simulator. Transparency is therefore treated with the same level of importance as the other two properties.

This thesis explores the current limitations of physics engines with respect to the verification of robotic systems using simulation and proof based upon the same set of equations, and presents novel solutions. This work aims to provide lower cost verification of robotic systems. It is also relevant to consistent model-based testing, where a model checker or theorem prover is used for automatic test generation.

The following example further motivates the need for consistency between proof and simulation.

Example 1.1: RoboStar Framework

The issue of consistency is illustrated by the following example, based on the use of RoboStar notations and techniques [11]. This framework provides domain-specific languages that can be used as part of the modelling, simulation, and verification of a robotic system in its environment. The domain specific language of the RoboStar framework relevant in this example is RoboSim. It allows the user to model a simulation diagrammatically in a platform-independent way, using a d-model (data-model), and to provide a **platform mapping** to a model of the robotic platform called the p-model (platform-model). Indeed the d-model is an example of the controller model as presented in this chapter. The environment model is not considered in this example, i.e. only the behaviour of a robotic system is of concern.

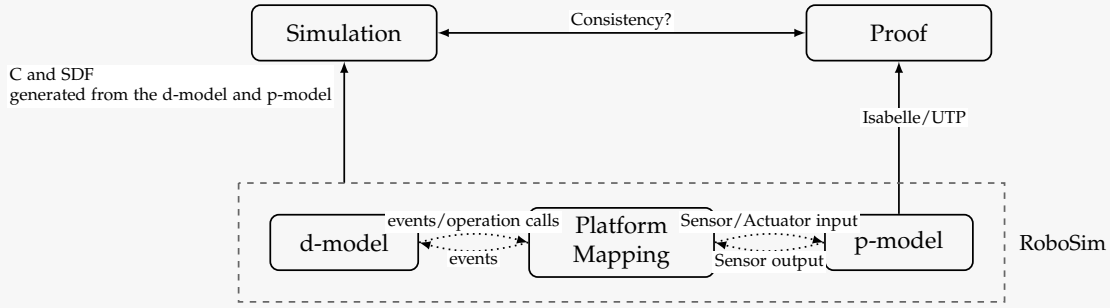


Figure 1.3: RoboStar Framework

Figure 1.3 illustrates what was possible using the RoboStar framework prior to the work presented in this thesis, demonstrating the challenges involved in establishing consistency between proof and simulation required for the verification of a robotic system.
With RoboSim terminology in place, Fig 1.4 provides a controller-centric view of the system model in Fig 1.2. The controller corresponds to the d-model, the robot dynamics correspond to the p-model, and their interface is the platform mapping; the environment model remains explicit in the dynamics block. This is a more concrete representation of the system model, for a specific controller, where the system can be viewed as a feedback loop. In general one could have several relationships between the components of the system model, not all of which are cyclic. There are also many architectures for which the controller is not viewed in this way [12] [13].

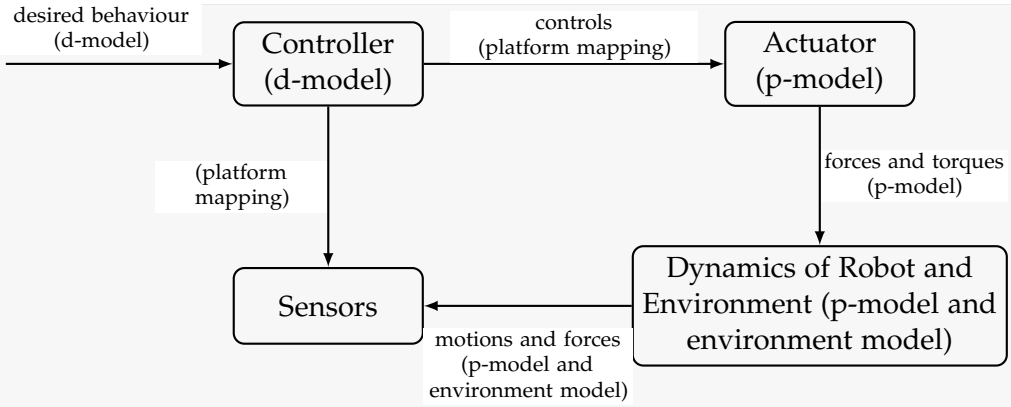


Figure 1.4: Feedback controller model, based on the presentation in [14].

The p-model provides a description of the robot including *some* of the underlying equations that need to be used in simulation and proof. Given a p-model, desired properties one would like to check are proven using a theorem prover such as Isabelle. Since robotic systems are hybrid, that is, they exhibit both continuous and discrete behaviour, proof of properties requires a logic that can be used to reason about both these forms of behaviour. Of interest in this thesis is the differential dynamic logic [15][16][17]. Once these properties have been proven, the behaviour guaranteed by proof must be observed in simulation. Conversely, if a behaviour is observed in simulation, and it holds for all possible simulation scenarios, it should be possible to prove that. All this is possible only if consistency between simulation and proof can be ensured.
In order to simulate the behaviour of a robot it is currently possible to use the p-model to automatically generate an XML-based file written using a standard called SDF [18] for use in simulation. SDF description captures information about the links, joints, sensors, and actuators. Several widely used robotics simulators accept an SDF description of the robotic platform as input.
The automatic generation of SDF is useful, as it allows to validate the p-model and lowers the cost of developing a simulation. In terms of ensuring consistency between simulation and proof-based verification, however, there are problems. The first problem one encounters is that SDF files do not include the behavioural equations of the p-model: the SDF notation does not provide support for capturing these equations. Instead the user is forced to use a combination of domain-specific physics-engine libraries and SDF files to simulate the desired behaviour. As a result, there is no way of determining the equations that are actually used, beyond, in some cases, reading the source-code of the physics engines or simulators. This is an issue not just for consistency with proof techniques, but for model-based robotics in general as it does not allow for configurability and extensibility without the user sufficiently understanding the library of the physics engine in the language it is written. Furthermore, this leads to a lack of transparency, whereby it is difficult to infer the correctness of equations without inspecting the source-code.

Throughout this thesis, the terms *robotic platform model* and *p-model*, *controller model* and *d-model*, and *controller-platform mapping* and *platform mapping* are used synonymously unless otherwise specified.

The behavioural equations of a robotic system can be specified for joints, links, sensors and actuators. Now consider using RoboSim for the pick and place example. In a p-model component-level equations can be written for joints, sensors and actuators. The sensor equations can, for example, capture probabilistic measurement models used in localisation. A running example is the beam-based range finder model from Probabilistic Robotics [19], where

each beam likelihood is modelled as a mixture of four causes (correct returns, unexpected obstacles, max-range failures, and random readings):

$$p(z_t^k \mid x_t, m) = z_{\text{hit}}\, p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}}\, p_{\text{short}}(z_t^k \mid x_t, m) + z_{\max}\, p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}}\, p_{\text{rand}}(z_t^k \mid x_t, m), \quad (1.1)$$

assuming conditional independence of beams. The implementation of this model in the PhysMod sensor library is discussed in Chapter 4.

The actuator equation could be [14]:

$$\tau = K_t \cdot i \quad (1.2)$$

where τ is the torque, K_t is the torque constant, and i is the current. The joint equations could be [20]:

$${}^{P}\mathbf{X}_B \equiv {}^{P}R_B(\theta)\ {}^{P}p_B(\theta) \quad (1.3)$$

$${}^{P}\mathbf{V}_B \equiv \begin{bmatrix} {}^{P}\omega_B(\theta, u) \\ {}^{P}v_B(\theta, u) \end{bmatrix} = {}^{P}H_B(\theta) \cdot u \quad (1.4)$$

$${}^{P}\mathbf{A}_B \equiv {}^{P}\dot{V}_B = {}^{P}H_B\dot{u} + {}^{P}\dot{H}_B u \quad (1.5)$$

$$\dot{\theta} = N(\theta)u \quad (1.6)$$

where ${}^{P}\mathbf{X}_B$, P, ${}^{P}\mathbf{V}_B$, ${}^{P}\mathbf{A}_B$, ${}^{P}\omega_B$, ${}^{P}v_B$, ${}^{P}H_B$, ${}^{P}p_B$, ${}^{P}R_B$ are the spatial position, spatial velocity, spatial acceleration, angular velocity, linear velocity, hinge matrix, translation vector, and rotation matrix of the child frame B with respect to the parent frame P respectively. θ is the generalised position coordinate and u is the generalised velocity coordinate. $N(\theta)$ is the kinematic coupling matrix.

However, as discussed in Chapter 2, whilst these equations do uniquely characterise the system at the component-level, it is not known how the equations are composed and solved at the system-level once the SDF has been generated and parsed by the physics engine. Indeed it is not possible to determine many of the important properties of a robotic platform as it evolves over time, such as the system-level equations (addressed in Chapter 2) and the solution to the equations of motion (addressed in Chapter 3). This implies that a RoboSim simulation is not transparent.

Now consider the case where the system-level variables for the dynamics of the robot were already known and could be written in a p-model. For example, the manipulator equation, which is commonly used to describe the dynamics of a robotic system, could be written.

$$\tau = M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + g(\theta) \quad (1.7)$$

Here the variables are acceleration $\ddot{\theta}$, torque τ, mass matrix $M(\theta)$, gravitational term $g(\theta)$, and Coriolis term $C(\theta, \dot{\theta})\dot{\theta}$. The equations of a joint can be composed into these system-level equations [21].

The primary purpose of a physics engine used for a robotic system is to calculate the evolution of these variables in this equation, alongside the variables in the equations of sensors and

actuators, which is called the **state**, over time. Note that the sensors and actuators play the role of both a component and system-level view of their equations.

The motivation in this thesis is to be able to write, simulate, and prove properties for any equations in a (rigid body) robotic system. Under this assumption the following could have equally been written:

$$L = T - U \tag{1.8}$$

where L is the Lagrangian, T is the kinetic energy, and U is the potential energy.

$$q = distance \tag{1.9}$$

$$\tau = \tau_{\text{in}} \tag{1.10}$$

.

Clearly these are very different ways of writing the equations and are at different levels of abstraction; making use of different representations and formulations. Furthermore, some of these are component-level equations and some of them are system-level equations. These issues are addressed more concretely by first developing the description of a robotic platform model in the next Chapter, and then testing these ideas in Chapter 4. Addressing this resolves the issues of configurability, and extensibility in the RoboStar framework.

1.2 Objectives

The overarching goal of this work is to enable consistency between simulation and proof for the verification of robotic systems. This goal is achieved through the following objectives.

1. To enable a modeller to modify and extend the equations in a physics engine at the model-level.

2. To enable the modeller to prove properties in a theorem prover, whilst knowing to what extent these properties will be observable within their simulation.

3. To provide the modeller with information about their equations such as whether or not they can be used by the physics engine.

These objectives are determined by considering the potential workflow that is useful for the modeller. The modeller should be able to use a physics engine that accepts their model as input and configures itself to be consistent with the model, or so that the the physics engine immediately extends its library to accommodate the model. The key point is that the modeller should not be expected to do anything with the source-code of the physics engine. The modeller should then be able to prove properties about their model that they can establish in simulation. Finally, if it were not possible for the modeller to configure or extend their equations at the model-level, the third objective ensures that the modeller is able to determine what is possible in the physics engine so that the modeller can consider how the model could be changed.

1.3 Contributions

The major contributions of this thesis are as follows:

1. Design and implementation of a simulation framework that maintains consistency with the complete RoboSim model, while automating the generation of transparent, configurable, and extensible physics-engine source code from p-models.

2. Extension of the p-model metamodel to support system-level equations and solutions, enabling the representation of system-level equations and their compositional relationship to component-level specifications already supported by the p-model.

3. The implementation of the framework across several rigid body dynamics formulations, and coverage of each kinematic topology (serial chains, trees, and closed loops).

4. Use of the framework to generate Isabelle proof data for the physical part of a cyber-physical model.

The additional minor contributions of this thesis are as follows:

1. The creation of slnRef and slnDF, enabling a platform-independent and transparent record of platform equations and their computational solution, beyond what other file formats such as SDF and URDF specify.

2. The prototype implementation of the mappingGen technique, which transforms the platform mappings used in the case studies into mapping constraints for downstream mapping-enabled code generation.

3. An extension of RoboTool to support the simulation framework, including formulation-specific equation definitions, a library of solution methods, and generation of C++ platform-engine code together with Isabelle definitions for the dataspace and equations of the physical model. The implementation is available in the RoboTool physics-modelling repository.

4. Generation of a simulation-orchestrator skeleton that coordinates the controller, platform engine, and mapping layer and provides integration points for scenario-specific environment/world code, which is supplied manually in source form because RoboSim currently has no environment-model language.

5. Evaluation of the framework through the Simple Arm, Acrobot, and Mobile Robot case studies, including Isabelle proofs that the Acrobot is underactuated and locally controllable at the upright equilibrium, a condition for LQR stabilisation.

1.4 Thesis Structure

This thesis is organised into five chapters and several appendices.

Chapter 2 provides the necessary background for this work. It introduces the RoboSim framework and its constituent models: the d-model for discrete controller behaviour, the p-model for physical platform properties, and the platform mapping that connects them. The chapter also presents an evaluation of existing physics engines, examining their capabilities

and limitations with respect to transparency, configurability, and extensibility.

Chapter 3 presents the core contribution of this thesis: a simulation framework comprising three techniques. Technique 3 (Equation Composition) takes a p-model and computes component and system-level equations for a chosen dynamics formulation, producing an extended p-model. Technique 4 (Guided Choice) accepts solution method selections from a library and unfolds them in the context of the p-model, producing solution references with all necessary inputs and constraints. Technique 5 (Code Generation) consumes the solution references and generates executable source code for target platforms including C++ and Isabelle/UTP.

Chapter 4 describes the implementation of the framework as an Eclipse-based toolchain and validates it through case studies. The case studies include a serial manipulator demonstrating forward dynamics generation, and an acrobot demonstrating controlled simulation with swing-up and balancing behaviour. The chapter also presents test coverage results and discusses the semantic alignment between generated code and the RoboSim operational semantics.

Chapter 5 summarises the contributions of this thesis and discusses directions for future work, including extensions to the p-model metamodel, additional solution methods, and enhanced code generation capabilities.

1.5 Glossary of Terms and Acronyms

Table 1.1: Glossary of terms, acronyms, and abbreviations.

Term or acronym	Definition or expansion
Behavioural equation	An equation that describes or constrains a robot's physical behaviour, including time-invariant relationships between physical quantities and the evolution of its state over time.
CGA	conformal geometric algebra
Co-verification	The combined use of simulation-based and proof-based verification.
Component-level equation	A behavioural equation attached to an individual link, joint, sensor, or actuator.
Configurability	The ability to modify the equations or solution methods used by a physics engine.
Consistency	The use of the same robot equations in simulation and proof.
Connectivity	The graph-like structural arrangement of a robotic platform, with physical components as nodes and fixed or flexible connections between them as edges.
CSP	Communicating Sequential Processes
CyPhyCircus	A hybrid, state-rich process algebra extending Circus to model reactive and continuous behaviour.
DAG	directed acyclic graph
DFS	depth-first search
D-model (controller model)	A model of platform-independent control logic and discrete data processing.
d-modelGen	The generation of source-code from a d-model.
DSL	domain-specific language
EBNF	extended Backus–Naur form

Table 1.1 (continued).

Term or acronym	Definition or expansion
EMF	Eclipse Modeling Framework
Environment mapping	The mapping from environment-model variables to corresponding physical properties in a robotic platform model.
Environment model	A model of the physical environment in which robotic platforms operate.
eqnComp	The composition of component- and system-level equations into a p-model+ for a selected dynamics formulation.
Equation library	A library of formulation-specific specifications used to generate equations.
Extensibility	The ability to add new equations or solution methods to a physics engine.
Forward and inverse dynamics	Forward dynamics computes accelerations from forces or torques; inverse dynamics computes the forces or torques required for a specified motion.
guidedChoice	The unfolding of solution-method selections into fully instantiated solution references.
HOL	higher-order logic
Hybrid-model	A RoboSim model composed of a d-model, p-model, and platform mapping.
IMU	inertial measurement unit
Kinematic topology	The structural arrangement of a platform's links and joints.
LCP	linear complementarity problem
LiDAR	light detection and ranging
LQR	linear–quadratic regulator
mappingGen	The prototype transformation implemented in this thesis for generating source-code from simple platform mappings.
Metamodel	A model that defines the abstract syntax of a modelling language.
MJCF	the MuJoCo XML model format
Model-level	The level at which models are changed without editing physics-engine source-code.
P-model (robotic platform model)	A model of a robotic platform's physical structure, sensors, actuators, and continuous behaviour.
P-model+	A p-model extended with equations, solution blocks, and supporting datatypes.
Physics engine	Software that evaluates physical equations to evolve a simulated state over time.
Platform engine	The component that computes robot dynamics from p-model+ equations.
Platform mapping (controller–platform mapping)	The mapping between d-model interfaces and p-model sensors and actuators.
Reality gap	The difference between simulated and physical-system behaviour.
RoboSim	A formal modelling language and framework for specifying, simulating, and verifying robotic systems in a tool-independent way.
Robotic platform	A physical robot system comprising links, joints, sensors, and actuators.

Table 1.1 (continued).

Term or acronym	Definition or expansion
Robotic system	The combination of a controller model, robotic-platform model, and platform mapping.
ROS	Robot Operating System
SDF	Simulation Description Format
Simulation	The execution of a controller–physical-system model over time as a computer program.
slnDF	A domain-specific language for describing executable solution implementations and their computation order.
slnRef	An implementation-independent reference specifying a target solution, method, ordering, inputs, and constraints.
Solution library	A repository of methods and algorithms for solving behavioural equations.
Solution method	The algorithm selected to compute a target expression.
Solution unfolding	The decomposition of a solution request into concrete dependent sub-solutions.
Source-code level	The level at which a physics engine is configured or extended through its source code, APIs, or supporting libraries.
State	The variables that describe a physical system at a given time.
SysML	Systems Modeling Language: a general-purpose graphical language for modelling systems.
System-level equation	A behavioural equation for the robotic platform as a whole.
System view	The highest level of abstraction of a robotic system, representing its platform as a single p-model block with sensor and actuator interfaces and optional system-level variables and equations.
Transparency	The ability to determine the equations and solution methods used by a physics engine.
URDF	Unified Robot Description Format
UTP	Unifying Theories of Programming
World engine	The component that manages environment entities and spatial queries.
World mapping	The mapping that computes sensor values from the world and platform states.
XML	Extensible Markup Language

2 Background and Physical Modelling

2.1 Physical Modelling

In this section, we discuss in detail the physical modelling requirements for a p-model. The p-model must be able to reflect system-level, local-level, and component-level physical properties of any rigid body robotic system. In the previous chapter we described what it means to be a robotic system: the high-level relationship between the continuous behaviour of the p-model, and the discrete behaviour of the d-model which was summarised in Fig. 1.2. This leads us to the following definition of a robotic platform which we refine throughout this chapter.

Definition 3 (Robotic Platform (1))**.** A **robotic platform** is a physical system which takes input and output to and from its sensors and actuators. These inputs and outputs map to the inputs and outputs of some control software. We associate the physical portion of a robotic system with a model of the robotic platform (the p-model).

In this chapter we first discuss the requirements of the p-model at the system-level, and then at the component-level, and local-level. Coincidentally, we introduce the metamodel and well-formedness conditions based on these requirements.

2.1.1 p-model Overview

A **p-model** is a SysML-like block diagram that provides a parametrised, modular, and hierarchical description of a robotic platform. The base p-model records the platform's physical components, their containment and connectivity, their poses, and the physical properties and behaviour associated with them. The RoboSim Physical Modelling Reference Manual and the p-model paper give the complete metamodel, notation, well-formedness conditions, and semantics [22], [23].

The root p-model block represents the platform as a whole and is the root of its containment hierarchy. It contains Links and may contain Parts. A Link represents a rigid element and can record inertial information in its definition. This information comprises mass, an inertia matrix, and an associated pose. A Link can also contain one or more Bodies, whose definitions can independently record inertial information, geometry (a box, cylinder, sphere, or mesh), contact properties (bounce and friction), and optical properties (laser reflectivity and transparency). Links are joined either directly by fixed connections, which permit no relative motion, or through Joints, which define flexible connections and their permitted motion. Sensors describe measured physical quantities and Actuators describe physical inputs; either may be contained by the relevant Link or Joint. Containment establishes the frame relative to which a component's Pose gives its position and orientation.

Each Link, Body, Joint, Sensor, or Actuator used in a p-model is a realisation: it either carries a local definition or refers to a reusable definition declared in a library. Definitions record the

relevant physical attributes and may declare input, output, bidirectional, local, and constant variables, together with equations and constraints. References to reusable Joint, Sensor, and Actuator definitions can instantiate values for open parameters. Constants support parameterised descriptions, while an Index compactly represents repeated elements. Annotations attach additional tool-oriented information to Joint, Sensor, and Actuator definitions or references without changing their mathematical behaviour.

A p-model package can contain p-models and platform mappings, together with reusable component definitions, constants, annotation templates, types, unit definitions, functions, and Fragments. Parts provide hierarchical reuse: a Part is a realisation of another p-model, with its own pose, optional index, parameter instantiations, and fixed or flexible connections to elements inside the incorporated p-model. A Fragment instead names a collection of Links, Joints, Sensors, and Actuators that does not itself form a p-model hierarchy. When a Fragment is used, its components and connections are incorporated directly into the containing p-model [22], [23].

Example 2.1: Simple Arm Inverted Pendulum: p-model and SKO formulation

This example introduces the p-model and formulation used for the Simple Arm inverted pendulum. Figure 2.1 shows its three Links, the two revolute Joints that connect them, and the components contained by those Links and Joints.
The root p-model, `SimpleArmSerial`, contains three local Link realisations: `BaseLink`, `IntermediateLink`, and `Gripper`. Each contains a local Body realisation with an inline definition that records its pose, mass, inertia matrix, and geometry: the `BaseLink` and `Gripper` Bodies are boxes, while the `IntermediateLink` Body is a cylinder. Thus, although the metamodel also permits inertial information to be recorded directly by a Link definition, this example records it in the contained Body definitions.
The other components are reference realisations. `BaseLink` contains `BaseSensor`, which refers to the reusable `TrivialSensor` definition, and `ElbowJoint`, which refers to `revolute_x`. `ElbowJoint` flexibly connects `BaseLink` to `IntermediateLink` and contains `ElbowActuator`, a reference to `TrivialMotor`. Similarly, `IntermediateLink` contains `WristJoint`, a reference to `revolute_z` that flexibly connects it to `Gripper`; the Joint contains `WristActuator`, which also refers to `TrivialMotor`. The notation *name* : *definition* in Figure 2.1 therefore identifies a reference component and the reusable definition that it realises.

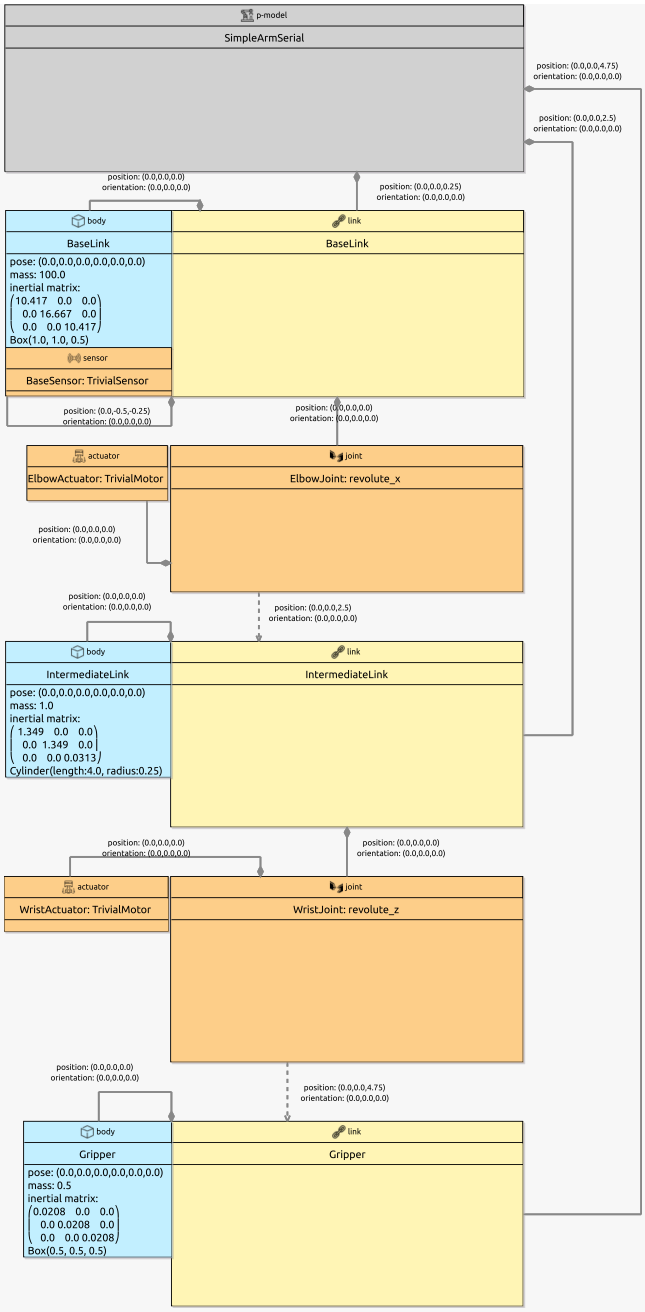


Figure 2.1: The Simple Arm serial-chain p-model.

Figure 2.2 shows the four reusable definitions referenced by these components. `TrivialSensor` has the input `SignalIn`, the output `MeasurementOut`, and an equation equating them. `TrivialMotor` similarly equates its `TorqueIn` input with its `TorqueOut` output. The `revolute_x` and `revolute_z` Joint definitions each provide a constant six-dimensional hinge vector, H, for rotation about the corresponding axis. These hinge vectors are specific to the spatial kernel operator (SKO) formulation [24], which is adopted here for didactic purposes because it clearly illustrates how system-level operators are composed from component-level equations.

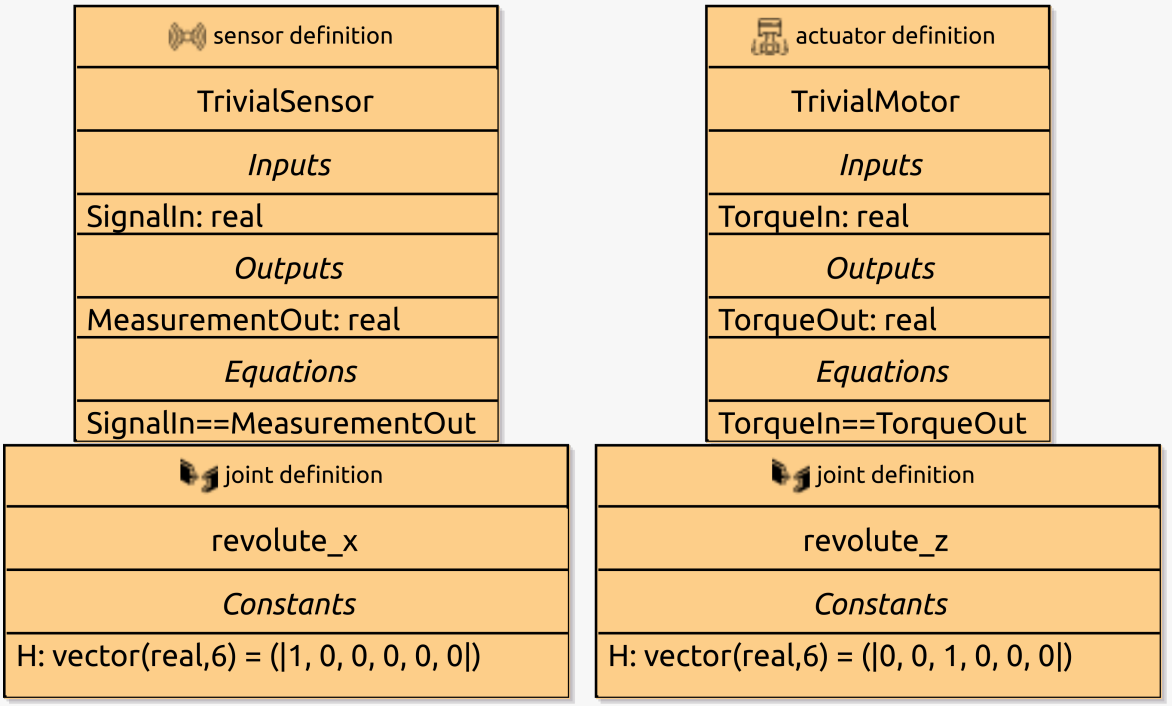


Figure 2.2: The joint, sensor, and actuator definitions used by the Simple Arm p-model.

This thesis uses a backward-compatible extension, called *p-model+*, that preserves this structural description while extending the mathematical information that it can express. It permits component-level equations to be attached uniformly to component definitions and system-level equations to be declared in the root p-model block. Supporting relations connect component quantities to those used at system level, while constraints, initial and boundary conditions, and vector and matrix types support the required mathematical statements. A component-level equation describes a physical relationship within an individual component; a system-level equation describes a relationship for the platform as a whole. In either case, an equation may express a time-invariant relationship or how physical properties and state change over time.

Equations and solutions have distinct roles. Equations state the physical relationships that the model must satisfy. A solution block records how a specified quantity is to be calculated, naming the method, its ordering and grouping, the required inputs, and any constraints. Thus, a solution records computational information about an equation without changing the physical relationship expressed by that equation. Section 2.1.3 presents the p-model+ metamodel additions, and Chapter 3 explains how the later framework uses them.

The remainder of this section first clarifies the mathematical meaning of the p-model's special physical variables. Section 2.1.2 then motivates the relationship between component- and system-level equations, before Section 2.1.3 presents the p-model+ additions.

Mathematical Clarification of p-model components

Whilst the mathematical relationship between most components in the p-model are general, there are some special variables which had a previously underspecified mathematical semantics, which have been included to make modelling easier. We extend the modelling approach of p-models by allowing the user to omit these special variables in favour of their own variables and constants. In particular, these include *inertial*, *geometry*, and *pose*.

The user may choose from different representations for these variables, and the code generator may generate the appropriate representation for the formulation specified by the user, as well as including the equations which relate different representations.

The following table summarises special variables in the p-model, and their mathematical semantics.

Variable	Semantics
pose	A 6D vector $[x, y, z, \phi, \theta, \psi]$ where x, y, z are position coordinates and ϕ, θ, ψ are Euler angles (ZXZ convention). Defines relative position/orientation of components with respect to parent. Link poses are defined relative to a world frame at origin.
inertial	A vector $[m, I_{xx}, I_{yy}, I_{zz}, I_{xy}, I_{xz}, I_{yz}]$ where m is mass, I_{xx}, I_{yy}, I_{zz} are moments of inertia, and I_{xy}, I_{xz}, I_{yz} are products of inertia. Defines mass and inertia properties of a component.
geometry	Geometry variables define the shape of a body for visualisation and collision detection. Supported primitives include `box(length, width, height)`, `cylinder(radius, length)`, and `sphere(radius)`. Dimensions are specified in the body's local reference frame: for a cylinder, `length` is along the local Z-axis and `radius` defines the cross-section in the XY plane; for a box, `length`, `width`, and `height` correspond to the X, Y, and Z axes respectively. The geometry is centred at the body's origin, with the body's pose defining its position relative to the parent link.

Table 2.1: Special variables in the p-model and their mathematical semantics.

2.1.2 Compositionality

The **system view** of the p-model is the highest level of abstraction of a robotic system. At this level, the robotic platform is represented by a single p-model block with (i) a set of sensor and actuator interfaces and (ii) optional system-level variables and equations capturing global behaviour.

Figure 2.3 illustrates a minimal system view in which the platform exposes sensors and actuators. Figure 2.4 extends this view with system-level variables and equations, for example the manipulator equation.

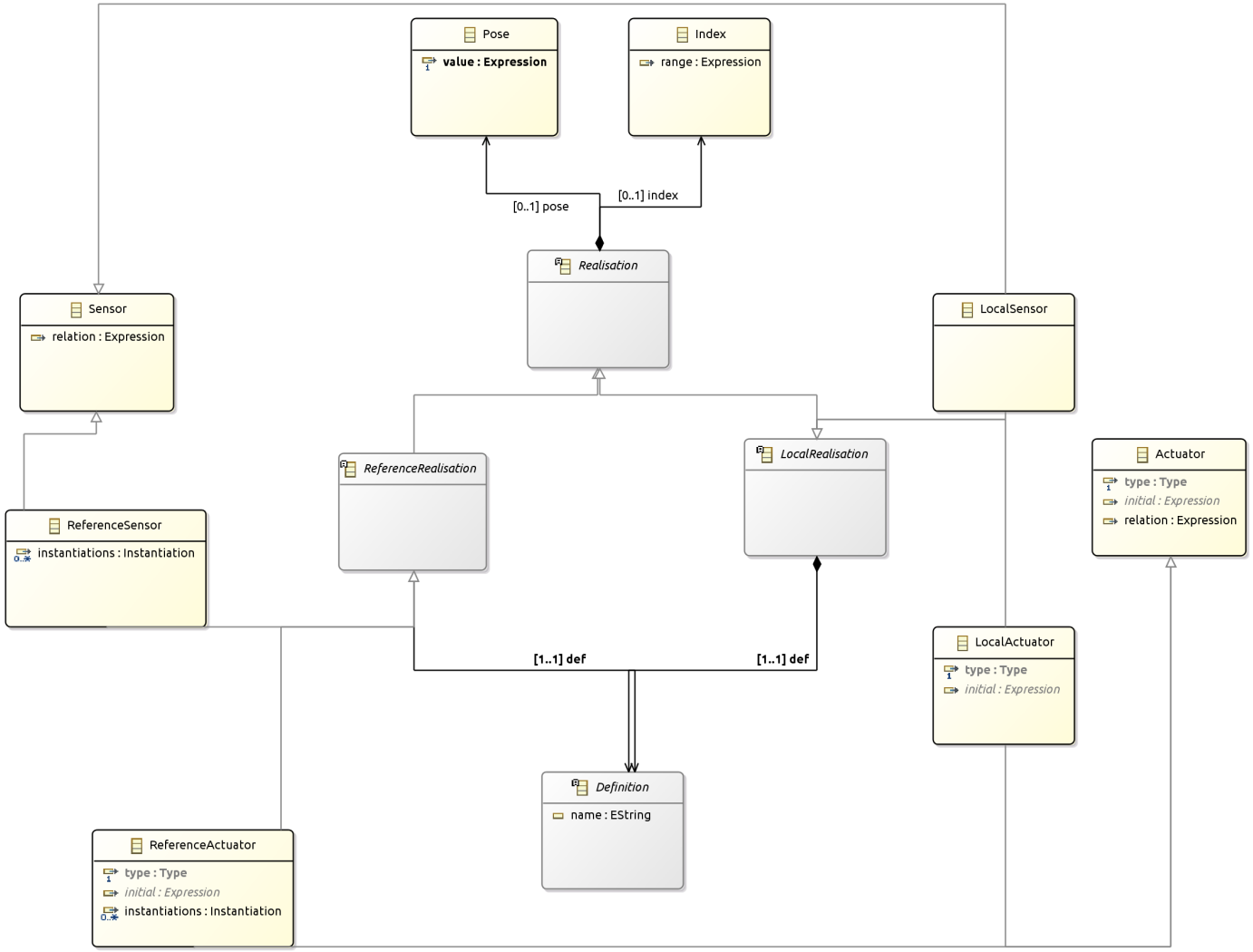


Figure 2.3: System view of a p-model showing the robotic platform with sensor and actuator blocks.

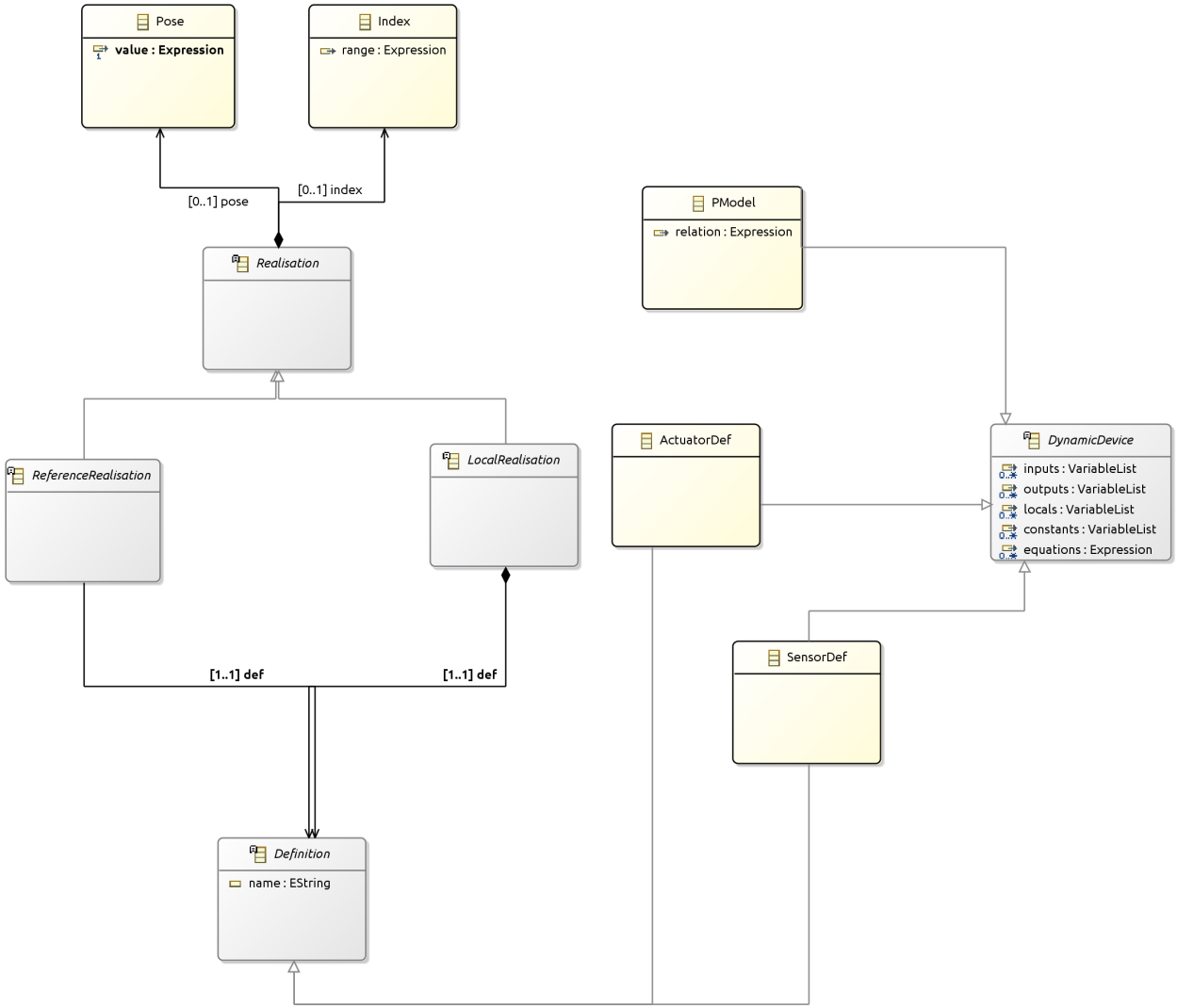


Figure 2.4: System view extended with system-level variables and equations (e.g., rigid-body dynamics).

A purely system-level presentation is useful for stating global properties, but it does not scale as the primary modelling interface: system-level dynamics depend on component properties (masses, inertias, joint geometry, sensor placement), and writing them monolithically obscures traceability from equations back to the physical structure. In practice, robotic platforms are authored compositionally from reusable component descriptions (e.g., links, joints, sensors, and actuators) and system-level dynamics are generated from these component models.

The **components** of a robotic platform are its links, joints, sensors, and actuators. Each component definition may introduce parameters, interface variables and local behavioural equations: rigid-body and inertial properties for links and bodies, kinematic constraints for joints, measurement models for sensors, and input–force/torque relations for actuators.

Compositionality requires that these component descriptions can be assembled into a single, well-defined set of equations for simulation while preserving the correspondence between component-level models and the generated system-level dynamics. This motivates the next section, where we extend the p-model so that equations and variables can be attached uniformly to p-models and to component definitions, supporting composition of dynamics, sensor and actuator equations and enabling automated equation generation and solution.

2.1.3 Extending the p-model

The simulation framework presented in Chapter 3 requires an extended p-model metamodel to support equation generation and solution specification. We refer to this extended version of the p-model as the p-model+, or PModel+ when referring to the metamodel type. The changes are strictly additive, so the metamodel remains backward-compatible with existing p-models and tooling, such as the SDF generator [23], while providing the infrastructure needed to generate physics engines from RoboSim models. Each figure below reproduces the relevant fragment from the metamodel Ecore presented in [23], with additions highlighted in blue.

The most significant extension is to the `PModel` class itself. In the original metamodel, `PModel` extended only `robochart::NamedElement`, providing naming but no support for system-level equations or variables. The extended metamodel updates this inheritance hierarchy so that `PModel` also extends `DynamicDevice` and `Solutions`. This enables system-level equations, variables, and solution strategies to be declared directly in the p-model, which is essential for representing the manipulator equation and other system-wide dynamics.

The `Definition` class, the abstract base for all library definitions (`LinkDef`, `BodyDef`, `JointDef`, `SensorDef`, `ActuatorDef`), now inherits from `DynamicDevice`. This aligns `LinkDef` and `BodyDef` with the earlier treatment of `JointDef`, so every definition can carry equations and variables. That uniformity supports composition by allowing component-level equations to be combined into system-level dynamics.

```
class PModel extends robochart::NamedElement,DynamicDevice,Solutions
{
   property parts : Part[*|1] { composes };
   property links : Link[*|1] { composes };
}

abstract class Definition extends DynamicDevice
{
   attribute name : String[?] = '';
}

class LinkDef extends Definition,DynamicDevice
{
   property inertial : Inertial[?] { composes };
}

class BodyDef extends Definition,DynamicDevice
{
   property geometry : Geometry[?] { composes };
   ...
}

class JointDef extends Definition,DynamicDevice,AnnotatedElement;
```

Figure 2.5: Extended `PModel` and `Definition` class hierarchy.

The `DynamicDevice` class itself now includes two additional properties. `InOuts` captures bidirectional variables that are both read and written during simulation, such as state variables updated at each step. `constraints` records algebraic constraints that must hold throughout simulation and complement the differential equations of motion.

```
abstract class DynamicDevice
{
   property inputs : robochart::VariableList[*|1] { composes };
   property outputs : robochart::VariableList[*|1] { composes };
   property InOuts :  robochart::VariableList[*|1] { composes };
   property locals : robochart::VariableList[*|1] { composes };
   property constants : robochart::VariableList[*|1] { ordered composes };
   property equations : robochart::Expression[*|1] { composes };
   property constraints :  robochart::Expression[*|1] { composes };
}
```

Figure 2.6: Extended `DynamicDevice` with InOuts and constraints.

A key addition for relating physical components to system or controller variables is the introduction of `relation` properties on the realisation classes. The `Link` class now includes both `relation` and `relation_flexi`, while `Body` and `Joint` now include `relation`. These properties extend the correspondence mechanism previously available on sensors and actuators, ensuring that component-level quantities can be linked to system-level variables during equation composition.

```
class Link extends Realisation,robochart::NamedExpression
{
   property fixed : FixedConnection[*|1] { composes };
   property bodies : Body[*|1] { composes };
   property sensors : Sensor[*|1] { composes };
   property actuators : Actuator[*|1] { composes };
   property joints : Joint[*|1] { composes };
   property relation :  robochart::Expression[?]  { composes };
   property relation_flexi :  robochart::Expression[?]  { composes };
}

class Body extends Realisation,robochart::NamedExpression
{
   property relation :  robochart::Expression[?]  { composes };
}

class Joint extends Realisation,robochart::NamedExpression
{
   property flexible : FlexibleConnection[?] { composes };
   property sensors : Sensor[*|1] { composes };
   property actuators : Actuator[*|1] { composes };
   property relation :  robochart::Expression[?]  { composes };
}
```

Figure 2.7: Relation properties on `Link`, `Body`, and `Joint` for system correspondence.

The extended metamodel introduces a new `Solutions` infrastructure to support the guided choice technique (T4). The abstract `Solutions` class aggregates `Solution` entries and is now inherited by `PModel`. Each `Solution` records the numerical method, order, and group, together with the variable to be solved (`solutionExpr`), its inputs, and any constraints. The `solutionVariable` class extends `robochart::NamedElement` and provides typed variables for use within solutions. This infrastructure makes the solution strategy explicit and supports correct and efficient simulation code generation.

```
abstract class Solutions
{
   property solutions :  Solution[*|1] { composes };
}

class Solution
{
   property solutionExpr :  solutionVariable[1] { composes };
   attribute method :  String[1];
   attribute order :  ecore::EInt[1];
   attribute group :  ecore::EInt[1];
   property inputs :  solutionVariable[*|1] { composes };
   property constraints :  robochart::Expression[*|1] { composes };
}

class solutionVariable extends robochart::NamedElement
{
   property type :  robochart::Type[1] { composes };
}
```

Figure 2.8: Solutions infrastructure for encoding numerical methods.

Three new expression types support mathematical constructs required by the simulation framework. The `IBCondition` class captures initial and boundary conditions, `SubMatrixExp` provides matrix extraction, and `AdjointExp` represents adjoint operations. They operate alongside the existing `PoseVariable` and `PoseExp` classes for pose quantities and the existing `Name`, `SName`, and `QName` classes for simple and qualified name expressions.

```
class IBCondition extends robochart::Expression
{
   property variable :  robochart::Expression[1] { composes };
   property exp :  robochart::Expression[1] { composes };
   property value :  robochart::Expression[1] { composes };
}

class SubMatrixExp extends robochart::Expression
{
   property value :  robochart::Expression[1] { composes };
   property rowStart :  robochart::Expression[1] { composes };
   property colStart :  robochart::Expression[1] { composes };
   property numRows :  robochart::Expression[1] { composes };
   property numCols :  robochart::Expression[1] { composes };
}

class AdjointExp extends robochart::Expression
{
   property value :  robochart::Expression[1] { composes };
}
```

Figure 2.9: New expression types for mathematical constructs.

The type system has also been extended with two abstract type classes. `AbstractVectorType` and `AbstractMatrixType` both extend `robochart::Type` and carry a base element type. These additions are essential for the code generation technique (T5), which must preserve typing for vector and matrix operations in the target language.

```
class AbstractVectorType extends robochart::Type
{
   property base :  robochart::Type[1] { composes };
}

class AbstractMatrixType extends robochart::Type
{
   property base :  robochart::Type[1] { composes };
}
```

Figure 2.10: New type classes for vector and matrix types.

Example 2.2: Simple Arm Inverted Pendulum: p-model+

In the root p-model block, the additions from p-model+ shown in this example are in/out and local-variable declarations, system-level equations and constraints, and solution blocks. The equations and constraints also use the added expression-language support for adjoint and submatrix operations and initial conditions. Figure 2.11 presents selected entries from the Simple Arm model introduced in Example 2.1. Each equation states a physical relationship, whereas each solution identifies a quantity to calculate and records its method inputs and constraints.

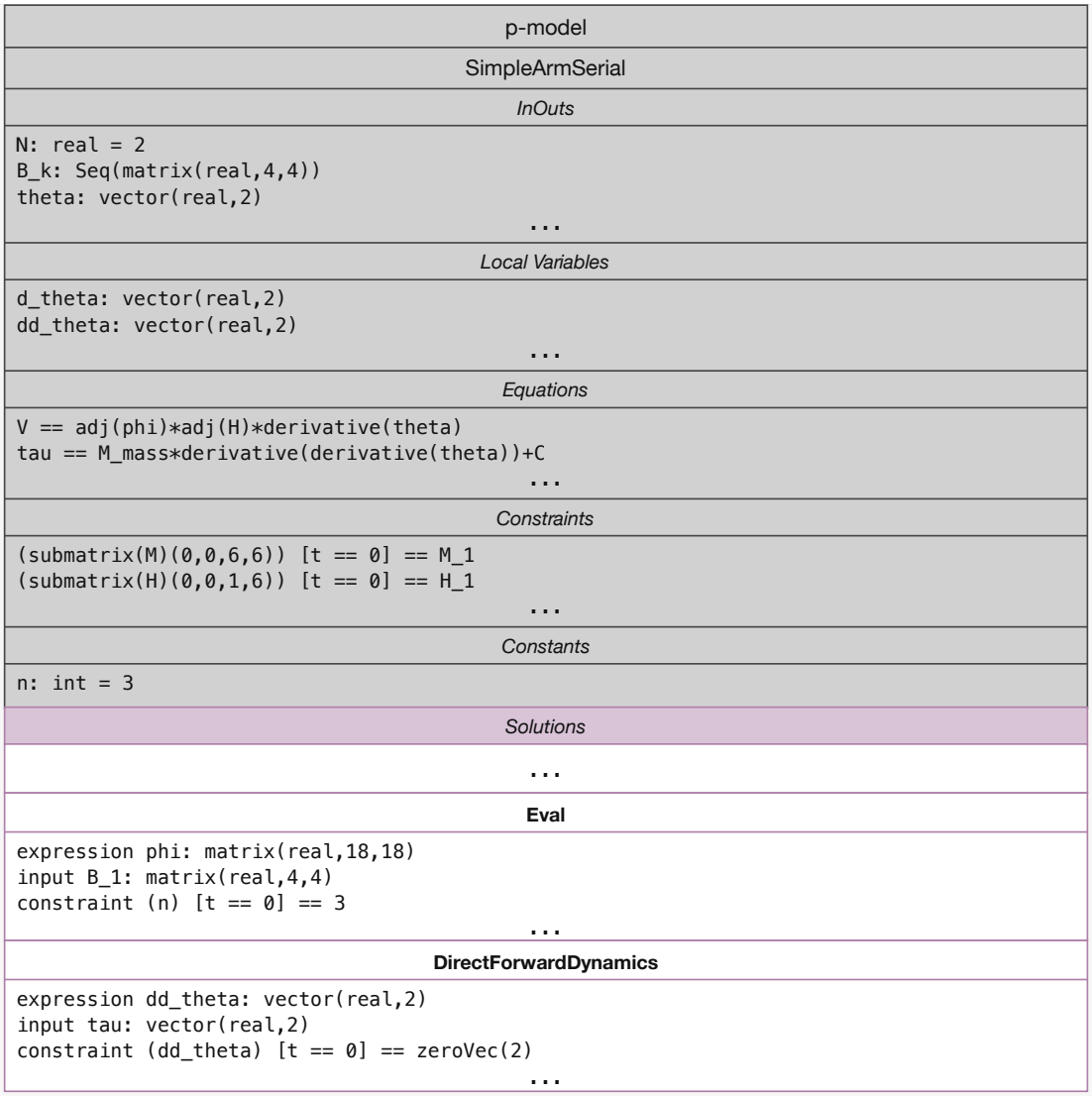


Figure 2.11: P-model+ content for the Simple Arm.

In addition to these major extensions, the `BodyDef.geometry` property is now optional (cardinality `[?]`) rather than required (`[1]`). This change recognises that not all body definitions require explicit geometry specification, particularly when the body is used purely for dynamic calculations rather than visualisation or collision detection.

The p-model metamodel also retains several legacy properties that are not used by the generators presented in Chapter 3. Specifically, the `Limit`, `Dynamics`, and `Bounce` properties on joints and bodies were inherited from robot description formats such as SDF but contradict the generality of the p-model we wish to support. These properties encode physics-engine-specific parameters that should instead be derived from the mathematical formulation of the dynamics equations. The generators in Chapter 3 ignore these properties, and they remain candidates for removal in future work to maintain the p-model's role as a physics-engine-agnostic specification.

The validator and type provider have also been updated to support the extended metamodel. The validator now permits vector types in derivative expressions, allowing expressions such as derivatives of vector-valued state variables required for system-level dynamics. The type provider introduces compatibility between `AbstractVectorType` and `VectorType`, and also treats row vectors ($1\times N$ matrices) as compatible with vectors of the same size, enabling

flexible use of vector and matrix representations in dynamics equations.

2.2 RoboSim

RoboSim is a formal modelling language and framework for specifying, simulating, and verifying robotic systems in a tool-independent way. It supports the joint modelling of discrete control software and continuous physical dynamics, together with the interfaces between them [23], [25].

As introduced in Chapter 1, RoboSim follows a *controller–platform* separation. The controller is modelled platform-independently, against an abstract interface of sensors, actuators, and services. The platform is modelled separately, capturing the mechanism, sensors, and actuators that realise that interface. An explicit mapping connects the two, documenting how controller-level inputs and outputs are interpreted at the platform level. Throughout the thesis we use *controller model* and d-model, *platform model* and p-model, and *controller–platform mapping* and platform-mapping synonymously, unless stated otherwise.

A complete RoboSim model is therefore a **hybrid-model** obtained by composing a d-model (controller), a p-model (platform), and a platform-mapping (interface realisation). Concrete instances of all three artefacts appear in the Chapter 4 case studies: Simple Arm pick-and-place in Section 4.3.1, Acrobot in Section 4.3.2, and mobile robot obstacle avoidance in Section 4.3.3.

The formal semantics of this hybrid model is defined using CSP-family notations: timed CSP (via `tock` events) for the discrete controller, and **CyPhyCircus** for the platform model, which combines CSP-style interaction with continuous evolution. In Chapter 2 we present only the semantic ingredients that are needed later in the thesis (in particular, the per-cycle interaction between controller and platform). The correspondence between these semantic concepts and the generated runtime is discussed separately in the tool implementation chapter.

The **d-model** (also called a *controller model* or *data model*) captures the platform-independent control logic of the robotic system: the state machines, control-cycle structure, and discrete data processing that decide which services to invoke and which commands to issue at each cycle. A d-model is written against an *abstract* platform interface (sensors, actuators, operations, and events), and therefore does not commit to a particular platform realisation or simulator [25], [26].

In Chapter 4 we give concrete d-model examples used with generated simulations: the Simple Arm pick-and-place controller in Section 4.3.1, the Acrobot controller state machine in Section 4.3.2, and the mobile robot obstacle-avoidance controller in Section 4.3.3.

The **p-model** (also called a *robotic platform model* or *platform model*) captures a platform realisation of the robot: its structural elements (such as links and joints), its sensors and actuators, and the continuous-time behaviour that constrains how the platform state evolves. RoboSim specifies p-models diagrammatically using block diagrams enriched with parameters and differential/algebraic equations, supporting modular descriptions of platforms and parts [22], [23].

Examples of p-models used in the Chapter 4 case studies are the Simple Arm platform in Section 4.3.1, the Acrobot mechanism in Section 4.3.2, and the mobile robot platform in Section 4.3.3.

The **platform mapping** (called *controller–platform mapping* in Chapter 1) is the interface that connects a d-model to a p-model. It records how each element of the controller's abstract platform interface (variables, events, and operations) is realised by the sensors and actuators of a particular platform model. In RoboSim, the platform mapping is a separate artefact that can be changed independently, enabling a single d-model to be retargeted to different platforms that provide the same abstract interface [23], [25].

In the Chapter 4 case studies, the platform mapping links the controller I/O of each d-model to the corresponding sensor/actuator quantities in the chosen p-model (for instance, mapping Acrobot joint measurements to the controller's sensor update event, and mapping the controller's torque command to the actuator input in Section 4.3.2).

2.2.1 RoboSim Semantics

The formal semantics of RoboSim is defined using a CSP-based semantic account of cyber–physical models. The controller (d-model) is given a timed CSP semantics (using `tock` events to represent discrete time), while the platform (p-model) is given a CyPhyCircus semantics that combines CSP-style interaction with continuous evolution. This section gives a lightweight overview of the semantic structure that is used throughout the thesis; detailed rules are presented in [23][27][17].

A complete RoboSim model consists of three artefacts (as introduced in the previous section): the d-model, the p-model, and the platform-mapping. At the semantic level, the overall system is captured by a parallel composition that synchronises controller and platform on their shared communication events:

$$System \triangleq DModel \parallel [|\{registerRead, registerWrite, tock\}|] PModel$$

In a closed model, these events are typically hidden from the external environment; they serve to coordinate the internal simulation cycle.

Operationally, the d-model interacts with the platform by reading the current sensor snapshot via `registerRead`, issuing actuator commands and operations via `registerWrite`, and advancing the controller clock through one or more `tock` events per control cycle.

The CyPhyCircus semantics structures the platform process as an initialisation followed by a repeated control cycle. Each cycle publishes a sensor snapshot to the controller (`SendToDModel`), accepts commands and operations from the controller (`ReceiveFromDModel`) until the end of the controller cycle (`Wait(cycle)`), then resets a local timer and continuously evolves the platform state for the cycle duration (`Evolve`). This cyclic structure is captured by the main action below.

$$\begin{array}{l} \bullet Init; \\ \quad \mu X @ \\ \qquad SendToDModel; \\ \qquad ReceiveFromDModel \triangle Wait(cycle); \\ \qquad timer := 0; \\ \qquad Evolve \triangle (timer \geq cycle \times tscale); \\ \qquad X \end{array}$$

The constants `cycle` and `tscale` relate the controller's discrete time (counted in `tock` events) to continuous time: the platform evolves until `timer` reaches `cycle × tscale`. In accordance with the simulation paradigm, `SendToDModel`, `ReceiveFromDModel`, and

`Wait(cycle)` take no continuous time; only `Evolve` advances the platform clock.

The channels **registerRead** and **registerWrite** define the synchronous communication protocol between the controller and platform. Informally, `registerRead` transfers a per-cycle sensor snapshot from p-model to d-model, and `registerWrite` transfers controller outputs (events, variable updates, and operation calls) from d-model to p-model.

2.3 Evaluation of Physics Engines

As mentioned in Chapter 1, the motivation of this work is to provide support to establish consistency between proof and simulation in robotics. The purpose of this chapter is to determine to what extent this is already possible, as well as to determine the *coverage* of physics engines with respect to fields of study relevant to robotics.

This chapter begins by outlining the criteria of assessment for physics engines, before outlining further preliminaries required for the evaluation. Subsequently, physics engines are evaluated based upon the criteria of assessment. Finally the evaluation is summarised through a discussion and conclusion including future work.

2.3.1 Criteria of Assessment

Now the criteria to evaluate physics engines to support consistency between proof and simulation in robotics is presented. In this thesis, a physics engine is responsible for computing a robot's continuous-time physical evolution (e.g., kinematics, dynamics, contact, and numerical integration).

To assess candidate engines, the criteria of transparency, configurability, and extensibility introduced in Chapter 1 are applied and interpreted here in terms of what must be evidenced to justify consistency between proof and simulation. In particular, the evaluation focuses on whether an engine's computational semantics can be inspected and, if necessary, aligned with a formal model.

In this chapter, transparency is assessed by the availability of artefacts that make an engine's modelling assumptions and numerical procedures traceable (e.g., documentation of implemented equations and algorithms, and practical access to the relevant implementation via source code or inspectable SDK components). Configurability is assessed by recording which modelling choices can be selected without modifying the engine: this is considered both at the model level (what can be expressed in robot/world descriptions) and at the source-code level (what can be selected via APIs and supporting libraries). Within each interface, configurability of numerical and physical parameters, of structural choices (e.g., state/geometry representations and interfaces), and of algorithmic choices (e.g., integrators, solvers, and dynamics/contact formulations) is recorded.

Extensibility is assessed by how readily an engine can be adapted when a required modelling construct, representation, or algorithm is not already available. Concretely, explicit extension mechanisms (such as plugins, callbacks, or well-defined subsystem boundaries) that support adding or replacing components with minimal invasive modification are sought, since verification often motivates alternative solvers or representations beyond an engine's defaults.

Robotics draws on a wide range of mathematical abstractions, and these abstractions evolve over time; for example, geometric formulations of soft-robot dynamics have introduced representations and methods that differ substantially from classical rigid-body pipelines [28], [29]. To capture this breadth, a fourth criterion is introduced: *coverage*. Coverage is considered along two axes: the mathematical structures supported for representing and manipulating physical quantities, and the robotics-relevant fields of study supported by the engine or exposed through its interfaces. A narrowly scoped engine may be easier to relate to a proof, but wide applicability in robotics is also important; moreover, limited coverage can force workarounds that undermine transparency, configurability, or extensibility.

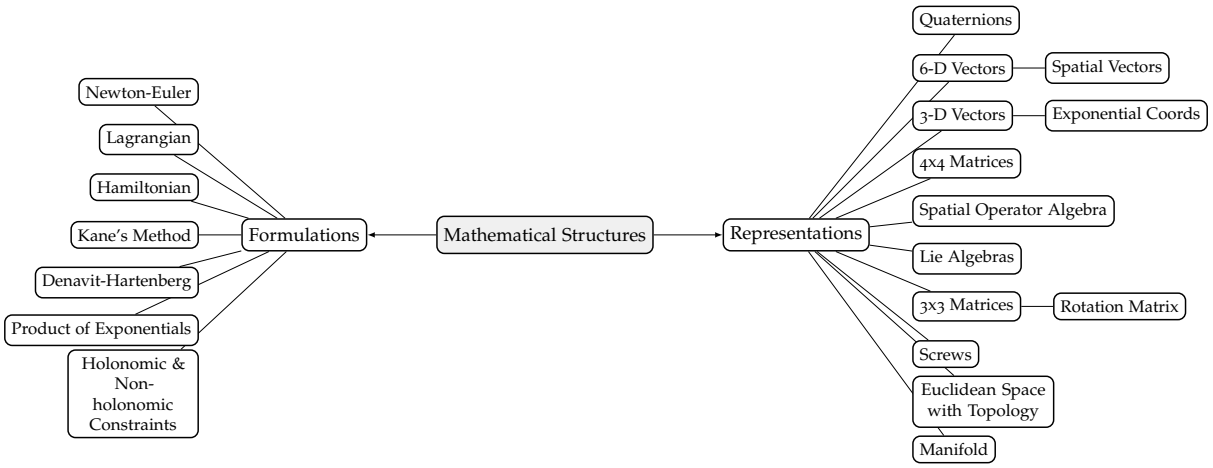


Figure 2.12: Examples of Mathematical Structures used in Robotics

Mathematical *structures* are used here in Bourbaki's broad sense (e.g., topology, metric, measure, algebra, geometry, vector space) [30]. In this thesis, a distinction is made between *representations* and *formulations*; Featherstone's treatment of spatial-vector notation and recursive dynamics algorithms is a useful point of reference for this distinction [31]. In this thesis, representations include structures that are primarily spatial (i.e., different ways of writing the same underlying geometric quantity), whereas formulations include both spatial and methodological abstractions (i.e., frameworks used to derive or structure computations). For example, in robot kinematics, pose may be represented using rotation matrices or quaternions (representations), whereas the Denavit–Hartenberg convention provides an intermediary method for structuring kinematic computation (a formulation). Figure 2.12 illustrates a non-exhaustive set of example structures that commonly arise in robotics.

Fields of study refer to the more detailed modelling domains that are not general enough to be treated as mathematical structures, but which materially affect the physical behaviour predicted by a simulator. In this thesis, these include kinematics, dynamics, contact and friction modelling, numerical methods, optimisation, and the modelling of physical components (e.g., sensors, actuators, links, and joints). Environment simulation may additionally draw on other fields (e.g., fluid dynamics and thermodynamics), but the evaluation in this chapter focuses on the domains most central to rigid-body robotics.

In the evaluation that follows, model-level configurability is assessed primarily through the XML-based robot description formats most commonly used in robotics simulation stacks: URDF, SDF, and MJCF (see Section 2.3.3).

The criteria above are applied at both the model level (what can be specified in robot/world descriptions) and the source-code level (what can be accessed or replaced via APIs). Throughout the evaluation tables in this chapter, a checkmark indicates that a capability is explicitly

supported at the relevant interface, a cross indicates that it is not supported, and a question mark indicates that the status is unclear or undocumented. To make the evaluation reproducible, each checkmark is backed by a concrete documentation or API reference (recorded in the appendix reference-location tables).

2.3.2 Preliminaries

A physics engine determines the relationship between positions, velocities, accelerations, and forces for all objects within a simulation at a given time step. In robotics simulation, this computation can be viewed as the platform-level procedure that, given the current state and inputs, evaluates kinematics and dynamics, resolves contacts and constraints, and then advances the state using a chosen numerical update rule. In practice, physics engines are organised into constituent parts; kinematics, rigid-body dynamics, collision detection with contact and friction modelling, numerical methods and optimisation routines, and the modelling of physical components such as joints, links, actuators, and sensors. This mirrors how robotics modelling decomposes physical problems [32].

The modelling domains that a simulation tool may cover are extensive. To keep the comparison focused and comparable across engines, the coverage assessment in this chapter concentrates on terrestrial rigid-body robotics, since many other robot domains build upon the same kinematic and rigid-body foundations while introducing additional effects. Accordingly, this evaluation does not consider aerial, soft, or underwater robots in detail, and it does not address rendering or ray-tracing (which are relevant to vision and environment modelling, but are left to future work on verification of environment models). Coverage also does not include the use of learned or identified real-world data to approximate physical properties, since the emphasis here is on explicit, equation-based modelling capabilities.

Within this scope, the fields of study considered in the evaluation are kinematics, dynamics, contact and friction modelling, numerical methods, optimisation, and component modelling. Some component models (notably sensors and actuators) are often provided by a wider simulation stack rather than by the physics engine alone. In this thesis they are treated as part of the physics engine insofar as they contribute equations whose solutions influence the simulated physical behaviour, while recognising that they are peripheral to the core functionality of dynamics and contact modelling.

Kinematics concerns the relationship between joint coordinates and poses. Forward kinematics computes poses given joint positions, whereas inverse kinematics computes joint positions that realise a desired pose or task-space constraint. Robotic mechanisms are most commonly represented in simulators as a kinematic tree. Kinematics algorithms in physics engines typically traverse this tree structure, propagating transforms through successive links to compute poses (and, when required, associated Jacobians) [33].

Dynamics algorithms seek to solve equations of motion in terms of forces, torques, and accelerations. For articulated rigid-body systems, the most widely used approaches in simulation engines are recursive algorithms, such as Featherstone's articulated-body method for forward dynamics and the recursive Newton–Euler algorithm for inverse dynamics [34][35]. Alternative formulations, such as Euler–Lagrange or Kane's methods, are also used in robotics modelling and can be advantageous when expressing constraints or deriving models analytically. However, they are less common in real-time engines because they often require differentiation and lead to dense expressions that are more expensive to evaluate numerically [31].

Building upon dynamics, contact and friction modelling captures how bodies interact with each other and with the environment. Collision handling couples geometric processing with a behavioural model of interaction forces. Engines typically separate this into a broadphase (identifying potentially colliding pairs) and a narrowphase (computing contact points, normals, and associated constraint data). The resulting contact laws and friction constraints are then solved using numerical methods, and are often reduced to constraint problems ranging from penalty methods to complementarity-based models [36].

Numerical methods and optimisation cover supporting topics such as numerical linear algebra, differentiation, time integration, differential equations, and constraint solving [37][38]. These methods appear throughout the pipeline (e.g., in integrators and contact solvers) and are often separated into distinct modules or libraries to enable clearer configuration and replacement of solvers. Although these areas determine many practical properties of simulation (including stability and accuracy), numerical errors and convergence guarantees are rarely made explicit in the user-facing abstractions provided by mainstream physics engines.

Finally, component modelling for robots is often the area with the least support in general-purpose engines. The components modelled are context-dependent but commonly include equations for sensors, motors, links, and joints. The parameters and constitutive assumptions embodied in these models can have a significant effect on the reality gap; for example, actuator models that include reflected inertia induced by motor gear ratios can materially change predicted system behaviour [39][40].

2.3.3 Robot Modelling Languages and File Formats

Robot simulation stacks typically combine a robot modelling language or description format that encodes kinematics, dynamics, and semantics, alongside asset file formats (meshes, materials, textures) referenced by the model for visualisation and collision geometry. This distinction matters for transparency and reproducibility: even if the kinematics are represented identically, different mesh formats, material conventions, or importer/exporter defaults can change geometry, collision approximations, and rendered appearance [41].

URDF (Unified Robot Description Format) is an XML-based format widely used in the ROS ecosystem to describe rigid-body robot mechanisms [42], [43]. A URDF model graph is expected to be a tree of links connected by joints (each link has at most one parent), which makes kinematic traversal straightforward but prevents direct representation of closed kinematic loops [44], [45]. URDF focuses on the robot mechanism itself: links may include inertial properties and collision/visual geometry, while simulator-specific aspects such as friction tuning, sensors, and controllers are commonly added via extensions rather than the core schema [43], [46], [47].

SDF (Simulation Description Format) is an XML-based format developed for Gazebo that can represent not only robots but also full simulation worlds including models, physics configuration, scene elements, and plugins [18], [48], [49]. Unlike URDF's tree restriction, SDFormat's kinematics are defined over a general kinematic graph, allowing closed kinematic loops [45], [50]. SDF is designed to describe sensors, actuators, and world-level elements in a single simulator-oriented file [49]. SDF also includes an explicit plugin mechanism, where plugins are documented as dynamically loaded code with parameters [51]. Where non-rigid phenomena are needed (e.g. deformables), they typically enter the ecosystem via simulator- or consumer-specific extensions and tags (rather than a single, portable core representation) [52].

MJCF (MuJoCo XML) is MuJoCo's native XML-based modelling language and is designed to provide access to MuJoCo's modelling and computation capabilities while supporting a scalable authoring workflow [4], [53], [54]. MJCF supports closed-chain mechanisms via explicit constraint constructs such as equality constraints (for example, weld constraints between bodies), and it provides fine-grained control over contact and collision behaviour through parameters such as `contype` and `conaffinity` [55]. MJCF embeds actuation constructs directly in the model, including flexible actuator definitions via the `general` actuator element and related shortcuts [53]. MuJoCo additionally supports extensibility via defaults/classes (parameter inheritance) and user-authored engine plugins [56], [57].

OpenUSD (Universal Scene Description) is a general-purpose scene description framework with a composition model based on layers and composition arcs (including sublayers, references, and variants) for non-destructive aggregation and overrides [58], [59]. For simulation, the `UsdPhysics` schemas define a rigid-body physics representation including constraints (joints) between rigid bodies and collision attributes (e.g. via `UsdPhysics.CollisionAPI`) [60], [61]. The initial scope of `UsdPhysics` concerns rigid-body simulation, and deformable/soft-body representations are typically provided via additional schemas or engine-specific extensions [60], [62]. OpenUSD is designed to be extended via custom schemas (and accompanying generated code) to add new domain semantics [63].

Simscape Multibody provides a block-diagram modelling environment for 3D multibody systems within MATLAB/Simulink, where bodies are connected via joint and constraint blocks and can be augmented with force elements and sensors [64], [65]. Simscape Multibody supports both kinematic trees and closed kinematic loops (closed-chain mechanisms) [66]. Contact interactions can be modelled using blocks such as Spatial Contact Force [67]. Simscape Multibody also supports flexible-body modelling via reduced-order flexible solids [68]. Beyond the block library, MATLAB provides the Simscape language for authoring custom physical components with equations, parameters, and conserving ports, enabling user-defined extensions to the modelling vocabulary [69].

Robot description files typically reference external assets for visual and collision geometry. Common mesh and asset formats include STL, a simple triangular surface mesh format widely used for collision geometry and rapid prototyping but which does not standardise appearance attributes such as materials or textures [70]; Wavefront OBJ with MTL, a widely supported format for polygonal surface geometry where material appearance is commonly provided via a separate MTL file [71], [72]; COLLADA (`.dae`), an XML-based digital asset schema that can encode scene hierarchies and visual assets [73]; glTF 2.0, a Khronos standard designed as an efficient runtime asset delivery format including physically based rendering (PBR) material descriptions [74], [75]; and USD layer formats, where OpenUSD layers can be stored as human-readable text (`.usda`) or a binary crate format (`.usdc`) [76], [77]. Simulator support varies: for example, Gazebo Classic expects mesh files in STL, COLLADA, or OBJ formats (with COLLADA and OBJ often preferred for visual fidelity) [41].

Table 2.2 summarises coverage of common modelling concerns (closed chains, soft/flexible bodies, collisions/contact, sensors/world constructs) and extensibility mechanisms provided by each approach. A consolidated list of the online documentation sources used for this section is provided in Appendix A.5.

Several practical considerations arise when selecting a representation format. Tree-only formats such as URDF require workarounds (constraint plugins or simulator-specific modelling) for closed-loop mechanisms, whereas formats with explicit constraints (SDF, MJCF, UsdPhysics, Simscape) can represent such mechanisms more directly but may increase coupling to a specific

Format	Coverage	Extensibility
URDF	Closed chains not representable in the core model graph (tree structure only) [44], [45]. Soft bodies not addressed in the core schema (links describe rigid bodies) [43]. Collision geometry and inertial properties can be specified on links [43]. Sensors and world elements are typically provided via simulator-specific extensions [46], [47].	Parameterisation via Xacro (XML macro expansion) in ROS toolchains [78]. Simulator hooks through `<gazebo>` tags that extend URDF with SDFormat content and are commonly converted to SDF for simulation [45], [79].
SDF	Supports kinematic loops (not restricted to a tree) [45], [50]. Non-rigid models (deformables/soft bodies) commonly appear via consumer- or simulator-specific extensions rather than a single portable core representation [52]. Links may contain multiple collision elements alongside visuals [80]. Designed to describe environments with sensors and actuators [48], [49].	Plugins as dynamically loaded code with parameters [51]. Composition via `<include>` tags for model reuse [81].
MJCF	Closed chains via equality constraints (e.g., weld) [55]. Composite and soft-object constructs (e.g. `flexcomp`) are supported [53]. Contact compatibility controlled via geom parameters such as `contype` and `conaffinity` [55]. Flexible actuation modelling via the `general` actuator element and related shortcuts [53].	Defaults/classes mechanism for scalable authoring and parameter inheritance [56]. Engine plugins and resource providers for user-authored extensions [57].
OpenUSD	Joints can be created between two rigid bodies or between one rigid body and the world [82]. The initial `UsdPhysics` schema is scoped to rigid-body simulation [60]. Collision via `UsdPhysics.CollisionAPI` on geometry [61]. General scene graph with time-sampled data and a layer-based composition model [58], [59].	Schema extensibility via `usdGenSchema` for generating new schema classes [63]. Composition arcs (e.g. references, variants) for non-destructive extension [58], [59].
Simscape	No restrictions on topology; supports kinematic loops [66]. Reduced-order flexible solids for flexible-body modelling [68]. Contact via Spatial Contact Force block [67]. Force elements and sensing within the Simulink environment [65].	Simscape language for custom physical components [69]. Library-based composition from reusable subsystems [65].

Table 2.2: Coverage and extensibility of selected robot modelling languages and description formats.

solver and simulator semantics [45], [55], [60], [66]. Soft and flexible body support is typically ecosystem-dependent: where available, deformables are implemented via engine- or vendor-specific features (such as reduced-order flexible solids in Simscape or schema extensions in USD-based ecosystems) rather than universally portable constructs [60], [62], [68]. URDF is primarily an interchange description for rigid mechanisms in ROS toolchains [43], while SDF, MJCF, and Simscape commonly embed simulator-facing semantics (sensors, actuators, contacts) that improve fidelity but reduce portability [49], [54], [65]. Even when kinematics and dynamics are expressed in URDF, SDF, or MJCF, geometry is typically offloaded to mesh formats, and conversions between asset formats can lose materials, coordinate conventions, or hierarchy information [41], [70], [71], [76].

Overall, the choice of representation is a trade-off between simplicity and tool compatibility (URDF), simulation-oriented richness (SDF, MJCF, Simscape), and general scene composition plus schema-driven extensibility (OpenUSD) [43], [49], [54], [58], [65].

2.3.4 Evaluation of Physics Engines

In this section, the tools compared in Tables 2.3–2.5 are evaluated based upon the criteria outlined previously. The end-to-end simulation engines considered are Bullet [83][84], Drake [85], MuJoCo [4], and PhysX 5 [86]. Tables 2.3–2.5 also include several rigid body dynamics and robotics-maths libraries that are frequently used alongside simulators in robotics toolchains: Rigid Body Dynamics Library (RBDL) [87], Pinocchio [88], DQRobotics [89], and GAFRO [90]. The simulation engines have been chosen because they have appeared most frequently in user surveys [91], appeared frequently in comparisons of physics engines [5][92] or simulators [64] used in robotics, or contribute something unique and significant to the discussion as well as having the potential for wide adoption in the future.

Whilst these surveys and comparisons can be combined to identify some potentially good physics engines for use in the verification of robotic systems, there is no single study which can provide a definitive answer. One reason for this is that some of the most highly cited comparisons, such as [5], are becoming less relevant; indeed many of their criticisms of physics engines, such as the lack of an account for Coriolis terms in PhysX, were fixed by developers in later versions [93].

Another reason is that recent studies often suffer from major omissions from their comparisons, for example, Drake was not included in most, if not all, recent reviews or surveys of physics engines or simulators with respect to robotics [1]. This is because it has not been widely adopted by the robotics community for reasons such as lack of ROS integration. However, it is being actively developed to be more accurate than many of the more popular robotic simulation tools, as demonstrated by its inclusion in a recent simulator comparison validating it against real-world data [39].

Several popular physics engines have not been included in this evaluation due to having closed source-code, which immediately violates the transparency criterion. These include Vortex Studio, Havok, RaiSim, and AGX Dynamics. Newton Dynamics Engine is not included due to its lack of presence in the literature cited above, despite still having widespread use in the video games industry, for example. Simbody is not included due to the fact that the original developer of the software is now collaborating on the Drake simulator, which seems under

[1] Based upon Google Scholar, IEEE, and ACM searches using terms *"robot" "physics engine" "comparison" "simulator" "simulation" "survey" "review"* since 2018 with more than two citations.

much more active development and engagement [94][95][96], primarily for robotic simulation.

Open Dynamics Engine (ODE) [97] is a physics engine used in a variety of robotic simulators, such as Gazebo. It only allows for use of first-order integrators, to make the simulation more stable, however this makes the simulation much less accurate for many robotic applications. Coriolis forces are also not documented in the ODE documentation, which are also crucial for accuracy. Finally, ODE does not appear to solve more general contact problems, compared to physics engines such as Bullet. Therefore ODE has been discarded from this evaluation.

There also exist code generation tools which provide an optimised implementation of a rigid body dynamics algorithm given a robot description, for example, RobCoGen [98], which performs better than several dynamics libraries in the benchmark of [99].

As mentioned, use of learned real-world data has not been considered as part of coverage as it is potential future work. Having said this, it is worth mentioning Tiny Differentiable simulator (TDS) [100], which makes use of learned real-world data combined with analytical data to perform various tasks from predicting link lengths, to stabilising continuous control tasks. It is not considered in this evaluation due to its fixed pipeline and lack of coverage compared to other simulation tools.

The remainder of this section discusses each tool in turn. Bullet, MuJoCo, Drake, and PhysX 5 are treated as end-to-end simulation engines, while RBDL, Pinocchio, DQRobotics, and GAFRO are discussed as complementary robotics dynamics/geometry libraries. A brief background of each tool is provided alongside the structure of its computation pipeline (where applicable), and each is discussed against the criteria of assessment, focusing on configurability and extensibility of algorithms, inputs/outputs, and modelling.

Bullet

Bullet is a physics engine primarily developed by Erwin Coumans that has received wide adoption within robotics, with applications ranging from NASA's Tensegrity robot simulation [101] to reinforcement learning [102]. This physics engine is primarily assessed through its Python API known as PyBullet [84]. Bullet does support unit testing, however it is limited compared to other simulation tools such as MuJoCo and Drake.

The Bullet physics engine is designed so that several parts of the software can be used in isolation or in conjunction, where these parts are soft body dynamics, Bullet multi threaded, rigid body dynamics, collision detection, linear math and memory, and plugins. The order of computation for the Bullet physics engine is as is indicated in figure 2.13. This pipeline is fixed, which means that when incrementing the simulation by one time-step, there is no option to configure the order of operations, and some parts such as the forwards dynamics cannot be configured without an understanding of the source-code beyond that which is provided in the documentation.

It is difficult to obtain transparency even with access to the source-code. The main reason for this is the fact that the functions used in the Python PyBullet API are oftentimes not clearly related through shared memory to the underlying C++ code of the Bullet engine. The documentation and Bullet user forums do not clearly provide the required information.

XML-model-level configurability and extensibility is mostly limited in Bullet to URDF. It is possible to use SDF and MJCF files, however this requires more work and an understanding of

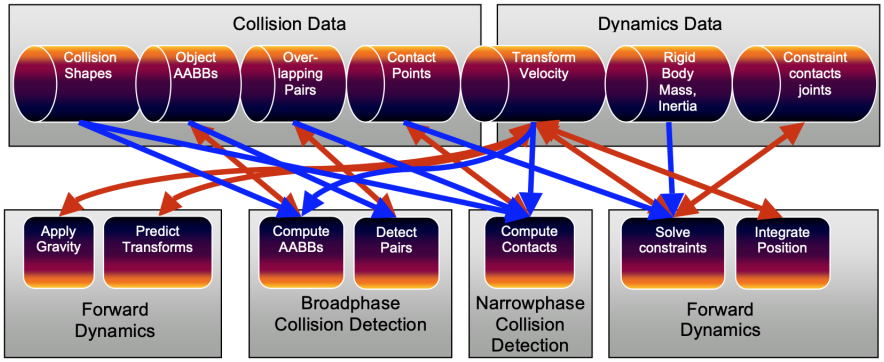


Figure 2.13: Rigid body pipeline in Bullet from left to right. Taken from [83].

the source-code on the behalf of the user.

PyBullet allows for conversions to and from all the representations of pose indicated in Table 2.4. PyBullet also allows for forward kinematics computations, for example, by setting a parameter to true when obtaining link positions, however, there is no direct way to compute pose from arbitrary joint angles [103]. It is not clear which equations are being used in the forward kinematics calculations even at the source-code-level. For inverse kinematics, damping least squares pseudoinverse [104] and Jacobian transpose methods are both available. Other inverse kinematics methods are not available in Bullet and it is does not seem possible to extend the C++ code of Bullet without a greater understanding of the library's use of shared memory.

Forward dynamics in Bullet [105] is computed using Featherstone's algorithm with Brian Mirtich's notation [106], and Coriolis terms are included. Inverse dynamics is computed using the Recursive Newton-Euler Algorithm [107][108]. The inverse dynamics is benchmarked [109] against variations introduced by Featherstone [110][111], which is included as part of the unit testing for Bullet. The inputs to forwards and inverse dynamics can be observed and configured using the *getDynamicsInfo changeDynamics* functions, however there are not multiple choices of representation of input and output, and the API is not designed such that they can be extended. These algorithms have fixed implementation in the source-code with no configurability, in particular in relation to forward dynamics, where only the symplectic Euler integrator is available. There is also no indication of how to extend the API at the source-code-level to use alternative algorithms, inputs or outputs. Bullet does allow for variable time step to control the constraint error, and therefore improves accuracy during the integration step of dynamics algorithms.

One of Bullet's main features is its collision detection, which is often used in other physics engines, for example DART [112]. Bullet makes a clear distinction between broadphase and narrowphase collision detection. It is possible to configure the collision detection to different shapes and meshes. The collision pipeline along with its rigid body dynamics is fixed in that there is no alternate choice of the order of algorithms, however it is possible to configure existing collision algorithms as well as extend at the source-code-level, as documented in [83]. Contact and friction problems in Bullet are often reduced to a Linear complementarity problem (LCP) which can be configured to be solved using Lemke's method [113], the PGS method [84], or Dantzig's method [114].

Gear ratios can be created as constraints in bullet, as well as changed in existing source-code using the change Constraint function. Erwin Coumans has suggested using the rayTestBatch function for the modelling of some sensors [115], however this is not an explicit way to create

sensors and so is not considered further. There are some examples that use sensors, however this requires a more detailed understanding of the source-code structure. Because of this, sensors in Bullet are not considered configurable or extensible here. There are no other means by which to describe physical properties of rigid robots beyond URDF and SDF. There are options to specify some of the physical properties of soft robots, however this is beyond the scope of this evaluation.

Finally, Bullet includes an OpenCL-based GPU pipeline for parts of the simulation [83]. In comparison, PhysX 5 exposes a dedicated GPU simulation pipeline oriented toward large-scale rigid-body workloads [116].

MuJoCo

MuJoCo is a physics engine introduced by Todorov *et al.* for model-based control [4]. It is widely used in robotics and continuous-control research where fast, stable rigid-body simulation is required [5]. A key practical advantage is its compact and expressive XML modelling language (MJCF), which provides strong model-level configurability over rigid-body structure, joints, actuation, and numerical parameters [4].

With respect to the criteria of assessment, MuJoCo's main strength is XML-model-level configurability, while its main limitation is extensibility of the contact and constraint-solving pipeline: its soft-constraint formulation and solver choices are tightly integrated into the engine and do not expose a straightforward path to alternative LCP-style contact formulations within the same tool [4]. As a result, MuJoCo is highly effective when the intended semantics align with its contact assumptions, but it is less suitable when the verification task requires swapping the core constraint/contact formulation.

In terms of coverage, MuJoCo supports a broad range of pose representations and dynamics computations internally, but these choices are largely fixed at the algorithmic level and exposed through a forward-simulation interface [4]. For example, it does not provide an inverse-kinematics API in the same sense as robotics toolkits, and the output structures available to users are determined by the engine's fixed computation pipeline rather than by a configurable dataflow graph.

Drake

Drake is an open-source robotics toolbox and simulator whose design emphasises a modular systems framework and a strong software-engineering culture (including extensive unit testing) [85]. This modularity is central to how Drake supports configurability and extensibility at the source-code level.

Drake is designed around compositional *systems* with explicit input and output ports, analogous to block-diagram modelling tools such as Simulink [85]. Many extension points are realised through reusable system components (often implemented using template patterns), allowing users to construct alternative simulation pipelines without editing a monolithic engine core.

In terms of transparency, the systems framework ensures that the source-code is consistent with how the user may model robotic systems, for example a controller and the physics engine (called the *multibody plant* in Drake) are separate systems connected through input and output

ports. Furthermore, it is under active open source development [96]. This has made it much easier to evaluate and assess it, as key technical challenges are discussed openly on GitHub rather than behind closed doors. Drake's support for unit testing is for the whole library, and the user can implement their own tests. This testing helps to improve the chances of correct simulation and reduces the risk when changing the source-code.

The kinematic structure of robots in Drake is represented in a separate kinematic topological tree class [85]. Transformations in Drake are executed using 6D spatial vectors with no other configurability of representation for computation. Specification using kinematic formulations such as through DH parameters is not available. It is possible to extract representations of pose as the 6D spatial vectors already specified, as well as Euler Angles, rotation matrices, and quaternions. Drake supports converting rotations into axis-angle form (e.g., via an AngleAxis representation / RotationMatrix conversion utilities) [85].

In terms of configurability for forwards dynamics using the API, the multibody plant remains fixed in a simulation pipeline, therefore offering potentially limited immediate extensibility. There does remain the choice to configure the code at the source-code-level, however in some instances this requires a knowledge of using Drake's memory cache, which is not well documented for the purpose of extensibility. In principle Drake's input and output ports can be configured and extended to any input or output, however all current examples make use of vectors or matrices using the Eigen C++ library. Furthermore, Drake has the widest coverage of integrators, and the option to extend to other integrators. Drake has an example of an inverse dynamics algorithm as well as an example of it being used in an inverse dynamics controller, each implemented using the leaf systems template. The library could easily be extended to include other implementations of inverse dynamics algorithms based upon this, however these examples make use of Drake's memory cache which requires and understanding of the source-code which, as mentioned, is not properly documented.

Collision detection and contact modelling are primarily handled by Drake's *scene graph* system, due to the geometric nature of these problems. There are examples in the library of configuring and extending the collision detection and contact modelling, however they require a deep understanding of the interplay between the multibody plant and the scene graph.

Configurability of gear ratios is possible in Drake at the source-code-level, and extending to other actuation models is possible through custom creation of a leaf system. There are also several examples of sensors, written as leaf systems, which can be configured, as well as extended.

Drake also provides multiple discrete-time contact solvers within `MultibodyPlant`, including a time-stepping nonsmooth formulation (TAMSI) and a compliant convex formulation (SAP). These are treated as part of Drake's source-code-level configurability and extensibility, and the relevant distinctions are summarised in the comparison tables below [117].

PhysX 5

NVIDIA PhysX is a high-performance rigid-body physics engine and SDK, supporting articulated rigid-body models ("articulations") and a broad set of collision/constraint features [86].

PhysX is included in this evaluation primarily for two reasons. First, it provides explicit GPU-oriented simulation pathways for rigid bodies and constraints, making it representative of the GPU-optimised class of engines [116]. Second, PhysX is increasingly used in robotics

through NVIDIA's Isaac ecosystem: Isaac Sim builds its physics stack on the PhysX backend, and Isaac Lab (built on Isaac Sim) exposes this capability to robotics research workflows [118], [119].

Relative to the assessment criteria, PhysX offers strong end-to-end pipeline coverage (collision, contact resolution, constraints, and time integration), but its configurability and extensibility are primarily exercised through the SDK and Omniverse/Isaac integration points rather than through robotics-first modelling abstractions [86], [118].

RBDL

The Rigid Body Dynamics Library (RBDL) is a lightweight C++ library for rigid-body dynamics in robotics, providing standard articulated-body algorithms (e.g., RNEA, CRBA, and ABA) in a compact implementation [87], [120].

RBDL is included because it is among the closest widely-used implementations to Featherstone's spatial-algebra presentation of rigid-body dynamics algorithms, and therefore provides a natural reference point for the spatial-vector-centric formulations used throughout this thesis [31], [87].

Under the assessment criteria, RBDL should be read as providing deep algorithmic coverage at the source-code level, but not the end-to-end simulation pipeline (collision detection, contact resolution, and time integration) provided by full simulators [87].

Pinocchio

Pinocchio is a robotics-focused C++ library designed for efficient rigid-body kinematics and dynamics, with an emphasis on computations suited to optimisation and control pipelines [88]. Like RBDL, it is transparent and extensible at the source-code level.

Pinocchio offers strong coverage of kinematics and dynamics computations (and related Jacobian-based quantities), but it intentionally does not constitute a full simulator: collision, contact models, and numerical integration are typically handled by other tools in the stack. This makes Pinocchio a good fit when the verification problem requires tight coupling to optimisation routines, but it does not replace a physics engine when contact-rich interactions are central.

DQRobotics

DQRobotics represents a different design point: it treats dual quaternion algebra as a first-class representation for robotics kinematics (and, in some formulations, dynamics) [89]. This can simplify some modelling and estimation problems by providing compact composition rules and avoiding singularities associated with some minimal coordinates.

In terms of the assessment criteria, DQRobotics broadens coverage over alternative pose representations (dual quaternions) and supports code-level extension of kinematics routines, but it does not provide the full collision/contact/integration pipeline expected of an end-to-end simulator. It is therefore best seen as complementary to physics engines rather than a

replacement for them.

GAFRO

GAFRO is included to represent the family of robotics tooling based on conformal geometric algebra (CGA), where geometric primitives and transformations are expressed in a unified algebraic framework [121]. This design choice can offer conceptual and computational benefits for some geometric reasoning tasks.

Under the assessment criteria, CGA-based tooling broadens coverage over representations (CGA), but it typically sits alongside simulation engines rather than supplanting them: contact modelling, integration, and sensors are not generally provided as a complete simulator pipeline. Its value in the present evaluation is therefore to make explicit how alternative algebraic frameworks relate to, and differ from, the spatial-vector-centric implementations in the mainstream engines.

Comparison of Simulation Engines

The qualitative discussion above is summarised in Tables 2.3–2.5. Each checkmark in these tables includes a citation and a lettered sub-reference. The lettered sub-references correspond to the documentation locations tabulated in Appendix A.4.

The configurability evaluation is summarised in Table 2.3 and is captured across both model-level (e.g., XML formats) and source-code-level. Coverage is summarised in Table 2.4 and extensibility in Table 2.5.

		MuJoCo	PyBullet	Drake	RBDL	PhysX 5	Pinocchio	DQRobotics	GAFRO
Pose Representation	Model-level	✓[4]a	✓[84]a	✓[85]a	✓[87]a	×	✓[88]a	×	×
Pose Representation	Source-code level	×	×	×	×	×	×	×	×
Kinematics Algorithms	Model-level input value	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source-code level input value	✓[4]b	✓[84]b	✓[85]b	✓[87]b	✓[86]a	✓[88]b	✓[89]a	✓[90]a
Kinematics Algorithms	Model-level output value	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source code level output value	✓[4]c	✓[84]c	✓[85]c	✓[87]c	✓[86]b	✓[88]c	✓[89]b	✓[90]b
Kinematics Algorithms	Model-level input structure	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source-code level input structure	×	×	×	×	×	×	×	×
Kinematics Algorithms	Model-level output structure	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source code level output structure	×	×	×	×	×	×	×	×
Kinematics Algorithms	Model-level alternative	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source-code level alternative	×	✓[84]d	✓[85]d	×	×	×	×	×
Dynamics Algorithms	Model-level input value	✓[4]d	×	×	×	×	×	×	×
Dynamics Algorithms	Source-code level input value	×	✓[84]e	✓[85]e	✓[87]d	✓[86]c	✓[88]d	×	✓[90]c
Dynamics Algorithms	Model-level output value	×	×	×	×	×	×	×	×
Dynamics Algorithms	Source code level output value	✓[4]e	✓[84]f	✓[85]f	✓[87]e	✓[86]d	✓[88]e	×	✓[90]d
Dynamics Algorithms	Model-level input structure	×	×	×	×	×	×	×	×
Dynamics Algorithms	Source-code level input structure	×	×	×	×	×	×	×	×
Dynamics Algorithms	Model-level output structure	×	×	×	×	×	×	×	×
Dynamics Algorithms	Source code level output structure	×	×	×	×	×	×	×	×
Dynamics Algorithms	Model-level alternative	×	×	×	×	×	×	×	×
Dynamics Algorithms	Source-code level alternative	×	✓[84]g	?	×	×	×	×	×
Collision Algorithms	Model-level input value	✓[4]f	✓[84]h	✓[85]g	?	?	?	×	×
Collision Algorithms	Source-code level input value	×	×	×	×	×	×	×	×
Collision Algorithms	Model-level output value	×	×	×	×	×	×	×	×
Collision Algorithms	Source code level output value	?	✓[84]i	?	?	✓[86]e	?	×	×
Collision Algorithms	Model-level input structure	×	×	×	×	×	×	×	×
Collision Algorithms	Source-code level input structure	×	×	×	×	×	×	×	×
Collision Algorithms	Model-level output structure	×	×	×	×	×	×	×	×
Collision Algorithms	Source code level output structure	×	×	×	×	×	×	×	×
Collision Algorithms	Model-level alternative	×	×	×	×	×	×	×	×
Collision Algorithms	Source-code level alternative	×	×	×	×	×	×	×	×
Integration	Model-level input value	✓[4]g	×	×	×	×	×	×	×
Integration	Source-code level input value	×	✓[84]j	✓[85]h	×	✓[86]f	×	×	×
Integration	Model-level output value	×	×	×	×	×	×	×	×
Integration	Source code level output value	?	✓[84]k	✓[85]i	×	✓[86]g	×	×	×
Integration	Model-level alternative	×	×	×	×	×	×	×	×
Integration	Source-code level alternative	×	×	✓[85]j	×	×	×	×	×
Constraint Solver	Model-level alternative	×	×	×	×	×	×	×	×
Constraint Solver	Source-code level alternative	×	×	×	×	×	×	×	×
Actuation and gearing	Model-level value	✓[4]h	✓[84]l	✓[85]k	×	?	×	×	×
Actuation and gearing	Source-code level value	×	×	✓[85]l	×	✓[86]h	×	×	×
Sensors	Model-level input value	✓[4]i	✓[84]m	✓[85]m	×	×	×	×	×
Sensors	Source-code level input value	×	×	✓[85]n	×	✓[86]i	×	×	×
Sensors	Model-level output value	✓[4]j	✓[84]n	✓[85]o	×	×	×	×	×
Sensors	Source code level output value	?	✓[84]o	✓[85]p	×	✓[86]j	×	×	×
Sensors	Model-level input structure	×	×	×	×	×	×	×	×
Sensors	Source-code level input structure	×	×	×	×	×	×	×	×
Sensors	Model-level output structure	✓[4]k	✓[84]p	✓[85]q	×	×	×	×	×
Sensors	Source code level output structure	×	×	×	×	×	×	×	×
Sensors	Model-level alternative	×	×	×	×	×	×	×	×
Sensors	Source-code level alternative	×	×	✓[85]r	×	×	×	×	×

Table 2.3: Configurability comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a).

		MuJoCo	PyBullet	Drake	RBDL	PhysX 5	Pinocchio	DQRobotics	GAFRO
Pose Structures	6D-Spatial Vectors	✓[4]l	✓[84]q	✓[85]s*	✓[87]f	✓[86]k	✓[88]f	×	×
Pose Structures	Euler-Angles	✓[4]m	✓[84]r	✓[85]t	✓[87]g	×	✓[88]g	?	×
Pose Structures	Axis-angle	✓[4]n	✓[84]s	✓[85]u	?	✓[86]l	✓[88]h	?	×
Pose Structures	Quaternions	✓[4]o	✓[84]t	✓[85]v	✓[87]h	✓[86]m	✓[88]i	✓[89]c	×
Pose Structures	Rotation Matrices	✓[4]p	✓[84]u	✓[85]w	✓[87]i	✓[86]n	✓[88]j	?	×
Pose Structures	Dual quaternions	×	×	×	×	×	×	✓[89]d	×
Pose Structures	Conformal Geometric Algebra (CGA)	×	×	×	×	×	×	×	✓[90]e
Forward Kinematics	D-H parameters	×	×	×	×	×	×	?	?
Forward Kinematics	Product of Exponentials	×	×	×	×	×	×	?	?
Forward Kinematics	Spatial vectors	✓[4]q	✓[84]v	✓[85]x	✓[87]j	×	✓[88]k	×	×
Inverse Kinematics	Damped least squares pseudoinverse	×	✓[84]w	×	✓[87]k	×	?	✓[89]e	?
Inverse Kinematics	Jacobian transpose	×	×	×	×	×	?	?	?
Inverse Kinematics	Analytical	×	×	×	×	×	×	×	×
Inverse Kinematics	Other	×	×	✓[85]y	×	×	?	?	?
Dynamics	Featherstone's algorithm	✓[4]r	✓[84]x	✓[85]z	✓[87]l	?	✓[88]l	×	?
Dynamics	Coriolis Terms	×	×	✓[85]aa	✓[87]m	✓[86]o	✓[88]m	×	?
Dynamics	Inverse dynamics	✓[4]s	✓[84]y	✓[85]ab	✓[87]n	✓[86]p	✓[88]n	×	?
LCP Problem	Lemke's method	×	✓[84]z	×	×	×	×	×	×
LCP Problem	PGS method	✓[4]t	✓[84]aa	×	×	✓[86]q	×	×	×
LCP Problem	Dantzig method	×	✓[84]ab	×	×	×	×	×	×
LCP Problem	Other	✓[4]u	✓[84]ac	✓[85]ac	?	✓[86]r	?	×	×
Other Constraint Problem	Implicit complementarity solver	?	×	?	?	×	?	×	×
Other Constraint Problem	Diagonal solver	?	?	×	?	×	?	×	×
Other Constraint Problem	Convex solver	✓[4]v	?	✓[85]ad	?	×	?	×	×
Other Constraint Problem	Other	✓[4]w	✓[84]ad	✓[85]ae	?	✓[86]s	?	×	×
Integration	Explicit Euler	×	×	✓[85]af	×	×	×	×	×
Integration	Symplectic Euler (semi-implicit / semi-explicit)	✓[4]x	✓[84]ae	✓[85]ag	×	✓[86]t	×	×	×
Integration	Implicit Euler	✓[4]y	×	✓[85]ah	×	×	×	×	×
Integration	Bogacki–Shampine	×	×	✓[85]ai	×	×	×	×	×
Integration	Radau	×	×	✓[85]aj	×	×	×	×	×
Integration	Runge–Kutta 2	×	×	✓[85]ak	×	×	×	×	×
Integration	Runge–Kutta 3	×	×	✓[85]al	×	×	×	×	×
Integration	Runge–Kutta 4	✓[4]z	×	✓[85]am	×	×	×	×	×
Integration	Runge–Kutta 5	×	×	✓[85]an	×	×	×	×	×
Component Modelling	gear ratio	✓[4]aa	✓[84]af	✓[85]ao	×	?	×	×	×
Component Modelling	sensors	✓[4]ab	×	✓[85]ap	×	✓[86]u	×	×	×
Component Modelling	rigid links	✓[4]ac	✓[84]ag	✓[85]aq	✓[87]o	✓[86]v	✓[88]o	✓[89]f	✓[90]f
Component Modelling	joints	✓[4]ad	✓[84]ah	✓[85]ar	✓[87]p	✓[86]w	✓[88]p	✓[89]g	✓[90]g
Component Modelling	deformable links / soft robots	✓[4]ae	?	✓[85]as	×	?	×	×	×
Computation	GPU acceleration	×	✓[83]a	×	×	✓[116]a	×	×	×

Table 2.4: Coverage comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a). *Drake uses 6D spatial vectors but not Plücker coordinates specifically.

		MuJoCo	PyBullet	Drake	RBDL	PhysX 5	Pinocchio	DQRobotics	GAFRO
Pose Representation	Model-level	✓[4]af	✓[84]ai	×	×	×	×	×	×
Pose Representation	Source-code level	×	×	×	×	×	×	×	×
Kinematics Algorithms	Source-code level extension	×	✓[84]aj	✓[85]at	✓[87]q	×	✓[88]q	✓[89]h	✓[90]h
Dynamics Algorithms	Source-code level extension	×	✓[84]ak	✓[85]au	✓[87]r	×	✓[88]r	×	✓[90]i
Collision Algorithms	Model-level extension	✓[4]ag	✓[84]al	✓[85]av	×	×	×	×	×
Collision Algorithms	Source-code level extension	×	✓[84]am	✓[85]aw	×	×	×	×	×
Integration	Source-code level extension	×	×	✓[85]ax	×	×	×	×	×
Constraint Solver	Source-code level extension	×	×	✓[85]ay	×	×	×	×	×
Actuation and gearing	Model-level extension	×	×	×	×	×	×	×	×
Actuation and gearing	Source-code level extension	×	✓[84]an	✓[85]az	×	×	×	×	×
Sensors	Model-level extension	×	×	×	×	×	×	×	×
Sensors	Source-code level extension	×	×	✓[85]ba	×	×	×	×	×

Table 2.5: Extensibility comparison across simulation engines and robotics dynamics/geometry libraries. A checkmark includes a supporting citation with a lettered sub-reference (e.g., 13a).

Taken together, Tables 2.3–2.5 make explicit a recurring pattern: end-to-end simulators tend to offer broad pipeline coverage (collision detection, contact resolution, time integration, and sensors), whereas robotics libraries tend to provide deeper access to individual kinematics/dynamics computations and mathematical representations.

It is also worth noting several rigid body dynamics and robotics-maths libraries that complement full physics engines: Rigid Body Dynamics Library (RBDL) [87], Pinocchio [88], DQRobotics [89], and GAFRO [90]. These libraries often provide faster or more specialised kinematics/dynamics computations (and in some cases alternative algebraic frameworks), but they typically do not provide a full simulator pipeline (collision detection, contact resolution, time integration, sensors, etc.) by themselves. They are included in the comparison tables in

order to make these trade-offs explicit.

Within the simulator engines, MuJoCo offers particularly rich model-level configurability via its MJCF modelling language, Drake provides the most explicit source-code-level extensibility across the simulation pipeline, and Bullet provides a pragmatic but comparatively fixed pipeline with wide adoption. PhysX 5 provides a GPU-capable rigid-body simulation stack and, through Isaac Sim and Isaac Lab, is increasingly used in robotics workflows [116], [118], [119].

2.4 Summary

This chapter established the modelling and simulation background used by the remainder of the thesis. Section 2.1 explained the p-model as a hierarchical description of a robotic platform, covering its physical structure, component connectivity, and component- and system-level equations. It also presented the backward-compatible p-model+ metamodel facilities for system-level equations, explicit solution specifications, constraints, relations, and supporting types, illustrated using the Simple Arm.

Section 2.2 described RoboSim's controller–platform separation. A d-model specifies the discrete controller, a p-model specifies the continuous robotic platform, and a platform mapping relates their interfaces; together, they form the hybrid model. The CyPhyCircus account provides the semantic basis for the controller–platform interaction used later when relating the models to the generated code.

The evaluation of selected physics engines and robotics dynamics libraries then considered transparency, configurability, and extensibility at both model and source-code levels. Existing tools provide useful but uneven support, and their equations, solution methods, or computation order are often fixed or exposed only through implementation details. These observations motivate the simulation framework presented in Chapter 3.

3 Simulation Framework

Building on the RoboSim models introduced in Chapter 2, this chapter presents a simulation framework that takes a d-model, p-model, and platform mapping as input and generates executable simulation artefacts by translating platform equations and solution methods selected by the user from a library into source code. By automating these translations, the framework reduces the need to edit intermediate representations or generated code and supports the consistent use of the same physical equations across modelling, simulation, and proof.

The framework comprises five techniques. The existing d-modelGen transformation generates controller source code from d-models [122], while the prototype mappingGen implemented in this thesis generates source code for the supported platform mappings. The three techniques contributed by this thesis are eqnComp, guidedChoice, and sourceCodeGen. Equation composition (eqnComp) generates the system-level equations required by the detected formulation. Guided choice (guidedChoice) resolves solution methods chosen by the user from the solution library into solution references (slnRef). Source-code generation (sourceCodeGen) transforms those solution references into a solution-description-format specification (slnDF) and target source code for the platform engine.

Section 3.1 summarises the simulation framework. Section 3.2 describes equation composition, and Section 3.3 introduces the solution-description languages used between the techniques. Sections 3.4 and 3.5 then present guided choice and generation for a platform engine, respectively. The techniques are illustrated throughout using the 2R Simple Arm inverted-pendulum model introduced in Example 3.1. Section 3.6 concludes the chapter.

3.1 Overview

The simulation framework enables carrying out simulation and proof in a consistent way, given a diagrammatic model of a simulation that defines both the software and the robotic platform's behaviour (via a d-model, a p-model, and a platform mapping). The framework is described in Fig. 3.1, which indicates the relationship between the RoboSim models and automatically generated artefacts. The arrows represent techniques that ensure semantics is preserved through each generation step, designed to preserve information and respect the operational semantics of the RoboSim models (see Chapter 2). This framework limits the need to modify intermediary representations and code, since they are automatically generated.

The **Simulation framework** is an integrated set of techniques that transform RoboSim models into executable simulation code.

This framework complements the existing **Technique 1: Generating source-code from d-model** (d-modelGen), which produces source code from a d-model (see [122]). For **Technique 2: Generating source-code from mapping** (mappingGen), this thesis implements a simple prototype that generates source code for platform mappings; it does not deal with the general mapping-transformation problem. The prototype and case studies are described in Chapter 4.

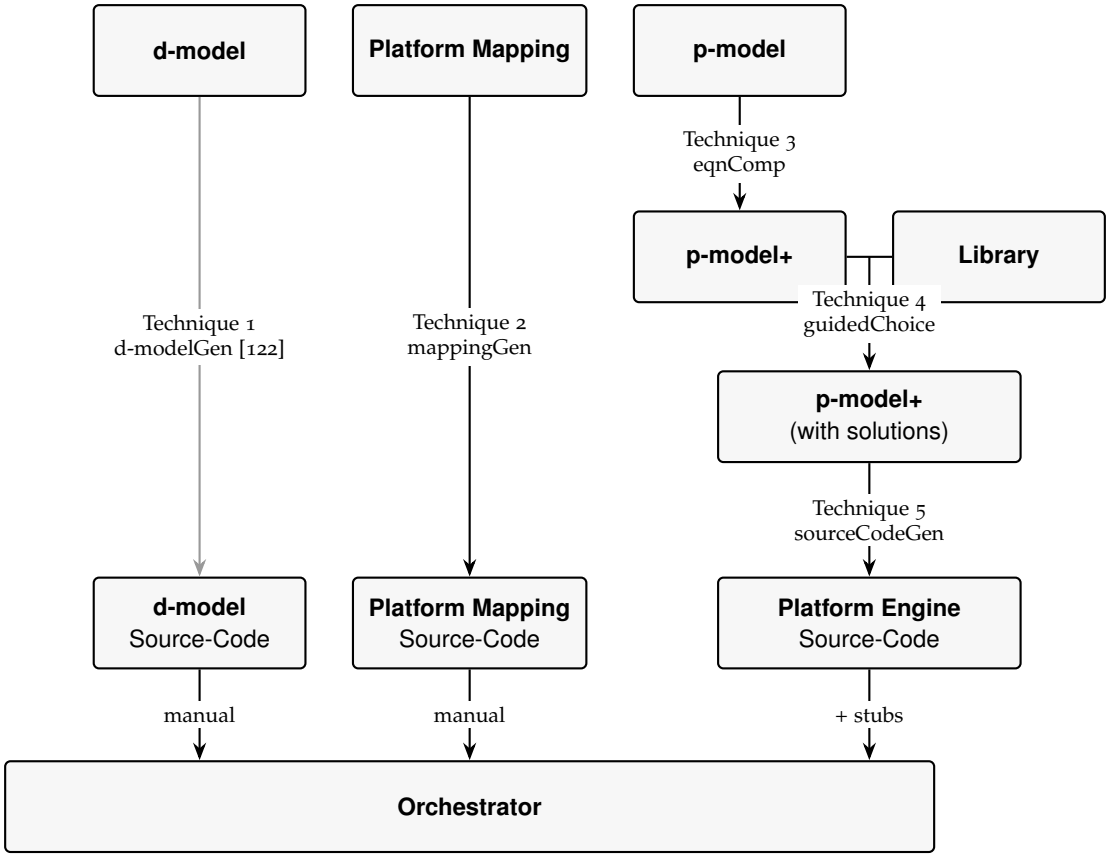


Figure 3.1: An overview of the RoboSim simulation framework.

The three principal techniques contributed by this thesis are as follows. Technique 3: Equation Composition (eqnComp) takes a p-model and computes component and system-level equations to yield the complete set of equations for a chosen formulation, thereby extending from a p-model to a p-model+. Technique 4: Guided Choice (guidedChoice) accepts solution-reference choices made by the user from a library, unfolds those solutions in the context of the rest of the p-model, and updates the model with references to the fully instantiated solutions and any required initialisations. Technique 5: Generation for a Platform Engine (sourceCodeGen) consumes the p-model+ produced by guidedChoice and generates target-language source-code for the platform engine along with an orchestrator that manages the simulation lifecycle. The orchestrator's role is to ensure that the individual d-model and p-model processes respect the RoboSim operational semantics described in Chapter 2. Since the discrete controller and mapping are generated by separate tools (the existing d-modelGen transformation and the prototype mappingGen), the orchestrator cannot fully determine the concrete controller entry point, event/operation types, and mapping interface without referencing those generated artefacts. Therefore sourceCodeGen emits a complete orchestrator skeleton with explicit integration stubs (e.g., the `registerRead/registerWrite` adapter, Listing A.14). Completing these stubs requires the user to provide the generated d-model and mapping code, select how controller inputs/outputs are represented, wire those representations to the platform-mapping variables, and supply any run-time configuration that is external to the models (e.g., cycle period and integration step).

Together, these techniques ensure that the source-code produced for the d-model, platform-mapping and p-model remains faithful to the original models, enabling full simulation of the RoboSim hybrid-model. In this hybrid model, the existing d-modelGen transformation handles discrete behaviour, the prototype mappingGen defines the discrete–continuous interface for

the mappings supported in this thesis, and the combination of eqnComp, guidedChoice, and sourceCodeGen realises the continuous dynamics.

In this approach, simulation development begins by authoring a d-model, a p-model, and a platform-mapping. Because users may supply arbitrary equations, the framework is designed to be general enough to support any relevant representations and formulations. In this chapter, Techniques 3–5 are illustrated using the canonical forward-dynamics example of the 2R Simple Arm inverted-pendulum model in Example 3.1, with additional use cases presented in Chapter 4.

Example 3.1: Simple Arm Inverted Pendulum: Simulation Framework

Example 2.1 introduced the physical structure of the Simple Arm p-model and the SKO formulation used for its joints. Here, the same p-model is used as the running input to Techniques 3–5. The complete Simple Arm d-model and platform mapping used with this p-model are presented later in Section 4.3.1. Figure 3.2 relates the input p-model to the resulting simulation and identifies three items absent from this particular input: (1) system-level equations in the system-level block; (2) relations composing component-level equations into system-level equations; and (3) a solution showing how the physics engine solves those equations. Technique 3 (eqnComp) addresses the first two omissions by generating the system-level equations and their composition relations; Technique 4 (guidedChoice) resolves the user's chosen solution methods; and Technique 5 (sourceCodeGen) generates the solution description and target source code for the platform engine. Together with the d-model and platform mapping, these outputs provide the executable artefacts used for the consistent simulation shown in the figure.
The p-model uses trivial sensor and actuator definitions because this Chapter 3 running example focuses on rigid-body dynamics.

Example 3.1 (continued): Simple Arm Inverted Pendulum: Simulation Framework

1. The input p-model has no system-level equations in its system-level block.

2. It has no relations composing component-level equations into system-level equations.

3. It has no solution showing how the physics engine solves the equations.

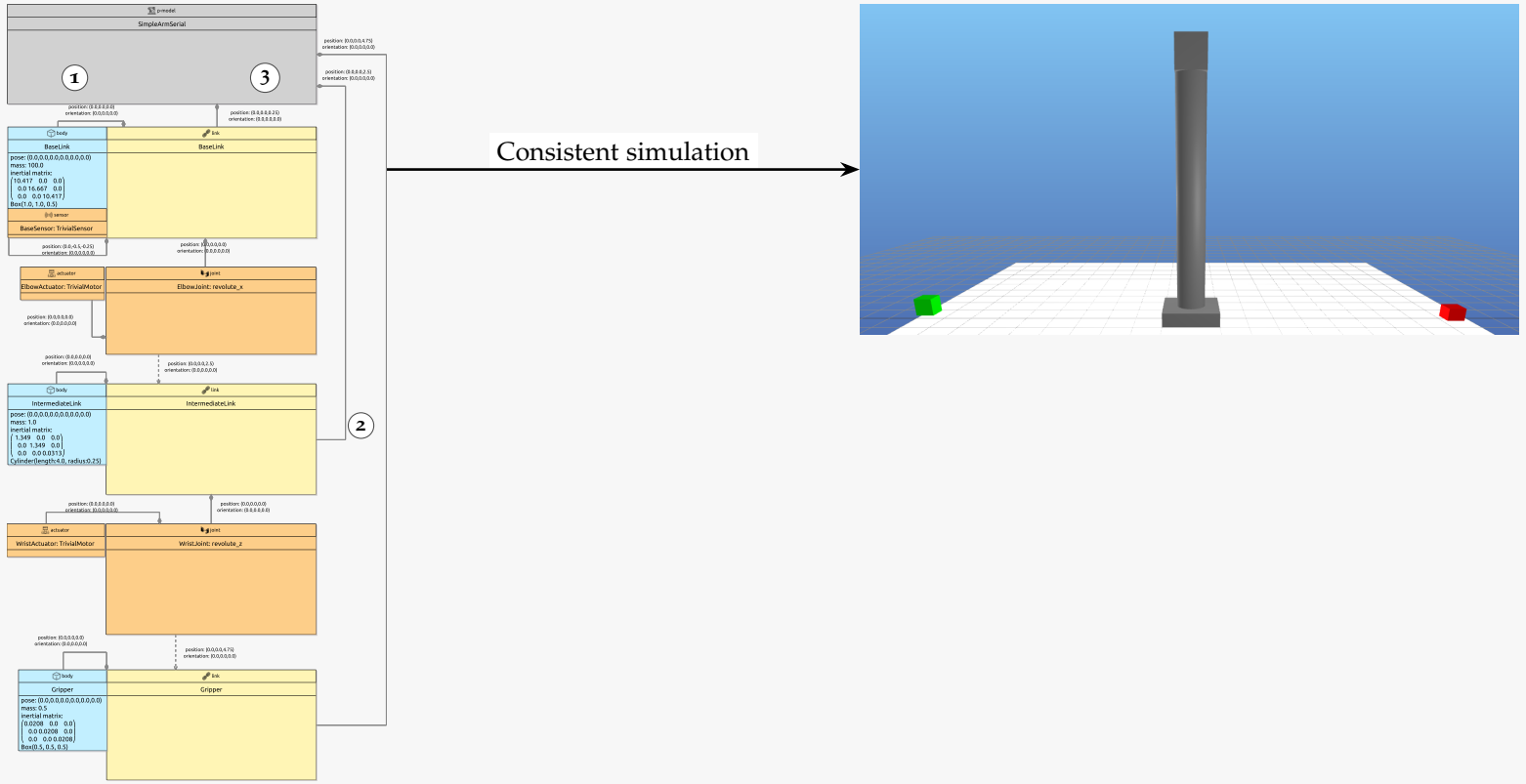


Figure 3.2: The Simulation Framework produces a consistent Simple Arm simulation from the input p-model, chosen solutions, and other workflow inputs.

As established in Chapter 2, rigid body dynamics provide the foundational behavioural equations for many robotic platforms, with forward dynamics being central to modern platform engines. To enable simulation of a complete robotic platform, while minimising modelling effort, our work derives solutions for forward dynamics by recursively composing equations from individual components up to the system-level, represented by the root p-model block. The framework also supports solving for variables within subsystems, defined as *parts* in the p-model.

The framework generates a complete set of behavioural equations from an initial p-model, enabling the simulation of the system's evolution over time.

This equation generation is handled by Technique 3: Equation Composition (eqnComp), which enriches an extended p-model (PModel+) with the required equations at both system and component levels (see Section 3.2). For the Simple Arm example, eqnComp populates system and subsystem blocks with new equations and variables, updating the main system block as shown in Fig. 3.3.

The framework also updates equations and variables for any component that is a *dynamic device*. In the BaseLink (Fig. 3.4), for instance, components of system-level operators become visible. Often, variables required for practical applications like visualisation are omitted from the theoretical literature, such as in [24]. Example 3.2 introduces such variables, including T_32 for computing link positions and Geom for defining link geometries. The Geom variables are also necessary to symbolically reference geometric elements in the p-model's equations. Other key identifiers, like the number assigned to links and joints, are detailed in Section 3.2.

Finally, eqnComp defines relations between components and between variables at the system and component levels. This supports bidirectional traceability between components themselves, and to and from the system-level, insofar as the formulation library exposes the required derivation information. These relations show how user-defined constants at the component-level, such as pose and geometry, propagate to the system-level. The specific relation between the p-model block and its links is detailed in Example 3.2 and illustrated in Fig. 3.4.

Example 3.2: Simple Arm Serial Manipulator: Equation Composition

For the Simple Arm example, the main result of eqnComp is the updating of the system-level block with system-level operator equations and variables, as presented in Fig. 3.3.
The variables relevant to the spatial kernel operator (SKO) formulation in this example are r_v1, r_v2, N, n, V, f, a, b, alpha, phi, M, theta, tau, C, H, and M_mass. The equations generated by eqnComp define how these system-level quantities are computed and how they compose from component-level equations.
B_1, B_2, and B_3 are the spatial positions of the links (at their centre of mass relative to the world frame), and B_k is the sequence of these positions. Spatial positions here are computed from the poses defined in the p-model. These variables, whilst not explicitly calculated in [24], are necessary to facilitate the common use case of calculating the spatial positions of the links. This is especially useful for visualisation and collision, since tools such as Meshcat [123] require the spatial positions of the links to be defined. The remaining variables of type Geom are also used for this purpose, but could also be used as part of collision algorithms. These variables are the geometries for each link computed based on the geometry attributes of link bodies.
As shown in Fig. 3.4, the components, such as BaseLink are updated with component-level variables and equations. The SKO component-level variables are M_3, a_3, b_3, alpha_3, f_3, _f_2, phi_3_2, r_v2, tau_3. I and m are calculated from the inertial properties of link or

corresponding bodies. The value of L3_geom is calculated from the geometry property of the body attached to the link.

p-model

SimpleArmSerial

InOuts

```
r_v1: int
r_v2: int
N: real = r_v1+r_v2
n: int = 3
B_1: matrix(real,4,4)
B_2: matrix(real,4,4)
B_3: matrix(real,4,4)
T_2_1: matrix(real,4,4)
T_3_2: matrix(real,4,4)
B_k: Seq(Frame)
L1_geom: Geom
L2_geom: Geom
L3_geom: Geom
Lk_geom: Seq(Geom)
V: vector(real,18)
f: vector(real,18)
a: vector(real,18)
b: vector(real,18)
alpha: vector(real,18)
phi: matrix(real,18,18)
M: matrix(real,18,18)
theta: vector(real,2)
tau: vector(real,2)
C: vector(real,2)
H: matrix(real,2,18)
M_mass: matrix(real,2,2)
```

...

Equations

```
B_k==<B_1, B_2, B_3>
Lk_geom==<L1_geom, L2_geom, L3_geom>
submatrix(phi)(0,0,3,3)==identity(3)
submatrix(phi)(0,3,3,3)==zeroes(3)
submatrix(phi)(0,6,3,3)==zeroes(3)
submatrix(phi)(3,0,3,3)==Phi(2, 1, B_k)
submatrix(phi)(3,3,3,3)==identity(3)
submatrix(phi)(3,6,3,3)==zeroes(3)
submatrix(phi)(6,0,3,3)==Phi(3, 1, B_k)
submatrix(phi)(6,3,3,3)==Phi(3, 2, B_k)
submatrix(phi)(6,6,3,3)==identity(3)
V==adj(phi)*adj(H)*derivative(theta)
alpha==adj(phi)*adj(H)*derivative(theta)+a
f==phi*(M*alpha+b)
tau==M_mass*derivative(derivative(theta))+C
M_mass==H*phi*M*adj(phi)*adj(H)
C==H*phi(M*adj(phi)*a+b)
B_2==B_3*T_3_2
B_1==B_3*T_3_2*T_2_1
```

...

Figure 3.3: The Simple Arm system-level block after applying eqnComp.

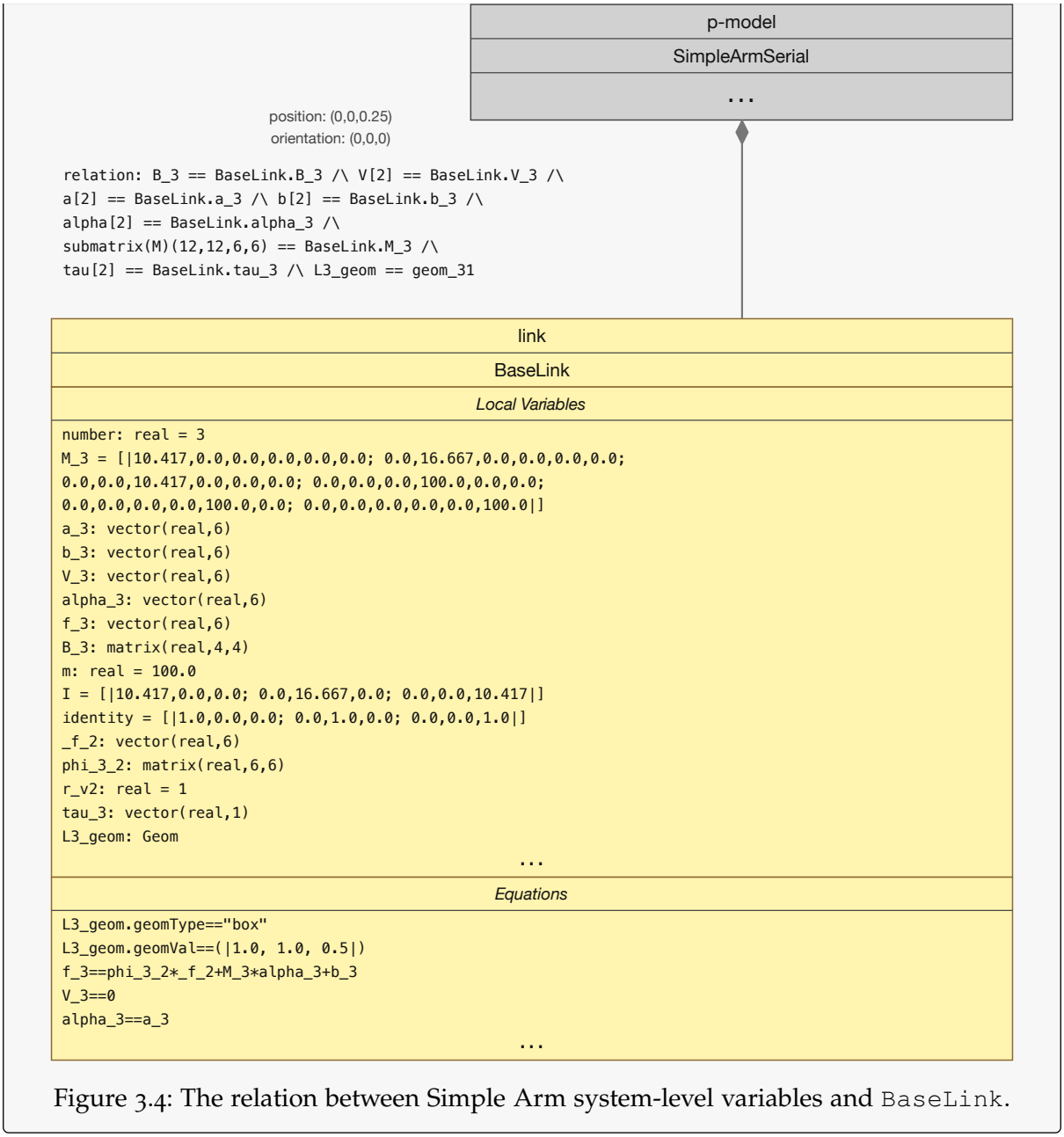


Figure 3.4: The relation between Simple Arm system-level variables and `BaseLink`.

Using the result of eqnComp, and a choice from a library of solution algorithms, a simplified description of all equations and solutions is obtained by generating another p-model+. It includes references to the solution of chosen variables, using Technique 4: Guided Choice(guidedChoice) described in Section 3.4. The library includes specialised solving algorithms recommended or applicable to specific variable-method pairs, and is described in Section 4.1.2. The user annotates the p-model+ with the variable-method pairs to be unfolded (if necessary) and solved. The guidedChoice technique automatically unfolds the solution methods for a given variable, hence the user need not know the details of the solution methods, but can instead focus on the variables they want to solve. The user can also specify the order in which the variables are solved, the group of variables that are solved together, the inputs of the solution, the constraints on any of the variables used in the solution, and the numerical error of the solution. These additional parameters also have default values which are computed based on the library and on the variables, constraints, and equations already in the p-model+.

In the p-model+ resulting from applying guidedChoice, the system-level block now contains references to the solution methods for each variable, as well as the constraints and numerical

error. The variables are also annotated with their order, which is used to determine the order of the solution methods, and group, to determine the variables that are solved together.

Any new methods and solution choices as a result of the unfolding are then added to the p-model+. The result of guidedChoice also ensures that all necessary inputs and constraints are added and initialised in the p-model+. The system-level block now contains all of the initial conditions for variables, which are used in solution, and additional variables required by the user's solution choices.

The unfolded solutions in the p-model+ are references that point to the full description of the corresponding solution method in the library, written in the Solution Description Format (slnDF). This is a specification language designed to explicitly represent all of the information necessary to describe the solutions to relevant variables; it is described in Section 3.3.2.

The p-model is also annotated with new information as a result of this technique. This includes new variables which did not previously exist in the model, and the constant dt, which is used to indicate the time step for the Euler integration method (if present). As mentioned in Chapter 2, the timer variable evolves according to $\frac{d\,timer}{dt} = 1$, ensuring proper synchronisation between the continuous dynamics and the discrete controller cycles.

Example 3.3: Simple Arm Serial Manipulator: Guided Choice of Solutions

For this example, the user begins by annotating the p-model+ with solution references. Concretely, these annotations are added to a dedicated `solutions` section on the system-level p-model block, as a list of `solution { ...}` entries. Each entry names the variable to be solved (`solutionExpr`), selects a solution-library method (`method`), and provides an `order` and `group`. The user-level annotations are intentionally lightweight (they omit explicit inputs and constraints); guidedChoice later enriches them by inferring dependencies and initialisation constraints and, when required, unfolding a single high-level request into a pipeline of sub-solutions. The full solution-reference format is introduced in Section 3.3.1. Listing 3.1 shows the initial user annotations.

```
solution {
   solutionExpr theta : vector ( real , 2 )
   order 1
   group 1
   method GeneralisedPosition_method1
}

solution {
   solutionExpr Bk : Seq ( matrix ( real , 4 , 4 ) )
   order 2
   group 1
   method Eval
}

solution {
   solutionExpr Bk : Seq ( matrix ( real , 4 , 4 ) )
   order 1
   group 2
   method Vis
}
```

Listing 3.1: User-authored solution references in the p-model+'s `solutions` section (before applying guidedChoice).

Group 1 requests the forward-dynamics pipeline that computes the generalised position $\boldsymbol{\theta}$ (via `GeneralisedPosition_method1`) and then evaluates the link transforms B_k (via `Eval`). Group 2 requests a separate visualisation computation (via `Vis`) so that geometry-related values are generated independently of the dynamics pipeline. Within a group, smaller `order` values are computed first; here $\boldsymbol{\theta}$ is ordered before B_k because evaluating B_k depends on the joint configuration.
Applying guidedChoice may unfold a single user request into multiple dependent solution references. Table 3.1 summarises the unfolded pipeline for the group 1 request for $\boldsymbol{\theta}$ (i.e., the expansion of `GeneralisedPosition_method1`). Each row corresponds to one generated solution reference, listed in increasing `order` within the same `group`. The columns record: the solved expression (*Expr*), the library method selected for it (*Method*), initial constraints inferred/propagated from the p-model (*Constraints*), the *Order* and *Group*, and the expected numerical error (zero for analytic steps; truncation order for numerical integration).
In this instance, the pipeline begins by obtaining the torque $\boldsymbol{\tau}$ from the platform mapping, then evaluates the rigid-shift operator $\boldsymbol{\phi}$, uses it to compute the inverse dynamics term $\boldsymbol{C}$, computes the mass matrix $\mathcal{M}$ (Composite Body Algorithm), inverts it (Cholesky), computes the acceleration $\ddot{\boldsymbol{\theta}}$ (Direct Forward Dynamics), and finally integrates twice using Euler to obtain $\dot{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}$.

Expr	Method	Constraints	Order	Group	Numerical Error
$\boldsymbol{\tau}$	PlatformMapping	$\boldsymbol{\tau}(0) = \mathbf{0}$	1	1	0
$\boldsymbol{\phi}$	Eval	$\boldsymbol{\phi}(0) = \mathbf{0}$	2	1	0
$\boldsymbol{C}$	Newton-Euler Inverse Dynamics	$\boldsymbol{C}(\mathbf{0}) = \mathbf{0}$	3	1	0
$\mathcal{M}$	Composite Body Algorithm	$\mathcal{M}(\mathbf{0}) = \mathcal{M}$	4	1	0
$\mathcal{M}^{-1}$	Cholesky Algorithm	$\mathcal{M}^{-1}(\mathbf{0}) = \mathbf{0}$	5	1	0
$\ddot{\boldsymbol{\theta}}$	Direct Forward Dynamics	$\ddot{\boldsymbol{\theta}}(0) = \mathbf{0}$	6	1	0
$\dot{\boldsymbol{\theta}}$	Euler	$\dot{\boldsymbol{\theta}}(0) = \mathbf{0}, dt = 0.1$	7	1	$\mathcal{O}(dt)$
$\boldsymbol{\theta}$	Euler	$\boldsymbol{\theta}(0) = 0, dt = 0.1$	8	1	$\mathcal{O}(dt)$

Table 3.1: The unfolded solution methods for $\boldsymbol{\theta}$ and the method *GeneralisedPosition_method1*.

Figure 3.5 shows how Table 3.1 is represented in the p-model+'s `solutions` section. The excerpt corresponds to rows 5–8 (from $\mathcal{M}^{-1}$ through $\boldsymbol{\theta}$): in addition to the table columns, guidedChoice has made data dependencies explicit via `Input` fields and has added/updated initialisation constraints. Earlier steps (rows 1–4) and the separate visualisation group are omitted for brevity.
Fig. 3.5 highlights in blue additional variables and constraints that have been added as a result of guidedChoice. The new additions are variables M_inv, d_theta, dd_theta, constraints for the initial conditions of any relevant variables which did not already have an initial condition, and the constant dt.

p-model

SimpleArmSerial

InOuts

```
theta: vector(real,2)
tau: vector(real,2)
M_mass: matrix(real,2,2)
...
```

Local Variables

```
d_theta: vector(real,2)
dd_theta: vector(real,2)
M_inv: matrix(real,2,2)
dt: real
...
```

Constraints

```
constraint (M_inv) [ t == 0] ==zeroMat(2, 2)
constraint (dd_theta) [ t == 0] ==zeroVec(2)
constraint (d_theta) [ t == 0] ==zeroVec(2)
...
```

Solutions

...

CholeskyAlgorithm

```
expression M_inv : matrix(real,2,2)
order 5
group 1
input M_mass : matrix(real,2,2)
constraint (M_inv) [ t == 0] ==zeroMat(2, 2)
```

DirectForwardDynamics

```
expression dd_theta : vector(real,2)
order 6
group 1
input tau : vector(real,2)
input M_inv : matrix(real,2,2)
constraint (dd_theta) [ t == 0] ==zeroVec(2)
```

Euler

```
expression d_theta : vector(real,2)
order 7
group 1
input dd_theta : vector(real,2)
input dt : real
constraint (d_theta) [ t == 0] ==zeroVec(2)
```

Euler

```
expression theta : vector(real,2)
order 8
group 1
input d_theta : vector(real,2)
input dt : real
constraint (theta) [ t == 0] ==zeroVec(2)
```

Figure 3.5: Selected Simple Arm p-model+ content after guidedChoice.

The final technique, Technique 5: Generation for a Platform Engine(sourceCodeGen), of the framework generates source-code in the target language given the p-model+ outputted from guidedChoice. In Example 3.4 this is C++. The generated code does not need to be modified, except for the stubs in the orchestrator code based on the output of the existing d-modelGen transformation and the prototype mappingGen.

The sourceCodeGen technique operates in two stages. First, it uses solution references from the p-model+ to generate a **Solution Description Format** (slnDF) file: a specification language that explicitly represents solution information through a `state` and a `computation` block, with

optional blocks for datatypes, functions, and procedures. This language provides a transparent account of platform engine computations, enabling consistency. Generation considers inputs, constraints, and order when forming the slnDF blocks.

The second stage generates source-code mirroring the slnDF structure. For Example 3.4, this produces a C++ implementation that compiles into a standalone simulation, integrating with code generated using the existing d-modelGen transformation and the prototype mappingGen. The computation block becomes the main function: after initialising state, it executes the slnDF computation in a while loop until program termination. Generators for other languages are discussed in Chapter 2.3.

In summary, the platform engine is fully automatically generated, while the orchestration layer is generated with stubs that must be completed for integration. A detailed breakdown of the generated artefacts is given in Chapter 4 (Section 4.1.3, Table 4.6). In terms of RoboSim semantics, the orchestrator realises the main action

$$\begin{aligned}&\mathit{Init};\ \mu X.\ \mathit{SendToDModel};\\&(\mathit{ReceiveFromDModel} \triangle \mathit{Wait}(\mathit{cycle}));\ \mathit{timer} := 0;\\&(\mathit{Evolve} \triangle (\mathit{timer} \geq \mathit{cycle} \times \mathit{tscale}));\ X\end{aligned}$$

while the generated platform engine implements *Evolve*.

The integration timestep dt used by *Evolve* is supplied from the slnDF. Actuator torques $\boldsymbol{\tau}$ are wired to the platform mapping via fully qualified names that mirror the mapping hierarchy, so names in the slnDF correspond 1:1 to the mapping code. The timer evolves according to $\frac{d\,timer}{dt} = 1$, ensuring cycle timing conforms to RoboSim semantics.

Example 3.4: Simple Arm Serial Manipulator: Source-Code Generation

The slnDF file generated by sourceCodeGen for the simple arm serial manipulator is presented in Listing 3.2 (line numbers shown). The excerpt begins with an (empty) `datatypes` block (lines 2–3), followed by the `state` block (lines 4–11), which declares global variables such as $\boldsymbol{\tau}$ (line 5) and the rigid-shift operator ϕ (line 6). The time-step for a cycle in the platform engine is given by dt (line 10).
The `procedures` block begins at line 12; the excerpt shows the procedure `SKOm_set` (lines 13–19), which sets the $(x, y)^{\text{th}}$ 6×6 submatrix of a spatial operator. The excerpt also includes part of the `functions` block (starting at line 23), showing `CalcPhi` (lines 24–31), which computes the rigid-shift operator block $\phi_{m,n}$ for links m and n. Finally, the `computation` block begins at line 35: lines 36–41 wire actuator torques from the platform mapping into $\boldsymbol{\tau}$, and lines 42–48 assemble ϕ by calling `SKOm_set` with either `Identity` blocks or `CalcPhi` blocks. The `datatypes` block is empty in this example because no additional datatypes are introduced.
The generated C++ excerpts are collected under "Generated C++ excerpts for the Simple Arm" in Appendix A.2: Listing A.13 presents the platform engine code for SimpleArm-Headless, and Listing A.14 presents the associated orchestrator. sourceCodeGen generates these two main components: the *platform engine* (platform1_engine.cpp) that implements the physics computation, and the *orchestrator* (orchestrator.cpp) that implements the RoboSim operational semantics from Chapter 2. The generated code is designed to respect the main action semantics described in Chapter 2, ensuring that the platform engine and orchestrator correctly implement the Init, SendToDModel, ReceiveFromDModel, Wait, and Evolve actions. The details of how sourceCodeGen achieves this are presented in Section 3.5.
The platform engine follows the structure of the slnDF file. In Listing A.13, state variables

(lines 2–7) include τ, ϕ, θ, $\dot{\theta}$, $\ddot{\theta}$, and the mass matrix $\mathcal{M}$, defined using the Eigen library for linear algebra. Procedures (lines 11–14) include `SKOm_set` for setting spatial operator matrix blocks and utility functions for matrix operations. Functions (lines 18–26) include `CalcPhi` for computing the rigid-shift operator ϕ and other helper functions. The main computation is in `physics_update` (lines 30–51), which implements the forward dynamics pipeline: computing τ from the platform mapping, evaluating ϕ, computing inverse dynamics $\boldsymbol{C}$, computing the mass matrix $\mathcal{M}$ via the composite body algorithm, inverting it using Cholesky decomposition, computing forward dynamics to obtain $\ddot{\theta}$, and integrating to obtain $\dot{\theta}$ and θ.

Listing A.14 shows excerpts from the orchestrator, which manages the simulation lifecycle and ensures both d-model and p-model processes respect the RoboSim semantics. The orchestrator implements the main action from Chapter 2: *Init* initialises the physics runtime, *SendToDModel* and *ReceiveFromDModel* handle communication via register channels using the shared state structures (lines 5–8), *Wait* monitors cycle boundaries, and *Evolve* (lines 11–29) performs continuous integration. The `Evolve` function first calls `EvolveWorldMapping()` to update sensor readings—this world mapping can be ignored for this example as it simply ensures sensor values are updated each timestep; details of the world mapping implementation are presented in Chapter 2.3. After updating sensors, `Evolve` waits for the timestep duration, updates the platform physics via `platform_engine->update()`, advances the timer according to $d\,\text{timer}/dt = 1$, and returns whether the cycle is complete. The orchestrator is generated with integration stubs: line 2 declares `extern "C" int pickplace_main(...)` expecting the d-model's main function from d-modelGen, and lines 33–52 define `DModelIOAdapter` and the `main()` function. `DModelIOAdapter` provides the interface stubs `registerRead`, `registerWrite`, and `tock` that implement the RoboSim channel semantics for communication between the d-model and p-model. The `main()` function orchestrates the complete simulation: initialising the physics runtime, registering the IO adapter to connect the d-model channels, launching the d-model via `pickplace_main`, and shutting down the engine when complete. These stubs are filled in based on the code generated by the existing d-modelGen transformation and the prototype mappingGen, integrating all three components into a complete executable simulation.

With the prototype mappingGen enabled, the additional integration stubs that were previously manual are filled automatically: the platform-mapping data structure (`platform_mapping.h`), the mapping adapter that mirrors sensor outputs into `p_mapping` and applies operations (`platform_mapping_adapter.cpp`), and the C-ABI d-model interface used by `registerRead/registerWrite` (`dmodel_interface.h/dmodel_interface.cpp`). What remains for the user is the environment-specific world engine/world mapping code and any runtime configuration (e.g., cycle period and initial world state), along with the d-model code produced by the existing d-modelGen transformation.

```
1  Solution {
2     datatypes {
3     }
4     state {
5        tau : vec(2) = zeroVec(2) ;
6        phi : mat(18,18) = zeroMat(18, 18);
7
8        . . .
9
10       dt : float = 0.1;
11    }
12    procedures {
13       procedure SKOm_set(val-res modifier: mat()
```

```
14        , val x: int
15        , val y: int
16        , val input: mat(6,6)
17        ) {
18           submatrix(modifier)(6 * x, 6 * y, 6, 6) = input;
19        }
20
21        . . .
22
23        functions {
24           function CalcPhi(m: int, n: int, Bk: seq(mat(4,4))): mat(6,6) {
25
26           submatrix(CalcPhi)(0, 0, 3, 3) == Identity(3, 3);
27
28           submatrix(CalcPhi)(0, 3, 3, 3) == lx(getFramePosition(m, Bk),
                 getFramePosition(n, Bk));
29
30           submatrix(CalcPhi)(3, 0, 3, 3) == Identity(3, 3);
31
32              }
33        . . .
34
35        computation {
36        {
37           tau[0] = p_mapping.SimpleArm.IntermediateLink.WristJoint.
38              WristActuator.TorqueIn;
39           tau[1] = p_mapping.SimpleArm.BaseLink.ElbowJoint.
40              ElbowActuator.TorqueIn;
41        }
42         {
43              SKOm_set(phi, 0, 0, Identity(6, 6));
44        SKOm_set(phi, 1, 0, CalcPhi(2, 1, Bk));
45        SKOm_set(phi, 1, 1, Identity(6, 6));
46        SKOm_set(phi, 2, 0, CalcPhi(3, 1, Bk));
47        SKOm_set(phi, 2, 1, CalcPhi(3, 2, Bk));
48        SKOm_set(phi, 2, 2, Identity(6, 6));
49        }
50
51       . . .
```

Listing 3.2: Excerpts from the Solution Description Format (slnDF) file generated by Technique 5 for Simple Arm.

When running a simulation, if a problem is found, given the level of automation of techniques, it is possible to return to the models and reapply them as appropriate. Problems here are defined loosely as observed physical structure or behaviour that does not match what is expected.

3.2 Technique 3: EquationComposition

This section describes eqnComp by first providing an overview of the technique and its corresponding algorithm, including each step and its purpose. The details of each step of the algorithm are then elaborated on in the following subsections. We continue to present this technique using the simple 2R manipulator example introduced in Example 3.1.

3.2.1 Overview

The algorithm describing **Technique 3: Equation Composition** (eqnComp) takes as input a p-model and returns as output an extended p-model including system-level equations.

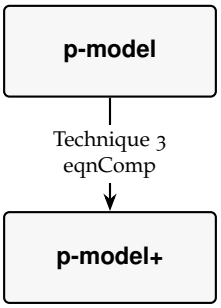


Figure 3.6: Technique 3: Equation Composition. This technique takes a p-model and returns an extended p-model with system-level equations.

Algorithm 1 has six steps. The algorithm accepts an optional *overrideFormulation* parameter that is passed to Step 1; when provided, the dynamics formulation is not auto-detected but uses the specified value instead. This supports edge cases such as single-link robots where no joints exist from which to infer the formulation type. The first step calculates the formulations to be used (using the override for dynamics if specified, otherwise auto-detecting), the second step localises reference components (joints, links, sensors, actuators, and bodies) in the p-model, and the third step validates that the generator can generate for the input model and formulations. The fourth step determines the topology of the robot. These four initial steps preprocess the p-model so that all the information required for the subsequent calculations is available. The fifth step assigns numbering to the links and joints of the robot. The sixth step calculates equations for all components (dynamics, sensors, and actuators) using the equation library, which is further detailed in Section 3.2.7. This involves defining datatypes and functions, calculating component-level equations for links, joints, sensors, and actuators, and finally determining the system-level equations.

Algorithm 1: eqnComp

Input: pmodel = SimpleArm: *pmodel*, overrideFormulation: *Formulation* (optional)
Output: pmodel_ : *pmodel+*
Variables: Topology: *{SerialChain, KinematicTree, ClosedChain}*, formulations: *Formulations*, dynamicsFormulation: *Formulation*

```
/* Step 1: Calculate Formulations (uses override if provided) */
formulations = calculateFormulations(pmodel, overrideFormulation);
dynamicsFormulation = formulations.dynamicsFormulation;
/* Step 2: Localise reference components */
pmodel_ = localiseRefs(pmodel);
/* Step 3: Validate generator compatibility */
eqnCompValidator(pmodel_, dynamicsFormulation);
/* Step 4: Determine topology */
topology = calculateTopology(pmodel_);
/* Step 5: Assign numbering to Links And Joints */
pmodel_ = assignNumbering(pmodel_, dynamicsFormulation);
/* Step 6: Calculate equations for all components */
pmodel_ = calculateEquations(pmodel_, topology, formulations);
return pmodel_;
```

3.2.2 Step 1: Calculate Formulations

The first step of eqnComp is to calculate the formulations for the robot dynamics, sensors, and actuators. This step analyses the p-model components to determine which mathematical formulation types should be used for each component category. Common formulations include SKO (Spatial Kernel Operators), Featherstone, and trivial formulations, though the algorithms presented here focus on SKO detection as an example.[1] Additional formulation types can be detected through similar pattern-matching approaches. These formulation types are then used in Steps 3, 5, and 6 of Algorithm 1.

Algorithm 2 shows the *calculateFormulations* procedure, which takes a *pmodel* and an optional *overrideFormulation* as input and returns either a *Formulations* object or *null* if no dynamics formulation can be determined. The *Formulations* object comprises a single *dynamicsFormulation* for the whole p-model, plus two maps that associate fully qualified component names with their detected formulation types (one map for sensors, one map for actuators).

The algorithm consists of four main steps: (Step 1) if an *overrideFormulation* is provided, it uses that value directly; otherwise it calls *findDynamicsFormulation* to determine the *dynamicsFormulation* based on analysing all joints in the p-model; (Step 2) it calls *findSensorFormulations* to build a *sensorFormulationMap* that maps each sensor's fully qualified name to its detected formulation type; (Step 3) it calls *findActuatorFormulations* to create an *actuatorFormulationMap* with similar mappings for actuators; and (Step 4) if a valid *dynamicsFormulation* is found (not null), it constructs and returns a *Formulations* object containing the dynamics formulation and both maps, otherwise it returns *null*.

The **Formulations data structure** contains three key fields: a *dynamicsFormulation* field that stores the single formulation type to be used for the entire p-model's dynamics (determined by

Algorithm 2: calculateFormulations

Input: *pmodel* : *PModel*, *overrideFormulation* : *Formulation* (optional)
Output: *result* : *Formulations*

```
   /* Step 1: Find dynamics formulation (use override if provided, otherwise
      auto-detect)                                                          */
1  if overrideFormulation ≠ null then
2  |  dynamicsFormulation = overrideFormulation;
3  end
4  else
5  |  dynamicsFormulation = findDynamicsFormulation(pmodel);
6  end
   /* Step 2: Find sensor formulations - returns map                        */
7  sensorFormulationMap = findSensorFormulations(pmodel);
   /* Step 3: Find actuator formulations - returns map                      */
8  actuatorFormulationMap = findActuatorFormulations(pmodel);
   /* Step 4: Return Formulations containing all formulations              */
9  if dynamicsFormulation ≠ null then
10 |  return Formulations(dynamicsFormulation, sensorFormulationMap, actuatorFormulationMap);
11 end
12 return null;
   /* No dynamics formulation found                                        */
```

[1] Throughout the algorithms in this chapter, the notation $\{\}$ denotes an empty ordered collection (set or map) that preserves insertion order. The **foreach** construct iterates over elements in their insertion order. The union operation $S = S \cup \{x\}$ appends element x to the end of S, maintaining the existing order. These ordering guarantees ensure that algorithms produce deterministic output: given the same input, the algorithms will always process elements in the same sequence and generate identical results.

analysing all joints for consistency), and *sensorFormulation* and *actuatorFormulation* maps that associate each sensor and actuator's fully qualified name with its specific detected formulation type.

Algorithm 3 details the *findDynamicsFormulation* procedure used in Step 1 of Algorithm 2. This procedure takes a *pmodel* as input and returns either a specific formulation type (such as SKO) or *null* if no consistent formulation can be determined.

The algorithm proceeds in three stages. First, it iterates through all *links* in the p-model and, for each link, iterates through its *joints*, adding joints to the collected set. Second, it checks formulation compatibility for each collected joint while tracking whether any joints were encountered. For *LocalJoint* instances it invokes *checkLocalJointForH(j)* to search for the characteristic "H" variable (SKO); for *ReferenceJoint* instances it invokes *checkReferenceForSKO(j)* to look for SKO library patterns; unknown joint types are treated as incompatible. If any joint is found incompatible with a formulation, the corresponding compatibility flag (e.g., *allJointsAreSKO*) is cleared. Third, it determines the return value in a prioritised manner: if joints were found and all are SKO-compatible it returns SKO; if joints were found but no single formulation matches, it returns *null*; if no joints were found, it returns *null*. Similar checks for other formulations can be added using the same structure.

For edge cases where the formulation cannot be automatically detected (such as single-link robots without joints), the user can pass an *overrideFormulation* parameter to bypass auto-detection for dynamics (see Section 4.1.1).

The helper functions *checkLocalJointForH* and *checkReferenceForSKO* encapsulate the specific logic for identifying SKO-compatible joints based on their type. Similar formulation detection procedures are applied to sensors and actuators as shown in Appendix A.2.1.1.

The formulation detection process analyses component references and local definitions to determine their types. For sensors and actuators specifically, the generator can detect various formulation types including "trivial" (from the physmod::trivial library), "unknown" (unrecognised types), and "none" (no components present). For dynamics (joints), different formulation types such as SKO and Featherstone are detected as shown in Algorithm 3. This approach enables the generator to select appropriate equation generation methods for different components.

The components are checked in the order of joints (for dynamics), then sensors, then actuators. This is because the dynamics form the primary equations of motion, while sensors and actuators are usually seen as additional constraints on the system. This order is important, since we assume we are modelling a whole robot, and not just a sensor or actuator; therefore the algorithm returns *null* if no formulation is found for the dynamics, indicating that the p-model cannot be processed.

For the sake of the speed of this algorithm, we assume that the p-model is well-formed, meaning that formulations are consistent across different components. We do not validate this assumption in Step 1; it could instead be checked by the equation composition validator during preprocessing (see Section 3.2.3 for eqnCompValidator).

Algorithm 3: findDynamicsFormulation (General)

Input: $pmodel$: $PModel$
Output: $formulation_{dyn}$: $Formulation_{dyn}$

```
   /* Step 1: Collect all joints from links                                  */
1  joints = {};
2  foreach link ∈ pmodel.links do
3  |   foreach joint ∈ link.joints do
4  |   |   if joint ≠ null then
5  |   |   |   joints = joints ∪ {joint};
6  |   |   end
7  |   end
8  end
   /* Step 2: Check if all joints are SKO compatible                         */
9  allJointsAreSKO = true;
10 hasJoints = false;
11 foreach j ∈ joints do
12 |   hasJoints = true;
13 |   if j instanceof LocalJoint then
14 |   |   isSKOJoint = checkLocalJointForH(j);
   |   |   /* Look for H variable                                           */
15 |   end
16 |   else if j instanceof ReferenceJoint then
17 |   |   isSKOJoint = checkReferenceForSKO(j);
   |   |   /* Check SKO patterns                                           */
18 |   end
19 |   else
20 |   |   isSKOJoint = false;
   |   |   /* Unknown joint type                                           */
21 |   end
22 |   if ¬isSKOJoint then
23 |   |   allJointsAreSKO = false;
24 |   end
25 end
   /* Step 3: Return formulation based on analysis                           */
26 if hasJoints ∧ allJointsAreSKO then
27 |   return SKO;
28 end
   /* Additional formulation checks (e.g., Featherstone) can be added here
      following the same pattern.                                           */
29 else if hasJoints then
30 |   return null;
   |   /* Mixed or unrecognised joints                                     */
31 end
32 else
33 |   return null;
   |   /* No joints found                                                  */
34 end
```

Example 3.5: SimpleArm Serial Manipulator: Calculate Formulation from References

In this example, we demonstrate how Algorithm 3 processes reference joints to determine the SKO formulation. When the algorithm encounters a *ReferenceJoint* (Step 2), it calls *checkReferenceForSKO* to examine the joint definition. In this case, the joint reference *SKO::revolute_x* contains the "SKO" namespace indicator and matches the pattern "Revolute_x", causing the function to return true. Since all joints in the model are similarly SKO-compatible reference joints, the algorithm sets *allJointsAreSKO* to true and ultimately returns the SKO formulation in Step 3. The resulting *Formulations* object contains SKO for dynamics, with separate maps for trivial sensor and actuator formulations as shown below.

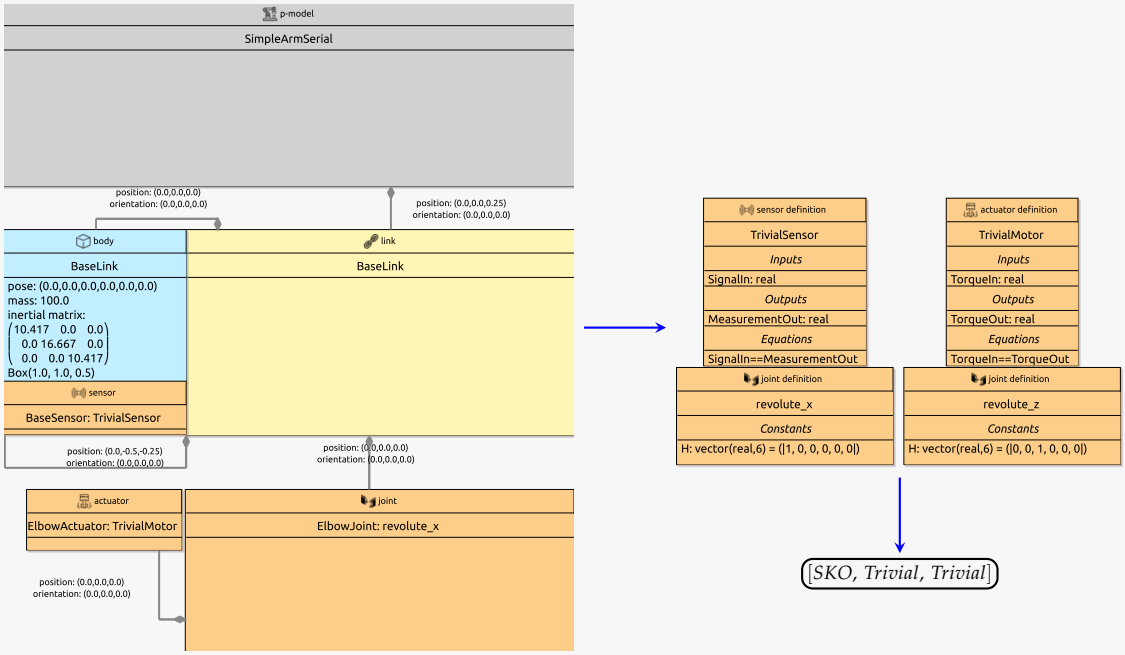


Figure 3.7: Calculating the formulation by checking reference components.

Example 3.6: SimpleArm Serial Manipulator: Calculate Formulation from Local Components

This example illustrates how Algorithm 3 processes local joints. When encountering a *LocalJoint* instance in Step 2, the algorithm calls *checkLocalJointForH*, which searches through the joint's definition constants. In Fig. 3.8, the local joint contains a constant $constH : vector(real, 6) = (|1, 0, 0, 0, 0, 0|)$. The presence of this variable named "H" is the characteristic marker of SKO formulation in local joints, causing *checkLocalJointForH* to return true and setting *isSKOJoint* to true.

When all joints in the model similarly contain the H variable (whether as vectors for revolute joints or matrices for other joint types), the algorithm maintains *allJointsAreSKO* as true throughout Step 2. In Step 3, with *hasJoints* and *allJointsAreSKO* both true, the algorithm returns SKO formulation. The complete *Formulations* object then contains SKO for dynamics and separate mappings for each sensor and actuator to their respective trivial formulations.

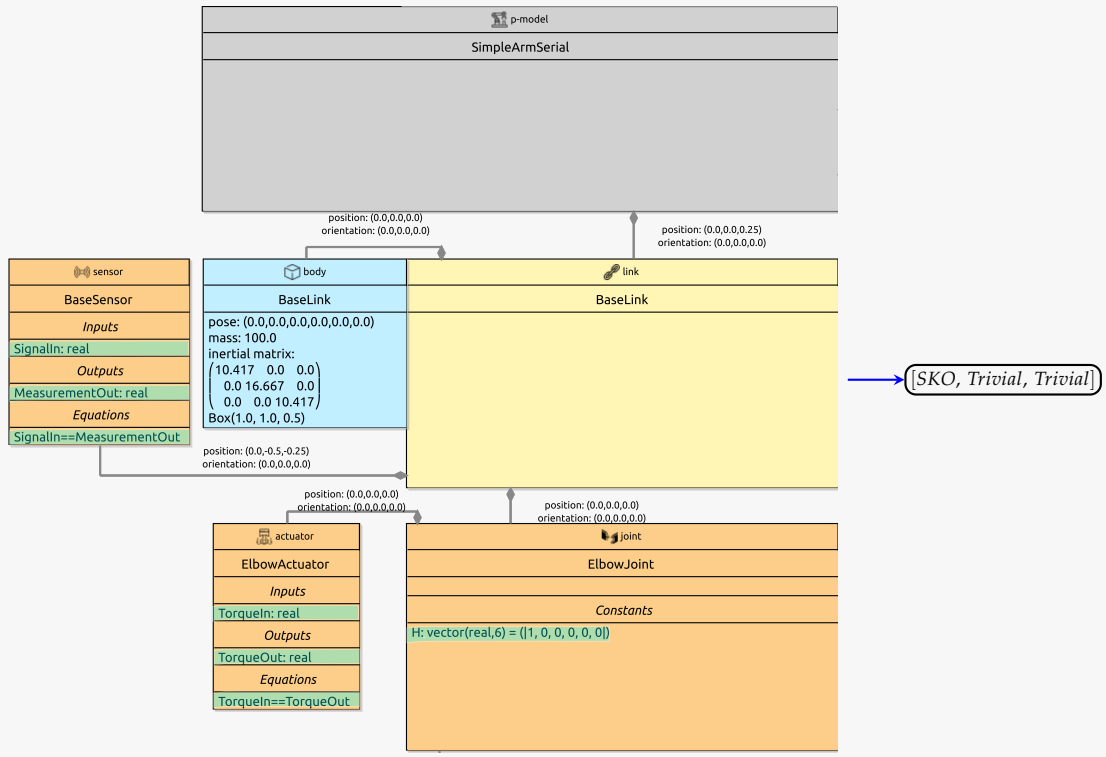


Figure 3.8: Calculating the formulation by checking variables and equations of local components.

It is important to note that formulation detection based on references (as shown in the first example) is preferred over detection based on local component definitions. Reference-based detection ensures consistency since all components reference well-defined library implementations with known formulation types. In contrast, local component definitions may inadvertently mix characteristics from different formulations, potentially leading to conflicts or ambiguous formulation determination. For instance, a locally-defined joint might contain variables that match patterns from both SKO and another formulation, making it impossible to determine a single consistent formulation. Therefore, the algorithm prioritises reference-based detection when available, falling back to local component analysis only when necessary.

3.2.3 Step 2: Localise Reference Components

The **localise references** procedure first expands any part instantiations and then transforms all reference components in the p-model (joints, links, sensors, actuators, bodies) into their local counterparts. This is necessary because reference components cannot be augmented with additional variables and equations that will be added in subsequent steps (*assignNumbering* and *generateEquations*). This section presents the *joints case*; analogous procedures for links, sensors, actuators, and bodies are given in Appendix A.2.1.2.

Algorithm 4: localiseRefs

Input: *pmodel* : *PModel*
Output: *pmodel_* : *PModel*

```
1 pmodel_ = pmodel;
2 expandParts(pmodel_);
3 convertReferenceJointsToLocal(pmodel_);
4 convertReferenceLinksToLocal(pmodel_);
5 convertReferenceSensorsToLocal(pmodel_);
6 convertReferenceActuatorsToLocal(pmodel_);
7 convertReferenceBodiesToLocal(pmodel_);
8 return pmodel_;
```

Algorithm 4 summarises the localisation. It first expands reusable part instantiations into links and connections before converting reference components. The joint case is given in Algorithm 5, which scans the model to collect reference joints and create their local counterparts (lines 1–10), performs an in-place substitution by overwriting each reference at its original index (lines 12–16), and finally returns the updated model (line 18). This ensures subsequent stages operate only on localised components.

Algorithm 5: convertReferenceJointsToLocal

Input: *pmodel* : *PModel*
Output: *pmodel* with all reference joints converted to local joints

```
1  foreach link ∈ pmodel.links do
2      jointsToReplace = {};
3      newJoints = {};
4      foreach joint ∈ link.joints do
5          if joint instanceof ReferenceJoint then
6              localJoint = createLocalJoint(joint);
7              jointsToReplace.add(joint);
8              newJoints.add(localJoint);
9          end
10     end
       /* Replace reference joints with local joints                    */
11     foreach i ∈ [0, size(jointsToReplace)) do
12         index = link.joints.indexOf(jointsToReplace[i]);
13         if index ≥ 0 then
14             link.joints[index] = newJoints[i];
15         end
16     end
17 end
18 return pmodel;
```

The procedure *createLocalJoint*, shown in Algorithm 6, first allocates a new *LocalJoint* and copies the original joint's name, pose, and flexible connection (lines 1–4) so that the joint remains named, positioned, and attached to the same link. When a definition is present, it deep-copies that definition and unsets its `name` to avoid duplicate identifiers once embedded locally (lines 5–7). The p-model metamodel allows sensors and actuators to be contained within joints (in addition to links); accordingly, lines 8–11 deep-copy any sensors or actuators that are nested inside the reference joint, preserving the containment structure. The constructed local joint is then returned (line 12).

Algorithm 6: createLocalJoint

Input: *refJoint* : *ReferenceJoint*
Output: *localJoint* : *LocalJoint*

```
1  localJoint = new LocalJoint();
2  localJoint.name = refJoint.name;
3  localJoint.pose = copy(refJoint.pose);
4  localJoint.flexible = copy(refJoint.flexible);
5  if refJoint.def ≠ null then
6  |   localJoint.def = copy(refJoint.def);
   |   /* Remove name from copied definition                    */
7  |   localJoint.def.unset("name");
8  end
   /* Preserve nested sensors and actuators                     */
9  foreach sensor ∈ refJoint.sensors do
10 |   localJoint.sensors.add(copy(sensor));
11 end
12 foreach actuator ∈ refJoint.actuators do
13 |   localJoint.actuators.add(copy(actuator));
14 end
15 return localJoint;
```

Example 3.7: Simple Arm Serial Manipulator: Converting Reference Joints

In this example, each reference joint (e.g., *SKO::revolute_x*) is replaced by a local joint in the same position within *link.joints*. The joint's name, pose, and flexible link association are preserved; the joint becomes self-contained as per Algorithm 6. This localisation prepares the model for subsequent numbering and equation generation. The transformation is illustrated in Fig. 3.9.

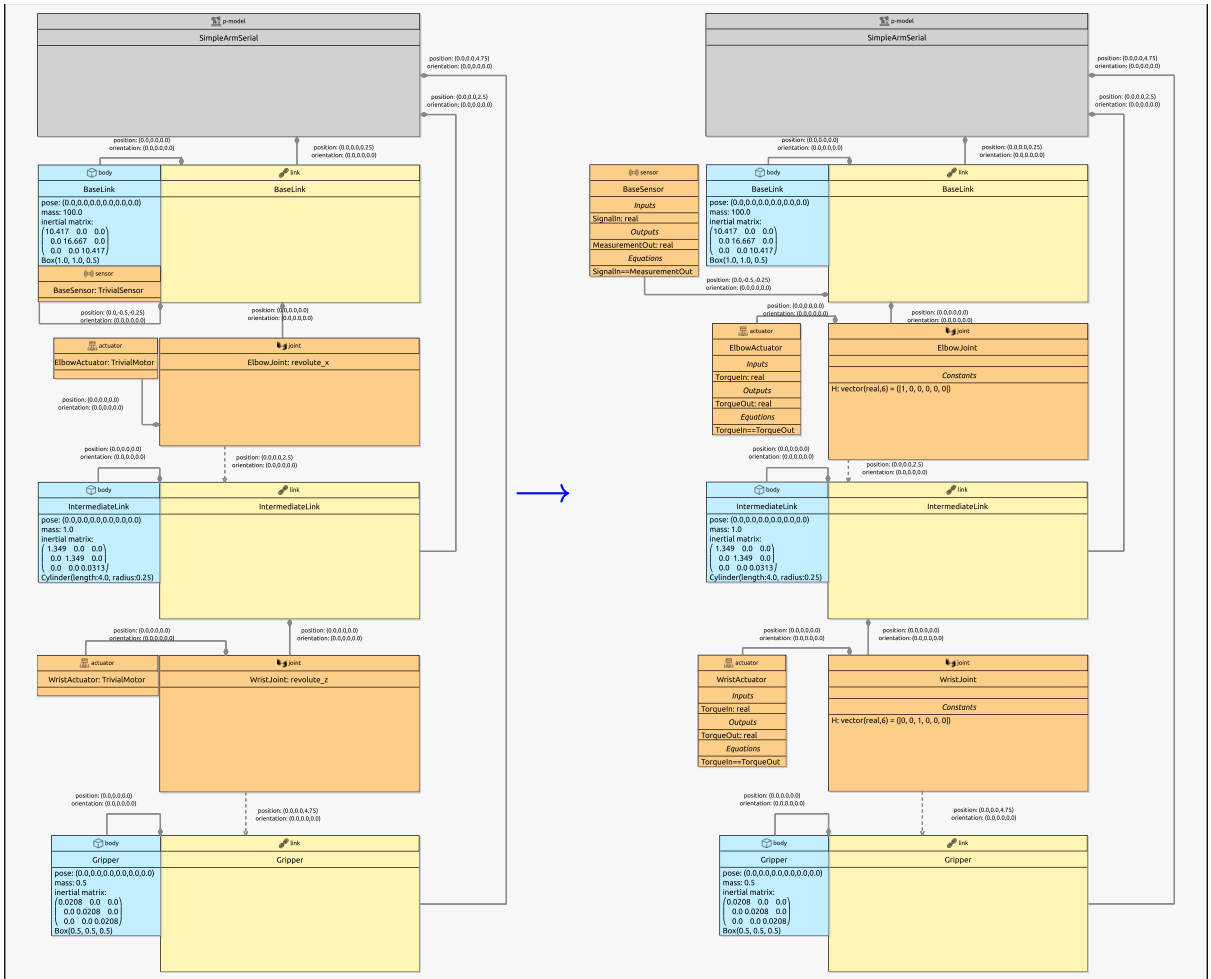


Figure 3.9: Translating a p-model with reference components to one with localised components.

3.2.4 Step 3: Validate Generator Compatibility

The **equation composition validator** checks that the p-model satisfies the formulation-specific well-formedness conditions required by the selected formulation. For the SKO formulation it verifies the availability of inertia information on links, which is necessary for dynamics computations.

In Step 3, the validator ensures that the p-model meets the formulation-specific requirements before equations are generated. General well-formedness constraints (such as links existing, being properly named, and having pose information) are enforced by the p-model's metamodel validation rules. The equation composition validator focuses on formulation-specific requirements. For the SKO formulation, the key requirement is that each link must have either a direct inertial definition or at least one body whose definition specifies inertial properties.

The algorithm verifies that each link has valid inertia information. If a link has neither direct inertia nor a body with inertia, validation fails and equation generation is aborted. Additional details about validation requirements for different formulations are provided in Appendix A.2.1.3.

Algorithm 7: validate (SKO formulation)

Input: *pmodel* : *PModel*
Output: *valid* : *Boolean*

```
// SKO-specific inertia requirements
foreach link ∈ pmodel.links do
    hasInertia ← (link.def.inertial ≠ null) or ∃ b ∈ link.bodies : b.def.inertial ≠ null;
    if ¬hasInertia then
        return false
    end
end
return true
```

Example 3.8: Simple Arm Serial Manipulator: Validation

In the Simple Arm Serial example under the SKO formulation, the validator checks that each link has valid inertia either directly or via a body. Because the model satisfies this requirement (each link has bodies with defined inertial properties), validation succeeds and the pipeline proceeds to topology calculation.

3.2.5 Step 4: Determine Topology

The fourth step calculates the robot topology based on the number of links and joints, and the number of connections between them. The function calculateTopology identifies serial chains, kinematic trees, and closed chains, which are all the possible topologies for a robot model. This step is crucial since the connectivity of the robot model influences the equations that can be generated alongside the formulation. For example, different methodologies such as constraint embeddings apply depending on whether the topology is open or closed chain. Additional details about topology types and detection logic are provided in Appendix A.2.1.4.

Algorithm 8: calculateTopology

Input: *pmodel* : *pmodel*
Output: Topology : {*SerialChain*, *KinematicTree*, *ClosedChain*}

```
if #pmodel.links = #getJoints(pmodel) + 1 then
    if ∃ link ∈ pmodel.links : #link.joints > 1 then
        Topology ← KinematicTree;
    end
    else
        Topology ← SerialChain;
    end
end
else
    Topology ← ClosedChain;
end
return Topology;
```

Example 3.9: Simple Arm Serial Manipulator: Topology

The function calculateTopology identifies serial chains, kinematic trees, and closed chains. In our example, we have a SerialChain: the number of links is one more than the number of joints, with the number of (fixed) link connections being equal to the number of links.

3.2.6 Step 5: Assign Numbering

The **assign numbering** step determines the appropriate ordering and numbering scheme for components based on their detected formulation types from Step 1.

This step involves several key algorithms. The main `assignNumbering` algorithm determines which numbering strategy to use based on the formulation type (e.g. Featherstone or SKO). The `orderPmodel` algorithm assigns sequential numbers to links and joints in the physical model following either a regular or reversed traversal of the spanning tree. The `getBaseLink` algorithm identifies the base link by determining which link contains the most child links in the model structure. Finally, the `getChildLinks` algorithm recursively identifies all descendant links for a given link in the model hierarchy. The full details of these algorithms are provided in Appendix A.2.1.5.

Example 3.10: Simple Arm Serial Manipulator: Numbering

The required order depends on the formulation. For the SKO formulation, we use a regular numbering as in [35, p.69] and then reverse the numbering so that it is consistent with that of the SKO formulation in [24]. The output of this step is shown in Fig. 3.10. The numbering is assigned to the links and joints.

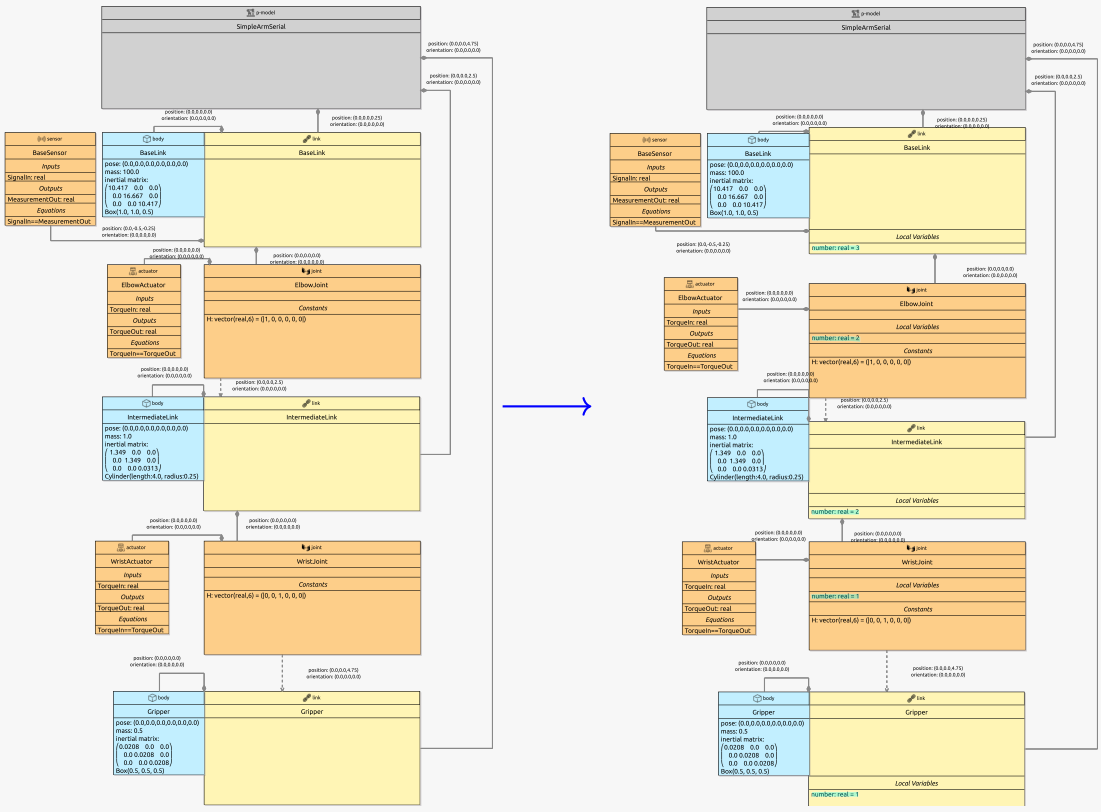


Figure 3.10: Assigning numbering to the links and joints of the p-model.

3.2.7 Step 6: Calculate Equations

Step 6 transforms the augmented p-model from Step 5 by generating the complete set of equations needed for simulation. This includes dynamics equations for the robotic platform, as well as sensor and actuator equations. The step delegates to the equation library (Section 4.1.1),

which generates equations according to the detected formulations and topology from previous steps.

Algorithm 9: calculateEquations

```
Input: pmodel : PModel, topology : Topology, formulations : Formulations
Output: pmodel′ : PModel

/* Resolve solution order based on formulation combination                    */
solutionOrder = resolveSolutionOrder(formulations);
if solutionOrder ≠ null then
    /* Execute in resolved custom order                                       */
    foreach step ∈ solutionOrder do
        if step.type == "dynamics" then
            pmodel′ = lib.calculateEquations(pmodel, topology, formulations.dynamicsFormulation);
        else if step.type == "sensors" then
            foreach sensorFQN ∈ dom(formulations.sensorFormulation) do
                formulation = formulations.sensorFormulation[sensorFQN];
                sensor = findSensorByFQN(pmodel′, sensorFQN);
                if sensor ≠ null then
                    lib.calculateEquations(sensor, topology, formulation);
                end
            end
        else if step.type == "actuators" then
            foreach actuatorFQN ∈ dom(formulations.actuatorFormulation) do
                formulation = formulations.actuatorFormulation[actuatorFQN];
                actuator = findActuatorByFQN(pmodel′, actuatorFQN);
                if actuator ≠ null then
                    lib.calculateEquations(actuator, topology, formulation);
                end
            end
        end
    end
else
    /* Execute in default order:  dynamics, sensors, actuators                */
    /* Step 6a:  Calculate dynamics equations                                 */
    pmodel′ = lib.calculateEquations(pmodel, topology, formulations.dynamicsFormulation);
    /* Step 6b:  Calculate sensor equations                                   */
    foreach sensorFQN ∈ dom(formulations.sensorFormulation) do
        formulation = formulations.sensorFormulation[sensorFQN];
        sensor = findSensorByFQN(pmodel′, sensorFQN);
        if sensor ≠ null then
            lib.calculateEquations(sensor, topology, formulation);
        end
    end
    /* Step 6c:  Calculate actuator equations                                 */
    foreach actuatorFQN ∈ dom(formulations.actuatorFormulation) do
        formulation = formulations.actuatorFormulation[actuatorFQN];
        actuator = findActuatorByFQN(pmodel′, actuatorFQN);
        if actuator ≠ null then
            lib.calculateEquations(actuator, topology, formulation);
        end
    end
end
return pmodel′;
```

Algorithm 9 proceeds in two main execution paths. First, it calls `resolveSolutionOrder` to determine whether the combination of formulations requires a specific execution order. If a custom order is resolved (for example, when sensors must be processed before dynamics), the algorithm executes dynamics, sensor, and actuator equation generation in that custom order. Otherwise, it uses the default order described below.

In the default execution path, the algorithm performs three sequential substeps. Step 6a generates dynamics equations for the entire p-model using the detected dynamics formulation, calling the equation library's `calculateEquations` method with the p-model and formulation type, which returns an augmented p-model with system-level equations. Step 6b iterates over the domain of the sensor formulation map (dom($formulations.sensorFormulation$) denotes the set of all fully qualified sensor names for which formulations have been detected), looks up each sensor by its fully qualified name (FQN) using `findSensorByFQN`, and generates equations based on the sensor's detected formulation type (such as trivial or custom sensor models). Step 6c performs the same process for actuators, iterating through the domain of the actuator formulation map and generating equations for each actuator using `findActuatorByFQN` to locate components.

The **equation library** provides an interface for equation generation across different component types and formulation types. For dynamics (PModel-level equations), the library delegates to formulation-specific implementations such as SKO or Featherstone based on the detected formulation and topology. For sensors and actuators, the library generates equations based on their detected formulation types.

The need for a library arises because equation generation is inherently formulation-specific: each dynamics formulation (SKO, Featherstone, etc.) produces different datatypes, variables, and equations based on its underlying mathematical framework. Rather than embedding this formulation-specific logic directly in the generator, each formulation is written as a specification in the slnDF language introduced in Section 3.3.2. The `calculateEquations` method interprets these specifications at runtime, executing the meta-operations that add datatypes, variables, equations, constraints, and relations to the p-model. Each formulation specification follows a common four-step structure: (1) define datatypes and functions, (2) define system variables, (3) generate component equations, and (4) generate system equations.

The lookup helpers (`findSensorByFQN`, `findActuatorByFQN`) parse fully qualified names in the format "linkName.componentName" and search through the p-model's hierarchical structure to locate specific components. Complete details of the lookup functions and equation library interfaces are provided in Appendix A.2.1.8 and Appendix A.2.1.9.

Example 3.11: Simple Arm Serial Manipulator: Equation Generation

We illustrate the algorithm's execution using the Simple Arm Serial Manipulator with the SKO formulation from [24]. The system consists of two links connected by revolute joints. In Step 6a, the equation library invokes the SKO-specific equation generator for serial chains, which follows a four-step internal process: (1) *defineDatatypesAndFunctions* creates the `Geom` datatype and utility functions (`identity`, `zeroes`, `Phi`, `zeroVec`, `zeroMat`); (2) *defineSystemVariables* declares system-level variables including state variables (θ, τ, C), kinematic variables (B_k, V, α), dynamic variables (f, M, H, M_{mass}), and structural constants ($n = 2$ for link count); (3) *componentEquations* performs two passes: the first declares component-local variables (m_k, I_k, M_k,pose variables, geometry variables) for each link and joint, the second constructs relations between system and component variables and generates component-level dynamics equations; (4) *systemEquations* assembles system-level matrices through diagonal-block constraints and formulates the complete SKO dynamics equations. The detailed algorithms for these four steps are provided in Appendix A.2.1.7. In Steps 6b and 6c, the algorithm processes sensors and actuators using their trivial formulations, generating simple input-output equality equations for each component. The complete derivation for the SimpleArm example, including all variable declarations and equation forms, is provided in Appendix A.1.1.

The implementation of the equation library is described in Section 4.1.1. The complete formulation specifications are maintained in the `physmod-libs` repository.

3.3 Solution Domain Specific Languages

This section introduces the solution description languages used in the simulation workflow. The slnRef language records solution references produced by guidedChoice and consumed by sourceCodeGen. The slnDF language is a concrete solution description generated from resolved slnRef entries and used by later code-generation stages.

3.3.1 Solution References (slnRef)

A **Solution Reference**(slnRef) is an implementation-agnostic specification that the modeller adds as an annotation to the p-model and that guidedChoice unfolds and emits. It describes the target solution without specifying the concrete computational implementation. It serves as an intermediate representation between high-level p-model solution specifications and the concrete slnDF format consumed by sourceCodeGen. It can be seen as an annotation to the p-model or in isolation.

Solution reference files (`.slnRef`) are the output of guidedChoice and serve as the input to sourceCodeGen. They provide all information required to solve for a given variable without explicitly providing the solution algorithm itself; that detail is deferred to the Solution Description Format format described in Section 3.3.2. These files contain structured specifications that bridge the gap between high-level solution requests and concrete algorithmic implementations.

Each solution reference contains several fields. The **Expression** field specifies the target variable to be computed as a typed local variable (name and type), for example `Local { name theta type "vector(real,2)" }`. The **Method** field contains the solution method name identifying which algorithm from the solution library should be used, such as `Euler` or `CompositeBodyAlgorithm`. The **Order** field is an integer specifying the processing priority, where lower values are processed first to ensure dependencies are resolved before dependent solutions; for example mass matrix computation (order 1) precedes forward dynamics (order 2). The optional **Group** field is an integer grouping related solutions into distinct computation groups. The **Inputs** field contains a list of typed input variables required by the solution method, where each input specifies a dependency on another variable that must be computed before this solution can execute. The **Constraints** field contains a list of algebraic constraints or initial conditions that must be satisfied, which may specify initial values (e.g., `(theta)[t == 0] == zeroVec(2)`) or algebraic relationships between variables that hold at all times (e.g., `(XJ_1)[t == t] == rotationMatrix(theta_1)`).

Example slnRef

Example 3.12: Solution Reference for Mass Matrix

Computing the mass matrix for the Simple Arm uses the Composite Body Algorithm:

```
SolutionRef {
    expression Local { name M_mass type "matrix(real,2,2)" }
    method CompositeBodyAlgorithm
    order 4
    input Local { name N type "real" }
    input Local { name M type "matrix(real,18,18)" }
    input Local { name phi type "matrix(real,18,18)" }
    constraint " (N) [ t == 0] == 2"
    constraint " (M_mass) [ t == 0] == zeroMat(2, 2)"
}
```

The implementation involves computing the generalised mass matrix $\mathcal{M}$ (a 2×2 matrix for 2 DOF) using the `CompositeBodyAlgorithm` method from SKO dynamics. This process occurs at processing order 4 (mid-sequence, after kinematics and before forward dynamics), requiring multiple inputs including the number of DOF N, spatial inertia matrix M, and rigid-shift operator ϕ. The constraints specify the number of DOF and initial mass matrix value.

slnRef Grammar

The slnRef format is defined by the following EBNF grammar.

Solution Reference File Structure

```
<SlnRefs> ::= <SlnRef>*
```

Solution Reference

```
<SlnRef> ::= "SolutionRef" "{"
             [ "expression" expression <Local> ]
             [ "method" method <QualifiedMethodName> ]
             [ "order" order <INT> ]
             [ "group" group <INT> ]
             <Input>*
             <Constraint>*
             <ErrorDecl>*
           "}"
```

Local Variable Declaration

```
<Local> ::= "Local" "{" "name" name <ID> "type" type <STRING> "}"
```

Inputs and Constraints

```
<Input> ::= "input" value <Local>

<Constraint> ::= "constraint" value <STRING>
```

```
4
5 <ErrorDecl> ::= "error" value <STRING>
```

Token Definitions

```
1 <QualifiedMethodName> ::= <ID> [ "::" <ID> ]
2 <ID> ::= letter (letter | digit | "_")*
3 <INT> ::= [0-9]+
4 <STRING> ::= '"' character* '"'
```

3.3.2 Solution Description Format (slnDF)

The **Solution Description format** (**slnDF**, file extension **`.slnDF`**) is a concrete domain-specific language for explicitly describing solution implementations, including state variables, pure functions, imperative procedures, and the computation sequence. It serves as an interface between sourceCodeGen and platform-specific code generation libraries.

Relative to the XML robot description formats surveyed in Section 2.3.3, slnDF is better suited to transparent, repeatable code generation and verification. Robot description formats like URDF/SDF/MJCF capture structure and parameters (links, joints, meshes, some contact settings), but leave algorithmic intent implicit or simulator-specific (plugins/options). In contrast, slnDF explicitly specifies executable algorithmic semantics: state, pure functions, procedures, and computation order. This engine-agnostic description ensures transparency between the model and the code. Listing 3.3 summarises the main syntactic categories.

The slnDF grammar is summarised by the following EBNF.

```
1 <Solution> ::= "Solution" <ID> "{" [ <DataTypes> ] <State> [
     <Procedures> ] [ <Functions> ] <Computation> [ <Proof> ] "}"
2 <State> ::= "state" "{" <VariableLine>* "}"
3 <VariableLine> ::= <ID> ":" <Type> [ "=" <Expression> ] ";"
4 <DataTypes> ::= "datatypes" "{" <CustomType>* "}"
5 <CustomType> ::= <ID> ( <DataType> | <EnumType> )
6 <DataType> ::= <Field>*
7 <Field> ::= <ID> ":" <Type>
8 <EnumType> ::= <ID> ( "," <ID> )*
9 <Procedures> ::= "procedures" "{" <Procedure>* "}"
10 <Procedure> ::= "procedure" <ID> "(" [ <ParameterDeclaration> ( ","
     <ParameterDeclaration> )* ] ")" "{" <Statement>* "}"
11 <ParameterDeclaration> ::= ("val"|"res"|"val-res") <Variable>
12 <Functions> ::= "functions" "{" <Function>+ "}"
13 <Function> ::= "function" <ID> "(" [ <Variable> ( "," <Variable> )* ]
     ")" ":" <Type> "{" <SemExprSeq>? "}"
14 <SemExprSeq> ::= <SemanticExpression> ( "/\\" <SemanticExpression> )*
15 <Computation> ::= "computation" "{" <Statement>* "}"
16 <Proof> ::= "proof" "{" ( <RelationalExpression> ";" )* "}"
17 <Statement> ::= ...
18 <Expression> ::= ...
19 <Type> ::= ...
```

Listing 3.3: slnDF EBNF Grammar (Summary)

Listing 3.3 summarises the main syntactic categories. The top-level `Solution` construct contains optional `datatypes`, mandatory `state` and `computation` blocks, and optional

`procedures` and `functions` blocks. The State subsection defines state variable declarations with optional initial values, which become member variables in the generated code.

The DataTypes section enables custom type definitions including records (with named fields) and enumerations. These support domain modelling, for example defining a `Params` record to group physical constants or an `IntegratorMode` enum to select integration methods.

The Procedures section defines imperative procedures with parameter modes (`val`, `res`, `val-res`) for call-by-value, call-by-result, and call-by-value-result semantics. Procedures contain statements and can modify state; they are used for operations like matrix assembly that require mutation.

The Functions section is distinguished by purity: function bodies consist only of semantic expressions combined with $\wedge$ (logical conjunction), not statements. Inside function bodies, semantic expressions are equalities (==), loops, or conditionals that define the function result. There are no statement terminators inside functions. Functions use == (semantic equality) rather than = (assignment), reinforcing their declarative nature. This contrasts with procedures and the computation block, where control flow is written as a sequence of *statements* separated by `;` using = for assignment. The $\wedge$ operator is exclusive to function bodies. This distinction is key to allow use of our results also in theorem proving as it facilitates the definition of the semantics of the library components.

The Left-expressions section distinguishes between assignment targets in statements (`LeftExpr`) and equality left-hand sides in function bodies (`FuncLeftExpr`). Both support indexed access, submatrix/subvector slicing, and record field access, but `FuncLeftExpr` additionally allows the function name itself (e.g., `Identity[i,j] == ...`) to define the return value element-wise.

The Statements section defines the imperative constructs available in procedures and the computation block: assignments (=), procedure calls, loops (`for`), conditionals (`if-then-else`), blocks, and the `skip` statement. These use statement terminators (;) and operate by side effects.

The Expressions section defines the standard arithmetic and relational expression grammar with operator precedence (relational < addition < multiplication < unary), supporting vectors, matrices, submatrix/subvector extraction, function calls, and atomic values. The Types section defines primitive types (`int`, `float`, `bool`, `String`), parameterised types (`vec(n)`, `mat(m,n)`, `seq(T)`), and custom type references.

Example 3.13: Simple Arm slnDF Specification

Listing 3.4 provides a short summary of the slnDF specification generated for the Simple Arm example, highlighting the top-level structure and the key language features used by the SKO forward dynamics pipeline.

```
Solution physics {
   state {
      // Declared state with explicit types and initial values
      tau : vec(2) = zeroVec(2);
      C : vec(2) = zeroVec(2);
      d_theta : vec(2) = zeroVec(2);
      dd_theta : vec(2) = zeroVec(2);
      M_mass : mat(2,2) = zeroMat(2, 2);
      M_inv : mat(2,2) = zeroMat(2, 2);
      theta : vec(2) = zeroVec(2);
```

```
11          dt : float = 0.01;
12      }
13      procedures {
14          // Imperative procedures use statements, assignment (=), and
                parameter modes
15          procedure SKOm_set(val-res modifier: mat(), val x: int, val y:
                int, val input: mat(6,6)) {
16              submatrix(modifier)(6 * x, 6 * y, 6, 6) = input;
17          }
18      }
19      functions {
20          // Pure functions use semantic equality (==) and conjunction
                (/\)
21          function skewSymmetric(v: vec(3)): mat(3,3) {
22              skewSymmetric[0,0] == 0 /\
23              skewSymmetric[0,1] == -v[2] /\
24              skewSymmetric[0,2] == v[1] /\
25              // remaining entries
26              skewSymmetric[2,2] == 0
27          }
28      }
29      computation {
30          // The computation block sequences the resolved solution steps
31          {
32              M_inv = LDLT(M_mass);
33              dd_theta = M_inv * (tau - C);
34              d_theta = d_theta + dt * dd_theta;
35              theta = theta + dt * d_theta;
36          }
37      }
38  }
```

Listing 3.4: Summary of the generated Simple Arm slnDF specification

This listing highlights the key language features used in generated slnDF. The state block declares all variables with explicit types and initial values. The procedures block defines imperative steps using = and parameter modes such as `val-res` for mutation. The functions block defines pure computations using == and conjunction (`/\`) without statement terminators. The computation block then sequences the resolved solution steps, in the order determined by the slnRef list emitted by guidedChoice.

Functions

In the slnDF notation, functions are defined using the following syntax structure:

```
function <name>(<parameters>): <return_type> {
    <semantic_expressions>
}
```

The Functions section is distinguished by purity: function bodies consist only of semantic expressions combined with $\wedge$ (logical conjunction), not statements. Inside function bodies, semantic expressions are equalities (==), loops, or conditionals that define the function result. There are no statement terminators inside functions. Functions use == (semantic equality)

rather than = (assignment), reinforcing their declarative nature. This contrasts with procedures and the computation block, where control flow is written as a sequence of statements separated by `;` using = for assignment. The $\wedge$ operator is exclusive to function bodies. This distinction is key to allow use of our results also in theorem proving as it facilitates the definition of the semantics of the library components.

The following well-formedness restriction must be satisfied:

F1 *Every equality inside a function body must define the function's result: the left-hand side must be rooted at the function being defined.* Allowed forms are: the function name itself (whole result), an indexed element of it (e.g., `f[i]` or `f[i,j]`), a submatrix/subvector of it (e.g., `submatrix(f)(...)`, `subvector(f)(...)`), or a record field of it (e.g., `f.field`).

The function syntax provides syntactic sugar that has a precise semantic representation in mathematical form. Inside the body of a function f, occurrences of the function name on the left-hand side refer to the value being defined by the function: a bare `f` denotes the whole return object; `f[i]` (vector results) denotes element i; `f[i,j]` (matrix results) denotes element (i, j); and for record results, `f.field` denotes the named field. The following example illustrates the relationship between syntactic sugar and semantic representation.

Example 3.14: Identity Matrix Function

Grammar (Syntactic Sugar):

```
function Identity(n: int): mat(n,n) {
  for(i: int in range(0, n, 1)) {
    for(j: int in range(0, n, 1)) {
      if (i == j) then Identity[i,j] == 1
      else Identity[i,j] == 0
    }
  }
}
```

Semantic Representation (Mathematical Form):

$$\forall i \mid 0 \leq i < n \bullet \forall j \mid 0 \leq j < n \bullet \big((i = j \wedge \text{Identity}[i, j] = 1) \vee (i \neq j \wedge \text{Identity}[i, j] = 0)\big).$$

This syntactic sugar makes function definitions more readable (it is consistent with the imperative style of the rest of the grammar) whilst maintaining the mathematical rigor of the underlying semantic representation [124].

3.4 Technique 4: Guided Choice of Solutions

This section presents guidedChoice, a technique that addresses the challenge of selecting appropriate solutions from multiple alternatives. We begin with a comprehensive overview of the guidedChoice generator in Section 3.4.1, detailing the rationale behind each step of its algorithm. Subsequent subsections (Sections 3.4.2–3.4.6) provide in-depth explanations of each of these steps and their specific roles in the solution resolution process.

3.4.1 Overview

The algorithm describing **Technique 4: Guided Choice** (guidedChoice) takes a p-model+ annotated with user SolutionRefs and returns the same model with each reference resolved to a slnDF description. Algorithm 10 summarises its five steps.

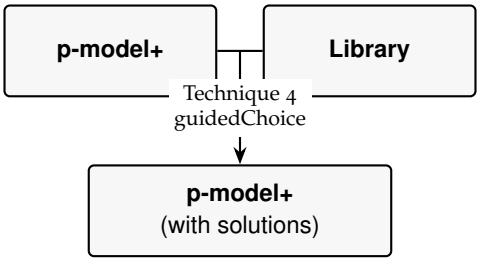


Figure 3.11: Technique 4 (Guided Choice): from a p-model+ and the solution library obtain a p-model+ with resolved solutions.

Algorithm 10: guidedChoice

Input: pmodel : *pmodel+* with user-annotated SolutionRefs
Output: pmodel' : *pmodel+*, slnRefs : *List<SolutionRef>*
Variables: formulation : *Formulation*, `solution_ref` : *List<SolutionRef>*, sln_ref : *SolutionRef*

```
/* Step 1:  Initialise sidecar and add Null type                          */
initialiseSidecar(pmodel);
addNullDatatype(pmodel);
/* Step 2:  Calculate formulation                                         */
formulation = calculateFormulation(pmodel);
/* Step 3:  Unfold user-annotated solutions into sub-solutions (first pass)  */
solution_ref = resolveSolutionFirstPass(pmodel, formulation);
;            /* computes [t==0] initial conditions; simplifies [t==t] algebraic
 constraints */
/* Step 4:  Reorder solutions by their order property                     */
pmodel.solutions.sortBy(solution → solution.order);
solution_ref.sortBy(element → element.order);
/* Step 5:  Resolve each solution in order (with iterations)              */
for each solution in pmodel.solutions do
    sln_ref = first element in solution_ref with matching expression name;
    for i = 0 to sln_ref.iterations do
        resolveSolution(pmodel, formulation, sln_ref);
    end
end
return pmodel, solution_ref;
```

The five steps of Technique 4 transform user-annotated slnRef entries into fully resolved solution references, each linked to a concrete slnDF description. The process begins with the user adding slnRef annotations to the p-model+ produced by eqnComp, specifying which solutions must be instantiated.

Step 1 initialises a sidecar data structure that records the resolution state of every variable and temporarily introduces a Null type so that unresolved symbols can be represented safely; the Null type is removed again at the end of the technique.

Step 2 reuses the formulation detection from eqnComp to select an appropriate mathematical formulation, which guides the generator's solution-library selection. Step 3 performs the initial unfolding: each user annotation is converted into one or more concrete sub-solutions by querying the solution library. Each of these unfolded slnRefs points to a concrete description

in the slnDF format. During this pass the technique aligns the solution expression and inputs with the model's types, adds any new variables required by the unfolded solutions to the p-model, derives initial-condition constraints $[t == 0]$ from available constants, and simplifies algebraic constraints $[t == t]$ that hold for all time. Step 4 orders both the solutions and their references by the `order` attribute so that when Step 5 processes them iteratively, each solution has access to the variables and results produced by earlier solutions.

Step 5 then iteratively refines every solution. Each iteration mirrors the processing pipeline from Step 3, but re-queries the SolutionLib component against the updated p-model state so that deferred selections and staged constraints can be completed. The `iterations/iteration` fields in the in-memory `SolutionRef` track this refinement process during unfolding. The outcome is the original p-model+ augmented with fully resolved, library-backed solutions ready for code generation, along with the accumulated solution references from Steps 3 and 5.

3.4.2 Step 1: Initialise Sidecar and Add Null Type

The first step of guidedChoice is to initialise the sidecar data structure and add a temporary null datatype to the p-model. This preparatory step sets up the infrastructure needed for tracking variable-resolution state and provides a special type for uninitialised values.

The **sidecar data structure** is a global map maintained throughout Technique 4 to track variable resolution across unfolding iterations. It associates the p-model root and each dynamic device (e.g. links, joints, sensors, actuators) with a per-device map from variable names to *VarInfo* records.

Each *VarInfo* records: (i) the current value or defining expression, if one is known; (ii) a *resolved* flag indicating that the variable can be treated as constant for the remainder of unfolding; (iii) the set of variable dependencies of the current expression; and (iv) a cache of reusable intermediate terms extracted from the device's equations and constraints. Locals are seeded with no value and marked unresolved; InOut, input, output, and constant variables are seeded from their declared initial expressions (when present) and are marked resolved when those expressions are literal. The cached intermediate terms support reuse across later steps, and the *resolved* flag allows subsequent iterations to focus evaluation on the variables that still require resolution.

The **Null type** is a special primitive type added to the p-model's datatype list to represent uninitialised values during solution resolution. This type allows the generator to distinguish between variables that have been assigned meaningful values and those that are still awaiting resolution. The Null type is temporary and is removed at the end of Technique 4.

Algorithm 11 sketches the *initialiseSidecar* procedure, which constructs sidecar entries for the p-model root and all contained dynamic devices. For each device it initialises a *VarInfo* record for every declared variable (using any available initial expressions), and caches reusable intermediate terms extracted from the device's equations and constraints for later reuse.

Algorithm 11: initialiseSidecar

Input: *pmodel* : *PModel*, *sidecar* : $Map\langle DynamicDevice, Map\langle String, VarInfo\rangle\rangle$
Output: *sidecar* : $Map\langle DynamicDevice, Map\langle String, VarInfo\rangle\rangle$

```
/* Collect devices:  root plus all contained dynamic devices                */
```
1 $devices = \{pmodel\} \cup pmodel.dynamicDevices$;
```
/* Seed per-device variable maps                                             */
```
2 **foreach** $device \in devices$ **do**
3 $vars = \{\}$;
4 **foreach** $v \in device.locals$ **do**
5 $vars$.put($v.name$, VarInfo(null, *false*, $\emptyset$, $\emptyset$))
6 **end**
7 ;
8 **foreach** $v \in device.inOuts \cup device.inputs \cup device.outputs \cup device.constants$ **do**
9 **if** $v.initial \neq null$ **then** $vars$.put($v.name$, VarInfo($v.initial$, isAllLiterals($v.initial$), $\emptyset$, $\emptyset$)) ;
10 **else** $vars$.put($v.name$, VarInfo(null, *false*, $\emptyset$, $\emptyset$)) ;
11 **end**
12 $sidecar$.put($device$, $vars$);
```
/* Cache reusable intermediate terms from constraints/equations             */
```
13 populateSubExprs($device$, $sidecar$);
14 **end**
15 **return** *sidecar*;

After initialisation, the temporary Null type is available as the placeholder type for any variable whose value has not yet been determined. Once all required solution methods have been unfolded and the sidecar has reached a stable resolved state, the Null type is removed from the p-model.

Example 3.15: Simple Arm Serial Manipulator: Sidecar initialisation

In the Simple Arm Serial example, the sidecar is initialised with entries for the root p-model and all dynamic devices. Each device is seeded with its variables. For instance, an excerpt from the initial map for `Link1` might look like:

```
sidecar["SimpleArmSerial.Link1"] = {
  "q":    VarInfo {
            value: null,
            resolved: false,
            dependencies: [],
            subExprs: []
          },
  "qdot": VarInfo {
            value: null,
            resolved: false,
            dependencies: [],
            subExprs: []
          },
  "M":    VarInfo {
            value: null,
            resolved: false,
            dependencies: [],
            subExprs: []
          },
  ...
}
```

As solution resolution proceeds, these entries are updated: constants remain resolved, algebraic simplifications may introduce new literals, and intermediate terms extracted from

constraints are cached for reuse. The Null type is present during this process to allow unresolved variables to be represented explicitly.

3.4.3 Step 2: Calculate Formulation

The second step of guidedChoice calculates the formulation type for the p-model. Since Technique 4 may be invoked independently of Technique 3, it recalculates the formulation using the same `calculateFormulation` helper detailed in Section 3.2.2. This formulation value (e.g., SKO, Featherstone, or `null`) is passed to the solution library to filter appropriate algorithms for the p-model structure.

3.4.4 Step 3: Resolve Solution (First Pass)

The third step of guidedChoice performs the first pass of solution resolution, which unfolds each high-level p-model solution into concrete sub-solutions by querying the solution library. As described in Section 4.1.2, the solution unfolding process converts abstract specifications (such as "solve for θ using method GeneralisedPosition_method1") into detailed SolutionRef structures specifying exact algorithms, intermediate variables, and computational steps.

The **first pass solution resolution** procedure unfolds each p-model solution by querying the solution library. For each solution, it constructs a *SolutionRef* containing the expression, method, order, constraints, and inputs, and passes this to the library to obtain one or more concrete sub-solution references. Each returned reference is then integrated into the p-model by registering required variables, computing initial conditions, simplifying time-invariant constraints, and creating the corresponding p-model solutions.

Algorithm 12 formalises this unfolding pass.

Algorithm 12 begins by allocating the context name and working lists (lines 1–4). It then iterates over a copy of the p-model solutions and translates each one into a *SolutionRef* (lines 6–20). The solution library is queried to unfold the reference (line 22), the original high-level solution is removed from the p-model (line 24), and each returned sub-solution reference is processed (lines 26–34). Within this processing loop, variables are registered and typed (lines 27–29), initial conditions are computed (line 30), and time-invariant constraints are simplified (line 31) before the reference is converted into a p-model solution (lines 32–34). Finally, the accumulated solutions are added back to the p-model (lines 36–37) and the list of resolved references is returned (line 38).

Algorithm 13 describes the variable-registration step used in Algorithm 12 (lines 27–29). It ensures that each variable mentioned by a library reference exists in the p-model and that its type is consistent. If the reference has no type, it is assigned the temporary Null type (lines 1–2). If the variable is absent from the p-model, it is added (lines 3–7); if it exists but still has type Null, it is updated to the concrete type supplied by the library (lines 8–10). Finally, the reference is rewritten to reflect the p-model's type (lines 11–13) and returned (line 14).

While *updateVariable* ensures that variables exist in the p-model with appropriate types, the remaining two procedures, *calculateInitialConditions* and *processAlgebraicConstraints*, determine the actual *values* of these variables. In Algorithm 12 these procedures are applied to each unfolded reference at lines 30–31. Both rely on a common resolution mechanism that constructs

a directed acyclic graph (DAG) of variable dependencies from the p-model's constraints, equations, and relations, then performs depth-first search (DFS) to propagate values through the dependency graph.

These procedures use the `sidecar` map (introduced in Step 1, Section 3.4.2) to track variable states during evaluation. The `sidecar` is initialised by scanning the model's constraints and equations to populate dependency information and cached intermediate terms for each variable.

Algorithm 12: resolveSolutionFirstPass (with solution unfolding)

Input: $pmodel$: *PModel*, $formulation$: *Formulation*
Output: $slnRefList$: $List\langle SolutionRef\rangle$

```
1  ctx = getFullyQualifiedName(pmodel);
2  slnRefList = {};
3  slnsPM = {};
4  pm2 = copy(pmodel);
   /* Step 1:  Convert each p-model solution to SolutionRef                */
5  foreach slnPM ∈ pm2.solutions do
6      slnRef = new SolutionRef();
7      slnRef.expression.name = slnPM.solutionExpr.name;
8      slnRef.expression.type = printType(slnPM.solutionExpr.type);
9      slnRef.method = slnPM.method;
10     slnRef.order = slnPM.order;
       /* Convert constraints and inputs to string representations         */
11     if slnPM.constraints ≠ null then
12         foreach constraint ∈ slnPM.constraints do
13             slnRef.constraints.add(printExpression(constraint));
14         end
15     end
16     if slnPM.inputs ≠ null then
17         foreach input ∈ slnPM.inputs do
18             inputRef.name = input.name;
19             inputRef.type = printType(input.type);
20             slnRef.inputs.add(inputRef);
21         end
22     end
       /* Step 2:  Unfold solution via solution library                    */
23     resolvedRef = SolutionLib.resolveSolution(formulation, slnRef);
       /* Step 3:  Remove original high-level solution from p-model        */
24     pmodel.solutions.remove(slnPM);
       /* Step 4:  Process each unfolded sub-solution reference            */
25     foreach ref ∈ ensureList(resolvedRef) do
26         ref.expression = updateVariable(pmodel, ref.expression);
27         foreach input ∈ ref.inputs do
28             input = updateVariable(pmodel, input);
29         end
30         ref = calculateInitialConditions(pmodel, ref);
31         ref = processAlgebraicConstraints(pmodel, ref);
32         sln = createSolution(pmodel, ctx, ref);
33         slnsPM.add(sln);
34         slnRefList.add(ref);
35     end
36 end
   /* Step 5:  Add unfolded sub-solutions to p-model                       */
37 foreach sln ∈ slnsPM do
38     pmodel.solutions.add(sln);
39 end
40 return slnRefList;
```

The *calculateInitialConditions* procedure resolves initial condition constraints that specify values at time $t = 0$. Solution references from the library often include constraints like $(expr)[t == 0] == value$, where the right-hand side value may need to be computed from the p-model's constants. This procedure constructs the dependency DAG from constraints and relations, performs DFS-based evaluation to propagate constant values through the graph, and substitutes computed values back into the constraints. When dimension placeholders like $zeroMat(0,0)$ appear, the procedure extracts the actual dimensions from the variable's declared type (e.g., $matrix(real, 2, 2)$) and replaces the placeholder with the correctly-sized value (e.g., $zeroMat(2,2)$).

The *processAlgebraicConstraints* procedure resolves algebraic constraints of the form $(expr)[t == t]$ that specify time-invariant relationships. It employs a three-stage process. First, it scans the model's relation hierarchy to establish a *system correspondence mapping*: for each component-level variable, it records the system-level indexed expression that denotes the same quantity, storing this in the `systemCorrespondence` field of the variable's `VarInfo` entry in the sidecar. Second, it identifies which input variables require algebraic constraint generation. Third, it performs DFS through the relation graph to locate defining equations, builds a constants map from the sidecar, and substitutes component variables with their system-level representatives.

For example, if the relation hierarchy contains a correspondence of the form $\theta[0] = \text{WristJoint}.\theta$, then a joint equation that uses the component-level symbol θ, such as $X_J == [|\cos(\theta), \dots |]$, can be rewritten as $[|\cos(\theta[0]), \dots |]$ by substituting θ with its recorded system-level representative $\theta[0]$. The procedure handles dependency cycles by tracking visited $(device, variable)$ pairs and returning null when cycles are detected.

Detailed algorithms for the DAG construction, DFS resolution, relation-based propagation, and system correspondence mapping are provided in Appendix A.2.2.1.

Algorithm 13: updateVariable

Input: *pmodel* : *PModel*, *varRef* : *Local* (variable reference from solution library)
Output: Updated *varRef* reflecting p-model state

```
1  if varRef.type = null then
2  |   varRef.type = "Null";
   |   /* Use Null placeholder                                          */
3  end
4  varPM = pmodel.getVariable(varRef.name);
5  if varPM = null and varRef.type ≠ null then
   |   /* Variable doesn't exist, add it to p-model                    */
6  |   pmodel.addLocal(varRef.name, varRef.type);
7  |   varPM = pmodel.getVariable(varRef.name);
8  end
9  else if varPM.type = "Null" then
   |   /* Variable has placeholder type, update it                    */
10 |   varPM.type = pmodel.createType(varRef.type);
11 end
12 if varPM ≠ null then
   |   /* Update reference to reflect p-model's type                  */
13 |   varRef.type = printType(varPM.type);
14 end
15 return varRef;
```

Example 3.16: Simple Arm Serial Manipulator: Solution Unfolding

In the Simple Arm Serial example with two movable joints (WristJoint and ElbowJoint), suppose the p-model initially contains a single high-level solution for "GeneralisedPosition_method1". The following walkthrough is an instance of Algorithm 12 (Step 3 of Algorithm 10). The numbered steps correspond to the major phases of Algorithm 12: constructing a `SolutionRef` from the p-model solution (lines 6–20), unfolding via the library (line 22), removing the original solution (line 24), processing each returned sub-solution reference (lines 26–34), and adding the resolved sub-solutions back to the p-model (lines 36–37).

Step 1: Convert p-model solution to SolutionRef. The high-level solution is converted to a solution specification:

```
SolutionRef {
    expression: "theta" (joint positions)
    method: "GeneralisedPosition_method1"
    order: 1
}
```

Step 2: Unfold via solution library. The solution library receives this specification and recognises that generalised position forward simulation requires computing eight intermediate quantities. It returns eight sub-solution references: Platform Mapping (tau), matrix evaluation (phi), Newton-Euler inverse dynamics (C), composite body algorithm (M_mass), Cholesky decomposition (M_inv), direct forward dynamics (dd_theta), and two Euler integrations (d_theta, theta).

Step 3: Remove original solution. The original high-level "GeneralisedPosition_method1" solution is removed from the p-model.

Step 4: Process each sub-solution reference. For each of the eight sub-solutions, the processing pipeline executes. Taking the Platform Mapping solution as an example:

(a) updateVariable: Variables *tau* (type $vector(real, 2)$) and *n* (type *int*) are added to the p-model.

(b) calculateInitialConditions: Initial conditions like $(n)[t == 0] == 3$ (indicating 3 links including base) and $(tau)[t == 0] == zeroVec(2)$ are resolved.

(c) processAlgebraicConstraints: Algebraic constraints map system-level joint torques to physical actuator inputs. For the Simple Arm Serial with WristJoint and ElbowJoint, the three-stage process establishes system correspondence between $tau(0)$ and the WristActuator's TorqueIn port, and between $tau(1)$ and the ElbowActuator's TorqueIn port:

```
(tau(0))[t == t] == p_mapping.SimpleArmSerial.IntermediateLink
                    .WristJoint.WristActuator.TorqueIn
(tau(1))[t == t] == p_mapping.SimpleArmSerial.BaseLink.ElbowJoint
                    .ElbowActuator.TorqueIn
```

(d) createSolution: The fully-resolved Platform Mapping solution is converted to a p-model solution object.

Step 5: Add resolved solutions. The eight fully-resolved concrete sub-solutions are added to the p-model in dependency order (Platform Mapping, phi evaluation, Newton-Euler, Composite Body, Cholesky, Forward Dynamics, Euler for velocities, Euler for positions). This unfolding makes the computational structure explicit and enables each sub-solution to be independently processed in subsequent steps.

Several helper algorithms support the solution resolution process across multiple steps. The *findMatchingSolutionRef* algorithm is used in Step 5 to match p-model solutions with their corresponding solution references by comparing expression names. The *formatSolutionRef* algorithm is used in Step 6 to generate formatted output representations of resolved solutions for documentation and downstream processing. These algorithms are detailed in Appendix A.2.2.8.

3.4.5 Step 4: Reorder Solutions

The fourth step of guidedChoice is to reorder both the p-model solutions and the solution references by their *order* property. This step ensures that solutions are processed in the correct dependency order during the subsequent iterative resolution phase.

Solutions in the p-model and their corresponding solution references each have an *order* property that specifies their processing priority. Solutions with lower order values should be resolved before solutions with higher order values, as earlier solutions may produce variables or constraints needed by later solutions. For example, a solution that calculates mass matrix elements (order 1) must be resolved before a solution that uses the mass matrix to compute forward dynamics (order 2).

The reordering step is a standard sort: both *pmodel.solutions* and the *solution_ref* list are sorted in ascending order by their *order* values. For completeness, Appendix A.2.2.8 presents the pseudocode (Algorithm 43).

Example 3.17: Simple Arm Serial Manipulator: Solution Reordering

In the Simple Arm Serial example, suppose the p-model contains three solutions after the first pass. The reordering process ensures proper dependency resolution:
Before reordering, the solutions appear in arbitrary order from the first pass:

```
pmodel.solutions = [
    Solution { expression: "C", method: "NewtonEulerInverseDynamics", order: 3
        },
    Solution { expression: "tau", method: "PlatformMapping", order: 1 },
    Solution { expression: "phi", method: "Eval", order: 2 }
]
```

After reordering, the solutions are sorted by order property:

```
pmodel.solutions = [
    Solution { expression: "tau", method: "PlatformMapping", order: 1 },
    Solution { expression: "phi", method: "Eval", order: 2 },
    Solution { expression: "C", method: "NewtonEulerInverseDynamics", order: 3
        }
]
```

The dependency relationships can be visualised as follows:

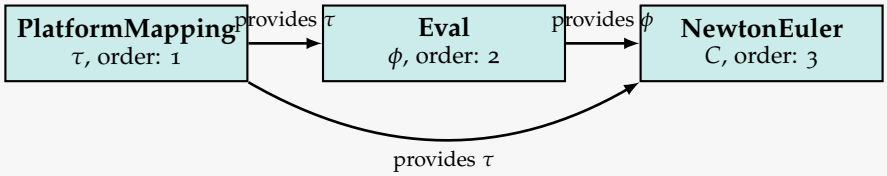


Figure 3.12: Solution Dependency Graph. Arrows show variable dependencies between solutions.

This ordering ensures that when Step 5 processes these solutions iteratively, the platform mapping (τ) is fully resolved before the evaluation (ϕ) begins, and both are complete before the Newton-Euler inverse dynamics (C) is computed. Without proper ordering, the inverse dynamics solution would fail because it depends on variables (τ and ϕ) that haven't been calculated yet.

3.4.6 Step 5: Resolve Each Solution in Order

The fifth step of guidedChoice iteratively resolves each solution in the reordered list. This step processes solutions that require multiple iterations, expanding them fully and updating the p-model with all necessary variables and constraints.

Algorithm 14 presents the iterative resolution routine used in this step.

Algorithm 14: resolveSolution (Iterative Resolution)

Input: $pmodel$: $PModel+$, $formulation$: $Formulation$, $slnRef$: $SolutionRef$
Output: $slnList$: $List\langle Solution\rangle$

```
1  slnList = {};
2  ctx = getFullyQualifiedName(pmodel);
   /* Perform iterations as specified by solution reference                    */
3  for i = 0 to slnRef.iterations do
       /* Step 1: Unfold solution reference using library templates            */
4      slnRef = SolutionLib.resolveSolutionRef(formulation, slnRef);
       /* Step 2: Update expression variable in p-model                        */
5      slnRef.expression = updateVariable(pmodel, slnRef.expression);
       /* Step 3: Update input variables in p-model                            */
6      foreach input ∈ slnRef.inputVariables do
7          input = updateInputVariable(pmodel, input);
8      end
       /* Step 4: Calculate or update initial conditions                       */
9      slnRef = calculateInitialConditions(pmodel, slnRef);
       /* Step 5: Process algebraic constraints                                */
10     slnRef = processAlgebraicConstraints(pmodel, slnRef);
       /* Step 6: Create solution and add to p-model                           */
11     sln = createSolution(pmodel, ctx, slnRef);
12     pmodel.solutions.add(sln);
13     slnList.add(sln);
14 end
15 return slnList;
```

The **iterative solution resolution** procedure is defined by Algorithm 14. It initialises an accumulator and context name (lines 1–2), then performs as many rounds as specified by $slnRef.iterations$ (line 3). In each round it queries the solution library for an updated reference (line 4) and re-applies the same processing pipeline used in Step 3 (Section 3.4.4): it aligns the expression variable and its inputs with the p-model (lines 5–7), computes missing initial-condition constraints (line 8), simplifies time-invariant constraints (line 9), and creates and registers the corresponding p-model solution (lines 10–12). Because each round runs against the updated p-model, later rounds can depend on information established earlier.

After solutions have been reordered in Step 4, Step 5 iterates through each solution in the p-model and finds its matching solution reference (by comparing solution expression names). For each matched pair, it calls the *resolveSolution* procedure to perform any additional iterations specified by the solution reference.

The iterative resolution mechanism supports solutions whose library specification must be refined in stages. In such cases, Step 5 associates the solution reference with a nonzero *iterations* bound and advances its *iteration* counter each time the corresponding library template is re-instantiated (Algorithm 14, line 4). Because each instantiation is performed against the updated p-model state, later stages can depend on variables, types, or correspondence constraints that only become available after earlier stages have been integrated. When a computation must be applied across multiple links or joints, this is typically handled by *solution unfolding* in Step 3, where the library yields a list of concrete sub-solution references rather than using the iteration counter.

The main loop in Step 5 (Algorithm 15) coordinates this process by iterating through each solution in the p-model, finding its matching solution reference, and calling the *resolveSolution* procedure (Algorithm 14):

Algorithm 15: Main Loop for Step 5

Input: $pmodel : PModel$, $formulation : Formulation$, $slnRefList : List\langle SolutionRef\rangle$
Output: Updated *pmodel* with all solutions resolved

```
   /* Process each solution in dependency order                    */
1  foreach slnPM ∈ pmodel.solutions do
       /* Find matching solution reference by expression name      */
2      slnRef = findMatchingSolutionRef(slnRefList, slnPM);
3      if slnRef ≠ null then
           /* Perform iterative resolution                         */
4          resolveSolution(pmodel, formulation, slnRef);
5      end
6  end
```

By processing solutions in the order established in Step 4, Step 5 ensures that all dependency relationships are respected. Solutions that depend on variables calculated by earlier solutions will find those variables already present and initialised in the p-model by the time they are processed.

Example 3.18: Simple Arm Serial Manipulator: Platform Mapping Iterative Resolution

In the Simple Arm Serial example with two movable joints (Wrist and Elbow), the Platform Mapping solution requires iterative resolution to establish the mapping from system-level joint torques (*tau*) to platform-specific actuator ports. The solution reference specifies 3 iterations:

```
SolutionRef {
    expression: { name: "tau", type: "vector(real, 2)" },
    method: "PlatformMapping",
    order: 1,
    iterations: 3
}
```

The iterative resolution proceeds as follows:
Iteration 0 (Initialise):

1. *Template instantiation* (Algorithm 14, line 4): the unfolding procedure instantiates the `PlatformMapping` template at iteration 0, yielding a reference that introduces the structural parameter n together with the placeholder constraint $(n)[t == 0] == 0$.
2. *Variable alignment* (Algorithm 14, lines 5–7): the variable n is registered in the p-model with type *int* so it can be referenced by later unfolding stages.

3. *Initial conditions* (Algorithm 14, line 8): the placeholder initial value for n is retained.
4. *Algebraic constraints* (Algorithm 14, line 9): no time-invariant mapping constraints are emitted at this stage.
5. *Solution registration* (Algorithm 14, lines 10–12): a partial p-model solution corresponding to this stage is created and stored.

Iteration 1 (Extract Link Count):

1. *Template instantiation* (Algorithm 14, line 4): the unfolding procedure instantiates the next stage of the `PlatformMapping` template, using the current p-model hierarchy to determine $n = 3$ (including the base link), and emits the updated initial constraint $(n)[t == 0] == 3$.
2. *Variable alignment* (Algorithm 14, lines 5–7): the system-level torque vector *tau* is introduced with type $vector(real, 2)$, where the dimension is derived from the number of movable joints.
3. *Initial conditions* (Algorithm 14, line 8): $(n)[t == 0]$ is updated to the concrete value 3.
4. *Algebraic constraints* (Algorithm 14, line 9): no actuator correspondence constraints are emitted yet.
5. *Solution registration* (Algorithm 14, lines 10–12): the stage is recorded as an additional p-model solution instance.

Iteration 2 (Establish Actuator Mapping):

1. *Template instantiation* (Algorithm 14, line 4): with actuator instances and their ports already present in the p-model, the unfolding procedure emits one correspondence constraint per actuator destination, mapping the indices of *tau* to platform-mapping paths.
2. *Variable alignment* (Algorithm 14, lines 5–7): no new variables are introduced.
3. *Initial conditions* (Algorithm 14, line 8): $(tau)[t == 0] == zeroVec(2)$ is added.
4. *Algebraic constraints* (Algorithm 14, line 9): the mapping constraints are emitted in time-invariant form, for example:

```
(tau(0))[t == t] == p_mapping.SimpleArmSerial
                    .IntermediateLink.WristJoint
                    .WristActuator.TorqueIn
(tau(1))[t == t] == p_mapping.SimpleArmSerial
                    .BaseLink.ElbowJoint
                    .ElbowActuator.TorqueIn
```

   These constraints map the system-level torque vector indices to the physical actuator input ports.
5. *Solution registration* (Algorithm 14, lines 10–12): the final mapping stage is created and stored.

Final state: After 3 iterations, the Platform Mapping solution has established the bridge between abstract system-level control inputs (*tau*) and concrete platform actuator ports. This mapping is then used by Technique 5 to generate platform-specific code that writes to the correct actuator interfaces.
The key aspect illustrated here is that each iteration refines the solution against an increasingly informative p-model state: iteration 0 introduces structural parameters, iteration 1 fixes the relevant dimensions, and iteration 2 emits correspondence constraints that require the actuator declarations already present in the model.

After all solutions have been iteratively resolved, Technique 4 returns two outputs: (1) the extended p-model with all solutions fully resolved and integrated, including all necessary variables, constraints, and initial conditions; and (2) the accumulated list of solution references (`.slnRef` file) from Steps 3 and 5, containing the complete specifications for each resolved solution including selected algorithms, inputs, and constraints. These outputs together provide the complete specification needed for code generation in Technique 5.

3.5 Technique 5: Generation for a Platform Engine

3.5.1 Overview

Technique 5: Generation for a Platform Engine (sourceCodeGen) takes a p-model+ with solutions and produces platform engine source-code.

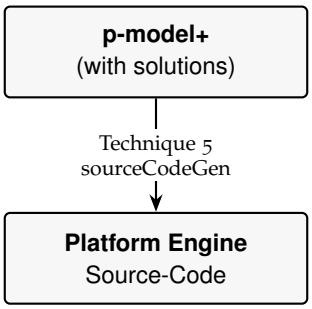


Figure 3.13: sourceCodeGen: Generation for a Platform Engine. This technique takes a p-model+ with solutions and produces platform engine source-code.

The sourceCodeGen technique transforms the resolved solutions from guidedChoice into executable platform engine code through a two-phase process.

In the first phase, each slnRef entry (solution reference, see Section 3.3.1) is resolved by querying the solution library for its corresponding solution template. The generator uses the library templates to return concrete algorithmic implementations in slnDF format (solution description format, see Section 3.3.2), which are then combined into a unified slnDF specification removing duplicate shared state variables, procedures, and functions. The resulting slnDF specification contains `state` blocks defining simulation variables with their types and initial values, `computation` blocks specifying the step-by-step evaluation sequence, and `functions` and `procedures` blocks implementing the computational algorithms. This phase instantiates abstract solution method references with concrete implementations that are ready for platform-specific code generation.

In the second phase, the unified slnDF specification is translated into platform-specific source code (as illustrated in Figure 3.13). A code generator for the target platform translates slnDF constructs to their target language equivalents: state variables become language-specific variable declarations, computation blocks become executable statements, and mathematical operations become library calls appropriate to the target platform (such as Eigen for C++, NumPy for Python).

The resulting platform engine source code provides the computational implementation derived from the p-model. However, to form a complete executable simulation, this generated code must be integrated with an orchestrator component (shown as the bottom layer in Figure 3.13). The orchestrator provides stubs for simulation timing, event scheduling, and communication between source-code generated from RoboSim models. In the current workflow, these stubs are filled in manually by the developer. Chapter 4 discusses an extended workflow where platform mapping information can be provided as additional input to automate this integration.

As an alternative to this two-phase approach, the slnRef format can also be used directly for platform engine translation, similar to existing formats like SDF. However, since slnRef annotations contain only algorithm references rather than their detailed implementations, this direct translation approach is not the primary focus of this chapter. Furthermore, working directly from slnRef makes it significantly more difficult to guarantee consistency between the generated code and the source models, and to ensure that the complete simulation faithfully matches the RoboSim semantics. The intermediate slnDF representation provides an explicit, inspectable specification of the computational implementation that facilitates verification of semantic preservation. This description is simulator and programming language independent. It is the gateway to facilitate extensibility of our framework. Providing support for additional programming languages requires only the creation of alternative translations (second step of this technique), without any impact on the rest of the approach. As we describe below, this translation is rather direct for imperative programming languages.

3.5.2 Step 1: Translate SolutionRef to slnDF

The first step of sourceCodeGen is to translate the slnRef annotation (generated by guidedChoice) into a slnDF specification. This translation process takes individual solution references with their constraints and inputs, selects and instantiates the corresponding templates from the solution library, and assembles them into a unified slnDF specification ready for code generation.

The **slnRef to slnDF translation** process takes a collection of slnRef entries and produces a single slnDF specification. Each slnRef entry specifies an expression to compute (with name and type), a solution method, an ordering, input variables, and constraints. The translation instantiates the library template for each method and then combines the resulting templates, removing duplicates between shared state variables, procedures, and functions.

The **Solution Library** is a repository of slnDF templates keyed by solution method. For a solution reference *sr*, we write library.*sr* for the template selected by *sr.method* and instantiated with the fields of *sr* (expression, inputs, and constraints).

Algorithm 46 (presented in Appendix A.2.3.1) shows the imperative formulation of the *translateSlnRefToSln* procedure, which coordinates the translation from a collection of slnRef entries to a unified slnDF specification. The algorithm iterates through each slnRef entry in order (the order property has already been established by Technique 4), selects and instantiates the

corresponding library template, extracts its components, and combines them while removing duplicates. The semantic translation rules presented below provide the declarative specification of this translation.

Rule notation. The following rules use bracket notation of the form $[\![x]\!]_K$, read as "translate (or extract from) x its K-component". The rule header states the translation and its type signature. The subscript $K \in \{\text{DT}, \text{STATE}, \text{PROC}, \text{FUNC}, \text{COMP}\}$ selects the corresponding slnDF block (datatypes, state, procedures, functions, and computation). A **where** clause introduces auxiliary meta-level functions used by the rule (e.g. sorting by `order`, set union, list concatenation, and deduplication by signature). The meta-functions datatypes$(\cdot)$, stateVars$(\cdot)$, procedures$(\cdot)$, functions$(\cdot)$, and computation$(\cdot)$ are projections from an instantiated template to its corresponding slnDF blocks (returning the empty set or empty list when a block is absent).

Rule 1: slnRef Collection Translation

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{slnDF}} : List\langle slnRef\rangle \to slnDF$$

Solution physics {
  datatypes $\{[\![slnRefs]\!]_{\text{DT}}\}$
  state $\{[\![slnRefs]\!]_{\text{STATE}}\}$
  procedures $\{[\![slnRefs]\!]_{\text{PROC}}\}$
  functions $\{[\![slnRefs]\!]_{\text{FUNC}}\}$
  computation $\{[\![\text{sorted}(slnRefs)]\!]_{\text{COMP}}\}$
}

where

$\text{sorted}(slnRefs) = \text{sort}(slnRefs, sr \mapsto sr.order)$

Rule 1 defines the top-level translation from a collection of slnRef entries to a unified slnDF specification. The rule applies five component-specific translations (datatypes, state, procedures, functions, and computation) to the input collection. The computation block processes entries in sorted order based on their `order` field. Each component translation is defined by subsequent rules.

Rule 2: Datatype Aggregation

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{DT}} : List\langle slnRef\rangle \to \mathcal{P}(Datatype)$$

$\bigcup_{sr \in slnRefs} \text{datatypes}(\text{library}.sr)$

Rule 2 collects all custom datatypes from the solution templates, taking the union to eliminate duplicates.

Rule 3: State Variable Deduplication

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{STATE}} : List\langle slnRef\rangle \to \mathcal{P}(StateVar)$$

$$\text{dedupByName}\left(\bigcup_{sr \in slnRefs} \text{stateVars}(\text{library}.sr)\right)$$

where
$\text{dedupByName}(V) = \{v \in V \mid \nexists v' \in V.\, v \neq v' \wedge v.name = v'.name\}$

Rule 3 extracts state variables from all templates and applies name-based deduplication. When multiple solution references declare variables with the same name (e.g., `dd_theta` shared between forward dynamics and Euler integration), only one declaration is retained in the unified state block.

Rule 4: Procedure Signature Deduplication

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{PROC}} : List\langle slnRef\rangle \to \mathcal{P}(Procedure)$$

$$\text{dedupBySignature}\left(\bigcup_{sr \in slnRefs} \text{procedures}(\text{library}.sr)\right)$$

where
$\text{dedupBySignature}(P) = \{p \in P \mid \nexists p' \in P.\, p \neq p' \wedge \text{sig}(p) = \text{sig}(p') \wedge \text{length}(p') > \text{length}(p)\}$
$\text{sig}(p) = (p.name, p.parameters)$
$\text{length}(p) =$ character count of textual representation of p

Rule 4 collects procedures from all templates and deduplicates by signature (name and parameter list). When multiple procedures share the same signature, the one with the longer textual representation is retained, preferring complete implementations over stub declarations.

Rule 5: Function Signature Deduplication

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{FUNC}} : List\langle slnRef\rangle \to \mathcal{P}(Function)$$

$$\text{dedupBySignature}\left(\bigcup_{sr \in slnRefs} \text{functions}(\text{library}.sr)\right)$$

where
$\text{dedupBySignature}(F) = \{f \in F \mid \nexists f' \in F.\, f \neq f' \wedge \text{sig}(f) = \text{sig}(f') \wedge \text{length}(f') > \text{length}(f)\}$
$\text{sig}(f) = (f.name, f.parameters, f.returnType)$
$\text{length}(f) =$ character count of textual representation of f

Rule 5 collects and deduplicates functions using the same strategy as procedures (comparing textual length when signatures match), but with function signatures that include the return type.

Rule 6: Computation Sequence

$$[\![slnRefs : List\langle slnRef\rangle]\!]_{\text{COMP}} : List\langle slnRef\rangle \rightarrow List\langle Statement\rangle$$

$\text{concat}([\,\text{computation}(\text{library}.sr) \mid sr \in slnRefs\,])$

Rule 6 preserves order by construction: the list comprehension ranges over `slnRefs` from left to right, and `concat` preserves the relative order of the statement sequences it flattens. Since the top-level rule supplies `slnRefs` as `sorted(slnRefs)` (Rule 1), and Technique 4 assigns the `order` fields to respect solution dependencies, the resulting computation block preserves the execution sequence established by Technique 4.

The resulting unified slnDF contains all necessary information for subsequent code generation: variable declarations with types and initial values, procedure and function definitions implementing the requested computational methods, and a computation block specifying the execution order. This intermediate representation bridges the gap between the high-level slnRef specifications (which reference abstract solution methods) and the platform-specific code (which will be generated in subsequent steps). The imperative formulation of the translation is presented in Appendix A.2.3.1.

Example 3.19: Simple Arm Serial Manipulator: SlnRef to slnDF Translation

Consider a simplified excerpt from the Simple Arm Serial manipulator's slnRef annotation containing two entries:

```
SolutionRef {
    expression Local { name "M_inv" type "matrix(real,2,2)" }
    method "SKO.CholeskyAlgorithm"
    order 1
    input Local { name "M_mass" type "matrix(real,2,2)" }
    constraint " (M_mass) [t==t] == H*phi*M*adj(phi)*adj(H)"
}
SolutionRef {
    expression Local { name "dd_theta" type "vector(real,2)" }
    method "SKO.DirectForwardDynamics"
    order 2
    input Local { name "M_inv" type "matrix(real,2,2)" }
    input Local { name "tau" type "vector(real,2)" }
    input Local { name "C" type "vector(real,2)" }
    constraint " (tau) [t==t] == M_mass*derivative(derivative(theta))+C"
}
```

The translation algorithm processes each entry in order. For the first entry (`SKO.CholeskyAlgorithm`), the Cholesky template is selected from the library and instantiated for the concrete variables and constraints of the slnRef entry. For the second entry (`SKO.DirectForwardDynamics`), the forward-dynamics template is selected and

instantiated in the same way.
After extracting components from both templates, the algorithm combines them:

```
Solution physics {
    state {
        M_inv: mat(2,2) = zeroMat(2, 2);
        M_mass: mat(2,2) = zeroMat(2, 2);
        tau: vec(2) = zeroVec(2);
        C: vec(2) = zeroVec(2);
        dd_theta: vec(2) = zeroVec(2);
    }
    functions {
        function LDLT(A: mat(2,2)): mat(2,2) {
            result == A.ldlt().solve(Identity(2,2))
        }
    }
    computation {
        M_inv = LDLT(M_mass);
        dd_theta = M_inv * (tau - C);
    }
}
```

Note that state variables are deduplicated (M_inv appears in both slnRefs but only once in the state block), and the computation block contains statements from both templates in their specified order. The variable M_mass is referenced by both solutions: it is an input to the first solution and appears in a constraint for the second solution, ensuring proper data flow between computational stages.

3.5.3 Step 2: Translate slnDF to Target Language

The second step of sourceCodeGen is to translate the unified slnDF file into executable code in a target programming language. This translation process converts the platform-independent slnDF representation into concrete executable code by mapping abstract mathematical operations to target-language constructs, translating types to language-specific representations, and generating the necessary scaffolding for execution.

The **slnDF to target language translation** process takes a slnDF specification and a code generator for the target platform, and produces source code in the target language. The translation is parameterised by the choice of code generator, which encapsulates all target-specific details such as type mappings, library dependencies, and code generation patterns. This design allows the same slnDF intermediate representation to be compiled to multiple target languages (C++, Python, Java, etc.) by providing different code generator implementations.

A **code generator** G provides the interface between the generic slnDF representation and a specific target language. Each code generator must implement a collection of translation functions: *includes* (which imports/headers to generate), *mapType* (mapping slnDF types to target types), *emitDataTypes* (generating custom type definitions), *emitStateDecls* (generating state variable declarations and initialisation), *emitFunction* (translating function definitions), *emitProcedure* (translating procedure definitions), *emitStatement* (translating computation statements), *emitEntrypoint* (generating the program entry point), and *assemble* (combining all components into final source code).

The translation is defined through a series of semantic rules that delegate to the code generator for target-specific details. Each rule shows how a component of the slnDF specification is translated using the code generator's functions.

Rule 7: slnDF to Target Translation

$$[\![sln : slnDF, G : CodeGenerator]\!]_{\text{TARGET}} : slnDF \times CodeGenerator \to String$$

G.assemble(
$[\![sln]\!]^G_{\text{HEADERS}}$,
$[\![sln.datatypes]\!]^G_{\text{DATATYPES}}$,
$[\![sln.state]\!]^G_{\text{STATEDECLS}}$,
$[\![sln.functions]\!]^G_{\text{FUNCTIONS}}$,
$[\![sln.procedures]\!]^G_{\text{PROCEDURES}}$,
$[\![sln.state, sln.computation]\!]^G_{\text{ENTRYPOINT}}$
)

Rule 7 defines the top-level translation from slnDF to target language source code. The code generator's `assemble` function combines the translated components (headers, datatypes, state declarations, functions, procedures, and entry point) into the final source code file. Each component is translated by a specialised rule.

Rule 8: Headers Translation

$$[\![sln : slnDF]\!]^G_{\text{HEADERS}} : slnDF \to String$$

G.includes()

Rule 8 generates the necessary imports or includes for the target language. The code generator determines which libraries and headers are needed based on the operations used in the slnDF specification.

Rule 9: Datatypes Translation

$$[\![datatypes : Datatypes]\!]^G_{\text{DATATYPES}} : Datatypes \to String$$

$$\begin{cases} G.\text{emitDataTypes}(datatypes) & \text{if } datatypes \neq null \\ \varepsilon & \text{otherwise} \end{cases}$$

Rule 9 translates custom datatype definitions (structs and enums) to target-language type

definitions, or produces an empty string if no custom datatypes are defined.

Rule 10: Type Mapping

$$G.\text{mapType}(t : \mathit{Type}) : \mathit{Type} \to \mathit{String}$$

---

target-language specific type mapping

Rule 10 defines the code generator's type mapping function, which translates slnDF types to target-language types. Each code generator provides its own implementation based on the target language's type system and available libraries.

Rule 11: State Declarations Translation

$$[\![\mathit{state} : \mathit{State}]\!]^{G}_{\text{STATEDECLS}} : \mathit{State} \to \mathit{String}$$

$G.\text{emitStateDecls}(\mathit{state})$

Rule 11 translates state variable declarations, including their types and initial values. The code generator determines the appropriate declaration syntax and initialisation strategy for the target language.

Rule 12: Functions Translation

$$[\![\mathit{functions} : \mathit{List}\langle \mathit{Function} \rangle]\!]^{G}_{\text{FUNCTIONS}} : \mathit{List}\langle \mathit{Function} \rangle \to \mathit{String}$$

$\text{concat}([\, G.\text{emitFunction}(f) \mid f \in \mathit{functions} \,])$

Rule 12 translates each function definition to target-language syntax, concatenating the results. The code generator handles the mapping of function signatures, parameters, return types, and body expressions.

Rule 13: Procedures Translation

$$[\![\mathit{procedures} : \mathit{List}\langle \mathit{Procedure} \rangle]\!]^{G}_{\text{PROCEDURES}} : \mathit{List}\langle \mathit{Procedure} \rangle \to \mathit{String}$$

$\text{concat}([\, G.\text{emitProcedure}(p) \mid p \in \mathit{procedures} \,])$

Rule 13 translates procedure definitions, concatenating the generated code. Procedures may have result parameters that are handled according to target-language conventions (e.g., reference parameters in C++, return tuples in Python).

Rule 14: Entry Point Translation

$$[\![state : State, computation : List\langle Statement\rangle]\!]^{G}_{\text{ENTRYPOINT}} : State \times List\langle Statement\rangle \to String$$

G.emitEntrypoint($[\![state]\!]^{G}_{\text{INIT}}, [\![computation]\!]^{G}_{\text{STMTS}}$)

where
$[\![state]\!]^{G}_{\text{INIT}} = \text{concat}([\, G.\text{emitStatement}(\text{initStmt}(v)) \mid v \in state.variables \,])$
$[\![computation]\!]^{G}_{\text{STMTS}} = \text{concat}([\, G.\text{emitStatement}(s) \mid s \in computation \,])$

Rule 14 generates the program entry point, wrapping initialisation code for state variables and the main computation statements. The code generator determines the appropriate entry point structure for the target language (e.g., `main()` in C++, a module-level script in Python).

C++ Code Generator Specialisation

We now illustrate a concrete C++ instance of the code generator interface, denoted $G_{\text{C++}}$. The goal is to explain the main specialisation choices (use of Eigen for linear algebra, `std::vector` for sequences, and a separate initialisation routine for non-trivial values). Rather than introduce additional formal translation rules for this instance, we give representative outputs of the key generator functions.

Example 3.20: C++ `includes()`

The C++ generator emits the header list required by the generated code:

```
#include <iostream>
#include <Eigen/Dense>
#include <vector>
#include <string>
```

Eigen provides matrix and vector operations; `iostream` supports console output; `vector` and `string` provide the standard containers used by `seq(T)` and `String`.

Example 3.21: C++ `mapType`

The function $G_{\text{C++}}$.mapType maps slnDF types to C++ types. The mapping is structural and recurses over sequence types:

slnDF type	C++ type
mat(m, n)	`Eigen::MatrixXd`
vec(n)	`Eigen::VectorXd`
float	`double`
int	`int`
bool	`bool`
String	`std::string`
seq(T)	`std::vector<`$G_{\text{C++}}$.mapType(T)`>`
custom type X	`X`

Example 3.22: C++ `emitDataTypes`: `Geom`

Consider the following slnDF datatype:

```
datatypes {
    Geom {
        geomType: String;
        geomVal: vec()
    }
}
```

The C++ generator emits a corresponding struct, applying $G_{\text{C++}}$.mapType to field types:

```
struct Geom {
    std::string geomType;
    Eigen::VectorXd geomVal;
};
```

Example 3.23: C++ `emitStateDecls`

For the slnDF state block from Section 3.5.2:

```
state {
    M_inv: mat(2,2) = zeroMat(2, 2);
    M_mass: mat(2,2) = zeroMat(2, 2);
    tau: vec(2) = zeroVec(2);
    C: vec(2) = zeroVec(2);
    dd_theta: vec(2) = zeroVec(2);
}
```

the C++ generator emits global declarations and an initialisation routine:

```
Eigen::MatrixXd M_inv;
Eigen::MatrixXd M_mass;
Eigen::VectorXd tau;
Eigen::VectorXd C;
Eigen::VectorXd dd_theta;

void initGlobals() {
    M_inv = Eigen::MatrixXd::Zero(2, 2);
    M_mass = Eigen::MatrixXd::Zero(2, 2);
    tau = Eigen::VectorXd::Zero(2);
    C = Eigen::VectorXd::Zero(2);
    dd_theta = Eigen::VectorXd::Zero(2);
}
```

This two-phase structure is required because C++ does not permit arbitrary run-time initialisation in global declarations.

Example 3.24: C++ `emitFunction: LDLT`

The slnDF function:

```
function LDLT(A: mat(2,2)): mat(2,2) {
    result == A.ldlt().solve(Identity(2,2))
}
```

is emitted as a C++ function returning an Eigen matrix:

```
Eigen::MatrixXd LDLT(const Eigen::MatrixXd& A) {
    return A.ldlt().solve(Eigen::MatrixXd::Identity(A.rows(), A.cols()));
}
```

The pure slnDF equality for `result` becomes a return statement; Eigen provides `ldlt()` and `solve()`.

Example 3.25: C++ `emitEntrypoint`

Given the state and computation blocks:

```
state { ... }
computation {
    M_inv = LDLT(M_mass);
    dd_theta = M_inv * (tau - C);
}
```

the C++ generator emits an entry point that initialises globals and then executes the computation statements in order:

```
int main() {
    initGlobals();
    M_inv = LDLT(M_mass);
    dd_theta = M_inv * (tau - C);
    return 0;
}
```

These examples illustrate one instantiation of the code generator interface for C++. Other target languages provide corresponding implementations of the same functions, differing only in type representations, library dependencies, and surface syntax for functions, procedures, and statements.

As mentioned in the overview of this chapter, the output of this technique fully automates generation of the platform engine and partially generates the orchestration layer. A more complete breakdown of the generated artefacts, and their role in the runtime architecture, is given in Chapter 4 (Section 4.1.3, Table 4.6).

3.6 Summary

This chapter presented a simulation framework that transforms RoboSim models into executable simulation code while preserving operational semantics. eqnComp composes system-level equations from component specifications; guidedChoice resolves user-selected solution references, propagating constraints and initial conditions; and sourceCodeGen generates a platform engine from the resulting p-model+ via slnDF. The platform engine is fully generated, while the orchestration layer is generated with integration stubs that are completed by incorporating the d-model and mapping artefacts (Chapter 4).

4 Tool and Case Studies

This chapter presents the implementation of the simulation framework tool and validates that it correctly realises the theoretical foundations from Chapter 3. The chapter is organised into four sections. Section 4.1 describes the tool implementation, covering the generator architectures and libraries for each technique, and validates that the implementation corresponds to the pseudocode algorithms from Chapter 3. Section 4.2 provides comprehensive test coverage demonstrating the framework's correctness across different robot configurations. Section 4.3 presents case studies demonstrating the practical application of the framework. Finally, Section 4.4 discusses extensibility considerations.

4.1 Tool Implementation

This section describes the design and capabilities of the simulation framework tool; usage instructions and installation details are available in the tool repository[1]. The implementation uses the Eclipse Modeling Framework (EMF) and Xtext for model manipulation and code generation [125], [126].

Users invoke the generators either by right-clicking on the relevant model files (p-model, mapping, or solution reference files) and selecting the appropriate generation action from the context menu, or by using the toolbar buttons shown in Figure 4.1.



Figure 4.1: Toolbar buttons for invoking the eqnComp, guidedChoice, and sourceCodeGen generators.

4.1.1 T3: Equation Composition

This section describes the implementation of eqnComp from Section 3.2 of Chapter 3.

Generator Architecture

The eqnComp generator is implemented as an Xtext code generator [125], [126]. In the standard Xtext generator pattern (Figure 4.2, left), a generator class extends `AbstractGenerator` and overrides the `doGenerate` method, which serves as the entry point. The `Resource`

[1] https://github.com/UoY-RoboStar/robotool-meta-physmod.git

parameter provides access to the parsed EMF model, while `IFileSystemAccess2` enables writing generated files. Services such as `IQualifiedNameProvider` and `ISerializer` are obtained via Guice dependency injection using the `@Inject` annotation. Xtend's `extension` keyword allows injected services to be used as if their methods were defined directly on the generator class.

The eqnComp generator extends this pattern through an intermediate class `AbstractPhysModGenerator` (Figure 4.2, right). The eqnComp generator adds an extended `doGenerate` signature that accepts an additional `PModel` parameter and an optional `Formulation` parameter, instantiates helper utilities and exposes them as Xtend `extensions`, and dispatches to the technique-specific method `eqnComp`.

Two key differences distinguish the eqnCompGenerator signature from the standard pattern. First, the method uses `def` rather than `override` because it defines a new method with an extended signature rather than overriding the inherited `doGenerate(Resource, fsa, context)`. Second, the method accepts additional parameters: a `PModel` parameter that specifies which p-model to generate for (supporting projects with multiple p-models), and an optional `Formulation` parameter that enables users to override formulation auto-detection for edge cases such as single-link robots without joints.

Figure 4.3 presents the overall architecture, which follows a pipeline pattern where the p-model flows through the six processing stages from the eqnComp algorithm presented in Chapter 3.

```
class MyGenerator
    extends AbstractGenerator {

  @Inject extension IQualifiedNameProvider
  @Inject extension ISerializer

  override doGenerate(
     Resource resource,
     IFileSystemAccess2 fsa,
     IGeneratorContext context) {
    // Generator implementation
  }
}
```

Listing 4.1: Standard Xtext generator

```
class eqnCompGenerator
   extends AbstractPhysModGenerator {

  public extension GeneratorState state
  public extension GeneratorUtils gu
  public extension EqnCompUtils utils
  @Inject EquationLib lib

  def void doGenerate(
     Resource resource, PModel pmodel,
     Formulation overrideFormulation,
     IFileSystemAccess2 fsa,
     IGeneratorContext context) {
    fsa.generateFile(pmodel.name + ".pm",
     eqnComp(pmodel, resource,
       overrideFormulation))
  }
}
```

Listing 4.2: eqnCompGenerator class

Figure 4.2: Side-by-side comparison of the standard Xtext generator pattern (left) and the eqnCompGenerator implementation (right). The implementation extends `AbstractPhysModGenerator`, uses extension utilities, accepts a `PModel` and optional `Formulation` parameter, and dispatches from `doGenerate` to the technique-specific `eqnComp` method.

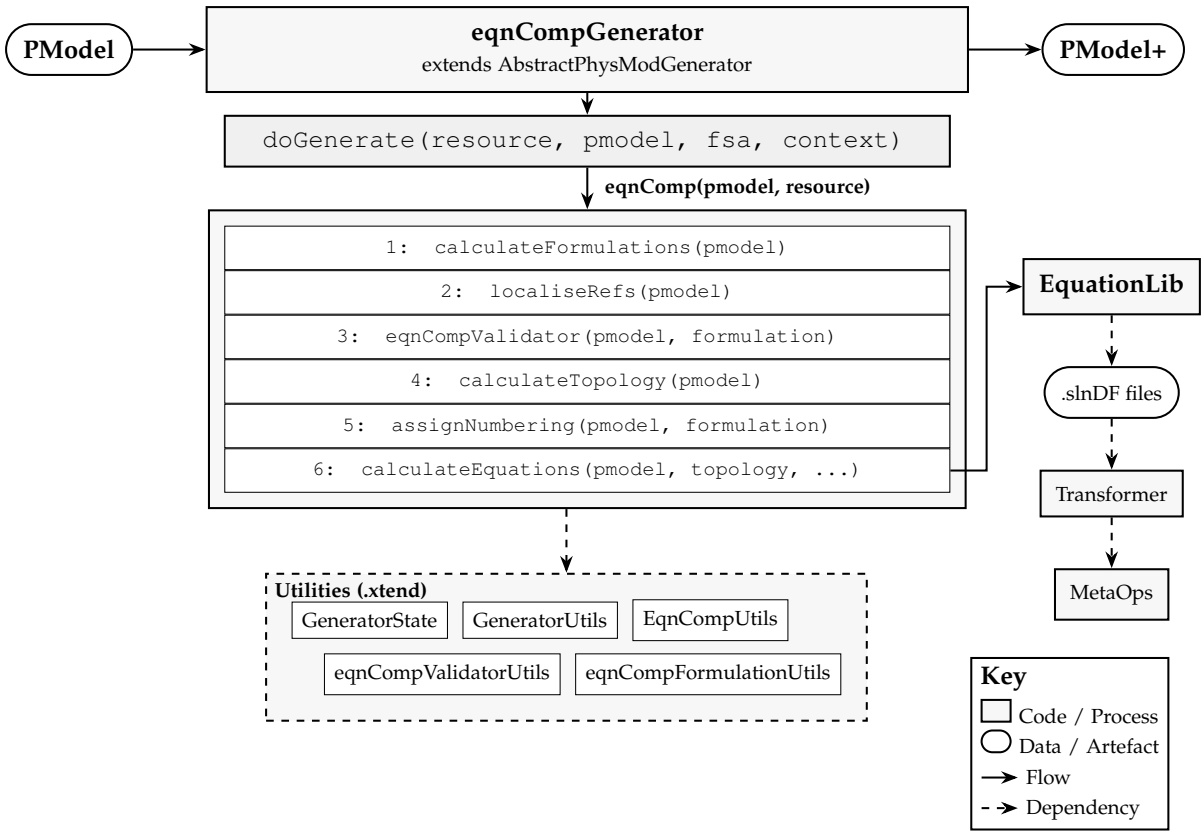


Figure 4.3: eqnComp generator architecture. Rectangular nodes represent code components, while rounded nodes represent data artefacts. The pipeline delegates equation generation to the external EquationLib subsystem.

The diagram distinguishes code components from data artefacts using shape: rectangular nodes represent code (the generator class, processing steps, external libraries, utilities, and transformers), while rounded nodes represent data (the input p-model, output p-model+, and specification files). Solid arrows indicate execution and data flow, tracing the main path from input p-model through the generator class, method invocation, and pipeline stages to the output p-model+. Dashed arrows indicate structural dependencies: components that the pipeline relies upon (such as utility classes) or internal composition relationships within subsystems (such as the EquationLib's use of specification files, the Transformer, and MetaOps).

The Extension Utilities (shown in the dashed box at the bottom of Figure 4.3) are separate classes instantiated in the generator's constructor. Each utility is instantiated once and shared across all generator methods. In the diagram, utilities appear in a grouped box to indicate their structural role as external helper classes that the generator delegates to, rather than methods defined within the generator itself. The `extension` declaration causes Xtend to make each utility's methods available as if they were defined directly on the generator class. The utilities provide the following functionality: `GeneratorState` maintains the generator mode (e.g., `Target.SKO`); `GeneratorUtils` handles uninitialised constants and user prompts; `EqnCompUtils` provides localisation helpers used in Step 2 (`expandParts` and conversion of reference joints/links/sensors/actuators/bodies into local components by deep-copying definitions) and numbering helpers used in Step 5 (e.g., `addNumber` and `addPoseVectorVariable`); `eqnCompFormulationUtils` supports formulation detection in Step 1; and `eqnCompValidatorUtils` supports validation in Step 3.

The right-hand side of Figure 4.3 shows the data-driven equation generation components. The `EquationLib` class dispatches to the appropriate formulation library based on the detected formulation. Rather than implementing each formulation as an Xtend class with methods, each formulation is specified as a solution description format file (`.slnDF`) stored in the `physmod-libs/eqnComp/{FORMULATION}/` directory. The slnDF transformer parses these specifications by reading the `.slnDF` file using the Xtext-generated parser to produce an abstract syntax tree (AST), maps variable names to values, and executes each statement in the `computation` block to apply the corresponding model operations. The `MetaOperations` class implements primitive functions that bridge the slnDF specifications to EMF model manipulation. Table 4.1 lists these functions, organised into model modification and accessor categories.

Function	Signature	Purpose
Model Modification		
`addInOut`	`(ctx, name, type, initial?)`	Add InOut variable to context
`addInput`	`(ctx, name, type)`	Add Input variable to context
`addGeomVariable`	`(ctx, name, body)`	Create Geom variable from Body EMF object
`createLocal`	`(ctx, name, type, initial)`	Create local variable
`createConstant`	`(ctx, name, value)`	Create constant
`createEquation`	`(ctx, expr)`	Create equation in context
`createConstraint`	`(ctx, expr)`	Create constraint in context
`createRelation`	`(ctx, expr)`	Create relation in context
`createFlexiRelation`	`(ctx, expr)`	Create flexible connection relation
`createDatatype`	`(ctx, name, fields)`	Create record type datatype
`createFunction`	`(ctx, name, params, retType)`	Create function declaration
Accessor Functions		
`getVariable`	`(ctx, name) : Object`	Get variable; supports dot notation for record fields (e.g., `geom.geomType`)
`getLinkNumber`	`(link) : int`	Get link number from def locals
`getJointNumber`	`(joint) : int`	Get joint number from def locals
`evaluateExpression`	`(expr) : real`	Evaluate RoboChart expression to numeric value

Table 4.1: MetaOperations functions for slnDF to EMF model manipulation.

Table 4.2 provides a mapping between the pseudocode notation used in Chapter 3 and the corresponding Xtend source code constructs. This mapping serves as a reference for understanding the implementation comparisons presented throughout this chapter for all three generators (eqnComp, guidedChoice, and sourceCodeGen) and in Appendix A.3.

Pseudocode	Xtend	Description
Control Flow		
`foreach x ∈ collection do`	`for (x : collection) {`	Iteration
`end`	`}`	Block terminator
`if condition then`	`if (condition) {`	Conditional
`else if condition then`	`} else if (condition) {`	Alternative branch
`else`	`} else {`	Default branch
Logical Operators		
$\wedge$	`&&`	Logical and
$\vee$	`||`	Logical or
$\neg$	`!`	Logical negation
$\exists x \in S : P(x)$	`for (x : S) { if (P(x)) return true }; return false`	Existential quantifier
Comparison Operators		
$=$	`===` or `.equals()`	Equality
$\neq$	`!==` or `!.equals()`	Inequality
$>, \geq, <, \leq$	`>, >=, <, <=`	Relational
Set/Collection Operations		
`{}`	`new LinkedHashSet<T>()`	Empty set (ordered)
`{}`	`new LinkedHashMap<K,V>()`	Empty map (ordered)
`[]`	`new ArrayList<T>()`	Empty list
$x \in S$	`S.contains(x)`	Membership test
$x \notin S$	`!S.contains(x)`	Non-membership test
$S \cup \{x\}$	`S.add(x)`	Add element
$S \cup T$	`S.addAll(T)`	Add collection
`size(S)`	`S.size`	Collection size
`S[i]`	`S.get(i)`	List index access
`S[i] = v`	`S.set(i, v)`	List index assignment
`M[k] = v`	`M.put(k, v)`	Map assignment
`dom(M)`	`M.keySet`	Domain of map (set of keys)
Object Construction		
`T(...)`	`new T(...)`	Constructor call
Null Handling		
`null`	`null`	Null reference
`x.field`	`x?.field`	Null-safe navigation
Variables		
`x = value`	`var x = value`	Mutable variable
`x = value`	`val x = value`	Immutable variable
`Output: x : T`	`var T x`	Output variable declaration

Table 4.2: Mapping between pseudocode notation and Xtend implementation.

As well as the mapping in the table, there are additional differences between the pseudocode and the implementation to consider. Namely, the pseudocode in Chapter 3 assumes *pass-by-value* semantics: when a variable is passed to a function, the function operates on a copy, leaving the original unchanged. The Xtend/Java implementation uses *pass-by-reference* for objects like the p-model, meaning functions receive a reference to the original object and can modify it in place. This is an intentional optimisation for large structures like the EMF p-model, where deep copying at every call would be prohibitively expensive.

To ensure the safe pattern is followed, the implementation creates one working copy at the start of `eqnComp` via `createWorkingCopy()`, which serialises the model to text and re-parses it into a new EMF resource. Step 1 (formulation detection) then operates on the original p-model in read-only mode, so no copy is needed for reading. Steps 2–6 operate on the copy, mutating it in place, and the original is never accessed after Step 1 completes. This pattern ensures correctness: the original is used only for read-only operations, and all mutations occur on the copy. The same pattern applies to guidedChoice, which creates a copy via `EcoreUtil.copy(pmodel)` before mutation.

The main algorithm is implemented in the `eqnComp` method, which orchestrates the six steps. Figure 4.4 shows the direct correspondence between the Chapter 3 pseudocode and its Xtend implementation:

The implementation preserves the structure of the pseudocode, with each step corresponding to a method call. However, several differences reflect practical implementation considerations.

The implementation creates a deep copy of the p-model via `createWorkingCopy()` before any mutations, with Step 1 using the original in read-only mode while Steps 2–6 mutate the copy; this implements the pass-by-value semantics of the pseudocode using pass-by-reference with explicit copying. The expression `formulations?.dynamicsFormulation` uses Xtend's null-safe operator, which returns `null` instead of throwing a `NullPointerException` if `formulations` is `null`. The `Resource resource` parameter is required by the Xtext framework for model serialisation and persistence. The pseudocode returns `pmodel_` directly, whereas the implementation serialises the model to its textual representation for the Xtext code generation framework.

The optional `overrideFormulation` parameter is passed to `calculateFormulations`, which uses it directly instead of auto-detecting; this enables generation for edge cases such as single-link robots where no joints exist from which to infer the formulation type. A backward-compatible overload without this parameter is provided for the common case where auto-detection is sufficient. The actual implementation also includes null checks, exception handling, and resource cleanup (try-finally blocks) that are omitted from the pseudocode for clarity. Detailed side-by-side comparisons of the remaining steps (`calculateFormulations`, `findDynamicsFormulation`, `localiseRefs`, etc.) are provided in Appendix A.3.

```
Input: pmodel = SimpleArm
Input: overrideFormulation (optional)
Output: pmodel_ : pmodel+

# Step 1: Calculate Formulations
formulations = calculateFormulations(
  pmodel, overrideFormulation)
formulation_dyn =
  formulations.formulation_dyn

# Step 2: Localise references
pmodel_ = localiseRefs(pmodel)

# Step 3: Validate
eqnCompValidator(pmodel_,
  formulation_dyn)

# Step 4: Determine topology
topology = calculateTopology(pmodel_)

# Step 5: Assign numbering
pmodel_ = assignNumbering(pmodel_,
  formulation_dyn)

# Step 6: Calculate equations
pmodel_ = calculateEquations(pmodel_,
  topology, formulations)

return pmodel_
```

Listing 4.3: Algorithm 3.1 (Pseudocode)

```
def CharSequence eqnComp(PModel pmodel,
    Resource resource,
    Formulation overrideFormulation) {
  // Create working copy
  val copyPair = createWorkingCopy(pmodel,
      resource)
  val pmodelCopy = copyPair.key
  val tempResource = copyPair.value

  // Step 1: Detect formulations
  var formulations = calculateFormulations(
    pmodel, overrideFormulation)
  var dynamicsFormulation =
    formulations?.dynamicsFormulation

  // Step 2: Localise refs
  var pmodel_ = localiseRefs(pmodelCopy)

  // Step 3: Validate model
  eqnCompValidator(pmodel_,
    dynamicsFormulation)

  // Step 4: Determine topology
  var topology = calculateTopology(pmodel_)

  // Step 5: Assign numbering
  pmodel_ = assignNumbering(
    pmodel_, dynamicsFormulation)

  // Step 6: Generate equations
  pmodel_ = calculateEquations(pmodel_,
    topology, formulations, tempResource)

  // Serialize and return
  state.idSanitizationMap.clear
  val outputStream = new
      ByteArrayOutputStream()
  tempResource.save(outputStream,
    SaveOptions.newBuilder().getOptions().
        toOptionsMap())
  var result = outputStream.toString("UTF-8
      ")
  result = result.replace("record Geom", "
      datatype Geom")
  return result
}
```

Listing 4.4: eqnComp Implementation

Figure 4.4: Side-by-side comparison of the eqnComp algorithm and its implementation, demonstrating direct correspondence between each step. Both accept an optional `overrideFormulation` parameter which is passed to `calculateFormulations` to skip auto-detection for edge cases.

Equation Library

The Equation Library provides formulation-specific equation generation for eqnComp Step 6 (`calculateEquations`). When the main generator calls `lib.calculateEquations()`, the library selects the appropriate formulation and topology-specific specification and applies it to the p-model. This dispatch mechanism enables the generator to support multiple dynamics formulations through a common interface.

In this thesis, *complete* is meant relative to the chosen dynamics formulation and the modelling scope of the p-model. After eqnComp, the extended p-model (PModel+) contains a closed, well-typed set of equations in which all formulation-specific intermediate quantities needed to state the system-level dynamics are explicit (for SKO, e.g., $\boldsymbol{\phi}$, H, M, $\mathcal{M}$, and $\boldsymbol{C}$), together with the required component-level equations from which they are composed. This does not imply that every physical phenomenon is modelled; rather, for the phenomena represented in the p-model, no additional hand-written dynamics equations are required downstream for solution selection and code generation. The initial model may be minimal, containing, for instance, only the joint definitions for the SKO formulation (as shown in Example 3.1).

To ensure transparency, the framework can decompose system-level equations, explicitly revealing their component-level origins. While a full decomposition can be complex, the SKO formulation is a prime example where system-level operators are transparently composed from their component-level counterparts. For other formulations, compositionality may be less obvious: component-level equations may not relate directly to system-level equations through simple block structure, and traceability may depend on what the formulation library chooses to expose. The onus is on the author of each formulation's equation library to preserve traceability as far as possible (e.g., by emitting explicit relations or auxiliary equations that show how system-level terms were obtained); otherwise, the derivation is implicitly encoded in the specification computations performed by the library algorithms themselves.

The `EquationLib` class routes equation generation requests based on the detected formulation (SKO, Featherstone, etc.) and topology (serial chain, tree topology, closed chain). Rather than implementing each formulation directly in generator code, each formulation is specified as a slnDF description that declaratively adds variables, equations, constraints, and relations to the p-model. The equation library dispatches to the appropriate specification and interprets it at runtime, applying the corresponding meta-operations to construct the resulting PModel+.

Each formulation library follows a four-step equation generation process as described in Section 3.2.7. The slnDF specification structure includes a procedures section for reusable computation logic and a computation section that implements the four-step process. For the SimpleArm serial chain, the generator selects the `SKO.serialChain` specification; Figure 4.6 illustrates this structure using excerpts from that specification.

The complete slnDF specifications for SKO, Featherstone, and other formulations are maintained in the `physmod-libs` repository. This thesis includes short excerpts to illustrate the structure. The architecture demonstrates how additional formulations can be supported: each formulation is captured by a specification that can be interpreted by the same execution engine, and `EquationLib` routes requests to the appropriate specification based on the detected formulation and topology. Future work to allow user-defined formulations is discussed in Section 4.4.1.

Certain variables are reserved across all formulations for visualisation and downstream pipeline compatibility. Table 4.3 lists these reserved variables.

```
Input: pmodel : PModel,
     topology : Topology,
     formulation : Formulation,
     resource : Resource
Output: pmodel' : PModel

switch formulation do
 case SKO:
   switch topology do
    case serialChain:
     return slnDF.execute(
       "SKO.serialChain", pmodel, resource);
    case treeTopology:
     return slnDF.execute(
       "SKO.treeTopology", pmodel, resource);
    case closedChain:
     return slnDF.execute(
       "SKO.closedChain", pmodel, resource);
   end
 case FEATHERSTONE:
   switch topology do
    case serialChain:
     return slnDF.execute(
       "FEATHERSTONE.serialChain", pmodel, resource);
    case treeTopology:
     return slnDF.execute(
       "FEATHERSTONE.treeTopology", pmodel, resource);
    case closedChain:
     return slnDF.execute(
       "FEATHERSTONE.closedChain", pmodel, resource);
   end
 ...
 default:
   error("Unsupported formulation/topology");
end
```

Listing 4.5: EquationLib Dispatch (Pseudocode)

```
class EquationLib {
 @Inject SlnDFExecutor slnDF

 def calculateEquations(
    PModel pmodel,
    Topology topology,
    Formulation formulation,
    Resource resource) {

  val spec = switch formulation {
   case Formulation::SKO:
    switch topology {
     case Topology::serialChain:
      "SKO.serialChain"
     case Topology::treeTopology:
      "SKO.treeTopology"
     case Topology::closedChain:
      "SKO.closedChain"
    }
   case Formulation::FEATHERSTONE:
    switch topology {
     case Topology::serialChain:
      "FEATHERSTONE.serialChain"
     case Topology::treeTopology:
      "FEATHERSTONE.treeTopology"
     case Topology::closedChain:
      "FEATHERSTONE.closedChain"
    }
   ...
   default:
    null
  }

  if (spec === null)
   throw new UnsupportedOperationException()

  return slnDF.execute(spec, pmodel, resource)
 }
}
```

Listing 4.6: EquationLib Implementation (Xtend)

Figure 4.5: Side-by-side comparison of EquationLib dispatch mechanism. The library routes equation generation to the appropriate formulation-specific specification based on the detected formulation and topology.

```
procedures {
   // Compute halfHeight from geometry type and values
   procedure computeHalfHeight(val gType: String, val gVal: seq(real)) {
      if (gType == "box") { halfHeight = gVal[2] / 2.0; }
      else if (gType == "cylinder") { halfHeight = gVal[1] / 2.0; }
      else if (gType == "sphere") { halfHeight = gVal[0]; }
      else { halfHeight = 0.0; };
   }
   // Create initial value constraint for B_k (joint frame transformation)
   procedure createBConstraint(val link: Link, val k: int, val z: real, val halfH: real) {
      jointZ: real = z - halfH;
      bMatStr = "[|1,0,0,0;0,1,0,0;0,0,1," + jointZ + ";0,0,0,1|]";
      createConstraint(link, "(B_" + k + ")[t==0]==" + bMatStr);
   }
}

computation {
   n = length(pmodel.links);

   // ===== Step 1: Define Datatypes and Functions =====
   createDatatype(pmodel, "Geom", "geomType: String; geomVal: vec()");
   createFunction(pmodel, "identity", "size: int", "mat(3,3)");
   createFunction(pmodel, "Phi", "m: int, n: int, B_k: seq(mat(4,4))", "mat(6,6)");


   // ===== Step 2: Define System Variables =====
   createConstant(pmodel, "n", n);
   addInOut(pmodel, "B_k", "seq(mat(4,4))");
   addInOut(pmodel, "T_geom", "seq(mat(4,4))");
   addInOut(pmodel, "T_offset", "seq(mat(4,4))");


   // ===== Step 3: Component Equations =====
   for (idx: int in range(0, n, 1)) {
      currentLink = pmodel.links[idx];
      linkNum = getVariable(currentLink, "number");
      currentBody = currentLink.bodies[0];
      // Create geometry variable from EMF Body object
      geomVarName = "geom_" + linkNum + "1";
      addGeomVariable(currentLink, geomVarName, currentBody);
      // Access geometry data and compute halfHeight
      geomTypeStr = getVariable(currentLink, geomVarName + ".geomType");
      geomVal = getVariable(currentLink, geomVarName + ".geomVal");
      computeHalfHeight(geomTypeStr, geomVal);
      // Access pose data for constraint computation
      poseVec = getVariable(currentLink, "poseVec");
      poseZ = poseVec[2];
      createBConstraint(currentLink, linkNum, poseZ, halfHeight);

   };

   // ===== Step 4: System Equations =====
   createEquation(pmodel, "V == adj(phi) * adj(H) * derivative(theta)");
   createEquation(pmodel, "tau == M_mass * derivative(derivative(theta)) + C");
   createEquation(pmodel, "T_geom == <T_geom_1,T_geom_2,T_geom_3>");
   createEquation(pmodel, "T_offset == <T_offset_1,T_offset_2,T_offset_3>");

}
```

Figure 4.6: SKO formulation: procedures and four-step equation generation structure.

Variable	Type	Semantics
Visualisation Transforms		
`T_geom_k`	`mat(4,4)`	Spatial position of geometry for link k
`T_offset_k`	`mat(4,4)`	Offset from joint frame to geometry centre for link k
`T_geom`	`seq(mat(4,4))`	Sequence of all geometry positions
`T_offset`	`seq(mat(4,4))`	Sequence of all geometry offsets
Geometry Data		
`geom_k1`	`Geom`	Geometry record for body 1 of link k (fields: geomType, geomVal)
Pose Data		
`poseVec`	`vec(6)`	Local variable storing body pose $[x, y, z, roll, pitch, yaw]$

Table 4.3: Reserved variables generated by all formulation libraries.

The visualisation transforms are 4×4 homogeneous matrices used by sourceCodeGen to generate code that positions visual geometry relative to joint frames. The sequence variables (`T_geom`, `T_offset`) aggregate the per-component transforms for system-level use. The geometry variables (`geom_k1`) are created using the `addGeomVariable` metaoperation, which extracts geometry type and dimensions from the EMF `Body` object and stores them in a typed record. The pose data (`poseVec`) is accessed from the link's local variables to compute initial constraint values for joint frame positions.

4.1.2 T4: Guided Choice

This section describes the implementation of guidedChoice from Section 3.4 of Chapter 3. The generator queries the solution library to find appropriate computational methods for each variable in the p-model+.

Generator Architecture

The guidedChoice generator follows the same Xtext code generator pattern described in Section 4.1.1: it extends `AbstractPhysModGenerator`, exposes supporting services via Xtend `extension` fields, and dispatches from `doGenerateT4` to `Technique4` (the guided-choice pipeline) and `genSlnRef` (the serialisation of resolved solution references). Figure 4.7 presents the architecture.

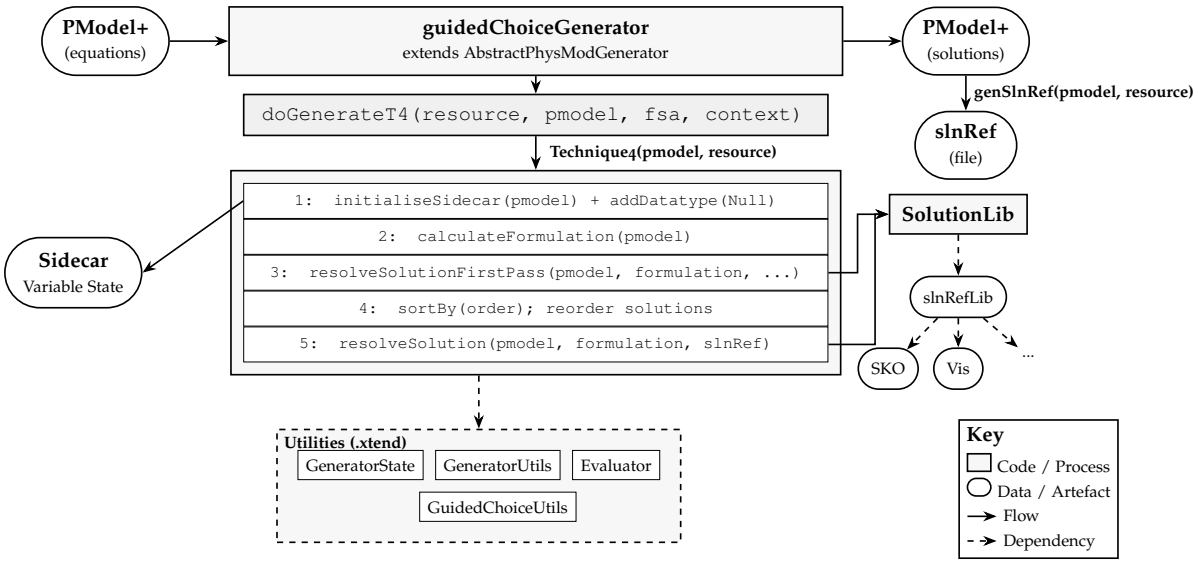


Figure 4.7: guidedChoice generator architecture. Rectangular nodes represent code components, while rounded nodes represent data artefacts. The Sidecar maintains mutable variable state across all steps. Steps 3 and 5 delegate solution resolution to the external SolutionLib subsystem; the resolved p-model is then rendered as a `slnRef` file.

In this instance, the dashed arrows indicate that the pipeline depends on the shared utility classes, and that SolutionLib depends on the slnRefLib repository of formulation-specific entries (including SKO, Vis, and other entries indicated by "..."). Although these entries are treated as data artefacts in the framework model (analogous to the slnDF files used by Technique 3), the current implementation realises them as Xtend code. Future work (Section 4.4.1) proposes migrating to a purely specification-based format.

As in the eqnComp generator, the implementation is written in Xtend and factors common functionality into reusable helper modules that are accessed via Xtend's `extension` mechanism:

```
class guidedChoiceGenerator extends AbstractPhysModGenerator {
   public extension GeneratorState state
   public extension GeneratorUtils gu
   public extension Evaluator ev
   public extension Preprocessing prep
   public extension QualifiedNameUtils qnu
   public extension PoseUtils poseg
   public extension GuidedChoiceUtils utils
   // ... other extension fields omitted for brevity ...
   @Inject ISerializer serializer
   @Inject IQualifiedNameProvider qnp
   @Inject extension PhysModUtils
   @Inject Provider<ResourceSet> resourceSetProvider
   @Inject IResourceValidator validator

   val Map<DynamicDevice, Map<String, VarInfo>> sidecar =
      Maps.<DynamicDevice, Map<String, VarInfo>>newHashMap
}
```

Figure 4.8: guidedChoiceGenerator class structure

The Extension Utilities (shown in the dashed box at the bottom of Figure 4.7) are helper

modules that the generator delegates to. In the current implementation most of these helpers are instantiated in the generator constructor, while Xtext services (e.g., the serializer and qualified-name provider) are injected. `GeneratorState` records the selected generation mode (validation, proof, mapping), set by the surrounding tooling. `GeneratorUtils` provides shared helper functions (e.g., constant collection and prompting, as in eqnComp). `Evaluator` implements expression evaluation and constant folding. `GuidedChoiceUtils` contains guidedChoice-specific helpers used by `calculateInitialConditions` and `processAlgebraicConstraints` (Algorithms 40 and 41).

The `sidecar` (shown as a rounded data node on the left of Figure 4.7) is a *global shared state structure* that distinguishes guidedChoice from eqnComp. Unlike the utilities, which are stateless helper classes, the sidecar is a class-level field that persists throughout the generator's execution. Its conceptual role and the `VarInfo` record structure are described in Section 3.4.2; here we focus on the implementation aspect. In the architecture diagram, the solid arrow from Step 1 to the Sidecar node indicates the initialisation flow via `initialiseSidecar(pmodel, sidecar)`, after which subsequent steps query and update the shared state.

The pseudocode-to-Xtend mapping from Table 4.2 in the eqnComp architecture section applies equally to the guidedChoice implementation. The pseudocode assumes pass-by-value, while the implementation relies on pass-by-reference and therefore introduces explicit copies when a stable snapshot is required. In particular, Step 3 makes a deep copy of the p-model's solutions (e.g., `pm2 = EcoreUtil.copy(pmodel)`) so that the algorithm can iterate over the original solution list while removing and replacing entries, and later steps create independent structures such as `PM_out` when constructing derived outputs. Intermediate `resource.save(...)` calls persist updates to the EMF resource as the algorithm progresses. The `sidecar` is passed explicitly to `initialiseSidecar(pmodel, sidecar)` to make this shared-state discipline explicit.

The main algorithm is implemented in the `Technique4` method, which orchestrates the five steps:

```
Input: pmodel+, formulation
Output: pmodel+ with solutions

// Step 1: Initialise sidecar
initialiseSidecar(pmodel)
addDatatype(Null)

// Step 2: Calculate formulation
formulation =
  calculateFormulation(pmodel)

// Step 3: First pass resolution
solution_ref =
  resolveSolutionFirstPass(
    pmodel, formulation)

// Step 4: Reorder solutions
solutions.sortBy(order)

// Step 5: Iterate resolution
for sln in pmodel.solutions:
  sln_ref = findMatchingSolutionRef(sln)
  for i in 0..sln_ref.iterations:
    resolveSolution(pmodel, formulation,
        sln_ref)

return pmodel
```

Listing 4.7: guidedChoice Algorithm (Pseudocode)

```
def String Technique4(
    PModel pmodel, Resource resource) {
  // Step 1: Init global sidecar and
      services
  initialiseSidecar(pmodel, sidecar)
  GeneratorServices.initialize()
  val Null = RoboChartFactory.eINSTANCE
    .createPrimitiveType
  Null.name = "Null"
  pmodel.addDatatype(Null)

  // Step 2: Calculate formulation
  var formulation =
    calculateFormulation(pmodel)

  // Step 3: First pass resolution
  var solution_ref =
    resolveSolutionFirstPass(
      pmodel, formulation, resource)

  // Step 4: Reorder solutions
  pmodel.solutions.sortBy[order]
  solution_ref.sortBy[order]

  // Step 5: Resolve each solution
  for (sln_PM : pmodel.solutions) {
    var sln_ref = solution_ref.findFirst[
      expression.name == sln_PM.solutionExpr
          .name]
    resolveSolution(pmodel,
      formulation, sln_ref, resource)
  }

  // Cleanup and serialize
  container.types.remove(nullType)
  rebindSolutionConstraintSNames(pmodel)
  return serializer.serialize(resource.
      contents.get(0))
}
```

Listing 4.8: guidedChoice Implementation

The implementation largely preserves the structure of the pseudocode, with each step realised as a method call. Several implementation-specific details are worth noting. The implementation uses `resource.save(...)` after major steps to keep the EMF resource and Xtext index consistent; these saves are a pragmatic safeguard but noticeably slow down execution. The temporary `Null` datatype is introduced to represent unresolved types during early passes; as types are discovered, `updateVariable` replaces Null-typed p-model variables with concrete types, and `Technique4` removes the Null datatype prior to final serialisation. Before serialising, `Technique4` also calls `rebindSolutionConstraintSNames` to normalise qualified-name references in constraints. Finally, although Algorithm 10 expresses iteration as an explicit inner loop in Step 5, the implementation performs the loop inside `resolveSolution`; the top-level method invokes `resolveSolution` once per solution, and `sln_ref.iterations` controls the number of refinement passes.

Steps 3 and 5 delegate solution resolution to the `SolutionLib` class, which is described in Section 4.1.2. Detailed side-by-side comparisons of all guidedChoice algorithms (including `initialiseSidecar`, `resolveSolutionFirstPass`, `updateVariable`, `calculateInitialConditions`, and `processAlgebraicConstraints`) are provided in Appendix A.3.2.

Solution Library

The **Solution Library** is a repository of solution methods and algorithms for solving behavioural equations in robotic systems.

It is used by guidedChoice to select appropriate algorithms based on formulation type and solution requirements, enabling the automatic unfolding of high-level solution specifications into concrete computational implementations. Figure 4.9 highlights this loop: unresolved `slnRef` annotations are loaded into an in-memory `SolutionRef`, which repeatedly queries the solution library until the reference is resolved and written back to the model.

A **solution query** consists of a target expression to compute and a **solution method** name that identifies which algorithm from the library should be used. For example, querying for variable $\boldsymbol{\theta}$ (generalised positions) with method `GeneralisedPosition_method1` in the SKO formulation.

Solution unfolding is the process by which guidedChoice uses the library to decompose a high-level solution request into one or more concrete sub-solutions with explicit algorithms, variables, constraints, and dependencies. This enables users to request solutions at an abstract level without needing to know the detailed computational steps.

Table 4.4 summarises the key concepts and artefacts involved in the solution library workflow which was presented in Figure 4.9 and Figure 4.14:

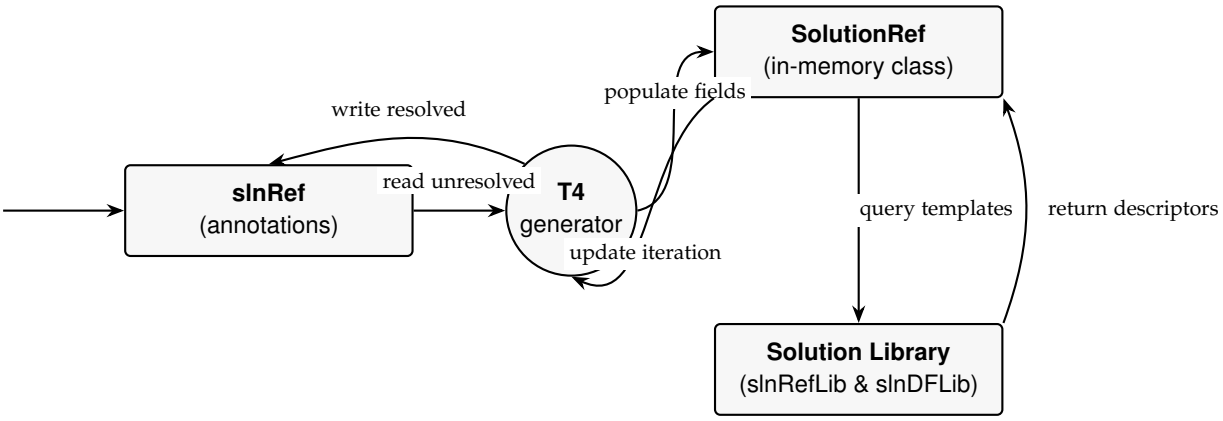


Figure 4.9: T4 loop: solution references are iteratively unfolded into resolved `slnRef` entries via the `SolutionRef` class and the shared solution library.

Concept/Artefact	Description
`SolutionRef` class	In-memory datastructure manipulated by guidedChoice. Extends the slnRef format with additional `iterations/iteration` fields used for iteration tracking during unfolding.
Solution reference specification (slnRef)	Schema describing the expression, method, order, inputs, and constraints captured for each solution reference. May exist as a standalone `.slnRef` file emitted by guidedChoice and consumed by sourceCodeGen, or be appended directly to the p-model's solutions block; when persisted, the `.slnRef` lists resolved `SolutionRef` entries in dependency order.
`slnRefLib` template library	Formulation-specific templates that guidedChoice instantiates to unfold high-level slnRef queries into detailed `SolutionRef` instances.
Solution Description Format specification (slnDF)	`.sln` file generated by sourceCodeGen by instantiating `slnDFLib` templates with data from `.slnRef` for platform-specific code generation.
`slnDFLib` template library	Repository of parameterised algorithm specifications keyed by method name; sourceCodeGen instantiates them to produce slnDF descriptions.

Table 4.4: Key concepts and artefacts in the solution library workflow.

A *solution reference file* (`.slnRef`) is the on-disk artefact produced by guidedChoice and consumed by sourceCodeGen. By contrast, the `SolutionRef` *class* is the in-memory data structure used by the library during the unfolding and resolution process within guidedChoice. The file format and class share the name "SolutionRef" but differ in their role and lifecycle: the file is a persistent artefact exchanged between techniques, while the class is a transient data structure used during algorithm selection.

Example 4.1: Simple Arm Serial Manipulator: Library Query

For the Simple Arm example, the user begins by annotating the p-model with a single slnRef instance specifying the target variable θ (generalised positions, type `vector(real,2)`) and the solution method `GeneralisedPosition_method1`. This slnRef is added to the p-model's solutions collection. During guidedChoice, the generator uses the SKO entries of the solution library (maintained in `physmod-libs`) to unfold this high-level request into an ordered pipeline of sub-solutions. In the full-simulation setting, a separate mapping solution provides the torque input τ from actuator commands. Table 4.5 summarises the resulting pipeline:

Order	Variable	Method	Purpose
1	$\boldsymbol{\tau}$	`PlatformMapping`	Obtain joint torques from actuator commands
2	$\boldsymbol{\phi}$	`Eval`	Evaluate rigid-shift operator
3	$\boldsymbol{C}$	`NewtonEuler-InverseDynamics`	Compute inverse dynamics (bias forces)
4	$\mathcal{M}$	`CompositeBody-Algorithm`	Compute mass matrix
5	$\mathcal{M}^{-1}$	`CholeskyAlgorithm`	Invert mass matrix
6	$\ddot{\boldsymbol{\theta}}$	`DirectForward-Dynamics`	Compute joint accelerations
7	$\dot{\boldsymbol{\theta}}$	`Euler`	Integrate accelerations to velocities
8	$\boldsymbol{\theta}$	`Euler`	Integrate velocities to positions

Table 4.5: Ordered solution pipeline for the Simple Arm example. The torque input $\boldsymbol{\tau}$ is supplied by the platform mapping solution; the remaining entries are unfolded from the user-level $\boldsymbol{\theta}$ request.

Each row in Table 4.5 represents a slnRef instance in the p-model's solutions block. The library infers dependencies, initial conditions, and execution order during unfolding so that later steps can generate correct code.

This unfolding demonstrates that users specify high-level goals by adding a single slnRef to the p-model ("solve for $\boldsymbol{\theta}$") without needing to know the detailed forward dynamics pipeline.

Listing 4.9 shows an excerpt from the SKO solution library entry for `GeneralisedPosition_method1` (from `physmod-libs`). The entry constructs the dynamics sub-solutions and assigns their order; it treats $\boldsymbol{\tau}$ as an input that is supplied by mapping or actuator solutions.

The library supports consistency because it is transparent, configurable, and extensible. Transparency comes from documenting selections and composition in the `.slnRef` and `.sln` artefacts, which enables inspection before code generation. Configurability is provided by variable parameters in the slnRef so solutions can be adapted across different robot models. Extensibility is achieved via the mechanisms in Section 4.1.2, which allow new methods and templates to be added without modifying the generators.

This section describes the implementation of the solution library, which is the core component of guidedChoice from Section 3.4. The library provides computational methods for solving the equations generated by eqnComp, enabling the automatic selection of appropriate algorithms based on the detected formulation type and computational requirements.

The **SolutionLib** class implements the solution query mechanism described in Algorithm 10 from Chapter 3. It takes a formulation type and a solution reference as input, and returns one or more solution references with concrete implementation details through the solution unfolding process. This class is the practical realisation of the guidedChoice query interface.

The `SolutionRef` class (in-memory) represents a fully-specified solution with all implementation details. It extends the high-level solution specification from the p-model with concrete algorithms, variables, constraints, equations, and iteration information needed for implementation. Do not confuse with the solution reference file (`.slnRef`). This data structure

corresponds to the solution representation in the guidedChoice technique.

The solution library is implemented as a collection of Xtend classes organised by formulation type, directly implementing the library architecture proposed in guidedChoice. The main entry point is the `SolutionLib` class, which dispatches solution queries to formulation-specific implementations (such as `SKO_gen` for SKO) based on the detected formulation type. This implementation enables the library to handle the Simple Arm example and other SKO-based models systematically. Listing 4.10 shows the *SolutionRef data structure*, which provides the in-memory representation of a resolved solution.

```
static class GeneralisedPosition_method1 {
  static def EList<SolutionRef> asReference(SolutionRef solution) {
    val dof = requireThetaDof(solution)
    val n = requireBodyCount(solution)

    // tau is expected to be provided as an input (e.g., PlatformMapping)

    var solution_phi = new SolutionRef()
    solution_phi.expression.name = "phi"
    solution_phi.expression.type = "matrix(real," + 6 * n + "," + 6 * n + ")"
    solution_phi = Eval.asReference(solution_phi)
    solution_phi.order = 2

    ...

    var solution_theta = new SolutionRef()
    solution_theta.expression.name = "theta"
    solution_theta.expression.type = "vector(real," + dof + ")"
    solution_theta = Euler.asReference(solution_theta)
    solution_theta.order = 8

    var solutions = new BasicEList<SolutionRef>()
    solutions.add(solution_phi)
    ...
    solutions.add(solution_theta)
    return solutions
  }
}
```

Listing 4.9: Excerpt from the guidedChoice SKO library: `GeneralisedPosition_method1`

```
class SolutionRef {
   public var Local expression // Variable to compute
   public var EList<String> constants // Constants used
   public var EList<Local> inputs // Input variables
   public var String method // Method name
   public var int order // Processing order
   public var int group // Solution group
   public var EList<String> constraints // Constraints
   public var EList<String> errors // Error terms
   public var EList<String> equations // Defining equations
   public var int iterations = 0 // Number of iterations
   public var int iteration = 0 // Current iteration
}
```

Listing 4.10: `SolutionRef` data structure

Listing 4.11 shows the *Local data structure* used to record a variable's name and type.

```
class Local {
   public var String name // Variable name
   public var String type // Type (e.g., "vector(real,2)")
}
```

Listing 4.11: `Local` data structure

These data structures capture all information needed to instantiate a solution from the library into the p-model and eventually generate source code. The `SolutionRef` class corresponds to the `.slnRef` file format but includes additional fields for runtime processing: `iterations` (total iteration count) and `iteration` (current iteration number). Core fields such as `expression`, `method`, `order`, `group`, `inputs`, `constraints`, `constants`, `errors`, and `equations` mirror the persisted format, while types are stored as `Local` objects with parsed name and type information for easier manipulation.

Main library interface. The `SolutionLib` class provides the main query interface through its `resolveSolution` method. This method implements a two-level dispatching mechanism: the outer switch statement selects the formulation-specific generator based on the formulation type parameter, and the inner switch statement routes to the appropriate solution method within that formulation. Listing 4.12 shows the structure for the SKO formulation.

```
class SolutionLib {
   static def resolveSolution(
      Formulation formulation,
      SolutionRef solution
   ) {
      switch formulation {
         case SKO:
            switch solution.method {
               case "Eval":
                  return SKO_gen.Eval.asReference(solution)
               case "GeneralisedPosition_method1":
                  return SKO_gen.GeneralisedPosition_method1
                     .asReference(solution)
               case "PlatformMapping":
                  return Mapping_gen.PlatformMapping
                     .asReference(solution)
               case "WorldMapping":
```

```
                    return Mapping_gen.WorldMapping
                        .asReference(solution)
                case "NewtonEulerInverseDynamics":
                    return SKO_gen.NewtonEulerInverseDynamics
                        .asReference(solution)
                case "CompositeBodyAlgorithm":
                    return SKO_gen.CompositeBodyAlgorithm
                        .asReference(solution)
                case "CholeskyAlgorithm":
                    return SKO_gen.CholeskyAlgorithm
                        .asReference(solution)
                case "DirectForwardDynamics":
                    return SKO_gen.DirectForwardDynamics
                        .asReference(solution)
                case "Euler":
                    return SKO_gen.Euler.asReference(solution)
                case "Visual":
                    return Vis_gen.Visual.asReference(solution)
            }
        }
        return solution
    }
}
```

Listing 4.12: `SolutionLib` dispatch for the SKO formulation

The library currently supports the SKO (Spatial Kernel Operator) formulation with multiple solution methods for different computational tasks. The switch statement routes each method name to its formulation-specific generator, and each generator returns a fully populated `SolutionRef` instance.

Interface Mappings: PlatformMapping and WorldMapping. The solution library provides two special methods for handling the interface between the physics model and the external world: `PlatformMapping` for actuator inputs and `WorldMapping` for sensor inputs. Unlike formulation-specific methods (such as `Eval` or `NewtonEulerInverseDynamics` which are in `SKO_gen`), these interface mapping methods are implemented in a separate `Mapping_gen` class since they are formulation-agnostic—the same mapping logic applies whether the physics is computed using SKO, Featherstone, Kane, or any other formulation.

`PlatformMapping` handles the mapping from abstract control inputs (τ) to physical actuator interfaces. Users must specify the actuator path constraints in their input solution to establish the correspondence between system-level indexed torque variables and platform-specific actuator ports. For example, with a serial manipulator having two joints, the user provides constraints of the form:

```
(tau(0)) [t == t] == p_mapping.RobotName.LinkName.JointName
                                .ActuatorName.TorqueIn
(tau(1)) [t == t] == p_mapping.RobotName.LinkName.JointName
                                .ActuatorName.TorqueIn
```

The `p_mapping` prefix indicates that these values are provided by the d-model (controller) through the platform mapping interface at runtime. The library validates the solution structure and adds default initial conditions (such as `(tau)[t == 0] == zeroVec(N)`) if not already specified.

`WorldMapping` handles the complementary case of sensor inputs from the world engine. Users must specify the sensor path constraints in their input solution to map sensor variables to the world mapping interface:

```
(RobotName.LinkName.SensorName.SignalIn) [t == t] ==
    w_mapping.RobotName.LinkName.SensorName.SignalIn
```

The `w_mapping` prefix indicates that these values are provided by the world engine (environment simulation) through the world mapping interface. As with `PlatformMapping`, the library validates the solution structure and adds default initial conditions if not specified.

This design places the responsibility for specifying the correct actuator and sensor paths on the user, ensuring that the mapping constraints accurately reflect the physical model structure. The user must ensure that the path names in the constraints match the component names defined in their p-model (robot name, link names, joint names, actuator/sensor names, and port names). Both mapping methods can be specified explicitly in the p-model or unfolded automatically when using higher-level formulation methods. For instance, `GeneralisedPosition_method1` automatically includes `PlatformMapping` as its first solution (order 1) to establish the actuator interface before computing dynamics. Users who require sensor inputs can add a separate `WorldMapping` solution to their p-model with an appropriate order number.

Formulation-Agnostic Visualisation: The Visual Method. The solution library provides a special `Visual` method that bridges the gap between formulation-specific physics frames and formulation-agnostic visualisation. This method is implemented in `Vis_gen` and addresses a key architectural challenge: while different physics formulations (SKO, Featherstone, Kane) compute joint frames (B_k) using different algorithms, the visualisation system needs body-centre frames for rendering.

The Visual method uses a two-frame approach:

- B_k **(joint frames):** Computed by the ForwardKinematics method specific to each formulation. These are the Jain convention frames at joint/hinge positions.
- $T_{\mathbf{geom},k}$ **(body centre frames):** Computed by the Visual method from B_k and a fixed offset $T_{\text{offset},k}$.
- $T_{\mathbf{offset},k}$ **(joint-to-body offset):** A fixed 4×4 homogeneous transform from joint frame to body centre, generated by eqnComp from the link geometry.

The relationship is:

$$T_{\text{geom},k} = B_k \cdot T_{\text{offset},k}$$

For example, with a cylindrical link of height 4m, if the joint is at $Z = 0.5$ and the body centre is at $Z = 2.5$, then T_{offset} encodes a +2.0m translation in the local Z-axis:

```
T_offset = [|1,0,0,0; 0,1,0,0; 0,0,1,2.0; 0,0,0,1|]
```

The Visual method generates algebraic constraints that recompute $T_{\text{geom},k}$ at each timestep:

```
(T_geom_1) [t == t] == B_1 * T_offset_1
```

```
(T_geom_2) [t == t] == B_2 * T_offset_2
(T_geom) [t == t] == <T_geom_1, T_geom_2, ...>
```

The sourceCodeGen generator (Section 3.5) then uses T_{geom} directly for visualisation.

Each method implementation is backed by a complete solution specification written in the Solution Description Format (slnDF, Section 3.3.2), which provides the detailed computational steps, functions, procedures, and state management needed for code generation. These specifications are consumed by sourceCodeGen (Section 3.5) to generate executable code. For the Simple Arm example, the `Euler` method specification produces the integration code shown in Listing A.13 (lines 460-462).

Example 4.2: Simple Arm Serial Manipulator: Library Implementation

For the Simple Arm example (Example 3.1), the library query for $\boldsymbol{\theta}$ using `GeneralisedPosition_method1` proceeds through several stages. First, guidedChoice detects the formulation type SKO based on the presence of hinge matrices in the joint definitions (ElbowJoint and WristJoint), using the same formulation detection mechanism as eqnComp. The `SolutionLib.resolveSolution` method then routes this query to `SKO_gen.GeneralisedPosition_method1`, which creates eight `SolutionRef` instances representing the complete forward dynamics pipeline as shown in Example 4.1.
This implementation demonstrates how guidedChoice bridges the gap between the abstract equations generated by eqnComp and the concrete computational methods needed for code generation in sourceCodeGen.
Each sub-solution undergoes recursive resolution through calls to `asReference` on its corresponding method class. For instance, `solution_phi = Eval.asReference(solution_phi)` resolves the rigid-shift operator computation, `solution_C = NewtonEulerInverseDynamics.asReference(solution_C)` resolves the inverse dynamics computation, and `solution_theta = Euler.asReference(solution_theta)` resolves the Euler integration step. During this process, the library automatically infers input dependencies from the algorithm structure: for example, `solution_theta_dd` (acceleration) is marked as an input to `solution_theta_d` (velocity), ensuring the correct processing order.
Type information is inferred from the p-model during unfolding. For the Simple Arm with 2 DOF, the variables $\boldsymbol{\theta}$, $\dot{\boldsymbol{\theta}}$, $\ddot{\boldsymbol{\theta}}$, $\boldsymbol{\tau}$, and $\boldsymbol{C}$ all have type `vector(real,2)`, while the mass matrix $\mathcal{M}$ has type `matrix(real,2,2)`. Initial conditions are also generated based on p-model constants, producing constraints such as $\boldsymbol{\theta}(0) = \mathbf{0}$, $\dot{\boldsymbol{\theta}}(0) = \mathbf{0}$, and the time step $h = 0.1$. The complete unfolding for Simple Arm is documented in Example 3.3 and Table 3.1, demonstrating how a single library query produces a detailed computational pipeline.

Extensibility is supported in two ways. First, additional formulation types (e.g., Featherstone) can be added by creating new formulation-specific generator classes (e.g., `Featherstone_gen`) and adding cases to the `resolveSolution` outer switch statement. Each new formulation requires its own generator class containing method implementations appropriate to that formulation's mathematical structure. Second, within each existing formulation, new solution methods can be added by implementing new static nested classes with the `asReference` method and adding corresponding cases to the inner method-name switch.

4.1.3 T5: Code Generation

This section describes the implementation of sourceCodeGen from Section 3.5 of Chapter 3. The generator transforms the p-model+ with solutions into executable code for multiple target platforms.

Generator Architecture

Figure 4.10 presents the architecture of the sourceCodeGen generator, implemented as an Xtext code generator. The architecture realises the two-phase process described in Section 3.5.1: first resolving solution references to the slnDF intermediate representation, then dispatching to target-specific generators based on configuration.

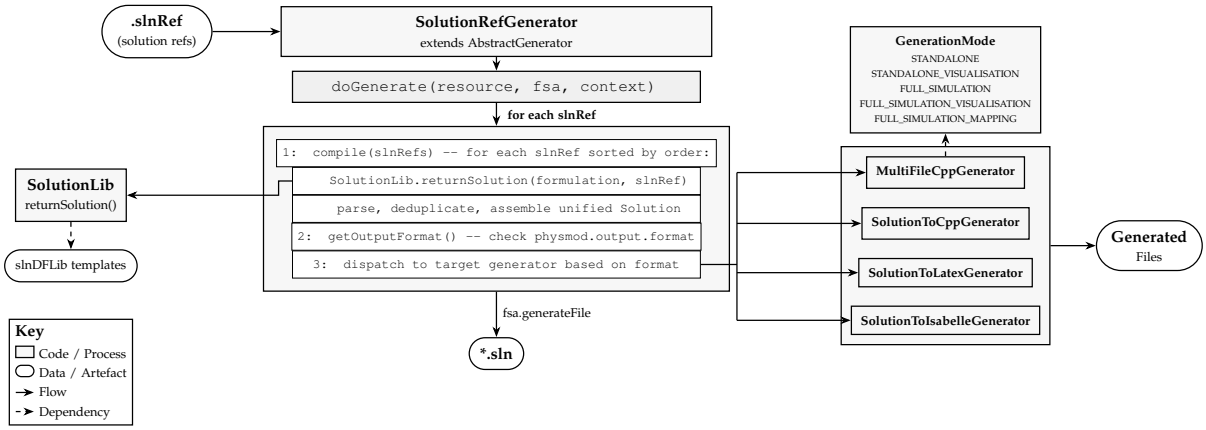


Figure 4.10: sourceCodeGen generator architecture. The generator takes `.slnRef` files as input, resolves each solution reference via SolutionLib to produce a unified `.sln` (slnDF) file, then dispatches to target-specific generators based on the `physmod.output.format` system property.

The `SolutionRefGenerator` class extends Xtext's `AbstractGenerator` and implements the `doGenerate` method as the main entry point.

In the first phase, the generator iterates over each slnRef entry sorted by their `order` attribute. For each entry, it queries `SolutionLib.returnSolution(formulation, slnRef)` to retrieve the corresponding slnDF template. The returned templates are parsed, deduplicated (removing shared state variables, procedures, and functions that appear in multiple solutions), and assembled into a unified `Solution` object. This unified solution is written to a `.sln` file (e.g., `mapping.sln`) as an inspectable intermediate artefact.

In the second phase, the generator checks the `physmod.output.format` system property to determine the target output format. For LaTeX output, it instantiates `SolutionToLatexGenerator` to produce mathematical documentation. For Isabelle output, it uses `SolutionToIsabelleGenerator` to generate theorem prover specifications. For the default C++ output, the generator uses `MultiFileCppGenerator` to produce a structured set of source files.

The pseudocode-to-Xtend mapping from Table 4.2 applies to sourceCodeGen as well. The main pipeline is implemented in `SolutionRefGenerator.doGenerate`; Figure 4.11 summarises the correspondence, with detailed comparisons in Appendix A.3.

```
Input: slnRefFile, resource
Output: generated files

slnRefs = parseSlnRefs(resource)
solution = compile(slnRefs)
write(solution, "*.sln")

mode = getGenerationMode()
format = getOutputFormat()
if format == "latex":
  generateLatex(solution)
else if format == "isabelle":
  generateIsabelle(solution)
else:
  generateCppMultiFile(solution, mode)
```

Listing 4.13: sourceCodeGen Pipeline (Pseudocode)

```
def void doGenerate(
    Resource resource,
    IFileSystemAccess2 fsa,
    IGeneratorContext context) {

  val slnRefs = extractSlnRefs(resource)
  val solution = compile(slnRefs)
  writeSln(solution, fsa)

  val mode = getGenerationMode()
  val format = getOutputFormat()
  switch format {
    case "latex":
      latexGen.generate(solution, fsa)
    case "isabelle":
      isabelleGen.generate(solution, fsa)
    default:
      multiFileGen.generate(solution, fsa,
          mode)
  }
}
```

Listing 4.14: SolutionRefGenerator Implementation

Figure 4.11: Side-by-side comparison of sourceCodeGen pipeline and implementation.

The `MultiFileCppGenerator` supports five generation modes that control the complexity of the generated output. The `STANDALONE` mode generates minimal code for testing physics computations in isolation, while `STANDALONE_VISUALISATION` adds MeshCat visualisation support. The `FULL_SIMULATION` mode generates complete orchestrator integration with interfaces for d-model communication and platform mapping, with `FULL_SIMULATION_VISUALISATION` adding visualisation. Finally, `FULL_SIMULATION_MAPPING` extends full simulation mode with auto-generated platform mapping adapters derived from mapping specifications.

The `SolutionToCppGenerator` translates slnDF constructs to C++ code through a structured process. State variables become Eigen matrix and vector declarations with appropriate dimensions and initial values. Computation blocks are translated to sequential C++ statements, with mathematical operations mapped to Eigen library calls. Procedures become C++ functions with pass-by-reference parameters for matrix outputs, while slnDF functions map to C++ functions with return values. The generator handles, for example, type inference, index translation between slnDF's 1-based and C++'s 0-based indexing, and proper scoping of loop variables.

Output Architecture

The code generated by sourceCodeGen follows a modular, interface-based architecture that preserves the semantics of the RoboSim models while enabling efficient real-time simulation. The architecture consists of five main components that directly correspond to the modelling abstractions.

The platform enginePlatform Engine implements robot dynamics from the p-model+ equations, computing forward kinematics, mass matrices, and integration following the formulation specified in eqnComp. The d-model controllerD-Model Controller is generated from the d-model state machine, preserving the discrete control logic. The platform mappingPlatform

Mapping component implements sensor/actuator interfaces from the mapping specification, converting between discrete operations and continuous signals. The world engineWorld Engine manages environment entities and spatial queries for sensor computation, while the **Orchestrator** coordinates execution following RoboSim cycle semantics at configurable rates (e.g., 200Hz).

Table 4.6 summarises the main output artefacts produced for a C++ target. It distinguishes files that are fully generated by sourceCodeGen, files that are generated with stubs that must be completed for integration, and artefacts supplied either by other techniques or by scenario-specific code.

Category	File	Description
Fully Automatically Generated (T5)		
	`platform1_engine.cpp`	Platform engine implementing the slnDF state, procedures, functions, and computation sequence
	`platform1_state.hpp`	State variable declarations, types, and initialisation support for the platform engine
	`interfaces.hpp`	Interface definitions used to connect orchestrator, platform engine, platform mapping, and world components
Partially Generated (T5 integration stubs)		
	`orchestrator.h`	Orchestrator interface declarations (thread lifecycle, timing, and adapter hooks)
	`orchestrator.cpp`	Orchestrator skeleton with stubs to bind the generated d-model and platform mapping interfaces
	`CMakeLists.txt`	Build configuration with project name and dependencies (may require project-specific adjustments)
Generated Elsewhere (other techniques)		
	`platform_mapping.h`	Platform mapping implementation generated by mappingGen
	`platform_mapping_adapter.cpp`	Adapter code connecting the platform mapping to the orchestrator runtime
	`dmodel_interface.h/cpp`	D-model interface and channel adapter used by the orchestrator
	`d-model generated (e.g. PickPlace.c)`	Discrete controller implementation generated by d-modelGen
Scenario-Specific (manual or external)		
	`world_engine.cpp/h`	World engine defining environment state and update logic (scenario-dependent)
	`world_mapping.cpp/h`	World mapping that computes world-dependent sensors from world and platform state (scenario-dependent)

Table 4.6: File generation categories for Technique 5 (C++ target)

Figure 4.12 shows the detailed runtime architecture of the generated simulation system. The numbered arrows correspond to one control cycle. The same numbers are used in the flow

description below, and for each transition a concrete subexample is provided from either the Simple Arm Serial Manipulator or the Acrobot controlled example.

(0) `initialise()` – world state. The orchestrator triggers initialisation for the environment before the first control cycle so that the world state is defined.

Example 4.3: Simple Arm Serial Manipulator: 0. initialise()

In the full-simulation reference orchestrator, `initialise_runtime_state()` calls `world_engine->initialise();` before the engine thread starts, establishing the initial world state.

(1) `SendToDModel`/`ReceiveFromDModel`/`Wait(cycle)` – cycle control. The orchestrator publishes input events to the d-model, waits to respect the cycle period, and buffers outputs for the next boundary.

Example 4.4: Simple Arm Serial Manipulator: 1. cycle control

In `engine_thread()`, the orchestrator calls `dmodel_publish_input_message();` then advances time with `Evolve(timer)`, which invokes `Wait(SIM_CONFIG.platform_dt)` to enforce the control cycle.

(2) `registerRead(type, data, size)` – input events. The d-model requests the next event through the input channel interface.

Example 4.5: Simple Arm Serial Manipulator: 2. registerRead()

In `PickPlace.c`, the controller calls `registerRead(&input_type, &event, sizeof(event))`. For predicate events such as `INPUT_DETECT_OBJECT`, the event data is the scalar `EventData` record (e.g., `event.occurred`).

Example 4.6: Acrobot Controlled: 2. registerRead()

In `AcrobotSwingUpLQRModule_generated.c`, the controller calls `registerRead(&input_type, &sensor_data, sizeof(sensor_data))`. For `INPUT_SENSOR_UPDATE`, the event data is an `AcrobotSensorState` struct that bundles multiple sensor values into a single event.

(3) `registerWrite(OperationData)` – operations. The d-model publishes discrete operations that will later be applied by the engine.

Example 4.7: Simple Arm Serial Manipulator: 3. registerWrite()

In `PickPlace.c`, operations such as `PrePick` are sent with `registerWrite(&(OperationData){OUTPUT_PRE_PICK, ...});`.

(4) `tock(INPUT_DONE)` – cycle done. At each cycle boundary, the orchestrator signals the d-model that inputs for the next cycle are ready.

Example 4.8: Simple Arm Serial Manipulator: 4. tock(INPUT_DONE)

The engine thread calls `tock(INPUT_DONE)` after finishing a cycle, aligning the discrete boundary with the CyPhyCircus cycle semantics defined in [25].

(5) `ReceiveFromDModel` – `OperationData`. Buffered operations are received and applied to the platform mapping at the cycle boundary.

Example 4.9: Simple Arm Serial Manipulator: 5. ReceiveFromDModel

The output-processing thread calls `ReceiveFromDModel(&operation_type, &parameter, &timestamp)` and then `process_operation_event(...)` to update `p_mapping.operation_name` and seed `p_mapping.k`.

(6) read `p_mapping` – `mapping_state_t`. The platform engine reads actuator inputs from the platform mapping during physics integration.

Example 4.10: Simple Arm Serial Manipulator: 6. read p_mapping

In `platform1_engine.cpp`, the physics step reads torques from `p_mapping.SimpleArm.*.TorqueIn` and assigns them to the actuator vector `tau`.

(7) `update()` – platform state. The orchestrator advances the platform engine, updating the platform state used by subsequent mapping steps.

Example 4.11: Simple Arm Serial Manipulator: 7. update()

In `Evolve()`, the orchestrator calls `platform_engine->update();` and then uses `platform_engine->getPlatform().getState()` for the next sensor computation. In `platform1_engine.cpp`, the physics step also synchronises `SimpleArmSerial_BaseLink_BaseSensor_SignalIn` from `w_mapping.SimpleArm.BaseLink.ObjectBeamSensor.measurement` before evaluating the dynamics.

(8) Sensor-reading computation. The world mapping function `computeSensorReadings(world_state, platform_state, sensor_output)` computes sensor outputs using the current environment and platform states.

Example 4.12: Simple Arm Serial Manipulator: 8. computeSensorReadings()

The orchestrator invokes `platform_world_mapping->computeSensorReadings(world_engine->state(), platform_engine->getPlatform().getState(), sensors);` to produce the sensor snapshot for the cycle.

(9) `sensor_output` – `sensor_data_t` (world+platform). The world mapping writes the sensor structure that combines environment-derived readings with platform-internal sensors.

Example 4.13: Simple Arm Serial Manipulator: 9. sensor_output

In `world_mapping.cpp`, `computeSensorReadings` writes the object and goal beam measurements into `out.SimpleArm.BaseLink.*`. These values are environment-derived (computed using the world model and the current platform state and forward kinematics (FK)), so this example does not use any additional platform-internal sensor fields.

(10) Update the platform-mapping sensor data. The `updateFromSensors(sensor_output)` function mirrors the world-and-platform `sensor_data_t` snapshot into `p_mapping` for controller access.

Example 4.14: Simple Arm Serial Manipulator: 10. updateFromSensors()

In `orchestrator.cpp`, `EvolveWorldMapping()` calls `platform_mapping->updateFromSensors(sensors);` after `computeSensorReadings`, mirroring the beam sensor measurements into `p_mapping.SimpleArm.*` for the next cycle.

(11) `SendToDModel` – input events. The orchestrator derives input events from the platform mapping and publishes them to the d-model.

Example 4.15: Simple Arm Serial Manipulator: 11. SendToDModel

`dmodel_publish_input_message()` reads `p_mapping.SimpleArm.BaseLink.ObjectBeamSensor.measurement` and `GoalBeamSensor.measurement`, then computes `Event1` and `Event2` for the input channel.

CyPhyCircus Semantic Alignment

The generated orchestrator implements the CyPhyCircus operational semantics for p-model processes as specified in [25]. The main action of the p-model process follows the pattern:

$$\texttt{Init; } \mu\ \texttt{X . SendToDModel; (ReceiveFromDModel } \triangle \texttt{ Wait(cycle)); ResetTimer; Evolve } \triangle \texttt{ (timer} \geq \texttt{cycle} \times \texttt{tscale); X}$$

The generated C++ orchestrator preserves this semantic structure through a direct correspondence between the formal specification and implementation components. The `Init` action is realised by `initialise_runtime_state()`, which performs physics runtime setup, logging initialisation, and priming of the first input-channel message before the main loop begins. The `SendToDModel` action is realised by publishing a per-cycle input snapshot (e.g., `dmodel_publish_input_message()` in the reference orchestrators), which prepares inputs for the d-model via `registerRead`. This publication computes boolean event values from platform mapping sensor registers and stores them in a channel buffer for the duration of the current cycle. The d-model can then call `registerRead` to consume these buffered inputs exactly once per cycle.

The `ReceiveFromDModel` action is realised by the `ReceiveFromDModel()` function, which accepts outputs from the d-model via `registerWrite` events. The d-model calls `registerWrite` to send operation commands (such as `PrePick`, `PrePlace`, or `Return`) along with timing parameters. The output processing thread receives these commands and buffers them

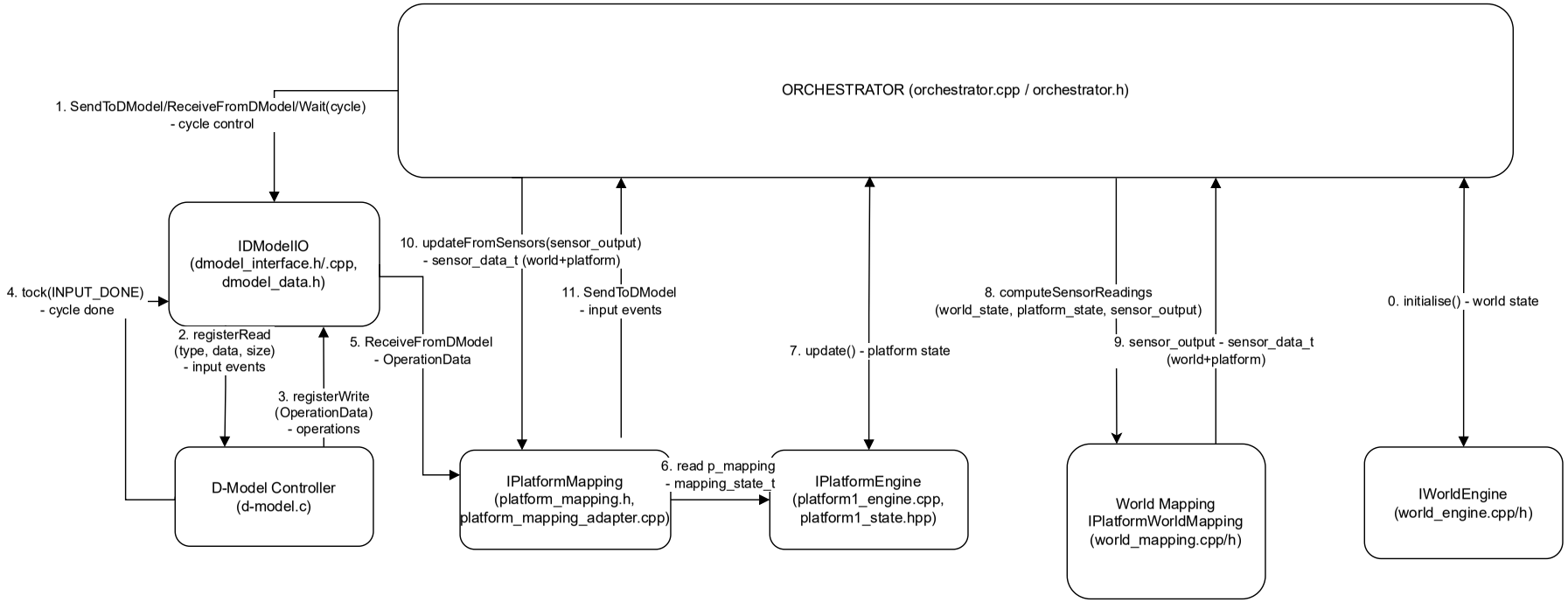


Figure 4.12: Runtime architecture showing CyPhyCircus semantic correspondence and numbered cycle transitions. The world mapping computes `sensor_output` from world and platform state; the platform mapping is updated via `updateFromSensors` and provides input events for `SendToDModel`.

for application at the cycle boundary, ensuring discrete effects align with control ticks. The interrupt operator ($\triangle$) in `(ReceiveFromDModel △ Wait(cycle))` is realised through the cycle boundary mechanism: the output thread monitors for the `cycle_boundary_flag` set by the engine thread when the cycle duration elapses, interrupting any pending receive operation.

The `Wait(cycle)` action is realised by allowing time to advance only within `Evolve` (which calls `Wait` for the physics timestep) until the cycle boundary condition holds. At the boundary, the engine signals cycle completion by making `INPUT_DONE` available to the d-model (via `tock(INPUT_DONE)`), and then prepares the next input snapshot. The `ResetTimer` action is implemented by `resetTimer()`, which sets the simulation timer back to zero at the end of each cycle in preparation for the next evolution phase.

The `Evolve` action is realised by the `Evolve()` function, which advances the continuous dynamics by one physics step. This function executes the world mapping (`computeSensorReadings`) to compute `sensor_output` from world and platform state, propagates those readings to the platform mapping via `updateFromSensors`, calls the platform engine's `update()` method to integrate dynamics equations, and continuously integrates the phase time parameter k while operations are active. The interrupt condition `(timer ≥ cycle × tscale)` is checked at the end of each `Evolve()` call, returning true when the accumulated time reaches the configured cycle duration.

The channel semantics are preserved through the `DModelIOAdapter` class, which implements the `IDModelIO` interface providing `registerRead`, `registerWrite`, and `tock` operations. The `registerRead` channel is called by the d-model to read inputs that were prepared by `SendToDModel`, while `registerWrite` is called by the d-model to send outputs that are received by `ReceiveFromDModel`. This bidirectional communication follows the CyPhyCircus channel semantics defined in [25]: inputs flow from p-model to d-model via `registerRead`, and outputs flow from d-model to p-model via `registerWrite`.

Figure 4.13 summarises the direct correspondence between the CyPhyCircus actions defined in [25] and their C++ implementations.

CyPhyCircus Action	C++ Implementation	Description
Init	`initialise_runtime_state()`	Physics setup, logging, initial message
SendToDModel	`dmodel_publish_input_message()`	Prepare inputs via registerRead
ReceiveFromDModel	`ReceiveFromDModel()`	Receive outputs via registerWrite
Wait(cycle)	`Wait(...)` + boundary check	Sleep and detect cycle boundary
ResetTimer	`resetTimer()`	Reset timer to 0
Evolve	`Evolve()`	Physics update with k integration
registerRead	`registerRead_impl()`	D-model reads buffered inputs
registerWrite	`registerWrite_impl()`	D-model sends operation commands

Figure 4.13: Correspondence between CyPhyCircus semantics and the generated C++ orchestrator implementation.

The timing model follows the CyPhyCircus semantics defined in [25], in which `SendToDModel`, `ReceiveFromDModel`, and `Wait(cycle)` take no simulation time, while time passes only during `Evolve`. The generated code realises this through the timer variable that accumulates only within the `Evolve()` function, with the cycle boundary triggered when `timer` $\geq$ `cycle` $\times$ `tscale`. The configuration structure `SIM_CONFIG` captures these timing parameters: `cycle` specifies the number of tock events per control cycle, `t_scale` specifies seconds per tock, and `platform_dt` specifies the physics integration timestep.

Code Generation Library

The Code Generation Library provides templates for generating code in each target language. Each solution method in the Solution Library has corresponding code templates that are instantiated with the specific parameters from the p-model+ and solution references. The templates are maintained in the `physmod-libs` repository.

Once the resolved `slnRef` list has been emitted, sourceCodeGen consumes it to generate `slnDF` implementations. Figure 4.14 shows how the T5 generator instantiates library templates and emits concrete `slnDF` descriptions that feed later code-generation stages.

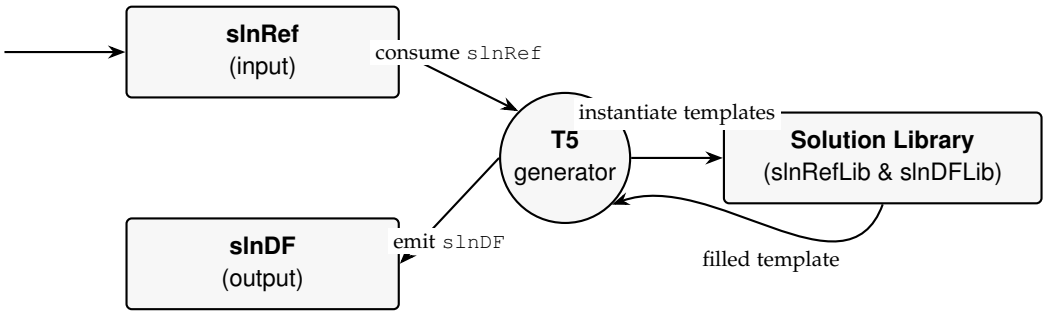


Figure 4.14: T5 flow: the generator consumes emitted `slnRef` files, instantiates solution templates from the shared library, and writes the resulting `slnDF` description.

For C++ generation, the library provides templates for state structures (defining the data types for generalised coordinates, velocities, and forces), computation functions (implementing the numerical algorithms for forward kinematics, dynamics, and integration), and interface implementations (realising the abstract interfaces for platform engine, mapping, and orchestration).

Example 4.16: Simple Arm Serial Manipulator: Template Excerpt

For the Simple Arm example, Table 4.5 includes a `CholeskyAlgorithm` step that computes M_{inv} from M_{mass}. Listing 4.15 shows an excerpt of the corresponding slnDF template instance produced by the SKO code generation library.

```
1  Solution temp {
2    state{
3      M_inv : mat(2,2) = ;
4      M_mass : mat(2,2);
5    }
6    functions{
7      function LDLT(matIn: mat()): mat() { }
8    }
9    computation{
10     M_inv = LDLT(M_mass);
11   }
12 }
```

Listing 4.15: Excerpt of the SKO `CholeskyAlgorithm` template as a slnDF solution

The library also includes explicit handling for platform mapping methods when mapping information is supplied. In `FULL_SIMULATION_MAPPING` mode, `PlatformMapping` and `SensorOutputMapping` solution methods are translated into mapping-aware templates that emit the `mapping_state_t/sensor_data_t` structures, the `updateFromSensors` adapter, and the concrete wiring between `p_mapping` fields and model-level inputs/outputs. This preserves the mapping semantics from `mapping.slnRef` in the generated C++ interfaces, so actuator inputs and sensor outputs are visible in the generated code rather than being left as hand-written stubs.

In addition to executable code templates, the library supports a proof-oriented use case. For SKO-based models, a formulation-specific method `SKO::proof` can be selected (via the proof-generation target in guidedChoice) to emit a proof system containing the SKO algebraic constraints as a `.sln` solution, which sourceCodeGen can then translate into an Isabelle theory (dataspace variables and equations). This provides a lightweight path from generated SKO equations to theorem-prover artefacts, complementing formulation-agnostic proof generation that only reflects the p-model state.

The template-based approach ensures that all generated code is traceable back to the original p-model and platform mapping, implementing the key principle of sourceCodeGen: maintaining semantic equivalence between the high-level models and the generated code. The transformation preserves variable names and types from the p-model+ specification, algorithm structure from the solution references produced by guidedChoice, operational semantics from the RoboSim framework, and physical constraints and equations from eqnComp.

4.1.4 Mapping Generation

This section describes the mapping-generation workflow used in this thesis. The simulation framework overview in Figure 3.1 presents Technique 2 (mappingGen) as a direct translation from `mapping.pm` to mapping source-code. In this thesis we implement this in two steps. First, mappingGen extracts a `mapping.pm` specification into a `mapping.slnRef` constraint file. Second, when mapping support is enabled, sourceCodeGen consumes `mapping.slnRef` and generates the mapping-layer integration artefacts (platform mapping adapter and d-model interface) used by the orchestrator. This two-step workflow produces the same mapping code as the direct translation in Figure 3.1. Figure 4.15 shows the pipeline used in this thesis.

A platform mapping connects the discrete controller interface to the continuous platform model by identifying which actuator input ports receive each element of the control vector and which sensor output ports populate the fields of the input events delivered to the d-model. Exporting this wiring as explicit slnRef constraints ensures that the generated adapter is consistent with the mapping.

The prototype mapping generator is invoked from the IDE and locates a `mapping.pm` file in the project. It parses the mapping model and emits an ordered list of `SolutionRef` blocks to `mapping.slnRef`.

For actuators, the generator exports at most one `SolutionRef` per `PlatformMapping`. The exported solution uses method `PlatformMapping`. It determines actuator destinations from the mapping equations, sizes a local control vector `u`, emits the initialisation `(u)[t == 0]==`

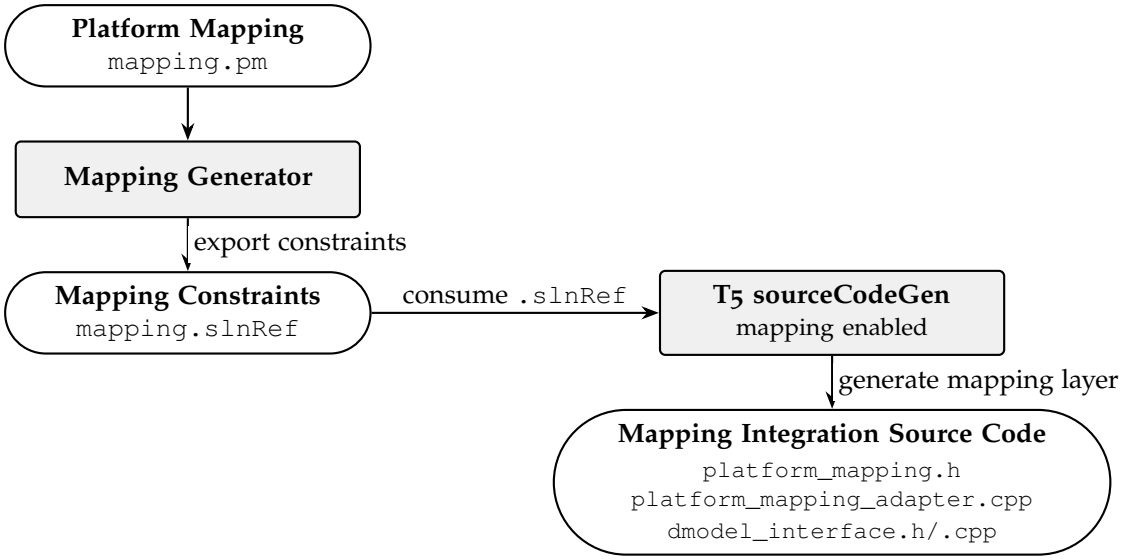


Figure 4.15: A refined view of the Technique 2 pipeline used in this thesis. The mapping generator exports `mapping.pm` into `mapping.slnRef`. In mapping-enabled mode, sourceCodeGen consumes `mapping.slnRef` and generates the mapping integration artefacts required by the orchestrator (platform mapping adapter and d-model interface).

`zeroVec(N)`, and adds one algebraic constraint per destination of the form `(u(i))[t == t]==p_mapping.....`

For sensors, it exports one `SolutionRef` per `InputEventMapping`. The exported solution uses method `SensorOutputMapping`. It emits constraints of the form `(p_mapping....)[t == t]==event.field`.

In both cases, p-model locations are detected syntactically (via `p_mapping.` or `::`) and normalised to the `p_mapping....` form in the emitted constraints.

Example 4.17 illustrates these rules for the Simple Arm case study, including the platform mapping diagram and the corresponding `mapping.pm` and `mapping.slnRef` excerpts.

Example 4.17: Simple Arm: Mapping generation

Figure 4.16 shows the platform mapping for the Simple Arm example. It specifies the actuator destinations and sensor bindings that mappingGen exports to `mapping.slnRef` for downstream code generation.

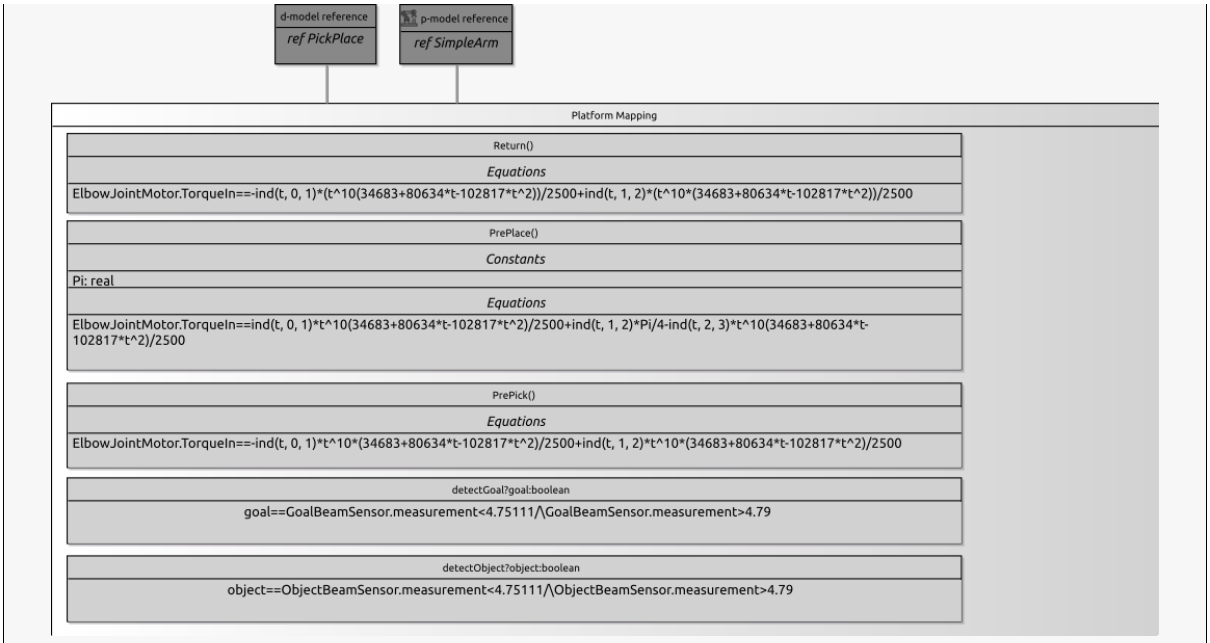


Figure 4.16: Simple Arm platform mapping that links controller events and operations to the Simple Arm sensors and actuators.

Listings 4.16 and 4.17 show a minimal worked example in terms of the printed equation strings that the generator processes. The first two equations are taken from an operation mapping and identify actuator destinations. The third equation is taken from an input event mapping and binds a sensor output to an event field.

```
SimpleArmControlled::BaseLink::ShoulderMotor.ControlIn == cmd.shoulderTorque;
p_mapping.SimpleArmControlled.BaseLink.WristMotor.ControlIn ==
    cmd.wristTorque;

su.shoulderAngle == SimpleArmControlled::BaseLink::ShoulderEncoder.AngleOut;
```

Listing 4.16: Worked example equation strings extracted from `mapping.pm`.

From these equations, the generator produces the `mapping.slnRef` excerpt in Listing 4.17. The actuator destinations determine the length of `u`, the control vector is initialised at time zero, and each element `u(i)` is constrained to the corresponding actuator input path. The sensor mapping becomes an algebraic constraint from the normalised `p_mapping` path to the event field.

```
SolutionRef {
   expression Local { name u type "vector(real,2)" }
   method PlatformMapping
   order 1
constraint " (u) [ t == 0] ==zeroVec(2)"
constraint " (u(0)) [ t == t]
    ==p_mapping.SimpleArmControlled.BaseLink.ShoulderMotor.ControlIn"
constraint " (u(1)) [ t == t]
    ==p_mapping.SimpleArmControlled.BaseLink.WristMotor.ControlIn"
}

SolutionRef {
   expression Local { name sensorUpdate type "real" }
   method SensorOutputMapping
   order 2
```

```
14  constraint "
        (p_mapping.SimpleArmControlled.BaseLink.ShoulderEncoder.AngleOut) [ t ==
        t] ==su.shoulderAngle"
15  }
```

Listing 4.17: Corresponding excerpt of the generated `mapping.slnRef`.

Example 4.18 shows how this mapping specification determines the mapping-related artefacts in the Technique 5 output architecture (Table 4.6).

Example 4.18: Simple Arm: mapping-enabled integration artefacts

In mapping-enabled mode, the pipeline fills the "Generated Elsewhere" artefacts from Table 4.6:

- `platform_mapping.h` – a nested `p_mapping` struct with fields for the actuator destinations and sensor locations referenced in `mapping.slnRef` (e.g. `p_mapping.SimpleArmControlled.BaseLink.ShoulderMotor.ControlIn`).
- `platform_mapping_adapter.cpp` – the adapter implementation used by the orchestrator, including the copy statements that mirror sensor snapshots into `p_mapping` each cycle.
- `dmodel_interface.h/cpp` – the C ABI used by the generated d-model to communicate with the orchestrator; the header defines `INPUT_*/OUTPUT_*` enums derived from the input events and operations in `mapping.pm` (e.g. `INPUT_DETECT_OBJECT`, `OUTPUT_PRE_PICK`).

The intermediate `mapping.slnRef` and unfolded `mapping.sln` are retained to make the generated mapping logic inspectable and debuggable.

Example 4.19: Simple Arm: remaining user stubs

After generating the mapping integration layer, the user still completes the partially generated and scenario-specific artefacts from Table 4.6:

- `world_engine.cpp/h` and `world_mapping.cpp/h` – implement the pick-and-place scenario world state and compute the object/goal sensor readings used by the d-model.
- `PickPlace.c` (from d-modelGen) – adapt the generated controller code to use `dmodel_interface.h` for input/output, following the `registerRead/registerWrite` pattern described in Section 4.1.3.

The generator is intentionally syntax directed and relies on simple string-level heuristics. It assumes that relevant mapping equations contain a single top-level equality and that a p-model location is recognisable by the presence of `p_mapping.` or `::`. It does not perform type checking, conflict detection, or symbolic manipulation of expressions. As a result, this prototype relies on two assumptions about how `mapping.pm` equations are written:

(A1) Sensor-mapping equations are written so that *exactly one* side of each relevant equality is recognisable as a p-model location. If both sides, or neither side, match the p-model-location heuristic, then the equation is ignored and no constraint is generated.

(A2) Operation-mapping equations avoid having both sides look like p-model locations. When both sides match the p-model-location heuristic, the prototype defaults to selecting the left-hand side as the actuator destination.

A more general implementation of mappingGen would replace these heuristics with a semantics-aware transformation that detects ambiguities and emits mapping code directly, rather than relying on string-level recognition.

4.1.5 Proof Generation

Figure 1.3 in Chapter 1 illustrates the consistency problem that motivates this thesis: the need to ensure that simulation and proof use the same underlying equations. In that figure, the arrow from the p-model to proof is annotated with "Isabelle/UTP", indicating the intended target for formal verification. The original RoboStar vision was to generate proof obligations from the CyPhyCircus model, which combines the d-model and p-model into a unified formal semantics expressed in Circus. From CyPhyCircus, one could then generate Isabelle/HOL theories that encode the hybrid system behaviour for verification using differential dynamic logic or related formalisms.

However, automated generation from CyPhyCircus to Isabelle is not yet implemented. The CyPhyCircus semantics and the associated translation infrastructure remain under development as part of ongoing RoboStar research. In the absence of this automation, establishing consistency between simulation and proof requires an alternative approach.

This thesis addresses this gap by implementing proof generation for the p-model portion of the RoboSim model. In this instance, the simulation framework developed in this thesis can generate Isabelle theories directly from the p-model via the same techniques used for simulation code generation. Figure 4.17 illustrates this approach: the p-model is processed by eqnComp to produce composed equations, then by guidedChoice to resolve solution methods, and finally by sourceCodeGen with the Isabelle target to produce formal theories.

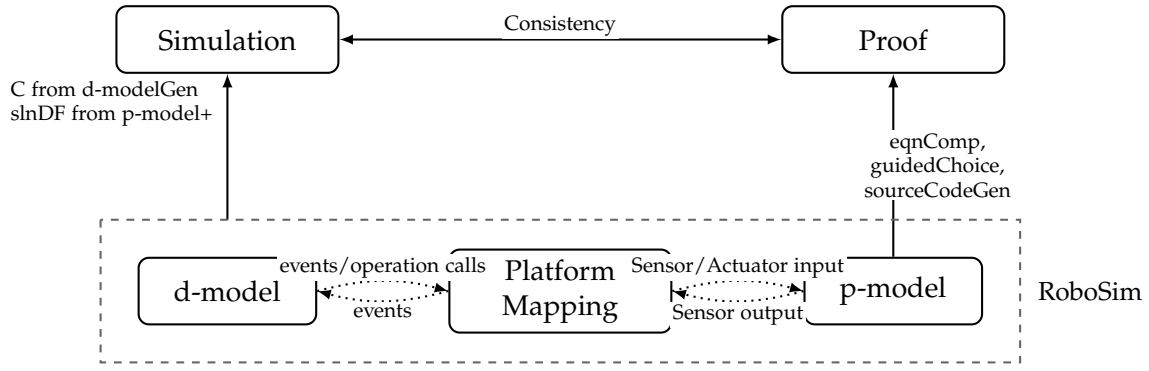


Figure 4.17: Proof generation for the p-model portion of the RoboSim model. The p-model is processed by the simulation framework techniques (eqnComp, guidedChoice, sourceCodeGen) to generate Isabelle theories, enabling consistency between simulation and proof for the physical model.

This approach provides partial but immediate support for consistency verification. The generated Isabelle theories encode the same equations and solution structures that are used in the generated C++ simulation code, ensuring that any properties proven about the p-model dynamics hold for the corresponding simulation behaviour.

Implementation in guidedChoice

The guidedChoice technique implements proof generation as a dedicated generator target (`PROOF_ON`) in the `guidedChoiceGenerator`. When this target is selected, Technique 4 executes its standard solution unfolding and resolution pipeline and then appends a proof solution that captures the equations and assumptions needed to construct a proof dataspace downstream.

Because the proof solution is derived from the *transformed* p-model, it reflects whatever information is present after unfolding the selected solution methods. In particular, if the modeller wants the proof dataspace to include auxiliary quantities introduced by optional methods (for example, the geometry variables introduced by visualisation, such as `T_geom` and per-link `Geom` records), those methods must be enabled in the solution selection prior to exporting the proof block. If desired, this can be staged: a modeller can first run guidedChoice to produce an enriched `slnRef` that includes the relevant method outputs (e.g., visualisation), and then re-run with `PROOF_ON` to export the final proof block over the enriched model.

Concretely, after all solutions have been resolved and their constraints populated, the generator increments the `order` of every existing p-model solution so that `order = 0` is reserved for proof generation, then prepends a new p-model `Solution` with `method = "proof"` and `order = 0`. It populates the proof solution constraints with all system-level equations (i.e., every `left == right` equation lifted into the system block, including actuator and sensor), rewriting them as algebraic (`[t==t]`) constraints for export, and appends the remaining per-solution constraints (notably `[t==0]` initial conditions and any unfolding-introduced mapping constraints). Finally, it adds every system-level variable (from the p-model, actuator, and sensor blocks) as an input of the proof solution.

Finally, the normal `genSlnRef` export path serialises this additional solution alongside the simulation solutions. Method names are namespaced by formulation during export, so for SKO models the proof entry appears as `method SKO::proof`.

Implementation in sourceCodeGen

The sourceCodeGen technique consumes the `.slnRef` file emitted by guidedChoice. Its entry point, `SolutionRefGenerator`, iterates over all `SolutionRef` blocks (ordered by `order` as established by Technique 4) and resolves each one by calling `SolutionLib.returnSolution(formulation, slnRef)`. The formulation is inferred from the `X::method` namespace (e.g., `SKO::proof`).

For proof generation, the `SKO::proof` entry is handled by the SKO library's `proof` template, which constructs an intermediate `Solution` named `proof_solution`. The implementation separates `[t==0]` constraints into initial conditions and `[t==t]` constraints into algebraic equations, normalises literals (sequence, vector, and matrix syntax), and flattens dotted identifiers so that they can be used as identifiers in downstream targets. It then builds a proof-oriented `state` block that reflects the full model state by collecting variables across all `SolutionRef` entries (from expressions, inputs, and constraint identifiers) and uses initial conditions to supply state initial values. The resulting `proof` block contains the algebraic constraints, while the `computation` block is `skip;` because the artefact is intended for reasoning about equations rather than execution.

When Isabelle is selected (`physmod.output.format=isabelle`), sourceCodeGen dispatches

to SolutionToIsabelleGenerator. In equations mode (physmod.isabelle.mode=equations), the generator emits an ISAVodes-style theory <name>_EQUATIONS.thy. This theory imports Hybrid-Verification.Hybrid_Verification. It generates a dataspace from the state block, lifts non-trivial initialisations into named initial-condition definitions, and translates each proof constraint into a named algebraic equation definition. Common constructs are rewritten into idiomatic Isabelle/ISAVodes syntax, for example derivative(theta) into theta' and vector indexing theta(0) into theta$0.

Example 4.20: Simple Arm Serial Manipulator: Proof-generation artefacts

Listing 4.18 shows the SKO::proof solution reference exported by guidedChoice when proof generation is enabled for the Simple Arm Serial Manipulator. The excerpt illustrates three classes of constraints that end up in the proof dataspace: (i) composed system-level dynamics equations from eqnComp (e.g., tau == ...); (ii) platform mapping constraints (e.g., tau(0) == p_mapping....); and (iii) auxiliary quantities introduced by optional methods, such as geometry transforms and per-link geometry descriptions (when visualisation is enabled). Listing 4.19 shows the corresponding beginning of the Isabelle theory produced by sourceCodeGen.

```
SolutionRef {
  expression Local { name proof_var type "int" }
  method SKO::proof
  order 0
  input Local { name theta type "vector(real,2)" }
  input Local { name tau type "vector(real,2)" }
  ...
  constraint " (tau(0)) [ t == t ] ==
     p_mapping.SimpleArmSerial.IntermediateLink.WristJoint.WristActuator.TorqueIn"
  constraint " (tau(1)) [ t == t ] ==
     p_mapping.SimpleArmSerial.BaseLink.ElbowJoint.ElbowActuator.TorqueIn"
  constraint " (BaseLink_BaseSensor_SignalIn) [ t == t] ==
     BaseLink_BaseSensor_MeasurementOut"
  constraint " (BaseLink_ElbowActuator_TorqueIn) [ t == t] ==
     BaseLink_ElbowActuator_TorqueOut"
  constraint " (V) [ t == t ] ==adj ( phi )*adj ( H )*derivative(theta)"
  constraint " (tau) [ t == t ] ==M_mass*derivative(derivative(theta))+C"
  constraint " (T_geom) [ t == t] ==<T_geom_1, T_geom_2, T_geom_3>"
  constraint " (geom_21 . geomType) [ t == 0] ==cylinder"
  constraint " (geom_21 . geomVal) [ t == 0] ==[| 0.25 , 4.0 |]"
  ...
  constraint " (tau) [ t == 0] ==zeroVec(2)"
  constraint " (dt) [ t == 0] ==0.01"
}
```

Listing 4.18: Excerpt of the generated SimpleArmSerial_WithProof.slnRef proof block.

```
section "Proof_solution Physics Equations"

theory Proof_solution_EQUATIONS
 imports
  "Hybrid-Verification.Hybrid_Verification"
begin

text "Geometry datatype (emitted only when Geom variables are
present)"
record Geom =
  geomType :: string
  geomVal :: "real list"
  meshUri :: string
  meshScale :: "real list"
```

```
dataspace proof_solution_system =
  variables
    theta :: "real ^2"
    tau :: "real ^2"
    theta' :: "real ^2"
    theta'' :: "real ^2"
    T_geom :: "(real mat[4, 4]) list"
    geom_21 :: "Geom"
    ...

context proof_solution_system
begin

definition n_init where
  "n_init ≡ (N = 2)ᵉ"
definition tau_eq where
  "tau_eq ≡ (tau = M_mass *v theta'' + C)ᵉ"
```

Listing 4.19: Excerpt of the generated `Proof_solution_EQUATIONS.thy` theory (dataspace and selected constraints).

4.2 Validation of Simulation Framework

This section validates that the tool implementation correctly realises the theoretical framework presented in Chapter 3. The validation consists of three parts: demonstrating the correspondence between the pseudocode algorithms and their implementations, presenting comprehensive test coverage for each technique, and comparing the forward dynamics outputs against established simulators. The implementation correspondences were discussed in Section 4.1 and Appendix A.3.

4.2.1 Framework Coverage Tests

The tool implementation has been thoroughly tested to validate that it correctly implements the algorithms presented in Chapter 3. The testing strategy consists of unit tests for individual algorithm components, integration tests for complete workflows, and coverage tests for different robot topologies and formulations.

Tests are organised to mirror the structure of the algorithms, with each high-level function from Chapter 3 having corresponding test cases. The test suite follows a systematic approach to validate both individual components and their integration. The test suite is designed to satisfy the following coverage criteria.

C1 *Unit test coverage requires that every algorithm defined in Chapter 3 has at least one corresponding test method exercising its core functionality, covering normal operation, boundary conditions, and error handling.*

C2 *Integration test coverage requires that end-to-end tests verify the complete pipeline produces correct output by exercising multiple algorithms in sequence and validating the generated artefacts.*

C3 *Formulation coverage requires that tests cover multiple formulations (SKO and Featherstone) to*

ensure the framework handles different dynamics approaches correctly.

C4 *Topology coverage requires that tests cover the supported robot topologies, including serial chains, tree topologies (branching), and closed chains (kinematic loops).*

C5 *P-model coverage requires that the test suite exercise the PhysMod p-model metamodel components to ensure the generator pipeline robustly handles different model structures. The original metamodel testing in [23] did not systematically cover all metamodel components as we do here. The metamodel coverage suite validates and exercises the p-model element types used throughout the pipeline, and generator-level metamodel coverage tests ensure that eqnComp can generate valid p-model+ artefacts from these structurally varied inputs. Table A.19 in Appendix A.6 maps p-model metamodel components to covering metamodel coverage tests.*

eqnComp Coverage

The eqnComp (Equation Composition) generator is validated through a combination of unit tests and integration tests. Unit tests target individual algorithm components such as validation, reference localisation, and equation generation helpers, while integration tests verify the complete pipeline from p-model input to p-model+ output. Table A.10 in Appendix A.6.1 summarises the test coverage for eqnComp, showing the number of tests and which coverage criteria are satisfied.

The unit tests are organised to mirror the algorithm structure from Chapter 3. `EqnCompMainTest` exercises the main entry point on a minimal SKO model. `CalculateFormulationsTest` covers Step 1 with nested test classes for `findDynamicsFormulation` (S1a), `findSensorFormulations` (S1b), and `findActuatorFormulations` (S1c). `LocaliseRefsTest` covers Step 2 with nested classes testing reference-to-local conversion for joints, links, sensors, actuators, and bodies. `ValidateTest` covers Step 3 with tests for valid models, missing pose variables, missing inertia definitions, and error handling across SKO and Featherstone. `CalculateTopologyTest` covers Step 4 with tests for serial chain, tree, and closed-chain topologies. `AssignNumberingTest` covers Step 5 with nested classes for `orderPmodel` (S5a), `getBaseLink` (S5b), and `getChildLinks` (S5c), along with tests for different topologies including serial chains, tree (branching) structures, and closed chains (kinematic loops). `CalculateEquationsTest` covers Step 6 with nested classes for `generatePhiEquations`, `createInertiaVariable`, and `createGeometryVariable`.

The integration tests verify that the complete eqnComp pipeline produces valid p-model+ output. These tests exercise Steps 1 through 6 in sequence: formulation detection (`calculateFormulations`), topology analysis (`calculateTopology`), numbering assignment (`assignNumbering`), and equation generation (`doGenerateEqnComp`). The generated output is validated using the PhysMod validator to ensure syntactic and semantic correctness. Integration tests cover serial chains, tree topologies, and closed chains across SKO and Featherstone formulations, including library-referenced components and controlled actuators. A complete algorithm-to-test traceability matrix is provided in Appendix A.6.

The test cases partition the input space for each algorithm to ensure sufficient coverage. The input to the `getFormulations` algorithm (S1) is a p-model containing links, joints, sensors, and actuators. The algorithm determines the dynamics formulation by inspecting joint definitions, and independently detects sensor/actuator formulations. For dynamics formulation detection, the algorithm examines joints which may be either locally defined or referenced from a library. Local joints are inspected for characteristic variables (`H` for SKO, `s` for Featherstone),

while reference joints are identified by their source package. Testing both definition styles for each formulation (`testSKOLocalJoint`/`testSKOReferenceJoint` for SKO, `testFeatherstoneLocalJoint`/`testFeatherstoneReferenceJoint` for Featherstone) ensures both detection code paths are exercised. While additional dynamics formulations beyond SKO and Featherstone are supported, testing with at least two distinct formulations demonstrates that the detection algorithm is not hardcoded for a single formulation and can correctly discriminate between different characteristic variables. Since this thesis considers only joint-based dynamics formulation detection (rather than alternatives such as link-level equation analysis), testing both formulations across both definition styles provides sufficient coverage of the dynamics detection logic. The test `testMixedFormulations` verifies the rejection path when joints indicate inconsistent formulations. For sensor/actuator formulation detection, the algorithm distinguishes "trivial" formulations (matching the library definitions) from "unknown" formulations. The tests `testTrivialSensor` and `testTrivialActuator` verify trivial detection for locally-defined components, while `testTrivialSensorLibraryImport` and `testTrivialActuatorLibraryImport` verify detection when components are imported from the library via reference. The tests `testUnknownSensor` and `testUnknownActuator` verify that non-matching definitions are correctly classified as unknown. The boundary cases of models with no joints (`testNoJoints`) and empty component maps (`testEmptyMaps`) complete the coverage by testing the algorithm's behaviour when optional inputs are absent.

The input to the `localiseRefs` algorithm (S2) is a p-model potentially containing reference components of five types: joints, links, sensors, actuators, and bodies. The algorithm iterates over each component type and converts references to local definitions by copying the name, pose, and definition. The key input dimensions are: the component type being converted and the composition of reference versus local components in the model. For component type coverage, the tests `testConvertRefJoints`, `testConvertRefLinks`, `testConvertRefSensors`, `testConvertRefActuators`, and `testConvertRefBodies` each exercise a specific component type, verifying that the conversion function correctly identifies reference components, creates local copies with preserved names and poses, and replaces the references in the model. For composition coverage, `testConvertReferenceJointsToLocalWithNoReferenceJoints` verifies that conversion is a no-op when no reference components exist, and `testConvertMixedJoints` verifies selective conversion in models containing both reference and local components. The helper function `createLocalJoint` (S2b) is directly tested by `testCreateLocalJoint`, which verifies that the name, pose, flexible connection, and definition are correctly deep-copied from the reference joint rather than shared by reference. This direct test complements the indirect coverage through `testConvertRefJoints` by explicitly verifying the deep-copy semantics. While the conversion logic follows a uniform pattern across component types, testing all five types ensures that any type-specific differences in the metamodel structure (such as the different containment hierarchies for sensors inside links versus actuators inside joints) are correctly handled.

The input to the `validate` algorithm (S3) is a formulation type and a p-model. The validator checks multiple independent preconditions: that all links have pose information, that all links have inertia definitions (for SKO), and that required variables exist. The tests partition these preconditions: `testValidPModel` establishes the baseline where all preconditions hold, while `testValidFeatherstoneModel` confirms the Featherstone formulation validates with the same structural checks in place. The tests `testMissingPose` and `testMissingInertia` each violate exactly one precondition. This isolation ensures that each validation check is independently exercised. The test `testEmptyPModel` covers defensive programming paths that guard against malformed inputs. The test `testMixedValidity` verifies that partial validity (some components valid, others invalid) is correctly detected, which tests the

iteration logic. Finally, testNonSKO verifies that non-SKO models fail SKO-specific validation, confirming the formulation-specific nature of the checks.

The input to the orderPmodel algorithm (S5a) is a p-model with links that have been assigned numeric identifiers by the assignNumbering algorithm. The algorithm's correctness depends on producing a correctly ordered sequence regardless of the input order. Testing with test-OrderedLinksBasic verifies normal operation with two links defined in reverse order. The test testOrderedLinksFourLinks extends this to four links with scrambled ordering, verifying that the algorithm scales correctly. The test testOrderedLinksSingleLink covers the edge case of a single-link model. Since the ordering algorithm is deterministic and the upstream assignNumbering always produces valid sequential numbering, these tests provide sufficient coverage of the input space.

The EqnCompUtils helper functions handle type conversion between the metamodel's numeric expression types. The function toFloatExp must handle two input types (FloatExp and IntegerExp), and testing both (testToFloatExpWithFloatExp, testToFloatExp-WithIntegerExp) provides complete input type coverage. Similarly, getNumericValue is tested with both types. The tests testHelperFunctionErrorHandling and testClass-CastExceptionFix verify that the functions correctly reject invalid inputs and handle the specific type mismatch scenario that motivated their introduction. Testing with multiple values of each type (testToFloatExpWithFloats, testGetNumericValueMixed) provides additional confidence that the conversion logic is correct across the value domain.

guidedChoice Coverage

The guidedChoice generator is validated through unit tests that exercise the sidecar initialisation and variable-resolution utilities and through integration tests that run the full five-step pipeline across formulations and generator modes. Table A.13 in Appendix A.6.2 summarises the coverage for guidedChoice, and Appendix A.6.2 provides full traceability.

The unit tests are organised to mirror the algorithm structure from Chapter 3. GuidedChoice UtilsTest provides broad unit coverage of sidecar initialisation and variable-resolution utilities, including indexed variables, submatrix access, relation traversal, and staged propagation. HeadlessInitializationTest and SimpleHeadlessTest cover the headless initialisation and printing services used when running the generator outside of the graphical tooling. AlgebraicConstraintsTest targets the algebraic-constraint processing utilities (constraint normalisation, simplification, and error paths), while HeadlessPrintingServicesTest checks that expressions and constraints can be emitted deterministically without UI dependencies. SimpleArmSerialGuidedChoiceTest exercises a representative serial-chain instance end-to-end at the unit level, grounding the helper tests in a realistic model.

The input to the initialiseSidecar step (S1) is the p-model+ variable environment (constants, locals, inputs/outputs, and nested devices) and an initially empty sidecar. The tests testInitialiseSidecar and testInitialiseSidecarSubExpressions partition this input space by variable kind and expression structure, ensuring that initial values are recorded for constants and that unresolved variables are initialised consistently, including indexed variables (e.g., a[0]) and structured expressions (e.g., submatrix(M)(...)). This validates both the baseline initialisation path and the subexpression bookkeeping required for later staged resolution.

The input to the staged resolution step resolveSolutionFirstPass (S3) is a partially-

initialised sidecar, a set of correspondence relations, and a p-model+ containing equations and constraints that define values for unresolved variables. The unit tests cover direct versus indirect relation traversal (`testTryDirectRelationLeftSide/testTryDirectRelationRightSide/testTryDirectRelationNoRelation`) and ensure that variables can be evaluated through correspondence chains (`testEvaluateXJ1_ThroughRelation`). Constraint evaluation is exercised on both scalar and structured RHS expressions (`testEvaluateVariableWRTConstraintsUpdatesSidecar`, `testEvaluateVariableWRTConstraintsSubmatrix`) and on mixed resolved/unresolved inputs (`testEvaluateVariableWRTConstraintsUpdatesSidecarOnMixedRHS`), ensuring the resolution logic correctly updates sidecar state while preserving dependency information when full resolution is not yet possible.

The ordering step `reorderSolutions` (S4) must produce a well-founded evaluation order for the resolved solution blocks, including proof and visualisation outputs. This is validated primarily through integration tests which assert ordering invariants (e.g., that prerequisite blocks appear before dependent blocks) and detect regressions in the topological ordering logic across the supported generator modes. Finally, the complete `resolveSolution` step (S5) is validated end-to-end by the integration suite, which checks that the exported slnRef artefacts match the expected structure for each mode and formulation.

The integration tests verify that the complete guidedChoice pipeline produces valid slnRef output from eqnComp-generated p-model+ inputs. These tests exercise Steps S1–S5 in sequence: sidecar initialisation and null-datatype seeding (S1), formulation selection (S2), staged solution resolution with initial-condition synthesis and algebraic constraints (S3/S3a/S3b), solution reordering (S4), and final solution export (S5). The integration suite partitions the input space along four primary axes: formulation (SKO and Featherstone), topology (serial and closed chain), generator mode (base, mapping, proof, and visualisation), and library usage (local versus referenced components). This ensures both normal and edge-case code paths are exercised, including proof block ordering and visualisation-specific outputs. A complete algorithm-to-test traceability matrix and integration test catalogue are provided in Appendix A.6.2.

Metamodel coverage for guidedChoice is provided by the shared PhysMod p-model metamodel test suite (Appendix A.6.4), validated by `MetamodelCoverageValidationTest`, and by eqnComp metamodel coverage generator tests (`MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`), which ensure that downstream stages operate on well-formed and structurally varied p-model+ inputs.

sourceCodeGen Coverage

The sourceCodeGen generator is validated through unit tests for SolutionDSL-to-C++ translation and integration tests for the full two-stage pipeline (slnRef to slnDF to target code) across C++, LaTeX, Isabelle, and visualisation targets. Table A.16 in Appendix A.6.3 summarises the coverage for sourceCodeGen, and Appendix A.6.3 provides full traceability.

The unit tests in `SolutionToCppGeneratorTest` provide data-driven coverage of the slnDF-to-target translation (S2), including datatypes, functions, procedures, statements, and expression forms used in the generated engines. The integration tests exercise both stages of the pipeline: stage 1 translates slnRef into slnDF (S1), and stage 2 generates target artefacts (C++, LaTeX, Isabelle, and visualisation code) from the resulting slnDF (S2). The integration suite partitions the input space across target type, generator mode (standalone versus full simulation,

with and without visualisation), formulation (SKO and Featherstone), and topology (including a four-bar closed chain), ensuring that both generic code-generation paths and target-specific emission logic are exercised. The integration test catalogue and algorithm traceability matrix are provided in Appendix A.6.3.

The input to the stage 1 translation (S1) is an slnRef specification produced by guidedChoice; the translation instantiates solution-library templates and produces a normalised slnDF representation. The input to stage 2 (S2) is the slnDF, which contains concrete datatypes, variable declarations, and executable procedure bodies. The tests cover both minimal and feature-rich slnRef/slnDF inputs: standalone systems, full simulation systems, visualisation-enabled systems (with geometry variables), mapping-augmented systems, and proof-oriented systems.

The input to the `slnRefToSlnTranslation` step (S1) is partitioned by generator mode (standalone versus full simulation), feature flags (visualisation, mapping, proofs), and the originating formulation/topology of the upstream slnRef. The integration tests exercise these partitions by translating representative slnRef artefacts produced from SKO and Featherstone robot examples and by including both local and library-referenced inputs.

The `slnDFToTargetTranslation` step (S2) is partitioned by target backend (C++, LaTeX, Isabelle, and visualisation artefacts) and by the Solution DSL constructs used in the generated slnDF (datatypes, procedure bodies, statements, and expression forms). The unit suite `SolutionToCppGeneratorTest` provides fine-grained construct coverage for the C++ backend, while integration tests validate end-to-end target generation for LaTeX and Isabelle outputs, including proof-annotated artefacts. Together these tests ensure that the target generators are not specialised to a single robot, formulation, or output mode and that translation remains correct across the supported output languages.

Metamodel coverage for sourceCodeGen is provided by the shared PhysMod p-model metamodel test suite (Appendix A.6.4), validated by `MetamodelCoverageValidationTest` and exercised through `MetamodelCoverageGeneratorTest`/`MetamodelCoverageSKOGeneratorTest`, and by the unit and integration tests that exercise the Solution DSL constructs required for code generation.

The test suite comprises 126 eqnComp test cases, 132 guidedChoice test cases, and 38 sourceCodeGen test cases.[2] Test data includes reference p-models for each robot topology, expected p-model+ outputs with equations, slnRef and slnDF artefacts, and generated target-language outputs (C++, LaTeX, Isabelle, visualisation).

4.2.2 Benchmarking

To validate the generated forward dynamics implementation, we benchmark the Generated SKO physics engine against six established physics engines on the standard 2-DOF Acrobot. The comparison uses identical model parameters (link lengths, masses, and inertias), applies gravity and viscous damping, and integrates over a 5-second horizon using Euler integration with $dt = 0.001\,\text{s}$. Initial conditions set both joints to 1.0 rad with zero initial velocities. Figure 4.18 shows the trajectory comparison across all seven physics engines. All engines produce identical trajectories, confirming that the generated forward dynamics is consistent with established simulators. The trajectory error relative to Generated SKO is effectively zero for all engines, demonstrating exact agreement in forward dynamics computation when using

[2] https://github.com/UoY-RoboStar/robotool-meta-physmod/tree/main/physmod-testdata

consistent model parameters and coordinate conventions.

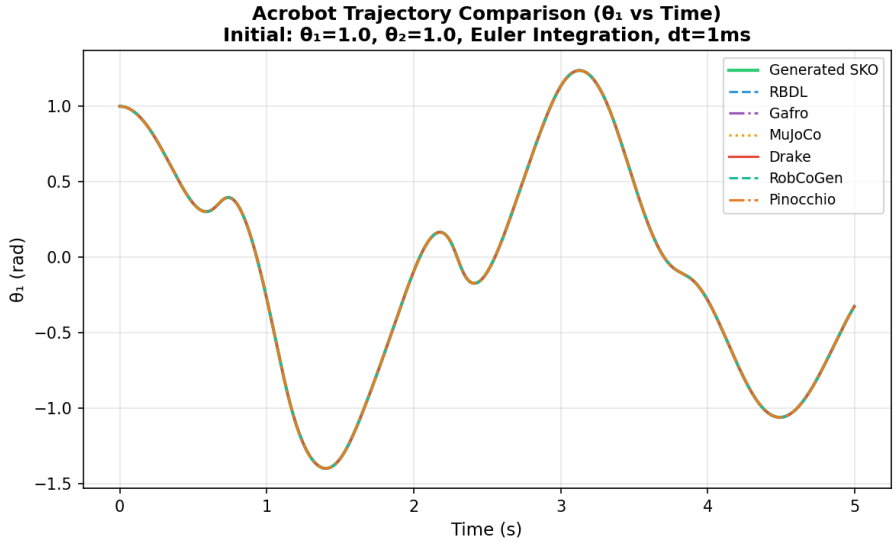


Figure 4.18: Trajectory comparison for the 2-DOF Acrobot across seven physics engines showing the shoulder angle θ_1 over time. All engines produce identical trajectories, validating the correctness of the generated dynamics.

Figure 4.19 compares the average forward dynamics computation time across the seven engines. RobCoGen achieves the fastest computation at approximately 0.1 µs per step, owing to its highly optimised code generation approach. RBDL follows at approximately 0.3 µs, then MuJoCo at 1.0 µs, the Generated SKO engine at 2.0 µs, and Gafro, Drake, and Pinocchio at 2.3–3.0 µs. The Generated SKO engine is competitive with established physics engines, achieving computation times within the same order of magnitude as MuJoCo, though RobCoGen demonstrates the performance advantage of specialised code generation.

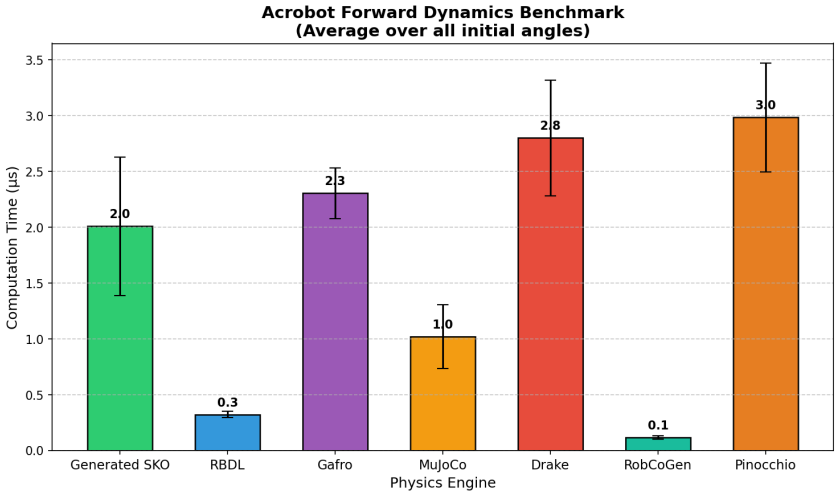


Figure 4.19: Average forward dynamics computation time for the Acrobot across seven physics engines. Error bars show standard deviation across multiple runs.

4.3 Case Studies

4.3.1 Simple Arm Pick-and-Place

This case study[3] presents the final piece of the full simulation integration of the RoboSim model for the Simple Arm example. Previous sections of the thesis have shown how the p-model and platform mapping can be input into the simulation framework to produce source-code consistent with their respective models. The last step is to generate code from the d-model and manually adapt the generated code to read and write from the interfaces generated in previous steps.

Figure 4.20 shows the pick-and-place d-model. The controller is a state machine that reacts to `detectObject` and `detectGoal` events, issuing operations such as `PrePick`, `PrePlace`, and `Return` to drive the arm through the task.

The platform mapping in Figure 4.16 connects these discrete operations and events to the continuous model. Sensor readings are mapped to the controller inputs, while the operation commands are mapped to torque profiles for the joint actuators derived from the requirements analysis in Appendix A.1.1.1.

The Simple Arm case study uses the standard SKO serial-chain formulation for equation composition and selects solutions that compute actuator torques, forward dynamics, and kinematic transforms for visualisation. The full-simulation visualisation mapping configuration generates the platform engine, mapping interfaces, and runtime scaffolding; the only manual edits for this case study are the d-model lines shown below. Table 4.7 presents the execution times for each transformation stage.

Generator	Time (s)
T3 (eqnComp)	12.84
T4 (guidedChoice)	38.52
T5 (sourceCodeGen)	0.21
Total	51.57

Table 4.7: Execution times for processing the Simple Arm model through each generator stage (5-run average).

The RoboSim-generated controller (`PickPlace.c`) is adapted to use the generated d-model interface by inserting the `registerRead`/`registerWrite` bridge calls. Listing 4.20 shows the exact lines added to the generated file; all other controller logic remains unchanged.

The d-model interface stub is completed by defining the input and output enumerations that correspond to the Simple Arm events and operations. Listing 4.21 lists the exact lines added to `dmodel_interface.h`.

```
1 // MANUAL: event and operation identifiers for Simple Arm
2 enum {
3     INPUT_DETECT_OBJECT,
4     INPUT_DETECT_GOAL,
5     INPUT_DONE,
6     INPUT_TERMINATE
7 };
8
```

[3] https://github.com/UoY-RoboStar/robotool-meta-physmod/tree/main/Examples/CaseStudies/SimpleArm/Model

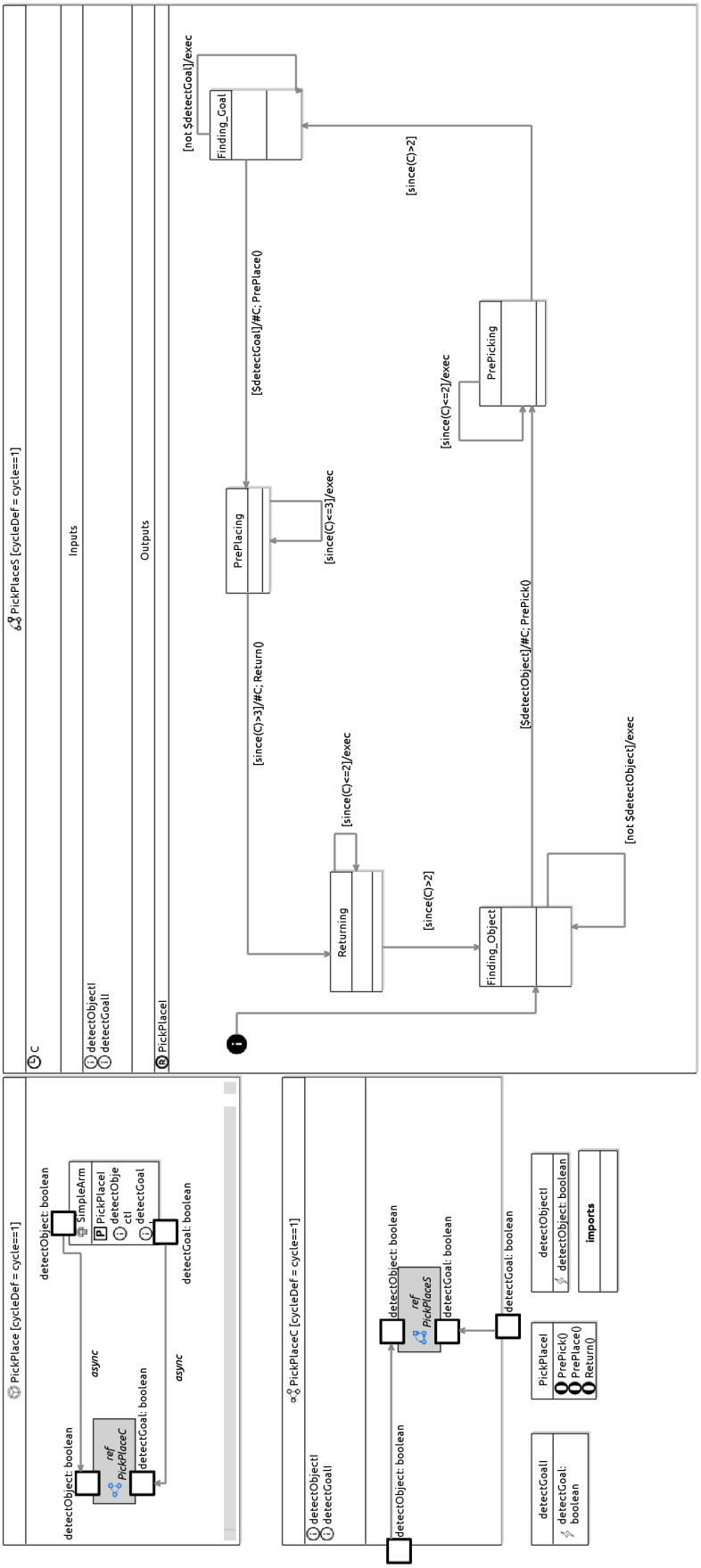


Figure 4.20: Pick-and-place controller d-model for the Simple Arm case study.

```
// MANUAL: include the d-model interface API
#include "dmodel_interface.h"

// MANUAL: replace stdin polling with registerRead-based events
EventData event;
int input_type = 0;
while (registerRead(&input_type, &event, sizeof(event))) {
   if (input_type == INPUT_DETECT_OBJECT) {
      detectObject = event.occurred;
   } else if (input_type == INPUT_DETECT_GOAL) {
      detectGoal = event.occurred;
   } else if (input_type == INPUT_DONE) {
      break;
   }
}

// MANUAL: replace stdout writes with registerWrite operations
if (out.type == M_PickPlace_output_PrePick) {
   registerWrite(&(OperationData){OUTPUT_PRE_PICK, /* ... */});
} else if (out.type == M_PickPlace_output_PrePlace) {
   registerWrite(&(OperationData){OUTPUT_PRE_PLACE, /* ... */});
} else if (out.type == M_PickPlace_output_Return) {
   registerWrite(&(OperationData){OUTPUT_RETURN, /* ... */});
}
```

Listing 4.20: Simple Arm: manual d-model bridge lines added to `PickPlace.c`

```
enum {
   OUTPUT_PRE_PICK,
   OUTPUT_PRE_PLACE,
   OUTPUT_RETURN,
   OUTPUT_DONE
};

// MANUAL: ABI used by the generated controller
bool registerRead(int* type, void* data, size_t size);
void registerWrite(const OperationData* op);
void tock(int type);
```

Listing 4.21: Simple Arm: manual lines added to `dmodel_interface.h`

4.3.2 Acrobot

The Acrobot[4] is a planar two-link robotic arm operating in the vertical plane against gravity, with an actuator at the elbow joint but no actuator at the shoulder joint [127]. The name derives from its resemblance to a gymnast on a parallel bar, whom controls motion predominantly through effort at the waist rather than the wrists. This case study demonstrates the application of the simulation framework to this robotic system.

In this section, the standalone simulation of the acrobot under gravity is presented for the SKO formulation, with Featherstone's spatial vector formulation and Conformal Geometric Algebra (CGA) demonstrated as additional supported dynamics formulations. Following this the implementation of the full simulation is presented for a control task. The control task studied for this case study is the swing-up and balance problem, in which the system must use the elbow torque to move from an initial downward-hanging configuration to a vertical upright configuration, and then maintain balance at the unstable equilibrium point [127]. An Isabelle proof template is also generated, from which the properties of underactuation and controllability about the upright equilibrium are proven.

[4] https://github.com/UoY-RoboStar/robotool-meta-physmod/tree/main/Examples/CaseStudies/Acrobot

Standalone Acrobot under Different Formulations

One of the main benefits of the simulation framework introduced in Chapter 3 is that it allows p-models to be written in a formulation-agnostic manner.

The physical description of the Acrobot is specified via a RoboSim p-model that captures the kinematic and dynamic properties of the mechanism. As shown in Figure 4.21, the model declares a platform component `AcrobotControlled` with three links (`BaseLink`, `Link1`, and `Link2`) connected by two revolute joints (`ShoulderJoint` and `ElbowJoint`), both instantiated as `Revolute_x`. The model contains three rigid bodies: a small `BaseBody` (box geometry) and two link bodies (`Link1Body` and `Link2Body`, cylinder geometries), each with declared mass and inertia. Joint encoders (`ShoulderEncoder` and `ElbowEncoder`) are instantiated as `JointEncoder` sensors and provide angle and velocity outputs for feedback. A single actuator (`ElbowMotor : ControlledMotor`) is attached at `ElbowJoint`; the control input matrix B_{ctrl} is defined so that the scalar controller output applies torque at the elbow joint.

The joint definition shown in Figure 4.21 can be swapped out with alternative formulations, allowing the same p-model structure to generate different dynamics equations. Three formulations are demonstrated: Spatial Kernel Operators (SKO), Featherstone's spatial vector formulation, and Conformal Geometric Algebra (CGA).

The SKO formulation uses Jain's spatial operator algebra to represent the kinematic and dynamic relationships between bodies. The revolute joint is defined using the motion subspace matrix H and the joint transformation $X_J(\theta)$, and is the same form as the SKO joint definition introduced in Chapter 3 (Section 3.2.1). Running the eqnComp generator with this joint definition produces the SKO equations used by the later pipeline stages.

Featherstone's spatial vector formulation uses 6-dimensional vectors to represent velocities and forces, with spatial transforms for coordinate changes between frames [35]. The revolute joint is defined using a motion subspace vector S and the corresponding joint transform $X_T(q)$, as shown in Figure 4.22. The eqnComp generator produces Featherstone-based dynamics equations which are then used for forward dynamics.

The Conformal Geometric Algebra (CGA) formulation [90] defines the revolute joint using a bivector generator and a rotor construction, as shown in Figure 4.23. The eqnComp generator produces CGA-based dynamics equations used in the standalone CGA engine.

Figure 4.24 compares the standalone trajectories produced by the SKO, Featherstone, and CGA formulations for the same initial conditions. Featherstone uses a base-to-tip indexing convention, so the joint order is swapped when plotting to align with the SKO/CGA ordering.

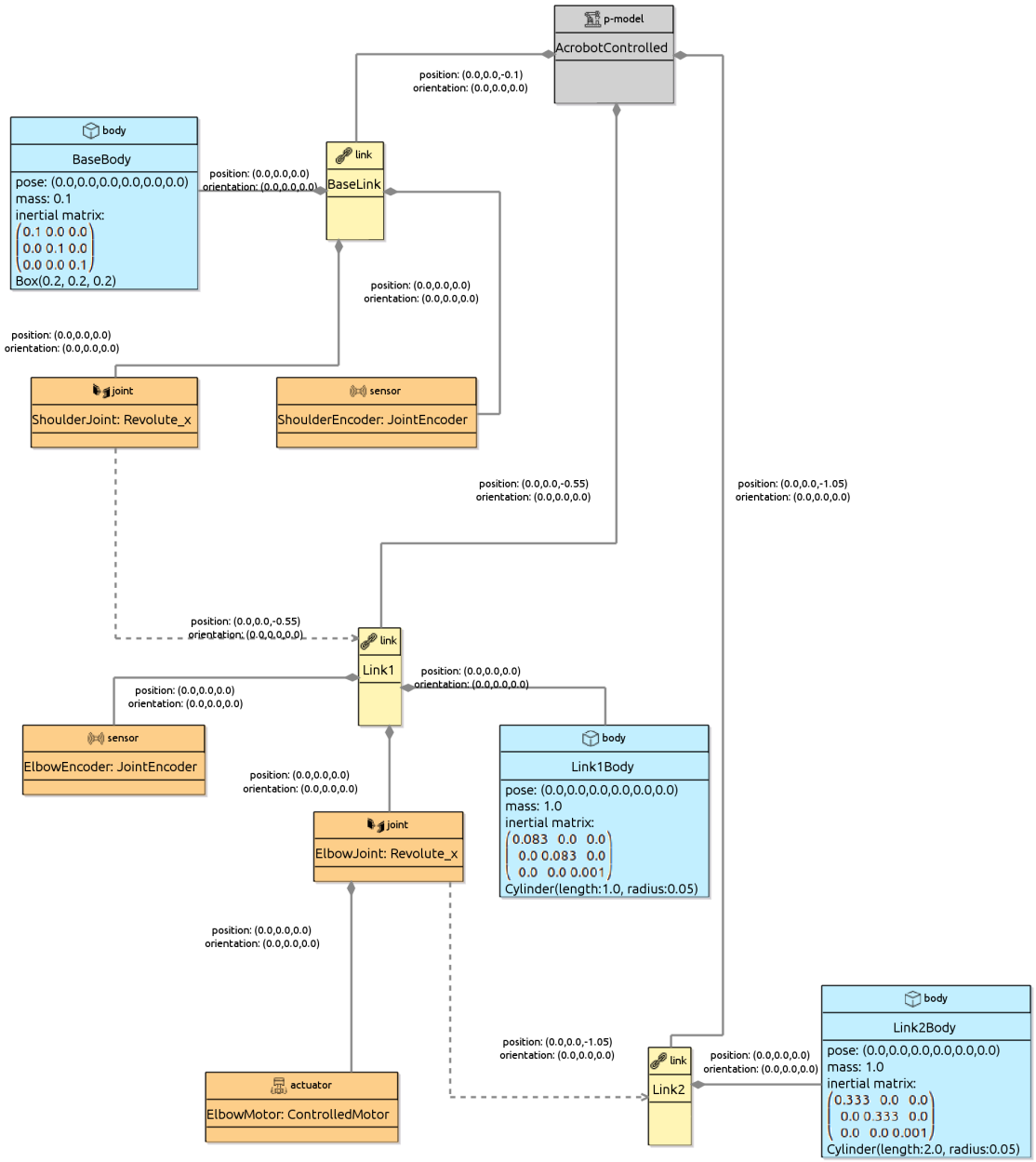


Figure 4.21: The p-model specification for the Acrobot (`AcrobotControlled`), showing links `BaseLink`, `Link1`, and `Link2`; revolute joints `ShoulderJoint` and `ElbowJoint` (both `Revolute_x`); joint encoders `ShoulderEncoder` and `ElbowEncoder` (`JointEncoder`); and the single actuator `ElbowMotor` (`ControlledMotor`) attached at the elbow joint.

joint definition

RevoluteJoint_Y

Constants

S: vector(real,6) = (|0, 1, 0, 0, 0, 0|)

Equations

X_T==[|cos(q),0,-sin(q),0,0,0; 0,1,0,0,0,0; sin(q),0,cos(q),0,0,0; 0,0,0,cos(q),0,-sin(q); 0,0,0,0,1,0; 0,0,0,sin(q),0,cos(q)|]
XJ==X_T
vJ==S*qd
cJ==(|0, 0, 0, 0, 0, 0|)

InOuts

q: real
qd: real
qdd: real
tau: real
X_T: matrix(real,6,6)
XJ: matrix(real,6,6)
vJ: vector(real,6)
cJ: vector(real,6)

Figure 4.22: Featherstone revolute joint definition using spatial vectors.

joint definition

RevoluteJoint_CGA

Equations

motor_joint==motorProduct(motor_frame, (|cos(0.5*q), sin(0.5*q)*axis_rot[0], sin(0.5*q)*axis_rot[1], sin(0.5*q)*axis_rot[2], 0, 0, 0, 0|))

InOuts

q: real
axis_rot: vector(real,3) = (|0, 1, 0|)
motor_frame: vector(real,8)
motor_joint: vector(real,8)

Figure 4.23: CGA revolute joint definition using rotors.

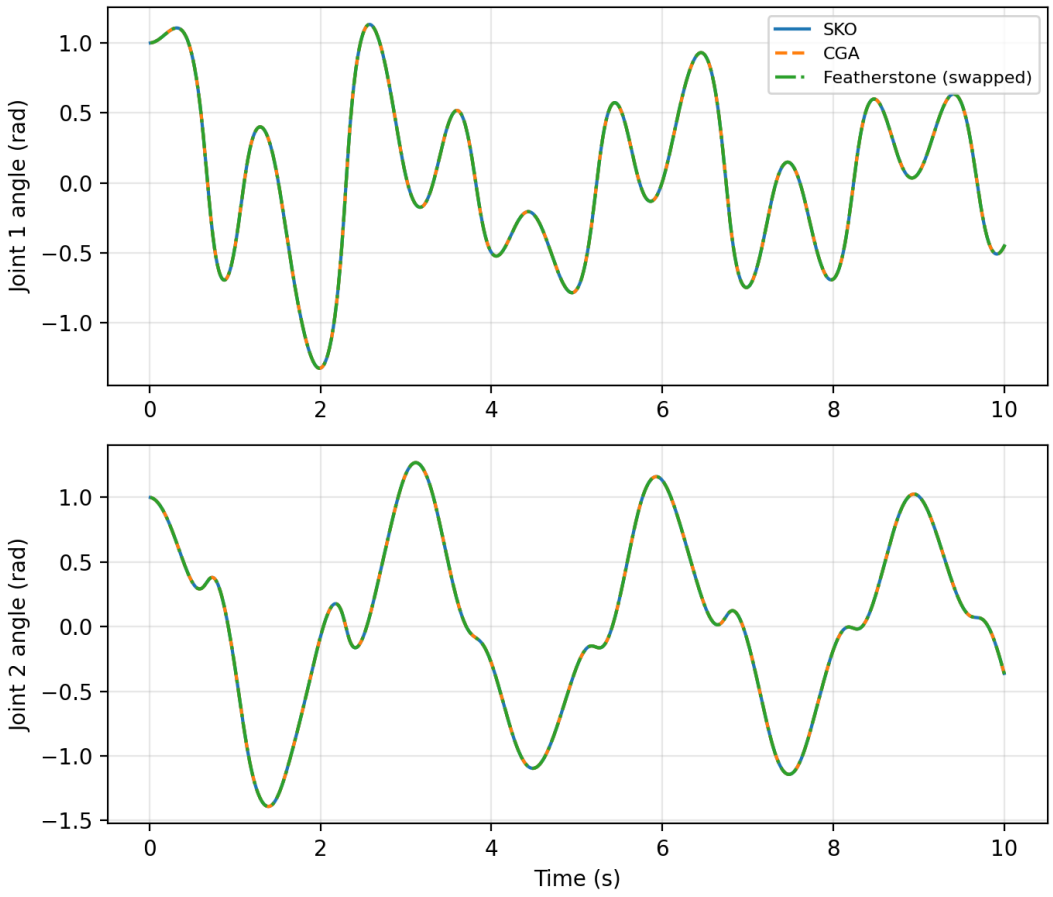


Figure 4.24: Standalone Acrobot joint trajectories for SKO, Featherstone, and CGA formulations over the first 10 seconds (same initial conditions). Featherstone joint ordering is swapped to align with SKO/CGA.

Swing-up Control

The control approach consists of two phases. In the swing-up phase, an energy-based controller pumps energy into the system by applying torques that increase the total mechanical energy until it reaches the energy level of the upright equilibrium. Once the system is near the upright configuration, control switches to a Linear Quadratic Regulator (LQR) that stabilises the system at the unstable fixed point. The LQR controller is designed by linearising the nonlinear manipulator equations around the upright equilibrium and solving the optimal control problem for a quadratic cost function.

The discrete controller behaviour is modelled as a RoboSim state machine, `AcrobotSwingUpLQR`, implementing the standard two-mode switching strategy used in Drake's Acrobot example [85]: an energy–based swing-up controller far from the upright configuration, and an LQR stabiliser near the upright equilibrium. As shown in Figure 4.25, the state machine receives periodic joint measurements via the `sensorUpdate` event (an `AcrobotSensorState` record containing joint angles and velocities) and issues a single actuator command per control cycle via `AcrobotActuatorI.ApplyTorque`.

The controller is parameterised using physical constants including link masses (m_1, m_2), link lengths (l_1, l_2), centre-of-mass offsets (l_{c1}, l_{c2}), moments of inertia about the centres of mass (I_{c1}, I_{c2}), and gravitational acceleration (g). The swing-up policy follows Spong's energy-shaping approach: the total mechanical energy $E = KE + PE$ is regulated toward the desired upright energy E_{desired} using the gain k_e, while additional proportional–derivative (PD) terms (k_p, k_d) shape the internal dynamics (via partial feedback linearisation) to keep the elbow motion well-behaved during energy injection. For balancing, the controller applies a precomputed linear state-feedback law obtained from the linearisation about the upright equilibrium, $\tau = -Kx_{\text{err}}$, with gains $(K_0, \ldots, K_3)$.

Mode switching is governed by a quadratic state-cost metric computed from the error state $x_{\text{err}} = [q_1 - \pi,\ q_2,\ \dot{q}_1,\ \dot{q}_2]^\top$. A symmetric weighting matrix S (entries $S_{00} \ldots S_{33}$) defines the cost $cost = x_{\text{err}}^\top S\, x_{\text{err}}$, and the Boolean `near_upright` is set when this cost falls below `balancing_threshold`, indicating the system is sufficiently close to the upright region of attraction for LQR stabilisation.

Internally, the state machine maintains local copies of the joint state (q_1, q_2, $\dot{q}_1$, $\dot{q}_2$), intermediate dynamics terms (mass matrix elements M_{11}, M_{12}, M_{22} and nonlinear terms C_1, C_2), energy quantities (KE, PE, E, E_{desired}), and the error components ($x_{\text{err}0} \ldots x_{\text{err}3}$). Control execution proceeds through five main states: `Wait` (idle, awaiting `sensorUpdate`), `Compute` (state wrapping, dynamics updates, and switching decision), `SwingUp` (energy-shaping and partial feedback linearisation to compute τ_{raw}), `Balance` (LQR computation of τ_{raw}), and `Output` (issues the command). After τ_{raw} is computed, clamping enforces the actuator bound `TAU_MAX`, ensuring the final torque τ respects hardware limits before being applied.

The platform mapping connects the discrete controller outputs to the physical actuator inputs. The mapping specifies how the control signal computed by the state machine is applied to the elbow motor, following the relationship $\tau = B_{\text{ctrl}} \cdot u$ where u is the scalar control input from the d-model.

The Acrobot model is processed through the three-stage generator pipeline, with each stage transforming the model representation and adding information required for simulation. The guidedChoice generator takes the extended p-model and a solution block that specifies which variables to compute and in what order. For the Acrobot with LQR control, the solution block

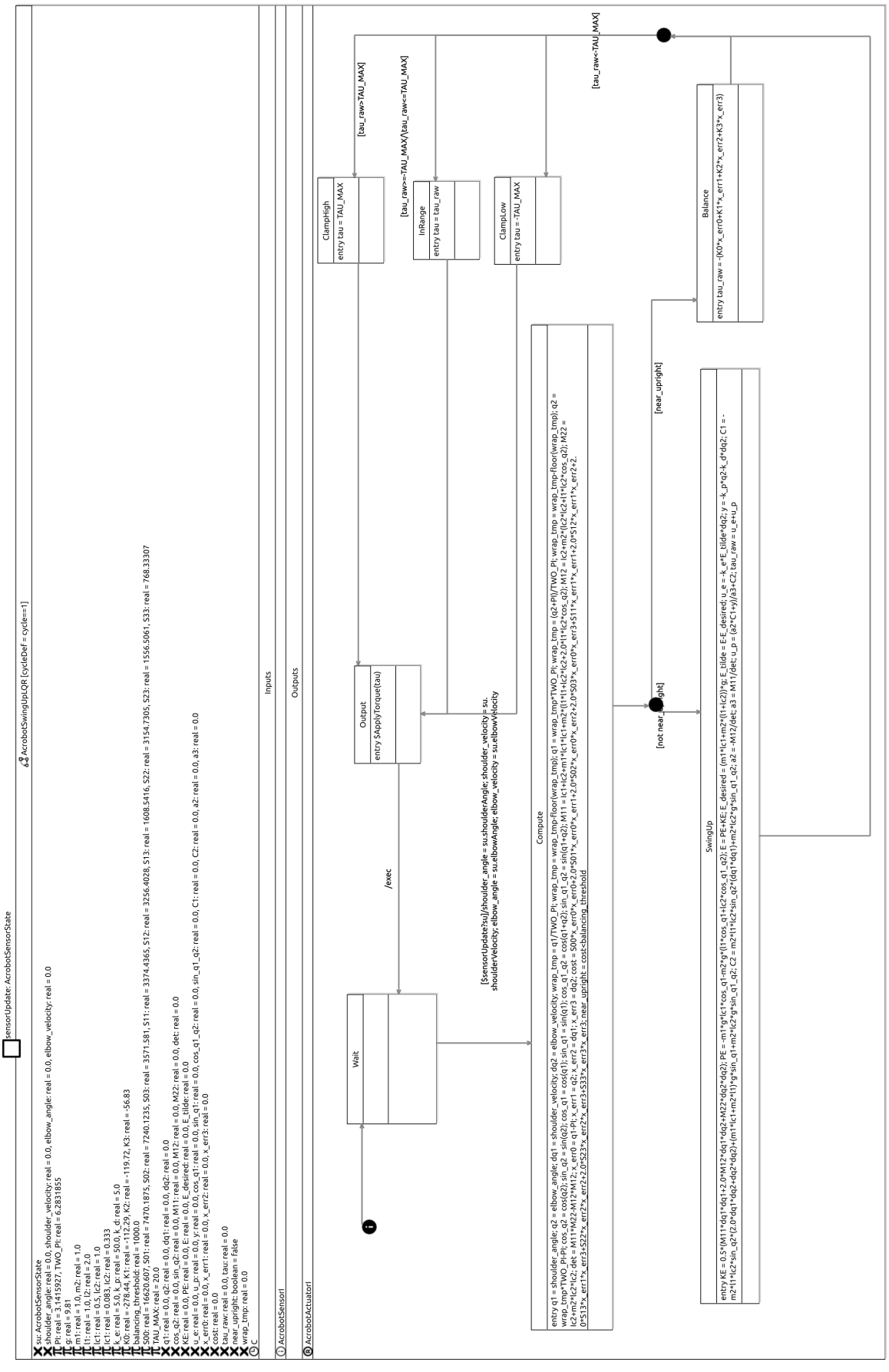


Figure 4.25: Acrobot d-model controller state machine (`AcrobotSwingUpLQR`).

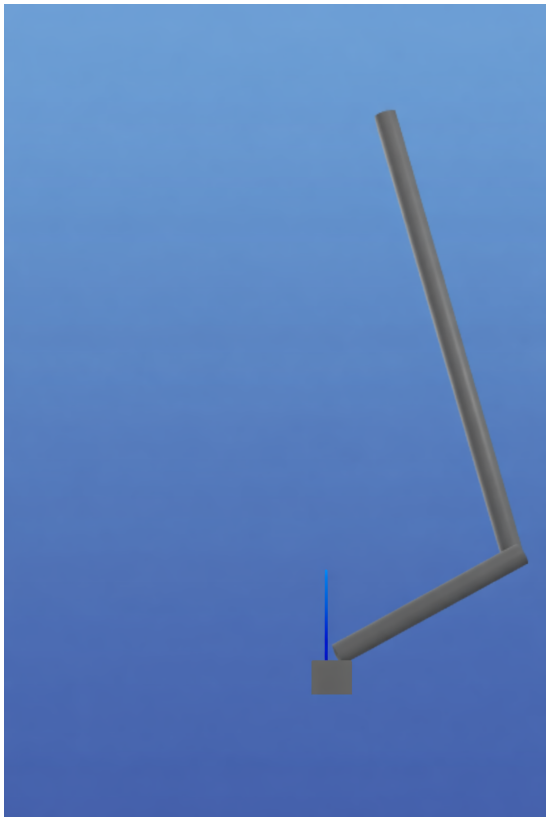


Figure 4.26: Visualisation of the Acrobot during simulation, showing the swing-up motion from the downward equilibrium.

specifies:

1. `u`: Map control input from d-model via `Mapping::PlatformMapping`
2. `tau`: Compute joint torques via `Actuator::ControlledActuator` using $\tau = B_{\text{ctrl}} \cdot u$
3. `theta`: Forward dynamics via `SKO::GeneralisedPosition` with gravity and damping
4. `B_k`: Evaluate transformation matrices via `SKO::Eval`
5. `T_geom`: Compute visualisation transforms via `SKO::Visualisation`

T5 generates skeleton files with `TODO/STUB` markers that require manual implementation. The detailed stub implementations for the Acrobot case study, including the d-model interface, platform mapping, world engine, orchestrator, and controller adaptation code, are provided in Appendix A.1.2.1.

As well as generating the full RoboSim simulation for this example, several properties of interest of the Acrobot are verified in Isabelle/UTP, starting from a proof system generated by sourceCodeGen. In particular, the system is proven to be underactuated, controllable, and to have a fixed point at the upright equilibrium. A key property verified is that despite being underactuated, the system is locally controllable around the upright equilibrium. This verification is formalised in Isabelle/HOL, providing a machine-checked proof that links the physical model properties to the control-theoretic requirements for LQR stabilisation.

A mechanical system is underactuated if the number of independent control inputs m is strictly less than the number of generalised coordinates n [127]. For the Acrobot, $n = 2$ (two revolute joints) and $m = 1$ (only the elbow joint is actuated), giving a degree of underactuation of $n - m = 1$. The control input matrix $B_{\text{ctrl}} \in \mathbb{R}^{2\times 1}$ maps the single control input to joint torques:

$$B_{\text{ctrl}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

In the generated model, the joint coordinates are ordered tip-to-base, so the first component corresponds to the elbow joint. Therefore, the unit entry indicates full actuation at the elbow and the zero entry indicates no direct actuation at the shoulder.

The Isabelle formalisation defines underactuation as a predicate and proves that the Acrobot satisfies this property:

```
definition is_underactuated :: "nat ⇒ nat ⇒ bool" where
  "is_underactuated n m ≡ m < n"
```

The theorem then establishes that the Acrobot satisfies this property:

```
theorem acrobot_is_underactuated:
  shows "is_underactuated (CARD(2)) (CARD(1))"
  unfolding is_underactuated_def
  by simp
```

For LQR design, the nonlinear dynamics are linearised around the upright equilibrium point $(\theta^*, \dot{\theta}^*, u^*)$. The linearised state-space form is:

$$\dot{x} = Ax + Bu$$

where the state vector $x = [\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2]^T \in \mathbb{R}^4$. The linearised A matrix has the structure:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & 0 & 0 \end{bmatrix}$$

where the a_{ij} terms arise from the partial derivatives of the gravitational torques, and the linearised B matrix is:

$$B_{\text{lin}} = \begin{bmatrix} 0 \\ 0 \\ b_{31} \\ b_{41} \end{bmatrix}$$

where the b_{ij} terms come from $M^{-1} \cdot B_{\text{ctrl}}$ evaluated at the equilibrium.

The system is controllable if and only if the controllability matrix $\mathcal{C} = [B \mid AB \mid A^2B \mid A^3B]$ has full rank.

The proof quantifies over a state s of the (generated) verification dataspace, and the assumptions in theorem `acrobot_controllability_upward` refer to dataspace fields by reading the values of `theta`, `theta'`, `M_mass_inv`, and `B_ctrl` at s. All other identifiers introduced in this proof section (`A_lin`, `B_lin`, `kalman2`, `K_upright`, etc.) are derived matrix definitions used to structure the controllability argument, and are not dataspace variables.

The linearised system matrices are defined in Isabelle as follows:

```
definition A_lin :: "real mat[2,2] ⇒ real mat[4,4]" where
  "A_lin K ≡ ⟦[0,0,1,0],
    [0,0,0,1],
    [K$0$0, K$0$1, 0, 0],
    [K$1$0, K$1$1, 0, 0]⟧"

definition B_lin :: "real mat[2,1] ⇒ real mat[4,1]" where
  "B_lin L ≡ ⟦[0],
    [0],
    [L$0$0],
    [L$1$0]⟧"

definition B_lin_from_B_ctrl :: "real mat[2,2] ⇒ real mat[2,1] ⇒ real
mat[4,1]" where
  "B_lin_from_B_ctrl M_inv B ≡ B_lin (M_inv ** B)"
```

The controllability matrix and controllability predicate are then defined:

```
definition controllability_matrix_4x1 ::
  "real mat[4,4] ⇒ real mat[4,1] ⇒ real mat[4,4]" where
  "controllability_matrix_4x1 A B ≡
    (let AB = A ** B;
         A2B = A ** AB;
         A3B = A ** A2B in
       ⟦[B$0$0,  AB$0$0,  A2B$0$0,  A3B$0$0],
        [B$1$0,  AB$1$0,  A2B$1$0,  A3B$1$0],
        [B$2$0,  AB$2$0,  A2B$2$0,  A3B$2$0],
        [B$3$0,  AB$3$0,  A2B$3$0,  A3B$3$0]⟧)"

definition is_controllable :: "real mat[4,4] ⇒ real mat[4,1] ⇒ bool"
where
  "is_controllable A B ≡ det (controllability_matrix_4x1 A B) ≠ 0"
```

For the parameters at the upright equilibrium, the concrete matrices are defined as:

```
definition B_ctrl_upright :: "real mat[2,1]" where
  "B_ctrl_upright ≡ ⟦[1], [0]⟧"

definition M_mass_inv_upright :: "real mat[2,2]" where
  "M_mass_inv_upright ≡ ⟦[2000/333, -1000/333],
    [-1000/333, 1333000/776889]⟧"

definition K_upright :: "real mat[2,2]" where
  "K_upright ≡ ⟦[1090/37, -545/37],
    [-1090000/86321, 1089455/86321]⟧"
```

The main controllability theorem establishes that the Acrobot is controllable at the upright equilibrium when the state satisfies the equilibrium conditions:

```
lemma acrobot_upright_controllable:
  shows "is_controllable (A_lin K_upright)
    (B_lin_from_B_ctrl M_mass_inv_upright B_ctrl_upright)"

theorem acrobot_controllability_upward:
```

```
fixes s :: 'st
assumes "theta⟨s⟩ = upward_equilibrium"
assumes "theta'⟨s⟩ = 0"
assumes "M_mass_inv⟨s⟩ = M_mass_inv_upright"
assumes "B_ctrl⟨s⟩ = B_ctrl_upright"
shows "is_controllable (A_lin K_upright)
  (B_lin_from_B_ctrl (M_mass_inv⟨s⟩) (B_ctrl⟨s⟩))"
```

Results

The Acrobot case study was processed through the complete generator pipeline. Table 4.8 presents the execution times for each transformation stage.

Generator	Time (s)
T3 (eqnComp)	14.39
T4 (guidedChoice)	47.06
T5 (sourceCodeGen)	0.27
Total	61.71

Table 4.8: Execution times for processing the Acrobot model through each generator stage (5-run average).

The generated physics engine correctly computes the forward dynamics for the underactuated system and supports closed-loop swing-up and stabilisation. Figure 4.27 shows the joint angle evolution for the generated simulation, illustrating the swing-up phase followed by convergence under the LQR controller. The shaded background indicates the controller mode (energy-shaping swing-up versus LQR balance), demonstrating successful switching and stabilisation at the upright equilibrium.

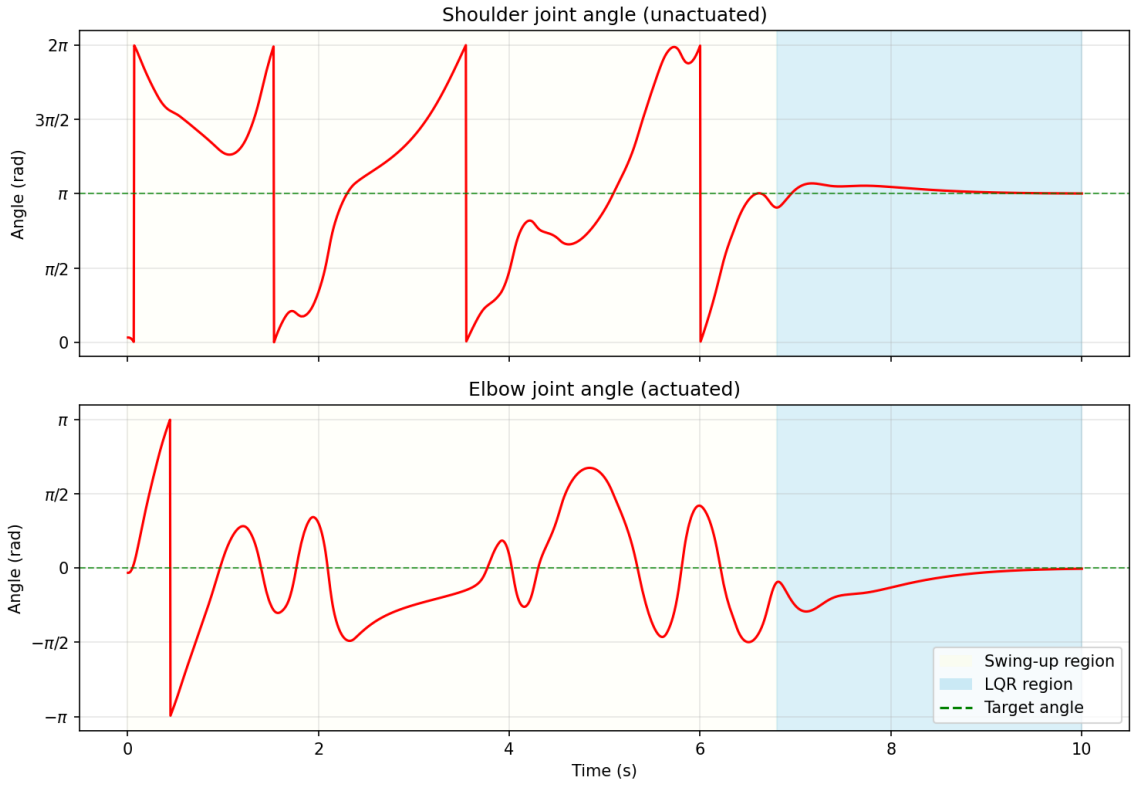


Figure 4.27: Joint angle trajectories for the controlled Acrobot produced by the generated physics engine. The swing-up controller drives the system from the downward-hanging initial configuration towards the upright equilibrium, after which the LQR controller stabilises the motion. The elbow angle is wrapped to $[-\pi, \pi)$ for presentation, since angles that differ by 2π correspond to the same physical configuration. Shaded regions indicate the controller mode (swing-up versus LQR).

Simulation results show successful swing-up from the downward equilibrium to the upright position, with the LQR controller maintaining balance at the unstable fixed point.

Summary

The Acrobot case study demonstrates the simulation framework's ability to generate a consistent RoboSim simulation from the p-model and d-model. The Isabelle formalisation provides machine-checked proofs of controllability properties, linking the physical model to control-theoretic requirements and demonstrating the framework's support for formal verification.

4.3.3 Mobile Robot Obstacle Avoidance

This case study[5] applies the simulation framework to a differential-drive mobile robot controlled by a RoboSim obstacle-avoidance state machine. The platform is modelled as a Turtlebot p-model, the controller as a RoboSim d-model, and their interaction as a platform mapping.

[5] https://github.com/UoY-RoboStar/robotool-meta-physmod/tree/main/Examples/CaseStudies/MobileRobot/MobileRobotOA

The control task is reactive obstacle avoidance using LiDAR sensing. The controller alternates between a wandering mode that generates a random walk and an avoidance mode that responds when an obstacle is detected. It produces angular and linear velocity commands that are mapped to wheel motor control inputs.

Figure 4.28 shows the p-model of the Turtlebot3 Burger platform. The model declares a base link with two controlled wheel joints (`LeftWheelJoint`, `RightWheelJoint`) and two controlled motors (`LeftMotor`, `RightMotor`). Two sensors are attached to the base link: a LiDAR sensor (`TBLidar`) and an IMU sensor (`TBIMU`). Mesh geometry is assigned to the chassis and wheels so that the generated visualisation can reuse the Turtlebot meshes.

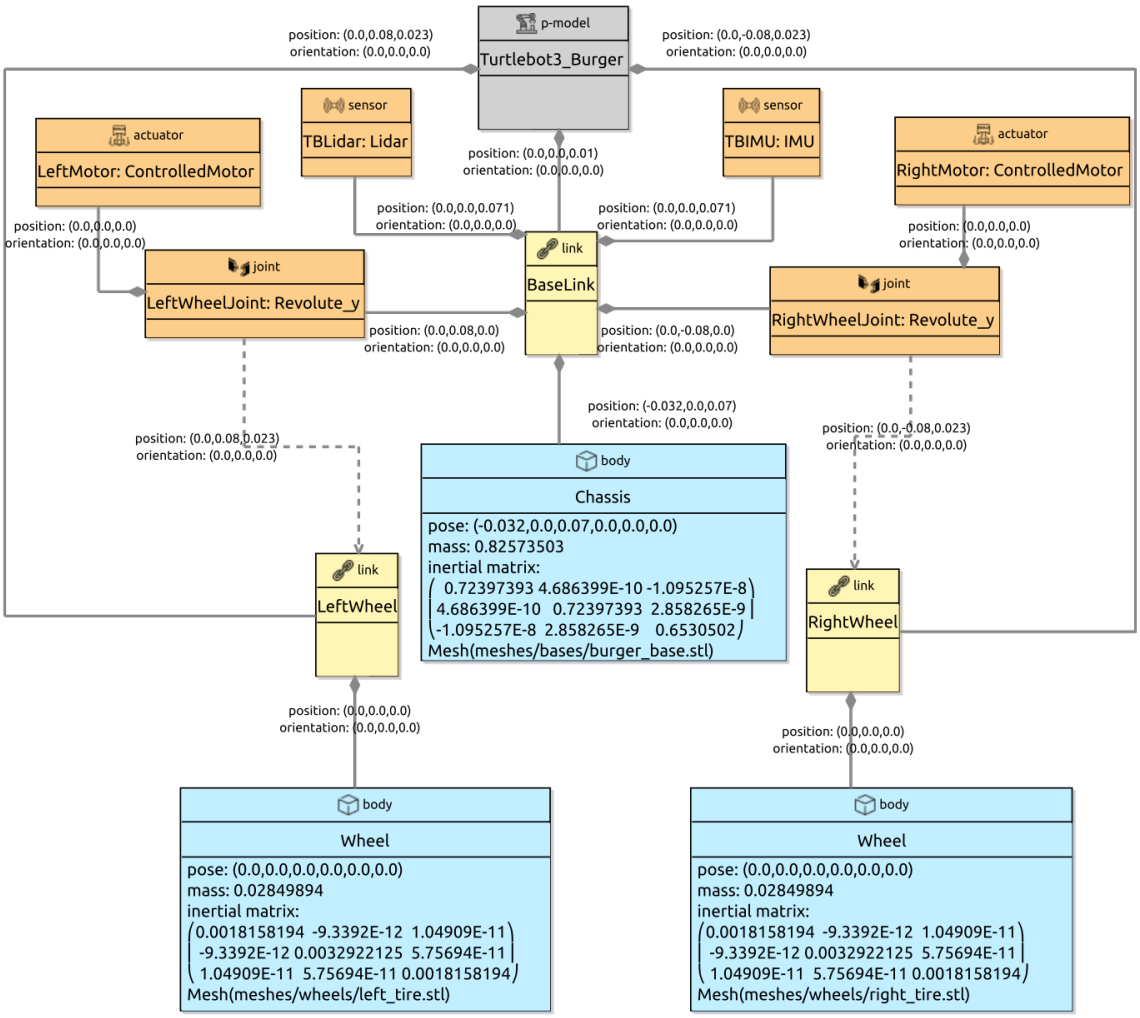


Figure 4.28: P-model for the Turtlebot3 Burger platform with controlled wheel motors and LiDAR/IMU sensors.

Figure 4.29 shows the LiDAR and IMU sensor definitions. The LiDAR uses a beam-model mixture and exposes the derived outputs `closestDistance` and `closestAngle`, which are the quantities consumed by the controller. The IMU exposes `currentLV` and `currentAV` as outputs corresponding to linear and angular velocity.

Figure 4.30 shows the RoboSim d-model for obstacle avoidance. The model is a state machine with a wandering region and an avoidance region. The wandering region toggles between turning and moving forward, and the avoidance region reduces speed and steers away from obstacles based on the closest obstacle angle and distance.

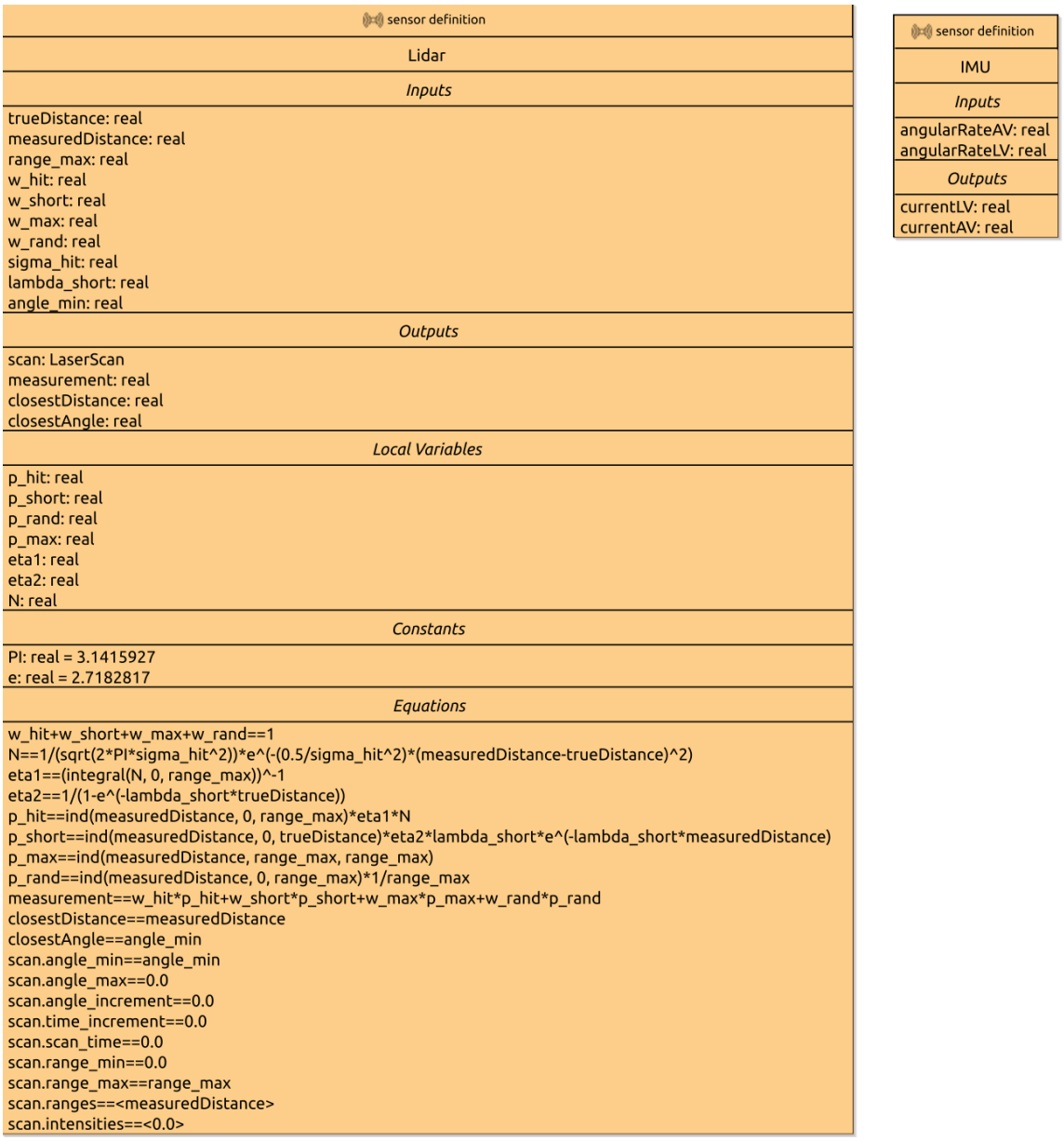


Figure 4.29: LiDAR and IMU sensor definitions used by the Turtlebot platform.

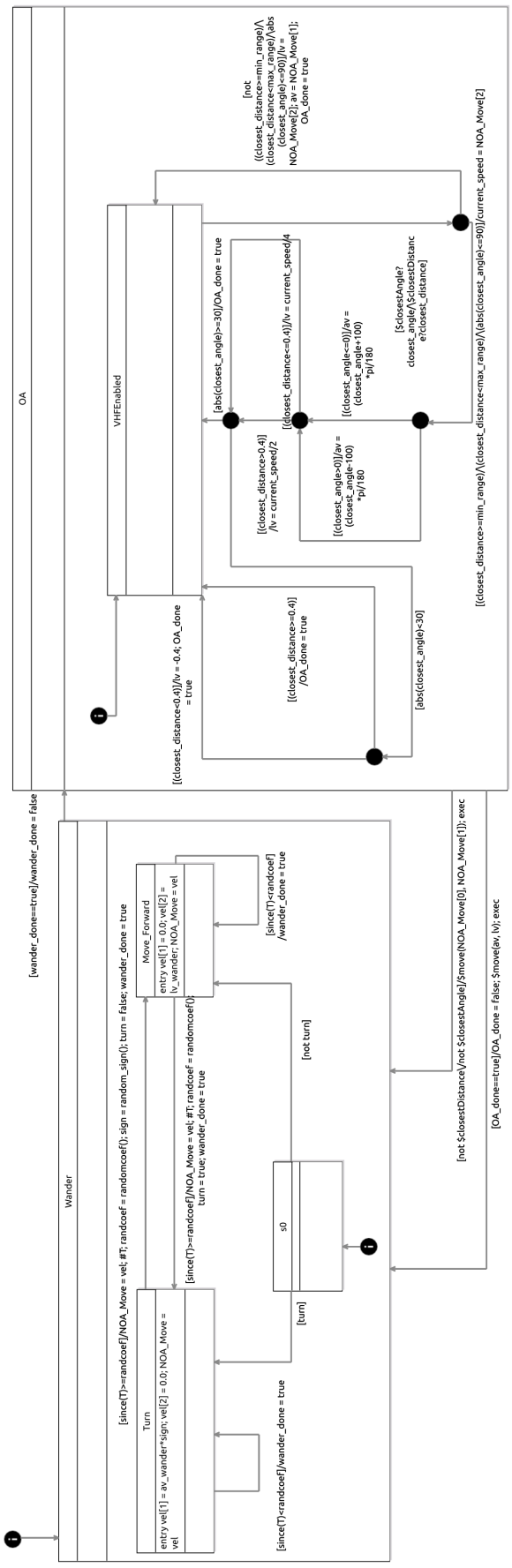


Figure 4.30: RoboSim d-model state machine for obstacle avoidance. A wandering mode generates random walk behaviour, and an avoidance mode reacts to obstacle proximity and bearing.

Figure 4.31 shows the platform mapping. The controller operation `move(av, lv)` is mapped to wheel motor control inputs using a differential-drive relation, and the controller events `closestDistance?` and `closestAngle?` are mapped to the LiDAR outputs `closestDistance` and `closestAngle`.

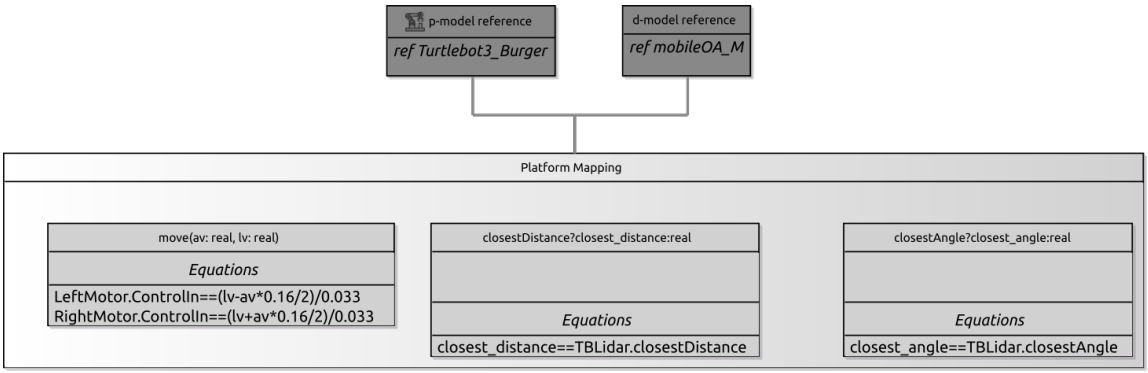


Figure 4.31: Platform mapping between the obstacle avoidance d-model and the Turtlebot p-model. The `move(av, lv)` operation is mapped to wheel motor control inputs, and LiDAR-derived `closestDistance` and `closestAngle` values are mapped to controller input events.

After equation composition, this case study selects the SKO tree-topology solutions needed for forward kinematics, mass matrix construction and inversion, direct forward dynamics with Euler integration, and visualisation transforms. In addition, the prototype mapping-generation pipeline produces explicit equations for the `move` operation and LiDAR events, exposing the controller/platform interface at the level of `Solution DSL` constraints.

Figure 4.32 shows a snapshot of the generated visualisation, using the Turtlebot mesh assets and the obstacle layout defined in the world model.

The generated simulation was executed for 10 minutes, logging the clearance to each obstacle and to the arena walls. Figure 4.33 focuses on the near-collision region in linear scale, with the collision band shaded in red; all traces approach but do not enter the collision region, and the clearance remains positive throughout the run.

Overall, the case study highlights the role of platform mappings in bridging the gap between controller-level velocity commands and platform-level actuation, and between world-dependent sensing and controller input events.

The Mobile Robot model was processed through the complete generator pipeline. Table 4.9 presents the execution times for each transformation stage.

Generator	**Time (s)**
T3 (eqnComp)	10.27
T4 (guidedChoice)	31.89
T5 (sourceCodeGen)	0.19
Total	42.35

Table 4.9: Execution times for processing the Mobile Robot model through each generator stage (5-run average).

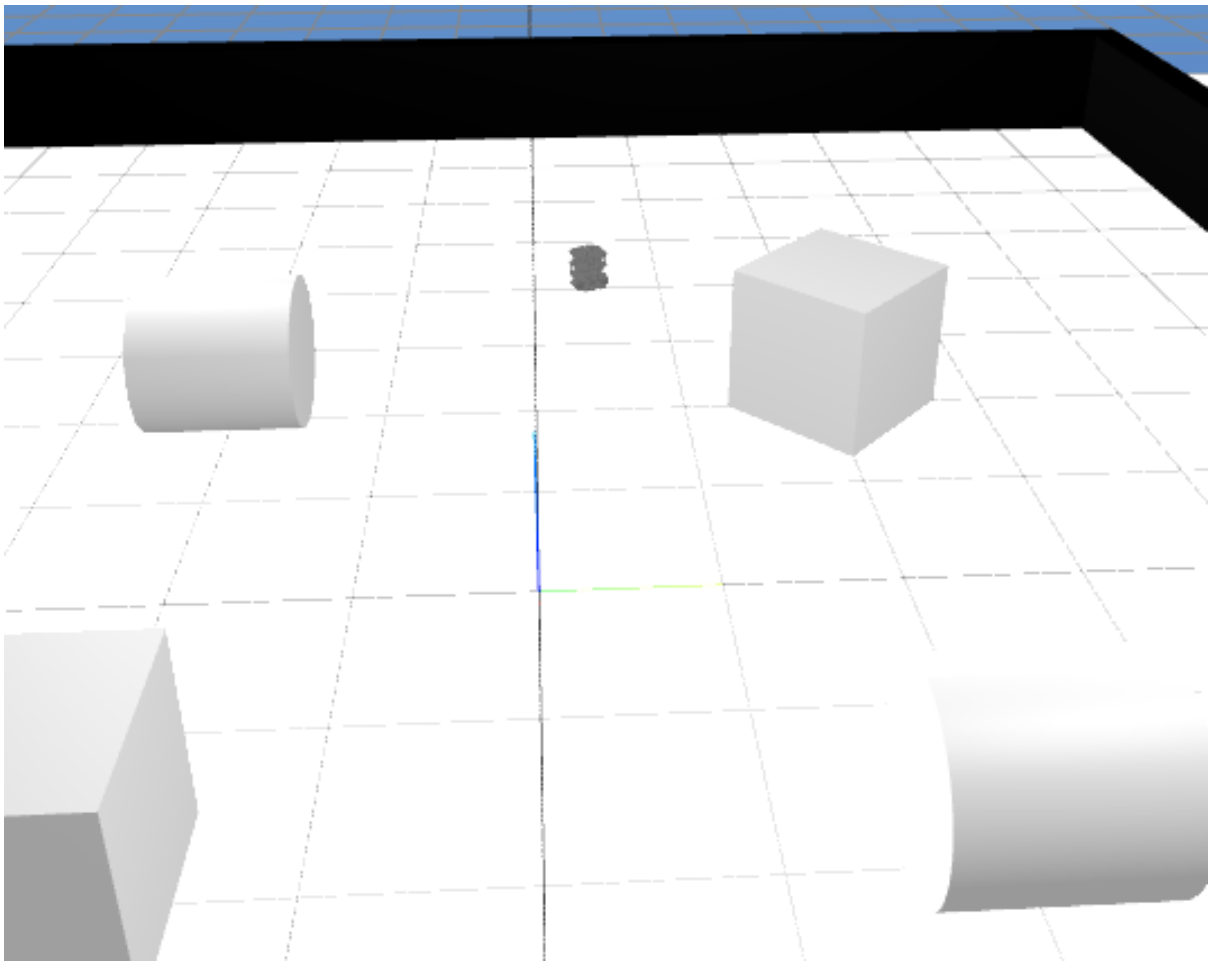


Figure 4.32: Generated visualisation for the mobile robot case study.

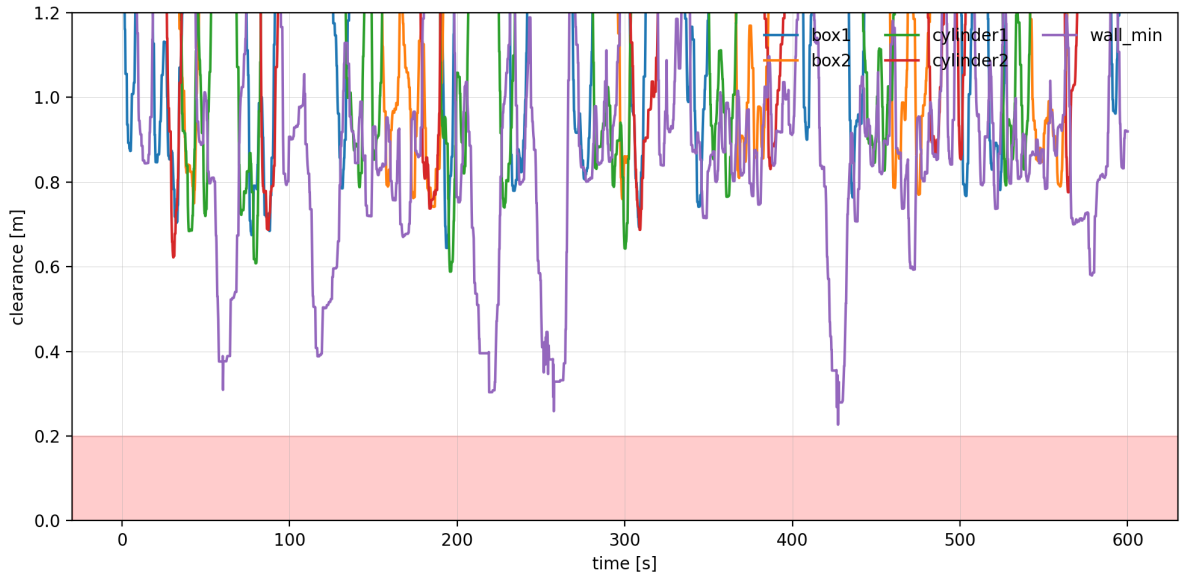


Figure 4.33: Minimum clearance to each obstacle and to the arena walls during a 10-minute run. The plot is zoomed to the near-collision range, and the red band marks the collision region.

4.4 Discussion

This section discusses the extensibility of the framework to new formulations, robot topologies, and target platforms, and reflects on the current limitations.

4.4.1 Extensibility

The framework has been designed with extensibility as a primary concern, allowing new capabilities to be added without modifying existing code. This section discusses the extension points available in each technique. Table 4.10 summarises the current extensibility status of each technique's library component.

Technique	Library	Status	Reason
T3	Equation Library	Partial	Equation templates in slnDF; formulation detection requires code
T4	Solution Library	Open	Solution methods dispatched via `SolutionLib` switch statement
T5	Code Templates	Open	Target generators can be added independently

Table 4.10: Extensibility status of library components in each technique. T4 and T5 support user extension. T3's equation library is now specified in slnDF, but formulation detection still requires generator modification.

The eqnComp equation library is implemented using the data-driven slnDF architecture described in Section 4.1.1. Each formulation (e.g., SKO, Featherstone) is specified as an `.slnDF` file in the `physmod-libs/eqnComp/` directory, containing declarative specifications of the variables, equations, and constraints to add to the p-model. This approach allows domain experts to define new equation templates without modifying Xtend code. Only the slnDF file needs to be created or edited. The slnDF specifications also serve as both implementation and documentation, ensuring the mathematical intent of each formulation remains clear and verifiable.

However, a limitation of the current implementation is that the formulation detection mechanism (`calculateFormulations` in Step 1) cannot be extended without modifying generator code. Currently, formulation detection relies on pattern matching within the p-model: SKO is detected by the presence of hinge matrices (`H` constants) in joint definitions, while Featherstone is detected by the presence of motion subspace matrices (`S` constants). Adding support for a new formulation requires updating the `Formulation` enumeration and the detection logic in `findDynamicsFormulation`, even though the equation generation itself can be specified entirely in slnDF.

Nevertheless, the slnDF specifications ensure there is a formal semantics to the formulation-specific portion of the generator that is independent of the Xtend implementation, making the equation generation logic easier to verify and reason about. The full slnDF specifications for each formulation are maintained in the `physmod-libs` repository.[6] Currently, the four-step structure (define datatypes and functions, define system variables, component equations, system equations) serves as a guideline for formulation authors. Future work could formalise this structure through well-formedness conditions on equation library specifications, enforced

[6] https://github.com/UoY-RoboStar/robotool-meta-physmod

by a dedicated validator that checks each slnDF file adheres to the expected structure before it is loaded by the transformer.

In contrast, the guidedChoice solution library and sourceCodeGen code generation templates are designed to be fully extensible. New solution methods can be added to guidedChoice by implementing additional cases in the `SolutionLib` dispatch mechanism, and new target platforms can be added to sourceCodeGen by creating new generator classes.

However, unlike the eqnComp equation library which uses declarative slnDF specifications, the guidedChoice solution library is currently implemented as Xtend code (SKO_gen, Vis_gen, Mapping_gen, etc.). Each solution method is a static class with an `asReference` method that programmatically constructs `SolutionRef` objects. While this approach provides full flexibility, it means that adding new solution methods requires Xtend programming expertise rather than the domain-expert-friendly declarative specifications used in eqnComp. Future work should migrate the T4 and T5 Xtend solution libraries to a specification-based format analogous to slnDF, and extend the formulation detection mechanism to be data-driven, enabling domain experts to define solution unfolding templates and new formulations in a format which is more obviously data than code.

4.4.2 Critical Reflections

The correctness of the data-driven architecture depends on the correctness of the meta-operations layer. While the slnDF specifications provide a declarative semantics for formulation-specific equation generation that is independent of the Xtend implementation, the correctness of the generated p-model+ ultimately depends on the faithful implementation of the eight primitive meta-operations (`createVariable`, `createConstant`, `createEquation`, `createConstraint`, `createRelation`, `addInOut`, `createDatatype`, `createFunction`). Any formal verification of the slnDF specifications would establish properties only up to the assumption that these meta-operations correctly manipulate the EMF model.

The meta-operations have been validated through comprehensive integration tests that exercise each primitive across multiple formulations (SKO and Featherstone) and robot configurations (serial chains, tree topologies). These tests verify that the generated p-model+ files parse correctly, pass the p-model validator, and produce the expected equations for known robot configurations. While this does not constitute formal verification, the testing strategy provides confidence that the meta-operations behave correctly for the equation patterns used in practice.

The execution time measurements from the case studies reveal that the guidedChoice generator (T4) consistently dominates the total generation time, taking approximately 2–4 times longer than eqnComp (T3) and orders of magnitude longer than sourceCodeGen (T5). This disparity arises from the architectural differences between the generators: T4 performs repeated solution library queries and recursive solution unfolding, whereas T3 executes a linear pass through the p-model and T5 performs straightforward template instantiation. The current T4 implementation does not cache intermediate results or optimise the dependency resolution graph, resulting in redundant computation when multiple solutions share common sub-expressions. Future work should investigate caching strategies and incremental resolution to reduce T4 execution time, particularly for larger models where the current performance characteristics may become prohibitive for interactive use.

The benchmarking evaluation was performed on a 2-DOF Acrobot system, which demonstrated that the generated forward dynamics produces identical trajectories and competitive computa-

tion times compared to established physics engines. However, this single configuration is not sufficient to generalise that the approach scales comparably with other engines across systems of varying complexity. A more comprehensive evaluation would require benchmarking across a range of robot topologies, degrees of freedom, and computational workloads to establish performance characteristics that generalise beyond the specific case study presented.

Finally, some parts of the current pipeline rely on heuristics rather than declarative specifications. In particular, the prototype mapping-generation pipeline implemented in this thesis relies on string-level recognition of common mapping patterns in order to emit adapter code. This is sufficient for the case studies, but it is a potential source of brittleness when mapping specifications deviate from the expected naming conventions. Future work could address this limitation by implementing a semantics-aware mapping transformation that infers mapping intent from model structure rather than string patterns.

5 Conclusions

5.1 Summary

The central contribution of this thesis is a modelling and generation approach that keeps the physical equations and the computational solutions in a form that can be reused consistently across both simulation and proof. The framework uses p-models (and the extended p-model, p-model+) to capture the physical structure and equations of a platform, and uses slnRef and slnDF artefacts to make solution choice and solution implementation explicit within the simulation pipeline. This enables the generation of executable platform-engine code and proof artefacts from the same underlying specification, which directly targets the goal of consistency between proof and simulation.

Compared to general-purpose physics engines and simulators such as MuJoCo, Bullet, Drake, and PhysX [4][84][85][93], this approach trades the breadth and maturity of a complete simulation stack for greater transparency and traceability of the computational steps that realise the model. A conventional simulator typically provides an end-to-end pipeline including collision detection, contact resolution, and time integration, but the semantics of that pipeline are largely embedded in engine code and configuration. In this approach, the equations and their associated solution methods remain explicit artefacts, which makes it easier to explain and audit how a particular simulation result relates to a particular proof obligation. The cost of this explicitness is that achieving feature parity with a mature engine is not immediate, and some aspects of high-performance simulation, such as specialised contact handling and GPU pipelines, remain more fully developed within dedicated engines.

When compared to robotics dynamics and robotics-maths libraries, such as RBDL and Pinocchio [87][88], and alternative pose and motion algebra toolkits such as DQRobotics and GAFRO [89][90], this framework occupies a different point in the design space. These libraries are effective for fast kinematics and rigid body dynamics computations within optimisation and control pipelines, but they typically assume that a wider simulation context is provided elsewhere. RobCoGen is the closest existing approach because it also treats kinematics and dynamics implementations as generated artefacts, producing robot-specific code from a domain-specific language [98], and it performs strongly in benchmarking studies [99]. This approach extends that idea by coupling model-level equation specification with explicit, inspectable solution specifications, and by generating proof artefacts alongside simulation code so that the relationship between the two can be maintained systematically.

5.2 Evaluation of Objectives

The overarching goal of this thesis was to enable consistency between simulation and proof for the verification of robotic systems. Three objectives were stated in Chapter 1, and the framework can now be evaluated against them.

The first objective was to enable modification and extension of equations at the model level. This is achieved through the design of the extended p-model (p-model+) and the equation composition technique (eqnComp, Technique 3). The framework allows component-level equations to be specified in the p-model and then automatically composed into system-level equations, and the slnDF-based equation library allows the generator to be extended with new equation templates without modifying its core implementation. The case studies in Chapter 4 demonstrate this capability across multiple dynamics formulations and kinematic topologies.

The second objective was to enable proving properties in a theorem prover while understanding to what extent those properties are observable within simulation. This objective is partially achieved. The framework generates Isabelle definitions that capture the dataspace and equations of the physical model, supporting proofs of physical properties that can be related back to generated simulations. The Acrobot case study demonstrates that properties proven using the generated Isabelle artefacts correspond to behaviour observable in the generated C++ simulation. However, the current implementation generates only the physical portion of the cyber-physical model, and complete end-to-end proof generation through CyPhyCircus remains future work as discussed in Section 5.3.

The third objective was to provide information about equation usability, including whether equations can be processed by the engine and which solution methods are applicable. This is achieved through the validation steps in eqnComp and the guided choice mechanism in guidedChoice (Technique 4). The framework validates that p-models are well-formed before equation composition, checks that requested solution methods are applicable to the model structure, and reports errors when equations cannot be processed. The solution library also provides feedback about which formulations and solution methods are available for a given model configuration, clarifying what the toolchain can and cannot do before committing to a particular simulation target.

Overall, the thesis demonstrates that a practical form of proof and simulation consistency can be obtained by generating both executable code and proof artefacts from a common model. The case studies establish that the framework can handle serial manipulators, mobile robots, and closed-loop mechanisms, and the benchmarking study shows that the generated dynamics can match established engines in both trajectories and performance for the evaluated configuration.

5.3 Future Work

5.3.1 Simulation Framework

Several extensions would strengthen the simulation framework and broaden the class of platforms and analyses that it can support. For Technique 3 (eqnComp), the current formulation calculation focuses on joint-based analysis for dynamics, which is sufficient for equations of motion generation in the presented case studies. Extending formulation calculation to analyse link equations in addition to joints would support formulations and constraints that are more naturally expressed at the link level, and it would enable hybrid approaches where different parts of the mechanism require different mathematical treatments. A related direction is to support formulation decisions determined by combinations of link and joint properties, and to move from a single system-wide formulation choice to component-specific formulation mappings where links, joints, sensors, and actuators may select different formulations when

that is computationally appropriate.

The current implementation of formulation calculation assumes that the p-model is well-formed so the algorithm can run efficiently. Making this assumption explicit, by adding cross-component consistency checks to the equation composition validator in Step 2 (Section 3.2.3), would improve error reporting and prevent subtle formulation incompatibilities from being discovered only after numbering and equation generation. In the same spirit, the current system supports "trivial" formulations from the physmod library, but it does not yet support additional standard or custom locally-defined formulations for sensors and actuators. Extending formulation detection and library lookup to handle such formulations, both from libraries and from within individual p-models, would make the approach more usable for practical modelling.

The evaluation in this thesis validated the framework primarily using the SKO formulation, with Featherstone's spatial vector formulation and Conformal Geometric Algebra (CGA) demonstrated only for serial chain topologies. A systematic evaluation across a broader range of dynamics formulations and for all supported kinematic topologies, including tree structures and closed-loop mechanisms, would provide stronger evidence about generality and would help identify formulation-specific or topology-specific limitations that require further work. A smaller but important practical extension is to distinguish between visual and collision geometries. The current implementation generates reserved geometry variables without distinguishing appearance meshes from collision shapes, even though these often differ significantly in simulation pipelines. Extending the p-model metamodel or adding annotations to mark geometries as visual or collision, and reflecting this in the reserved naming convention, would make generated simulations less ambiguous and would simplify the integration of collision detection backends.

For Technique 4 (guidedChoice), the dominant future direction is to complete the shift to a specification-based architecture. Unlike the eqnComp equation library, which is specified declaratively in slnDF, the current solution library is implemented as Xtend code that constructs `SolutionRef` objects programmatically. Migrating this solution library to a declarative format analogous to slnDF would lower the barrier to extension and would allow domain experts to contribute new solution unfolding templates without modifying generator code. Beyond the representation of the library, the current library coverage focuses mainly on SKO formulation dynamics, and expanding coverage to other formulations, to sensors and actuators, and to constraint handling would improve applicability.

The current guided choice implementation performs repeated library queries and recursive unfolding without caching. Introducing incremental resolution and caching of solution queries would improve performance for larger models and interactive use. Several related directions become possible once the resolution process is treated as a dependency problem over explicit constraints: automatic dependency analysis to infer minimal ordering requirements, richer constraint satisfaction that propagates known values through constraint networks, and verification steps that check whether resolved solutions satisfy their preconditions and postconditions. Finally, while the current framework relies on a solution library, allowing user-defined solution methods within p-models would support domain-specific optimisations while still benefiting from the consistency checks and traceability of the guided choice mechanism.

For Technique 5 (sourceCodeGen), the current framework assumes discrete-time controllers that execute in cycles, following the RoboSim operational semantics where the controller and platform alternate execution at fixed intervals. Supporting continuous-time controllers would require extending orchestrator generation to integrate controller and platform states

concurrently, and extending platform mapping to handle continuous rather than sampled sensor and actuator signals. A second important direction is numerical error tracking and propagation. The slnRef language includes support for specifying numerical error bounds on solution methods, but the case studies did not use this capability. Propagating numerical error through the solution pipeline, and generating formal error certificates alongside simulation code, would make the relationship between proven properties and simulated behaviour more explicit, particularly when numerical integration is unavoidable. Finally, the prototype mapping-generation pipeline implemented in this thesis relies on string-level heuristics to recognise mapping patterns. Replacing these heuristics with a semantics-aware transformation that detects mappings from their structure would improve robustness and would reduce the number of trusted assumptions in the end-to-end toolchain.

5.3.2 p-model Metamodel and Semantics

The current p-model and CyPhyCircus semantics assume that the equations of motion are expressed in continuous time, with discrete-time behaviour handled separately by the controller. Extending the p-model metamodel and the CyPhyCircus semantics to allow discrete-time equations of motion within the physical model would be valuable for contact models and event-driven dynamics, where impulsive effects are naturally expressed as discrete updates, and for sensor models where sampling is an inherent part of the behaviour. Supporting discrete-time dynamics at the physical level would also provide a clearer semantics for hybrid systems whose physical evolution is not purely continuous.

The current p-model treats sensor equations deterministically, even though real sensors exhibit stochastic behaviour due to noise, calibration uncertainty, and environmental factors. Extending the metamodel to support probabilistic semantics for equations, particularly for sensors, would enable more realistic modelling and would open the door to probabilistic verification techniques. This would require representing sensor outputs as probability distributions or stochastic processes and extending the semantic layer to reason about those distributions in a way that remains compatible with the generated proof artefacts.

A further direction is to extend RoboSim to include environment models. The current framework focuses on platform models (p-models) and controller models (d-models), but does not provide a structured representation of the environment in which the robot operates. Introducing environment models would allow obstacles, terrain, and other environmental features to be specified at the model level, and extending the code generation pipeline to produce source code from these environment models would enable complete simulation scenarios to be generated from a unified modelling framework.

5.3.3 Verification and Proof

The current framework generates Isabelle definitions corresponding to the dataspace and equations of the physical portion of the cyber-physical model. The main future direction is to extend proof generation to produce complete CyPhyCircus proofs directly from RoboSim models, covering both the discrete controller (d-model) and the continuous physical platform (p-model) together with their interaction through the platform mapping. This would involve generating the full CyPhyCircus process algebraic specifications, producing hybrid program representations in differential dynamic logic, and generating the proof obligations required to establish end-to-end properties of the complete cyber-physical system. Achieving this would complete the intended workflow, from high-level RoboSim models through to executable simulations and machine-checked proofs that are demonstrably consistent by construction.

A Appendices

A.1 Examples

A.1.1 SimpleArm: Pick and Place

A.1.1.1 Requirements analysis for Pick-and-Place case study

The torque of the actuator now needs to be determined by calculating the trajectories of the platform mapping. A trajectory must be chosen so that it is nice enough to satisfy properties such as the velocity always being positive or negative (depending which way the arm is supposed to be moving), as well as not being too steep at the beginning or end. A degree 13 polynomial was chosen in this instance:

$$\begin{aligned}\theta(t) = \cancel{a_0} + \cancel{a_1 t} + \cancel{a_2 t^2} + \cancel{a_3 t^3} + \cancel{a_4 t^4} + \cancel{a_5 t^5} + \cancel{a_6 t^6} + \cancel{a_7 t^7} + \cancel{a_8 t^8} + \cancel{a_9 t^9} + \cancel{a_{10} t^{10}} + \\ + a_{11} t^{11} + a_{12} t^{12} + a_{13} t^{13}\end{aligned} \tag{A.1}$$

Where lower order terms have been crossed out due to the initial condition

$$\theta(0) = \dot{\theta}(t) = \ddot{\theta}(t) = ... = \theta^{(10)}(t) = 0 \tag{A.2}$$

As indicated by the d-model, it is required that the following boundary conditions are imposed at $t = 1$ for the PrePick() trajectory:

$$\theta(1) = a_{11} + a_{12} + a_{13} = \frac{\pi}{4} \tag{A.3}$$

$$\dot{\theta}(1) = 11a_{11} + 12a_{12} + 13a_{13} = 5 \tag{A.4}$$

$$\ddot{\theta}(1) = 110a_{11} + 132a_{12} + 156a_{13} = 0 \tag{A.5}$$

Solving these equations yields

$$\theta(t) = \frac{3153}{2500}t^{11} + \frac{13439}{5000}t^{12} - \frac{7909}{2500}t^{13} \tag{A.6}$$

for $0 \leq t \leq 1$.

The full PrePick() trajectory may then be written as

$$\begin{aligned}\theta(t) = &-\chi_{[0,1]}(t) \times (\frac{3153}{2500}t^{11} + \frac{13439}{5000}t^{12} - \frac{7909}{2500}t^{13}) \\ &- \chi_{[1,2]}(t) \times (\frac{3153}{2500}(2-t)^{11} + \frac{13439}{5000}(2-t)^{12} - \frac{7909}{2500}(2-t)^{13})\end{aligned} \tag{A.7}$$

where χ is an indicator function.

Differentiating twice yields the acceleration:

$$\begin{aligned}\ddot{\theta}(t) = &-\chi_{[0,1]}(t) \times -33t^9 \times \frac{18694t^2 - 13439t - 5255}{1250} \\ &- \chi_{[1,2]}(t) \times 33(2-t)^9 \times \frac{18694(2-t)^2 - 13439(2-t) - 5255}{1250}\end{aligned} \tag{A.8}$$

It now remains to calculate the torque for the actuator with respect to time. Consider Figure A.1, which illustrates the links $L1$, $L2$, the torque τ, the centres of mass x_2, x_3, and the centre of rotation x_1. The easiest way to do this is to make use of the parallel axis theorem:

Theorem 1 (Parallel axis theorem)**.**

$$I = I' + md^2 \tag{A.9}$$

where I and I' are the moments of inertia about a given axis, and a parallel axis which is displaced a distance d from the given axis. m is the mass of the object, which is being rotated about the new axis I.

Therefore, using the information of the p-model and the parallel axis theorem, one can deduce the total moment of inertia I_1 about x_1 by summing the moments of inertia about x_1 for x_2 and x_3 respectively. Let I_2 be the moment of inertia parallel to the moment about x_2 and I_3 be the moment of inertia parallel to the moment about x_3. Then,

$$I_1 = 1.349 + 1 \times 2^2 \tag{A.10}$$

$$I_2 = 0.0208 + 0.5 \times 4.25^2 \tag{A.11}$$

$$\implies I_1 + I_2 + I_3 = 14.40105. \tag{A.12}$$

Now, using Newton's second law:

$$\begin{aligned}\tau_{PrePick()} = I_1\ddot{\theta}(t) = 14.40105 \times (&-\chi_{[0,1]}(t) \times -33t^9 \times \frac{18694t^2 - 13439t - 5255}{1250} \\ &- \chi_{[1,2]}(t) \times 33(2-t)^9 \times \frac{18694(2-t)^2 - 13439(2-t) - 5255}{1250})\end{aligned} \tag{A.13}$$

Symmetry of the trajectory yields the following torques for PrePlace() and Return():

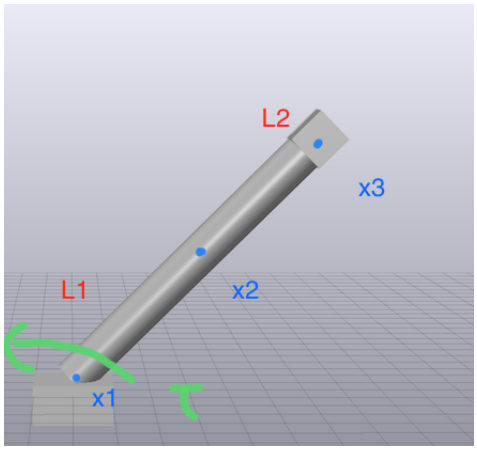


Figure A.1: The SimpleArm p-model, and d-model being animated in the Drake simulator.

$$\tau_{PrePlace()} = I_1\ddot{\theta}(t) = -2 \times 14.40105 \times (-\chi_{[0,1]}(t) \times -33t^9 \times \frac{18694t^2 - 13439t - 5255}{1250} - \chi_{[1,2]}(t) \times 33(2-t)^9 \times \frac{18694(2-t)^2 - 13439(2-t) - 5255}{1250}) \tag{A.14}$$

$$\tau_{Return()} = I_1\ddot{\theta}(t) = 14.40105 \times (-\chi_{[0,1]}(t) \times -33t^9 \times \frac{18694t^2 - 13439t - 5255}{1250} - \chi_{[1,2]}(t) \times 33(2-t)^9 \times \frac{18694(2-t)^2 - 13439(2-t) - 5255}{1250}) \tag{A.15}$$

A.1.1.2 Computation Pipeline

This section presents the computation pipeline for the SimpleArm Pick and Place case study, including key intermediate quantities, solver steps, and integration scheme.

A.1.2 Acrobot: Swing-up Control

A.1.2.1 T5 Stub Implementations

T5 generates skeleton files with `TODO/STUB` markers. Because we do not assume a mapping generator exists, the stubs were completed manually by transcribing the p-model and mapping specification into C/C++ as follows:

1. `platform_mapping.h`:
   define `mapping_state_t` to mirror the p-model sensor/actuator hierarchy, separating world properties from platform sensors/actuators.

2. `platform_mapping_adapter.cpp`: copy world-dependent sensors from `sensor_data_t` and platform sensors directly from `IPlatformState` into `p_mapping`.

3. `world_mapping.cpp`: compute only world-dependent sensors (gravity), leaving joint encoders to the adapter.

4. `world_engine.cpp`: initialise a minimal static world (constant gravity) with an empty `update()`.

5. `dmodel_interface.h/cpp` and `orchestrator.cpp`:
   wire `registerRead/registerWrite/tock` to the controller and implement the cycle semantics.

D-model interface (`dmodel_interface.h/cpp`) Defines the C ABI for communication between the orchestrator and the d-model controller. The header declares event types and functions:

```
// Input event types (engine -> d-model)
enum {
   INPUT_SHOULDER_ANGLE, INPUT_SHOULDER_VELOCITY,
   INPUT_ELBOW_ANGLE, INPUT_ELBOW_VELOCITY,
   INPUT_DONE, INPUT_TERMINATE
};
// Output operation types (d-model -> engine)
enum { OUTPUT_CONTROL_IN, OUTPUT_DONE };

// D-model C ABI functions
bool registerRead(int* type, double* value);
void registerWrite(int operation_type, double parameter, double current_time);
```

Listing A.1: D-model interface header (manual implementation)

The implementation (`dmodel_interface.cpp`) delegates to an active `IDModelIO` instance set by the orchestrator.

Platform mapping header (`platform_mapping.h`) Defines the C struct that mirrors the p-model sensor/actuator hierarchy. World properties (gravity) and platform sensors (joint encoders) are clearly separated:

```
typedef struct mapping_state_t {
   // World properties (from world_mapping.cpp)
   struct { double gravity_world[3]; } World;
   // Platform sensors and actuators (from platform state)
   struct {
      struct {
         struct { double AngleOut; double VelocityOut; } ShoulderEncoder;
      } BaseLink;
      struct {
         struct { struct { double ControlIn; } ElbowMotor; } ElbowJoint;
         struct { double AngleOut; double VelocityOut; } ElbowEncoder;
      } Link1;
   } AcrobotControlled;
} mapping_state_t;
// NOTE: Dynamics (M_inv, bias) are computed internally by the controller.
```

Listing A.2: Platform mapping structure (manual implementation)

World engine (`world_engine.cpp`) Implements `IWorldEngine` to provide environment state. For the Acrobot, this only supplies constant gravity:

```
class MinimalWorldEngine final : public IWorldEngine {
   void initialise() override {
```

```
        p_mapping.World.gravity_world[0] = 0.0;
        p_mapping.World.gravity_world[1] = 0.0;
        p_mapping.World.gravity_world[2] = -9.81;
    }
    void update() override {}
};
```

Listing A.3: World engine (manual implementation)

World mapping (`world_mapping.cpp`) Implements `IPlatformWorldMapping` to compute *world-dependent* sensors only (e.g., gravity, object positions). Pure platform sensors (joint encoders) are *not* handled here:

```
void computeSensorReadings(const IWorldState&, const IPlatformState&,
                    sensor_data_t& out) override {
    // World sensor only - gravity vector
    out.World.gravity_world[0] = 0.0;
    out.World.gravity_world[1] = 0.0;
    out.World.gravity_world[2] = -9.81;
    // NOTE: Platform sensors (joint encoders) are NOT set here.
    // They are copied directly from platform state by platform_mapping_adapter.cpp.
}
```

Listing A.4: World mapping: only world-dependent sensors (manual implementation)

Platform mapping adapter (`platform_mapping_adapter.cpp`) Copies sensor values into the platform mapping from two sources: world sensors from `sensor_data_t`, and platform sensors directly from platform state:

```
void updateFromSensors(const sensor_data_t& sensor_output) override {
    // Copy world sensors (from world_mapping.cpp)
    p_mapping.World.gravity_world[0] = sensor_output.World.gravity_world[0];
    // ...
}
void updateFromPlatformState(const IPlatformState& platform_state) override {
    // Copy platform sensors directly from platform state
    const platform1::State& st = static_cast<const platform1::State&>(platform_state);
    p_mapping.AcrobotControlled.BaseLink.ShoulderEncoder.AngleOut =
        st.BaseLink_ShoulderEncoder_AngleOut;
    // ... (other joint encoders)
}
```

Listing A.5: Platform mapping adapter: separates world and platform sensors

Orchestrator (`orchestrator.cpp`) The most substantial manual file (∼700 lines). Implements the main simulation loop: initialisation, threading for the d-model, the physics update cycle, and sensor/actuator routing via `registerRead`/`registerWrite`. The core loop structure follows the RoboSim semantics:

```
// Semantic outline from orchestrator.cpp:
// Init;
// mu X . SendToDModel;
// (ReceiveFromDModel interrupted by Wait(cycle));
// ResetTimer;
// Evolve interrupted by (timer >= cycle);
// X
```

Listing A.6: Orchestrator main loop structure (manual implementation)

D-model controller adaptation In addition to the T5 stubs, the generated d-model controller requires adaptation to integrate with the `dmodel_interface` ABI. The following listings show the exact additional C lines inserted beyond the RoboSim generator output to integrate the controller with the physics engine orchestrator. These blocks are extracted from `AcrobotSwingUpLQRModule_generated.c`.

```
// MANUAL: orchestrator bridge needs d-model interface API
#include "dmodel_interface.h"

// MANUAL: Wrap angle to [-pi, pi] range
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
static inline float wrap_angle(float v) {
   while (v > M_PI) v -= 2.0f * M_PI;
   while (v < -M_PI) v += 2.0f * M_PI;
   return v;
}
```

Listing A.7: Manual bridge additions at the top of the generated controller C file (include and angle-wrapping helper)

```
// MANUAL: Accumulate sensor readings from platform via registerRead()
struct AcrobotStateUpdate _value_ = (struct AcrobotStateUpdate) {
   .dtheta1 = 0.0,
   .theta1 = 0.0,
   .theta2 = 0.0,
   .dtheta2 = 0.0
};

while (!inputdone) {
   int input_type;
   double input_value;
   if (!registerRead(&input_type, &input_value)) {
      // propagate termination into the generated channel protocol
      pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_write);
      start_AcrobotSwingUpLQRModule->value =
          create_M_AcrobotSwingUpLQRModule_input__terminate_();
      pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_read);
      terminate__ = true;
      inputdone = true;
      break;
   }
   switch (input_type) {
      case INPUT_SHOULDER_ANGLE:
         // ShoulderEncoder = TIP joint = Drake's theta2
         // MANUAL: Wrap angle to [-pi, pi]
         _value_.theta2 = wrap_angle((float)input_value);
         break;
      case INPUT_SHOULDER_VELOCITY:
         _value_.dtheta2 = (float)input_value;
         break;
      case INPUT_ELBOW_ANGLE:
         // ElbowEncoder = BASE joint = Drake's theta1
         // MANUAL: Wrap angle to [-pi, pi]
         _value_.theta1 = wrap_angle((float)input_value);
         break;
      case INPUT_ELBOW_VELOCITY:
         _value_.dtheta1 = (float)input_value;
         break;
      case INPUT_DONE:
         // send the consolidated stateUpdate input into the generated channel protocol
         pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_write);
         start_AcrobotSwingUpLQRModule->value =
             create_M_AcrobotSwingUpLQRModule_input_stateUpdate(_value_);
         pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_read);
         // then send the generated internal _done_ marker
         pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_write);
         start_AcrobotSwingUpLQRModule->value =
             create_M_AcrobotSwingUpLQRModule_input__done_();
         pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_read);
```

```
47            inputdone = true;
48            break;
49         case INPUT_TERMINATE:
50            pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_write);
51            start_AcrobotSwingUpLQRModule->value =
                  create_M_AcrobotSwingUpLQRModule_input__terminate_();
52            pthread_barrier_wait(&start_AcrobotSwingUpLQRModule->can_read);
53            terminate__ = true;
54            inputdone = true;
55            break;
56         default:
57            break;
58      }
59   }
```

Listing A.8: Manual replacement of stdin-based input with registerRead-based sensor events (plus end-of-cycle handling)

```
1  if (_output_.type == M_AcrobotSwingUpLQRModule_output_ApplyTorque) {
2     float _aux1_ = _output_.data.ApplyTorque.v1;
3     // MANUAL: Send actuator command to orchestrator
4     registerWrite(OUTPUT_CONTROL_IN, (double)_aux1_, (double)cycle_count * 0.005);
5  } else if (_output_.type == M_AcrobotSwingUpLQRModule_output__done_) {
6     // MANUAL: Send done signal to orchestrator
7     registerWrite(OUTPUT_DONE, 0.0, 0.0);
8     outputdone = true;
9  }
```

Listing A.9: Manual replacement of stdout-based outputs with registerWrite actuator commands

```
1  // Generated RoboSim controller
2  extern "C" int acrobot_generated_main(int argc, char* argv[]);
3  // Manual implementation (for comparison)
4  // extern "C" int acrobot_main(int argc, char* argv[]);
5
6  // ...
7
8  int pickplace_exit_code = acrobot_generated_main(argc, argv);
```

Listing A.10: Manual changes in orchestrator.cpp to call the generated controller entry point

```
1  add_executable(physics_test
2     src/orchestrator.cpp
3     src/dmodel_interface.cpp
4     # Toggle between generated and manual controller:
5     src/AcrobotSwingUpLQRModule_generated.c # Generated from RoboSim
6     # src/acrobot_controller.c # Manual implementation
7  )
```

Listing A.11: Manual change in CMakeLists.txt to link the generated controller instead of the manual controller

```
1  float acrobot_cos(float x) {
2     return cosf(x);
3  }
4  float acrobot_sin(float x) {
5     return sinf(x);
6  }
```

Listing A.12: Manual renaming of sin/cos stubs to avoid collisions with libm and ensure standard math is used

A.2 Simulation Framework Appendix

A.2.1 Technique 3: Equation Composition

A.2.1.1 T3S1: Formulation Detection

This appendix presents the algorithms for detecting sensor and actuator formulations, which complement the dynamics formulation detection shown in Algorithm 3 of the main text. These algorithms build maps that associate each sensor and actuator's fully qualified name with its detected formulation type.

For reference sensors, the algorithm checks whether the definition string contains "physmod::trivial::sensors" for trivial sensors; any other reference is marked as unknown. For reference actuators, it checks for "physmod::trivial::actuators" for trivial actuators; any other reference is marked as unknown. For local components, the algorithms check whether the definition name matches known trivial patterns ("TrivialSensor" for sensors, "TrivialMotor" for actuators); any other local definition is marked as unknown. These checks ensure that each sensor and actuator is assigned an appropriate formulation type that will be used in the subsequent equation generation steps.

Algorithm 16: findSensorFormulations

```
   Input: pmodel : PModel
   Output: sensorFormulationMap : Map⟨String, String⟩
1  sensorFormulationMap = {};
   /* Step 1: Iterate through all sensors                              */
2  foreach sensor ∈ pmodel.sensors do
3      sensorFQN = getFullyQualifiedName(sensor);
       /* Step 2: Check if reference sensor                            */
4      if sensor instanceof ReferenceSensor then
5          defString = sensor.def.toString();
6          if defString.contains("physmod::trivial") then
7              sensorFormulationMap.put(sensorFQN, "trivial");
8          end
9          else
10             sensorFormulationMap.put(sensorFQN, "unknown");
11         end
12     end
       /* Step 3: Check if local sensor                                */
13     else if sensor instanceof LocalSensor then
14         defName = sensor.def.name;
15         if defName = "TrivialSensor" then
16             sensorFormulationMap.put(sensorFQN, "trivial");
17         end
18         else
19             sensorFormulationMap.put(sensorFQN, "unknown");
20         end
21     end
22 end
23 return sensorFormulationMap;
```

Algorithm 17: findActuatorFormulations

Input: *pmodel* : *PModel*
Output: *actuatorFormulationMap* : *Map*⟨*String*, *String*⟩

```
1  actuatorFormulationMap = {};
   /* Step 1:  Iterate through all actuators                          */
2  foreach actuator ∈ pmodel.actuators do
3      actuatorFQN = getFullyQualifiedName(actuator);
       /* Step 2:  Check if reference actuator                        */
4      if actuator instanceof ReferenceActuator then
5          defString = actuator.def.toString();
6          if defString.contains("physmod::trivial") then
7              actuatorFormulationMap.put(actuatorFQN, "trivial");
8          end
9          else
10             actuatorFormulationMap.put(actuatorFQN, "unknown");
11         end
12     end
       /* Step 3:  Check if local actuator                            */
13     else if actuator instanceof LocalActuator then
14         defName = actuator.def.name;
15         if defName = "TrivialMotor" then
16             actuatorFormulationMap.put(actuatorFQN, "trivial");
17         end
18         else
19             actuatorFormulationMap.put(actuatorFQN, "unknown");
20         end
21     end
22 end
23 return actuatorFormulationMap;
```

A.2.1.2 T3S2: Localise Reference Components

This subsection provides the detailed algorithms for converting reference components (links, sensors, actuators, and bodies) to their local counterparts. The main algorithm `localiseRefs` is presented in the main text (Algorithm 4), which calls component-specific conversion procedures. The joint conversion procedure is detailed in Algorithm 5 of the main text. This appendix presents the analogous procedures for the remaining component types.

The conversion procedures follow the same pattern as the joint conversion: they scan the model to collect reference components, create local counterparts using helper functions, and perform in-place substitution. Each conversion ensures that subsequent stages operate only on localised components, which can be augmented with additional variables and equations.

Algorithm 18 converts reference links to local links. The algorithm iterates through `pmodel.links` directly (lines 1–8), collecting reference links and creating local replacements. In-place substitution occurs at lines 10–14.

Algorithm 18: convertReferenceLinksToLocal

Input: *pmodel* : *PModel*
Output: *pmodel* with all reference links converted to local links

```
1  linksToReplace = {};
2  newLinks = {};
3  foreach link ∈ pmodel.links do
4      if link instanceof ReferenceLink then
5          localLink = createLocalLink(link);
6          linksToReplace.add(link);
7          newLinks.add(localLink);
8      end
9  end
   /* Replace reference links with local links                    */
10 foreach i ∈ [0, size(linksToReplace)) do
11     index = pmodel.links.indexOf(linksToReplace[i]);
12     if index ≥ 0 then
13         pmodel.links[index] = newLinks[i];
14     end
15 end
16 return pmodel;
```

The `createLocalLink` helper (Algorithm 19) allocates a new LocalLink, copies the name and pose (lines 1–3), deep-copies the definition if present, and unsets the definition's name to avoid duplicate identifiers (lines 4–6).

Algorithm 19: createLocalLink

Input: *refLink* : *ReferenceLink*
Output: *localLink* : *LocalLink*

```
1  localLink = new LocalLink();
2  localLink.name = refLink.name;
3  localLink.pose = copy(refLink.pose);
4  if refLink.def ≠ null then
5      localLink.def = copy(refLink.def);
6      localLink.def.unset("name");
7  end
8  return localLink;
```

Algorithm 20 converts reference sensors to local sensors. The p-model metamodel allows sensors to be contained within both links and joints, so the algorithm iterates through links and their sensor lists (lines 1–13) as well as each joint's sensor list (lines 14–26). Reference sensors are collected and converted, then substituted in place within each container's sensor list.

Algorithm 20: convertReferenceSensorsToLocal

Input: *pmodel* : *PModel*
Output: *pmodel* with all reference sensors converted to local sensors

```
1  foreach link ∈ pmodel.links do
       /* Convert sensors at link scope                                  */
2      sensorsToReplace = {}; newSensors = {};
3      foreach sensor ∈ link.sensors do
4          if sensor instanceof ReferenceSensor then
5              sensorsToReplace.add(sensor);
6              newSensors.add(createLocalSensor(sensor));
7          end
8      end
9      foreach i ∈ [0, size(sensorsToReplace)) do
10         index = link.sensors.indexOf(sensorsToReplace[i]);
11         if index ≥ 0 then
12             link.sensors[index] = newSensors[i];
13         end
14     end
       /* Convert sensors at joint scope                                 */
15     foreach joint ∈ link.joints do
16         if joint instanceof LocalJoint then
17             jSensorsToReplace = {}; newJSensors = {};
18             foreach sensor ∈ joint.sensors do
19                 if sensor instanceof ReferenceSensor then
20                     jSensorsToReplace.add(sensor);
21                     newJSensors.add(createLocalSensor(sensor));
22                 end
23             end
24             foreach i ∈ [0, size(jSensorsToReplace)) do
25                 index = joint.sensors.indexOf(jSensorsToReplace[i]);
26                 if index ≥ 0 then
27                     joint.sensors[index] = newJSensors[i];
28                 end
29             end
30         end
31     end
32 end
33 return pmodel;
```

The `createLocalSensor` helper (Algorithm 21) follows the same pattern: allocate a new LocalSensor, copy the name and pose, and deep-copy the definition with name unset.

Algorithm 21: createLocalSensor

Input: *refSensor* : *ReferenceSensor*
Output: *localSensor* : *LocalSensor*

```
1  localSensor = new LocalSensor();
2  localSensor.name = refSensor.name;
3  localSensor.pose = copy(refSensor.pose);
4  if refSensor.def ≠ null then
5      localSensor.def = copy(refSensor.def);
6      localSensor.def.unset("name");
7  end
8  return localSensor;
```

Algorithm 22 converts reference actuators to local actuators. Like sensors, actuators may be contained within both links and joints, so the algorithm iterates through links and their actuator lists as well as each joint's actuator list.

Algorithm 22: convertReferenceActuatorsToLocal

Input: *pmodel* : *PModel*
Output: *pmodel* with all reference actuators converted to local actuators

```
1  foreach link ∈ pmodel.links do
       /* Convert actuators at link scope                                  */
2      actuatorsToReplace = {}; newActuators = {};
3      foreach actuator ∈ link.actuators do
4          if actuator instanceof ReferenceActuator then
5              actuatorsToReplace.add(actuator);
6              newActuators.add(createLocalActuator(actuator));
7          end
8      end
9      foreach i ∈ [0, size(actuatorsToReplace)) do
10         index = link.actuators.indexOf(actuatorsToReplace[i]);
11         if index ≥ 0 then
12             link.actuators[index] = newActuators[i];
13         end
14     end
       /* Convert actuators at joint scope                                 */
15     foreach joint ∈ link.joints do
16         if joint instanceof LocalJoint then
17             jActuatorsToReplace = {}; newJActuators = {};
18             foreach actuator ∈ joint.actuators do
19                 if actuator instanceof ReferenceActuator then
20                     jActuatorsToReplace.add(actuator);
21                     newJActuators.add(createLocalActuator(actuator));
22                 end
23             end
24             foreach i ∈ [0, size(jActuatorsToReplace)) do
25                 index = joint.actuators.indexOf(jActuatorsToReplace[i]);
26                 if index ≥ 0 then
27                     joint.actuators[index] = newJActuators[i];
28                 end
29             end
30         end
31     end
32 end
33 return pmodel;
```

The `createLocalActuator` helper (Algorithm 23) follows the established pattern.

Algorithm 23: createLocalActuator

Input: *refActuator* : *ReferenceActuator*
Output: *localActuator* : *LocalActuator*

```
1  localActuator = new LocalActuator();
2  localActuator.name = refActuator.name;
3  localActuator.pose = copy(refActuator.pose);
4  if refActuator.def ≠ null then
5      localActuator.def = copy(refActuator.def);
6      localActuator.def.unset("name");
7  end
8  return localActuator;
```

Algorithm 24 converts reference bodies to local bodies. Bodies are also contained within links, following the same nested iteration pattern as sensors and actuators.

Algorithm 24: convertReferenceBodiesToLocal

Input: *pmodel* : *PModel*
Output: *pmodel* with all reference bodies converted to local bodies

```
1  foreach link ∈ pmodel.links do
2      bodiesToReplace = {};
3      newBodies = {};
4      foreach body ∈ link.bodies do
5          if body instanceof ReferenceBody then
6              localBody = createLocalBody(body);
7              bodiesToReplace.add(body);
8              newBodies.add(localBody);
9          end
10     end
       /* Replace reference bodies with local bodies                    */
11     foreach i ∈ [0, size(bodiesToReplace)) do
12         index = link.bodies.indexOf(bodiesToReplace[i]);
13         if index ≥ 0 then
14             link.bodies[index] = newBodies[i];
15         end
16     end
17 end
18 return pmodel;
```

The `createLocalBody` helper (Algorithm 25) completes the set of conversion helpers, following the same pattern as the other components.

Algorithm 25: createLocalBody

Input: *refBody* : *ReferenceBody*
Output: *localBody* : *LocalBody*

```
1  localBody = new LocalBody();
2  localBody.name = refBody.name;
3  localBody.pose = copy(refBody.pose);
4  if refBody.def ≠ null then
5      localBody.def = copy(refBody.def);
6      localBody.def.unset("name");
7  end
8  return localBody;
```

A.2.1.3 T3S3: Validate Generator Compatibility

The validation step ensures that the p-model meets the preconditions required by the detected formulation before equation generation proceeds. The main validation algorithm is presented in the main text (Algorithm 7) for the SKO formulation.

Different formulations may have different validation requirements. For the SKO formulation, the validator enforces three categories of checks. First, basic structure checks verify that links exist and are non-null, and that all links have valid names. Second, pose requirements ensure that every link has a pose definition, making pose information available for spatial transform calculations. Third, inertia requirements verify that each link has either direct inertial definition or at least one body with inertial properties, ensuring sufficient inertial information for dynamics calculations.

If any validation check fails, the generator raises a validation exception and aborts equation generation. This early detection prevents downstream errors and provides clear feedback about missing or invalid model information.

Other formulations may require additional or different validation checks. For example, formulations that require specific joint types or constraints would add corresponding validation logic.

A.2.1.4 T3S4: Determine Topology

The topology calculation step identifies the structural connectivity of the robot model, which influences the equation generation strategy. The main algorithm `calculateTopology` is presented in the main text (Algorithm in Section 3.2.5).

The algorithm distinguishes three topology categories: serial chains, where the number of links equals the number of joints plus one with simple linear connectivity; kinematic trees, where the number of links equals the number of joints plus one with branching structure (multiple fixed connections); and closed chains, where the number of links does not satisfy the open-chain relationship, indicating kinematic loops.

The detection logic uses two criteria. First, the link-joint relationship checks if #links $=$ #joints $+ 1$ to distinguish open versus closed chains. Second, for open chains, the connectivity pattern is examined by checking the number of fixed connections to distinguish serial chains from kinematic trees.

The topology information is used in subsequent steps to select appropriate equation generation algorithms and traversal strategies. For example, serial chains can use specialised recursive algorithms, while kinematic trees require more general tree traversal, and closed chains may require constraint embedding or loop-cutting techniques.

A.2.1.5 T3S5: Assign Numbering

The numbering step assigns sequential indices to components based on the detected formulation and topology. The main text (Section 3.2.6) describes the purpose and high-level approach. This appendix provides additional algorithmic details.

Different formulations require different numbering strategies. The Featherstone formulation uses regular numbering (base to tip), while the SKO formulation uses reversed numbering (tip to base) as per [24]. Other formulations may use alternative numbering schemes based on their mathematical structure.

The numbering process involves several key algorithms: `assignNumbering`, which dispatches to the appropriate numbering strategy based on formulation type; `orderPmodel`, which assigns sequential numbers to links and joints following the chosen traversal order (regular or reversed); `getBaseLink`, which identifies the base link by finding the link with the most child links in the model structure; and `getChildLinks`, which recursively identifies all descendant links for a given link in the model hierarchy.

The following algorithms implement the numbering process:

Algorithm 26: assignNumbering

Input: *pmodel* : *PModel*, *formulation* : *Formulation*
Output: *pmodel′* : *PModel* with numbering assigned

```
1  switch formulation do
2      case Formulation :: SKO do
3          return orderPmodel(pmodel, Ordering :: reverse);
4      end
5      case Formulation :: Featherstone do
6          return orderPmodel(pmodel, Ordering :: regular);
7      end
8      otherwise do return null;
9      ;
10 end
```

The main `assignNumbering` algorithm (Algorithm 26) acts as a dispatcher that selects the appropriate numbering strategy based on the detected formulation. For the SKO formulation, it calls `orderPmodel` with reversed ordering (tip to base); for the Featherstone formulation, it calls `orderPmodel` with regular ordering (base to tip).

Algorithm 27: orderPmodel

Input: *pmodel* : *PModel*, *ordering* : *Ordering*
Output: *pmodel′* : *PModel* with sequential numbering assigned

```
1  baseLink = getBaseLink(pmodel);
2  loopLinks = {baseLink};
3  spanningList = getChildLinks(pmodel, baseLink, loopLinks);
4  spanningList.addFirst(baseLink);
5  if ordering = Ordering :: reverse then
6  |  spanningList.reverse();
7  end
8  n = 0;
   /* Assign number to base link                                   */
9  if ordering = Ordering :: regular then
10 |  addNumber(baseLink, n, pmodel);
11 else
12 |  addNumber(baseLink, size(pmodel.links), pmodel);
13 end
14 addPoseVectorVariable(baseLink, pmodel);
   /* Assign numbers to bodies in base link                        */
15 bodyNum = 0;
16 foreach body ∈ baseLink.bodies do
17 |  addNumber(body, bodyNum, pmodel);
18 |  bodyNum = bodyNum + 1;
19 end
   /* Assign numbers to remaining links and joints                 */
20 loopLinks = {};
21 loopJoints = {};
22 for i ∈ [0, size(spanningList) − 1) do
23 |  foreach joint ∈ spanningList[i + 1].joints do
24 |  |  if joint.flexible.link ∉ loopLinks then
25 |  |  |  n = n + 1;
26 |  |  |  addNumber(joint, n, pmodel);
27 |  |  |  addPoseVectorVariable(joint, pmodel);
28 |  |  |  addNumber(joint.flexible.link, n, pmodel);
29 |  |  |  addPoseVectorVariable(joint.flexible.link, pmodel);
30 |  |  |  loopLinks.add(joint.flexible.link);
   |  |  |  /* Assign numbers to bodies                           */
31 |  |  |  bodyNum = 0;
32 |  |  |  foreach body ∈ joint.flexible.link.bodies do
33 |  |  |  |  addNumber(body, bodyNum, pmodel);
34 |  |  |  |  bodyNum = bodyNum + 1;
35 |  |  |  end
36 |  |  end
37 |  |  else
38 |  |  |  loopJoints.add(joint);
39 |  |  end
40 |  end
41 end
   /* Assign numbers to loop-closing joints                        */
42 foreach joint ∈ loopJoints do
43 |  addNumber(joint, n, pmodel);
44 |  addPoseVectorVariable(joint, pmodel);
45 end
46 return pmodel;
```

The `orderPmodel` algorithm (Algorithm 27) performs the actual numbering assignment by traversing the kinematic structure in the specified order. It first identifies the base link and constructs a spanning list of all links using depth-first traversal. For reversed ordering (used by SKO), the spanning list is reversed. The algorithm then assigns sequential numbers to joints

and links during traversal, handling loop-closing joints separately. Bodies within each link are also numbered sequentially.

Algorithm 28: getBaseLink

```
Input: pmodel : PModel
Output: baseLink : Link
baseLink = pmodel.links[0];
maxContainedLinks = 0;
foreach link ∈ pmodel.links do
    loopLinks = {};
    containedLinks = getChildLinks(pmodel, link, loopLinks);
    if size(containedLinks) > maxContainedLinks then
        maxContainedLinks = size(containedLinks);
        baseLink = link;
    end
end
return baseLink;
```

The `getBaseLink` algorithm (Algorithm 28) identifies the base link by finding the link with the most descendant links in the kinematic tree. It initialises the base link to the first link in the p-model so that single-link models are handled correctly, then iterates through all links, counts their descendants using `getChildLinks`, and replaces the current base only when a link has a strictly greater descendant count. This preserves first-encountered tie-breaking while avoiding a null base link for single-link models.

Algorithm 29: getChildLinks

```
Input: pmodel : PModel, link : Link, loopLinks : Set⟨Link⟩
Output: containedLinks : List⟨Link⟩
containedLinks = {};
if link.joints = null then
    return containedLinks;
end
foreach joint ∈ link.joints do
    if joint.flexible ≠ null then
        childLink = joint.flexible.link;
        if childLink ∉ loopLinks then
            containedLinks.add(childLink);
            loopLinks.add(childLink);
            /* Recursively add descendants                    */
            descendants = getChildLinks(pmodel, childLink, loopLinks);
            containedLinks.addAll(descendants);
        end
    end
end
return containedLinks;
```

The `getChildLinks` algorithm (Algorithm 29) recursively identifies all descendant links for a given parent link. It traverses the kinematic tree by following joint connections, tracking visited links in `loopLinks` to prevent infinite recursion in closed-chain structures. For each child link discovered, it recursively collects the child's descendants and combines them into the result.

The numbering process proceeds in several steps: first identifying the base link using `getBaseLink`, then constructing the spanning tree of the robot structure, traversing the tree in the order required by the formulation (regular or reversed), assigning sequential indices

to links and joints during traversal, and storing the numbering information in the p-model for use in equation generation. The numbering establishes the mathematical order of operations for recursive dynamics algorithms and determines the indexing of system-level vectors and matrices.

A.2.1.6 T3S6: Calculate Equations

This subsection provides the detailed helper functions, formulation-specific algorithms, and equation library interfaces used during equation generation. The main text (Section 3.2.7) presents the overall `calculateEquations` procedure (Algorithm 9) and its substeps.

The appendix is organised into three main parts: SKO formulation-specific equation generation algorithms, component lookup functions, and equation library functions.

A.2.1.7 SKO Formulation: Four-Step Equation Generation

When the equation library processes a p-model with SKO formulation and serial chain topology (as detected in Steps 1 and 4 of Technique 3), it invokes a four-step equation generation process. This structure corresponds directly to the reference implementation in the equation composition generator. Each step progressively builds the mathematical model by adding datatypes, variables, component equations, and system-level equations. The following algorithms detail these four steps for the SKO formulation.

Algorithm 30: defineDatatypesAndFunctions (SKO formulation)

Input: *pmodel* : *PModel*
Output: *pmodel* : *PModel* with datatypes and functions added

`/* Add Geom datatype for geometry representation */`
1 Geom ← Geom(*geomType*:`string`,*geomVal*:`vector(real)`) ;
2 pmodel.addDatatype(Geom) ;

`/* Add utility functions for matrix operations */`
3 $identity_func \leftarrow identity(size : \mathbb{R}) : \mathbb{R}^{3\times3}$;
4 pmodel.addFunction(identity_func) ;
5 $zeroes_func \leftarrow zeroes(size : \mathbb{R}) : \mathbb{R}^{3\times3}$;
6 pmodel.addFunction(zeroes_func) ;

`/* Add Phi function for spatial transform calculations */`
7 $Phi_func \leftarrow Phi(m : \mathbb{R}, n : \mathbb{R}, B_k : \text{Seq}(\mathbb{R}^{4\times4})) : \mathbb{R}^{6\times6}$;
8 pmodel.addFunction(Phi_func) ;

`/* Add zero vector and matrix constructors */`
9 $zeroVec_func \leftarrow zeroVec(a : \mathbb{R}) : \mathbb{R}^{1}$;
10 pmodel.addFunction(zeroVec_func) ;
11 $zeroMat_func \leftarrow zeroMat(a : \mathbb{R}, b : \mathbb{R}) : \mathbb{R}^{0\times0}$;
12 pmodel.addFunction(zeroMat_func) ;
13 **return** *pmodel*;

The first step creates the formulation-specific datatype and utility functions. The `Geom` datatype encapsulates geometric information (type and value) used in body definitions. The utility functions support matrix operations (`identity`, `zeroes`), spatial transforms (`Phi`), and zero constructors (`zeroVec`, `zeroMat`) required by the SKO formulation's recursive algorithms.

Algorithm 31: defineSystemVariables (SKO formulation)

Input: *pmodel* : *PModel*
Output: *pmodel* : *PModel* with system-level variables added

```
/* Create constant n for number of links                                  */
n ← |pmodel.links|;
pmodel.addConstant("n", n);
/* Declare system-level variables                                         */
B_k ← Seq(ℝ^{4×4}) // Frame transforms
XJ_k ← Seq(ℝ^{6×6}) // Spatial transforms
V, f, a, b, α ← ℝ^{6n} // Spatial vectors
φ, M ← ℝ^{6n×6n} // Spatial matrices
θ, τ, C ← ℝ^N // Joint space vectors
H ← ℝ^{N×6n} // Joint coupling matrix
M_mass ← ℝ^{N×N} // Mass matrix
N ← ℝ // Total DOF count
/* Add all variables to pmodel                                            */
foreach var ∈ {B_k, XJ_k, V, f, a, b, α, φ, M, θ, τ, C, H, M_mass, N} do
    pmodel.addInOut(var);
end
return pmodel;
```

The second step declares system-level variables that serve as containers for aggregating component information. These include frame and spatial transforms (B_k, XJ_k), spatial vectors representing velocities, forces, accelerations, and biases (V, f, a, b, α), spatial matrices (ϕ, M), joint-space quantities (θ, τ, C), and coupling matrices (H, M_{mass}). For the SimpleArm example with $n = 2$ links, the spatial vectors have dimension $6n = 12$ and the mass matrix has dimension $N \times N = 2 \times 2$.

Algorithm 32: componentEquations (SKO formulation)

Input: *pmodel* : *PModel*
Output: *pmodel* : *PModel* with component variables, relations, and equations added

$n \leftarrow |pmodel.links|$;
/* First pass: Declare variables for all components */
foreach $k \in [1..n]$ **do**
  /* Create link-local variables: $m_k, I_k, M_k, a_k, b_k, V_k, \alpha_k, f_k, B_k$ */
  declareLinkVariables($pmodel.getLink(k)$, k);
  /* For each body in link, create geometry variables with numbering */
  $bodyNum \leftarrow 0$;
  foreach $body \in pmodel.getLink(k).bodies$ **do**
    createGeometryVariable($body$, k, $bodyNum$);
    $bodyNum \leftarrow bodyNum + 1$;
  end
  /* Create joint variables if not last link */
  if $k < n$ **then**
    declareJointVariables($pmodel.getJoint(k)$, k);
    /* Calculate r_v_k from rank of joint's H matrix */
    $r_v_k \leftarrow rank(pmodel.getJoint(k).H)$;
    pmodel.addInOut(r_v_k);
  end
end
/* Second pass: Create equations, relations, and constraints */
foreach $k \in [1..n]$ **do**
  /* Add H_k from joint definitions */
  if $k < n$ **then**
    $H_k \leftarrow pmodel.getJoint(k).H$;
    pmodel.getLink(k).addInOut(H_k);
  end
  /* Create relations between system and link variables */
  $link_k \leftarrow pmodel.getLink(k)$;
  $ctx \leftarrow$ fullyQualifiedName($link_k$);
  /* Construct link relation: conjunction of 7 equalities */
  $relation \leftarrow "B_k == link_k.B_k \wedge V_k == link_k.V_k \wedge \ldots \wedge \tau[k-1] == link_k.\tau_k"$;
  createRelation($link_k$, ctx, $relation$);
  /* If link has flexible joint, add flexible relation */
  if $k < n$ **then**
    $flexiRelation \leftarrow "B_k == link_k.B_k \wedge T_{k+1,k} == link_k.T_{k+1,k} \wedge \ldots"$;
    createFlexiRelation($link_k$, ctx, $flexiRelation$);
  end
  /* Create link dynamics equations */
  createLinkEquations($link_k$, k);
  /* Create pose homogeneous matrix constraint */
  createPoseConstraint($link_k$, k);
  /* Create spatial inertia composition equations */
  createInertiaEquations($link_k$, k);
  /* Create joint relations and equations */
  if $k > 1$ **then**
    $joint_{k-1} \leftarrow pmodel.getJoint(k-1)$;
    $jctx \leftarrow$ fullyQualifiedName($joint_{k-1}$);
    /* Construct joint relation: conjunction of 3 equalities */
    $jrelation \leftarrow "B_k == joint_{k-1}.B_k \wedge V_k == joint_{k-1}.V_k \wedge \ldots"$;
    createRelation($joint_{k-1}$, $jctx$, $jrelation$);
    createJointEquations($joint_{k-1}$, k);
  end
end
return *pmodel*;

The third step uses a two-pass approach to handle dependency ordering. The first pass declares component-local variables for links (mass m_k, inertia I_k, spatial inertia M_k, dynamic vectors

$a_k, b_k, V_k, \alpha_k, f_k, B_k$) and joints, including geometry variables for each body with sequential numbering. The second pass constructs relations that connect system-level variables to component-level variables. For links, a relation establishes a conjunction of seven equality constraints binding system quantities to link quantities; for links with flexible joints, an additional flexible relation is created. The pass also generates component-level dynamics equations, pose homogeneity constraints, spatial inertia composition equations, and joint relations. For the SimpleArm example with 2 links, this creates variables for $k \in \{1, 2\}$ and establishes relations for both links and the connecting joint.

Algorithm 33: systemEquations (SKO formulation)

Input: *pmodel* : *PModel*, n: number of links
Output: *pmodel* : *PModel* with system-level equations added

```
/* Initialise B_k as aggregate of link frames                    */
```
$B_k \leftarrow \langle B_1, B_2, \ldots, B_n \rangle$;
pmodel.initializeVariable(B_k);

```
/* Generate diagonal block constraints for spatial inertia matrix M  */
```
foreach $k \in [1..n]$ **do**
  $constraint \leftarrow$ "$submatrix(M)(6(k-1), 6(k-1), 6, 6) == M_k$";
  pmodel.addConstraint(*constraint*);
end

```
/* Generate diagonal block constraints for joint coupling matrix H   */
```
foreach $k \in [1..n-1]$ **do**
  $constraint \leftarrow$ "$submatrix(H)((k-1), 6(k-1), 1, 6) == H_k$";
  pmodel.addConstraint(*constraint*);
end

```
/* Generate phi spatial transform propagation equations              */
```
generatePhiEquations(*pmodel*, n);

```
/* Create system-level dynamics equations                            */
```
pmodel.addEquation("$B_k == \langle B_1, B_2, \ldots, B_n \rangle$");
pmodel.addEquation("$V == adj(\phi) * adj(H) * \dot{\theta}$");
pmodel.addEquation("$\alpha == adj(\phi) * adj(H) * \ddot{\theta} + a$");
pmodel.addEquation("$f == \phi * (M * \alpha + b)$");
pmodel.addEquation("$\tau == M_{mass} * \ddot{\theta} + C$");
pmodel.addEquation("$M_{mass} == H * \phi * M * adj(\phi) * adj(H)$");
pmodel.addEquation("$C == H * \phi * (M * adj(\phi) * a + b)$");
return *pmodel*;

The fourth step assembles the system-level equations from component blocks. It initialises B_k as a sequence of link frames, generates diagonal-block constraints that embed component-level spatial inertias (M_k) and joint coupling vectors (H_k) into system-level matrices, and formulates the complete SKO dynamics equations. These equations relate joint-space accelerations ($\ddot{\theta}$), velocities ($\dot{\theta}$), and forces (τ) through the mass matrix (M_{mass}), bias forces (C), and spatial transforms (ϕ). For the SimpleArm with 2 links, this produces a 12×12 spatial inertia matrix M and a 2×2 joint-space mass matrix M_{mass}.

A.2.1.8 Component Lookup Functions

The following helper functions are used to find components by their fully qualified names:

These functions parse the fully qualified name format "linkName.componentName" and search through the p-model's hierarchical structure to locate the specific sensor or actuator. The formulation detection algorithms classify components as either "trivial" (simple input-output equality) or "unknown" (unrecognised types). The fully qualified name ensures unique

Algorithm 34: findSensorByFQN

Input: *pmodel* : *PModel*, *sensorFQN* : *String*
Output: *sensor* : *Sensor*

```
parts = split(sensorFQN, ".");
if parts.length ≠ 2 then
    return null;
end
linkName = parts[0];
sensorName = parts[1];
foreach link ∈ pmodel.links do
    if link.name = linkName then
        foreach sensor ∈ link.sensors do
            if sensor.name = sensorName then
                return sensor;
            end
        end
    end
end
return null;
```

Algorithm 35: findActuatorByFQN

Input: *pmodel* : *PModel*, *actuatorFQN* : *String*
Output: *actuator* : *Actuator*

```
parts = split(actuatorFQN, ".");
if parts.length ≠ 2 then
    return null;
end
linkName = parts[0];
actuatorName = parts[1];
foreach link ∈ pmodel.links do
    if link.name = linkName then
        foreach actuator ∈ link.actuators do
            if actuator.name = actuatorName then
                return actuator;
            end
        end
    end
end
return null;
```

identification of components within the p-model, which is essential for associating the correct formulation type with each component during equation generation.

A.2.1.9 T3S6: Equation Library Functions

The equation library provides an interface for generating equations across different component types and formulation types. The following algorithms show the pseudocode for the library's core functions:

The equation library provides a consistent interface for all component types, where each component is processed according to its specific formulation type. For dynamics (PModel), the library delegates to formulation-specific implementations (SKO, TEST), while sensors and actuators use simpler equation generation based on their detected formulation types.

Algorithm 36: calculateEquations (PModel)

Input: *pmodel* : *PModel*, *topology* : *Topology*, *formulation* : *Formulation*, *resource* : *Resource*
Output: *pmodel′* : *PModel*

```
switch formulation do
    case Formulation :: SKO do
        switch topology do
            case Topology :: serialChain do
                return sko.calculateEquations_serial(pmodel, resource);
            end
        end
    end
    case Formulation :: TEST do
        return test.calculateEquations_serial(pmodel, resource);
    end
end
```

Algorithm 37: calculateEquations (Sensor)

Input: *sensor* : *Sensor*, *topology* : *Topology*, *formulation* : *String*, *resource* : *Resource*
Output: *void*

```
switch formulation do
    case "trivial" do
        eqn = "SignalIn == MeasurementOut";
        ctx = sensor.name;
        GeneratorServices.createEquation(sensor, ctx, eqn, resource);
    end
    case "unknown" do
        return;
    end
end
```

Algorithm 38: calculateEquations (Actuator)

Input: *actuator* : *Actuator*, *topology* : *Topology*, *formulation* : *String*, *resource* : *Resource*
Output: *void*

```
switch formulation do
    case "trivial" do
        eqn = "TorqueIn == TorqueOut";
        ctx = actuator.name;
        GeneratorServices.createEquation(actuator, ctx, eqn, resource);
    end
    case "unknown" do
        return;
    end
end
```

A.2.2 Technique 4: Guided Choice of Solutions

A.2.2.1 Step 3: Variable Resolution Details

This section provides detailed algorithms for the DAG-based variable resolution mechanisms used in Step 3 (Section 3.4.4), specifically for the *calculateInitialConditions* and *processAlgebraicConstraints* procedures.

A.2.2.2 Relation-Based Value Propagation

A key aspect of variable resolution is the ability to propagate constant values through relation chains in the model hierarchy. Consider the joint screw axis matrix H as an example: Each joint has a constant H specifying its axis of motion, each link has a variable H_i that must be resolved, and the system level relates the full H matrix to link-level submatrices through relations like:

$$\begin{aligned} \text{submatrix}(H)(i,j,1,6) &== \text{SimpleArmSerial.Link}_k.H_k \\ H &== \text{SimpleArmSerial.Link}_k.H_k \end{aligned}$$

The first relation connects the system-level H matrix to the link's H_k variable, while the second (in the link's `relation_flexi`) connects the link's H_k to the joint's constant H. When evaluating H_k on a link, the resolution algorithm examines the link's `relation_flexi` to find patterns of the form `H == SimpleArmSerial.Link`$_k$`.H`$_k$, extracts the joint variable name, determines the parent joint index using the link number, retrieves the joint constant's value from the sidecar, and copies this value into the link's variable, marking it as resolved. This relation-based propagation allows values to flow from their definition site through the hierarchical model structure to where they are needed.

Sub-expressions resolve an expression if their closure (all sub-expressions combined) resolves to a constant value. This is accomplished by checking if all sub-expressions are resolved, and if so, evaluating the expression using the values of the sub-expressions. The closure is verified by evaluating the indices of the sub-expressions and checking whether combining them leaves any indices missing.

A.2.2.3 The evaluateVariable Function

The *evaluateVariable* function evaluates a variable within a dynamic device or p-model by performing depth-first search (DFS) through constraints and relations to propagate values. In the first phase, it identifies the target device and checks if the variable has already been resolved, returning its value immediately if so. It then resolves direct constraints or equations on the device, recursively evaluating right-hand side dependencies in a DFS manner and collecting them. This DFS traversal ensures all transitive dependencies are resolved before the current variable. If the variable resolves to a constant after this constraint evaluation phase, that value is returned.

In the second phase, it attempts to resolve the variable using direct relations in the p-model. If this relation-based resolution produces a literal value, it is returned immediately. For each dependency discovered during this process, the function recursively attempts to resolve their direct relations as well, enabling propagation through relation chains via DFS.

Finally, it re-applies constraint folding to incorporate newly imported constants from the relation resolution phase and returns the final resolved expression from the `sidecar` map. The function throws an `IllegalArgumentException` if the input object is not a supported device type, and an `IllegalStateException` if no initial/boundary condition is found for the specified variable.

Algorithm 39: evaluateVariable (DFS-based Variable Resolution)

Input: *pmodel* : *PModel*, *object* : *EObject*, *varName* : *String*, *sidecar* : *Map*
Output: *Expression* – the resolved value or partially resolved expression

```
   /* Phase 1:  Identify device and check cache                              */
1  device = identifyDevice(object);
2  deviceVars = sidecar.get(device);
3  if deviceVars ≠ null and deviceVars.get(varName).resolved then
4  |  return deviceVars.get(varName).value;
5  end
   /* Phase 2:  Check initial condition constraints [t==0]                   */
6  foreach constraint ∈ device.constraints do
7  |  if constraint is IBCondition with [t == 0] matching varName then
8  |  |  rhs = constraint.right;
9  |  |  constants = buildConstantsMap(sidecar, device);
10 |  |  resolved = resolveExpression(rhs, constants);
11 |  |  updateSidecarWithValue(device, varName, resolved, sidecar);
12 |  |  if sidecar.get(device).get(varName).resolved then
13 |  |  |  return sidecar.get(device).get(varName).value;
14 |  |  end
15 |  end
16 end
   /* Phase 3:  First pass - resolve constraints via DFS                    */
17 deps = evaluateVariableWRTConstraints(varName, device, sidecar, ∅);
18 if sidecar.get(device).get(varName).resolved then
19 |  return sidecar.get(device).get(varName).value;
20 end
   /* Phase 4:  Try direct relation resolution                              */
21 direct = tryDirectRelation(pmodel, object, varName, sidecar);
22 if isAllLiterals(direct) then
23 |  return direct;
24 end
   /* Phase 5:  Try relations for each dependency                           */
25 foreach (depDevice, depVars) ∈ deps do
26 |  foreach depVar ∈ depVars.keySet() do
27 |  |  tryDirectRelation(pmodel, object, depVar, sidecar);
28 |  end
29 end
   /* Phase 6:  Final constraint folding with imported constants            */
30 evaluateVariableWRTConstraints(varName, device, sidecar, ∅);
31 return sidecar.get(device).get(varName).value;
```

A.2.2.4 calculateInitialConditions Algorithm

Algorithm 40: calculateInitialConditions (Detailed)

Input: $pmodel$: *PModel*, $slnRef$: *SolutionRef*
Output: Updated $slnRef$ with resolved constraints

foreach $constraint \in slnRef.constraints$ **do**
  /* Skip algebraic constraints [t==t] - handled by processAlgebraicConstraints */
  if *constraint matches pattern* "$[t == t]$" **then**
    continue;
  end
  if *constraint matches pattern* "$[t == 0]$" **then**
    $expr = \text{extractLeftExpression}(constraint)$;
    $varName = \text{printExpression}(expr)$;
    /* Evaluate variable using p-model constants and sidecar */
    $\text{evaluateVariable}(pmodel, varName, sidecar)$;
    $varInfo = \text{getSidecarInfo}(sidecar, pmodel, varName)$;
    if $varInfo.resolved$ **then**
      /* Variable has been resolved, substitute value */
      $resolvedValue = \text{printExpression}(varInfo.value)$;
      /* Clean artefacts and handle dimension placeholders */
      $resolvedValue = \text{cleanEquationArtifacts}(resolvedValue)$;
      if $resolvedValue =$ *"zeroMat(o,o)"* **then**
        /* Dimension placeholder, compute actual dimensions */
        $dims = \text{extractDimensions}(pmodel, expr)$;
        $resolvedValue =$ "zeroMat(" $+ dims_0 +$ ", " $+ dims_1 +$ ")";
      end
      $constraint = (varName) +$ " [t=o] ==" $+ resolvedValue$;
    end
  end
end
return $slnRef$;

A.2.2.5 Algebraic Constraint Resolution: Three-Stage Process

The *processAlgebraicConstraints* procedure employs a three-stage process that correctly maps component-level variables to system-level indexed counterparts. Constraint processing occurs in three ordered passes: initial/boundary (IB) constraints $[t == 0]$ establish initial conditions first, existing algebraic constraints $[t == t]$ are processed next, and finally algebraic derivation generates missing $[t == t]$ constraints. This ordering ensures initial conditions take precedence and derived constraints only fill gaps.

Stage 1: Propagate System Correspondence. This stage scans the model's relation hierarchy to establish mappings between component-level variables and system-level indexed expressions. It examines relations at both link and joint levels, identifying patterns where system variables (like $\theta[0]$, $\theta[1]$) are related to component variables (like joint θ in WristJoint, ElbowJoint). For each such relation, the algorithm updates the `sidecar` to record a `systemCorrespondence` field on the component variable, storing the indexed system expression. For example, when it finds the relation $\theta[0] ==$ WristJoint.theta in a link's definition, it sets the `systemCorrespondence` of the joint's local θ variable to $\theta[0]$. The algorithm maintains a set of processed expressions to avoid redundant work and ensure efficient traversal of the model's structural hierarchy.

Stage 2: Detect Variables Needing Algebraic Constraints. This stage examines each input

variable in the solution reference to determine whether it requires an algebraic constraint $[t == t]$. Variables that already have initial condition constraints $[t == 0]$ are resolved through *calculateInitialConditions* and typically don't need algebraic constraints. Variables without initial conditions are flagged for constraint generation. For each such variable, the algorithm checks if an algebraic constraint already exists; if not, it proceeds to Stage 3.

Stage 3: Evaluate Algebraic Expressions via DFS-to-Equation. For variables requiring algebraic constraints, this stage performs depth-first search through the model's relation graph to locate the defining equation. The algorithm follows relations (link $\leftrightarrow$ joint, including `relation_flexi`) with a cycle guard that tracks $(device, variable)$ pairs in an active set. When a cycle is detected (the same variable on the same device is encountered again), the algorithm gracefully returns null to break the cycle instead of entering infinite recursion. This ensures robust handling of circular dependencies while allowing partial resolution to proceed.

The traversal continues until either a device-scoped $[t == t]$ constraint is found that fully resolves via constants and systemCorrespondence, or a defining equation is located. When an equation is found (typically at a joint or actuator defining a kinematic relationship), the algorithm calls `getEquation` to fetch the native right-hand side expression in its original component context. It then builds a `constants` map from the component's sidecar, containing variables already resolved to constant values from IB constraints and variables with `systemCorrespondence` mappings established in Stage 1. The `resolveExpression` function recursively substitutes component-local variables with their system-level indexed forms throughout the expression tree, processing all expression types including matrix and vector literals. For example, a joint equation $XJ == [|\cos(\theta), -\sin(\theta), \dots|]$ is transformed to $[|\cos(\theta[0]), -\sin(\theta[0]), \dots|]$ by substituting θ with its `systemCorrespondence` value $\theta[0]$. The resulting expression may remain symbolic (e.g., containing trigonometric functions of indexed variables) and does not need to be fully numeric. Finally, the algorithm emits a system-level constraint $(varName)[t == t] == expr$ and updates the sidecar with the derived expression.

The dispatch mechanism `tryDirectRelation` examines relations at different hierarchy levels: for PModel contexts, it scans link-level relations; for Link contexts, it scans the link's own relations and nested joint relations; for Joint contexts, it checks the joint definition's equations. This hierarchical resolution enables correct navigation from system-level variables (like X_J_1) through relations (like X_J_1 == ElbowJoint.XJ) down to component-level definitions (like the joint's kinematic equation for XJ).

The sidecar data structure plays a central role in this process. Each `VarInfo` entry stores not only the variable's current value and resolution status, but also its `systemCorrespondence` (the system-level indexed expression it maps to) and `subExprs` (device-scoped sub-expressions). The `resolved` flag indicates whether a variable has reached a constant value (which may be symbolic), and this flag takes precedence: variables with IB constraints are marked resolved and their values are never overwritten by algebraic derivation. The `subExprs` field tracks partial resolutions for indexed and submatrix accesses within a specific device context. When a variable is evaluated across device boundaries (e.g., from system to component), the algorithm uses `systemCorrespondence` to map the expression and `resolveExpression` to perform substitution, then stores the result as a device-scoped `subExpr` in the target context.

Algorithm 41: processAlgebraicConstraints (Three-Stage Resolution, Detailed)

Input: $pmodel$: $PModel$, $slnRef$: $SolutionRef$, $sidecar$: $Map\langle \text{String}, VarInfo \rangle$
Output: Updated $slnRef$ with resolved constraints

```
/* Stage 1: Propagate system correspondence through model relations */
propagateSystemCorrespondence(pmodel, sidecar);
/* Stage 2: Detect variables needing algebraic constraints */
if slnRef.inputs ≠ null then
    foreach input ∈ slnRef.inputs do
        varName = input.name;
        hasInitialCondition = ∃c ∈ slnRef.constraints : c.matches("(varName)[t == 0]");
        if ¬hasInitialCondition then
            hasAlgebraicConstraint = ∃c ∈ slnRef.constraints : c.matches("(varName)[t == t]");
            if ¬hasAlgebraicConstraint then
                /* Stage 3: Generate algebraic constraint */
                equation = getEquation(pmodel, varName, sidecar);
                if equation ≠ null and equation ≠ "" then
                    algebraicConstraint = (varName) + " [t == t] ==" + equation;
                    slnRef.constraints.add(algebraicConstraint);
                end
            end
        end
    end
end
/* Process existing [t==t] constraints, simplifying RHS */
foreach constraint ∈ slnRef.constraints do
    if constraint matches pattern "[t == t]" then
        expr = extractLeftExpression(constraint);
        varName = printExpression(expr);
        equation = getEquation(pmodel, varName, sidecar);
        if equation = null then
            resolvedExpr = evaluateVariableAlgebraic(pmodel, varName, sidecar);
            equation = printExpression(resolvedExpr);
        end
        if equation ≠ null then
            equation = cleanEquationArtifacts(equation);
            constraint = (varName) + " [t=t] ==" + equation;
        end
    end
end
return slnRef;
```

A.2.2.6 Implementation: propagateSystemCorrespondence

Stage 1 of Algorithm 41 is implemented by the `propagateSystemCorrespondence` function, which addresses a critical challenge in hierarchical model resolution: mapping component-level variables to their corresponding system-level indexed expressions.

The function operates by scanning relations at two levels of the model hierarchy. First, it examines link-level relations to identify patterns where system variables are indexed to component variables. For example, a relation like $\theta[0] ==$ WristJoint.theta indicates that the system's first joint angle corresponds to the WristJoint's local θ variable. The function parses such relations by checking both sides: if the left side is an indexed expression (detected using `ArrayExp`, `CallExp`, or `SubMatrixExp` patterns), it extracts the component variable reference from the right side, resolves which component it refers to, and records the mapping in that component's sidecar entry.

Second, the function also scans joint-level relations nested within links. Since joints are defined within link structures, their relations must be processed separately. For each joint in each link, the function examines its relation clause to find additional indexed expression patterns. This two-level scanning ensures that all component variables are correctly mapped to their system-level counterparts.

The mapping information is stored in the `sidecar` using a `systemCorrespondence` field within each component's `VarInfo` entry. When Stage 3 later evaluates an algebraic expression defined on a component, it populates the `constants` map with entries from the sidecar, including any `systemCorrespondence` values. The `resolveExpression` function then substitutes component variables with their system-level indexed equivalents, producing expressions like $[|\cos(\theta[0]), \dots|]$ that correctly reference the system's configuration space.

A.2.2.7 Implementation: processAlgebraicConstraints Processing Loop

The core implementation of Algorithm 41 centres on processing existing $[t == t]$ constraints from the solution library. The function signature is `def SolutionRef processAlgebraic Constraints(PModel pmodel, SolutionRef solution)`, and it operates in a single pass through the solution's constraint list.

For each constraint matching the pattern $[t == t]$, the algorithm:

1. Parses the constraint string to extract the left-hand side variable name and right-hand side expression
2. Calls `getEquation(pmodel, varName, sidecar)` to locate the variable's defining equation in the model structure
3. If found, substitutes component-level variables with their system-level indexed equivalents using the `systemCorrespondence` mappings
4. Falls back to `utils.evaluateVariableAlgebraic( pmodel, varName, sidecar)` if direct equation lookup fails
5. Cleans the resulting expression to remove formatting artefacts
6. Updates the constraint with the resolved right-hand side

The `getEquation` helper method implements a systematic search through the model hierarchy:

1. For variables matching numbered patterns (e.g., `X_J_1`, `M_2`), extracts the base name and index
2. Searches through model links and joints for matching variables in their equation clauses
3. When found, retrieves the equation's right-hand side expression
4. Builds a constants map from the component's sidecar entries, including `systemCorrespondence` values
5. Calls `resolveExpression` to recursively substitute all component variables with their system-level equivalents

6. Returns the fully resolved equation string

Algorithm 42: getEquation (Hierarchical Equation Lookup)

Input: *pmodel* : *PModel*, *varName* : *String*, *sidecar* : *Map*
Output: *String* – the resolved equation or *null* if not found

```
/* Step 1: Search global model equations */
equation = findEquationInGlobalModel(pmodel, varName);
if equation ≠ null then
    return equation;
end
/* Step 2: Parse variable name for index pattern (e.g., M_1, X_J_2) */
(baseVarName, varIndex) = parseIndexedVariable(varName);
/* Step 3: Search numbered elements if indexed variable */
if varIndex > 0 then
    equation = findEquationInNumberedElement(pmodel, baseVarName, varIndex);
    if equation ≠ null then
        return equation;
    end
end
/* Step 4: Try to resolve using sidecar data */
if sidecar ≠ null then
    equation = resolveFromSidecar(pmodel, varName, sidecar);
    if equation ≠ null and ¬isIdentityMatrix(equation) then
        return equation;
    end
end
/* Step 5: For X_J_i patterns, use evaluateVariableAlgebraic */
if varName matches pattern "X_J_\d+" then
    expr = evaluateVariableAlgebraic(pmodel, pmodel, varName, sidecar);
    if expr ≠ null then
        return printExpression(expr);
    end
end
/* Step 6: Derive equation from constraints */
equation = deriveEquationFromConstraints(pmodel, varName, baseVarName, varIndex);
return equation;
```

The resolution process preserves symbolic expressions: equations like $XJ == [|\cos(\theta), \ldots|]$ are transformed to $[|\cos(\theta[0]), \ldots|]$ through variable substitution, maintaining the parametric form rather than attempting numeric evaluation.

A.2.2.8 Helper Algorithms

This appendix collects small helper routines referenced across Technique 4.

Algorithm 43: Reorder Solutions

Input: *pmodel* : *PModel+*, *slnRefList* : *List⟨SolutionRef⟩*
Output: *pmodel* and *slnRefList* with elements sorted by order

```
/* Sort p-model solutions by order property */
pmodel.solutions.sortBy(sln → sln.order);
/* Sort solution references by order property */
slnRefList.sortBy(ref → ref.order);
```

Algorithm 44: findMatchingSolutionRef

Input: $slnRefList : List\langle SolutionRef\rangle$, $slnPM : Solution$
Output: $slnRef : SolutionRef$ or *null*

```
1 foreach ref ∈ slnRefList do
2     if ref.expression.name = slnPM.solutionExpr.name then
3         return ref;
4     end
5 end
6 return null;
```

Algorithm 45: formatSolutionRef

Input: $slnRef : SolutionRef$
Output: $text : String$

```
1 text = "SolutionRef {";
2 text = text + printField("expression", slnRef.expression);
3 text = text + printField("method", slnRef.method);
4 text = text + printField("order", slnRef.order);
5 text = text + printField("iterations", slnRef.iterations);
6 text = text + printFieldList("inputVariables", slnRef.inputVariables);
7 text = text + printFieldList("constraints", slnRef.constraints);
8 text = text + "}";
9 return text;
```

A.2.3 Technique 5: Generation for a Platform Engine

A.2.3.1 Imperative Algorithm Formulation

This appendix presents the imperative algorithm formulations for the SolutionRef to slnDF translation described in Section 3.5.2. While the main text uses semantic translation rules (Rules 1 through 6) to define the translation declaratively, these imperative algorithms show the operational steps for implementing the translation. We explain the correspondence between each semantic rule and the relevant sections of the imperative algorithms.

Main Translation Algorithm Algorithm 46 presents the imperative formulation of the SolutionRef collection translation.

Algorithm 46: translateSlnRefToSln (Imperative Formulation)

Input: $slnRefs$: $List\langle SolutionRef\rangle$
Output: $solution$: $slnDF$ (unified slnDF file)

```
1  dtAcc ← [];
2  stateAcc ← [];
3  seenVarNames ← {} ;                          // Track added state variable names
4  procAcc ← [];
5  funcAcc ← [];
6  compAcc ← [];
   /* Process each slnRef entry in order                                          */
7  foreach slnRef ∈ slnRefs do
       /* Select and instantiate the corresponding library template              */
8      tmpl ← library.slnRef;
9      if tmpl = null then
10         error "No template for method " + slnRef.method;
11     end
       /* Accumulate datatypes (if present)                                      */
12     dtAcc ← dtAcc ∪ datatypes(tmpl);
       /* Extract state variables (deduplicate by name)                          */
13     foreach var ∈ stateVars(tmpl) do
14         if var.name ∉ seenVarNames then
15             stateAcc.add(var);
16             seenVarNames.add(var.name);
17         end
18     end
       /* Extract procedures and functions                                       */
19     procAcc ← procAcc ∪ procedures(tmpl);
20     funcAcc ← funcAcc ∪ functions(tmpl);
       /* Extract computation statements                                         */
21     compAcc ← concat([compAcc, computation(tmpl)]);
22 end
   /* Deduplicate procedures and functions by signature                          */
23 procAcc ← deduplicateProcedures(procAcc);
24 funcAcc ← deduplicateFunctions(funcAcc);
   /* Assemble final unified solution                                            */
25 solution.datatypes ← dtAcc;
26 solution.state ← stateAcc;
27 solution.procedures ← procAcc;
28 solution.functions ← funcAcc;
29 solution.computation ← compAcc;
30 return solution;
```

Deduplication Algorithm Algorithm 47 presents the signature-based deduplication for procedures (the same process applies to functions).

Algorithm 47: deduplicateProcedures (Imperative Formulation)

Input: *procedures* : *List*⟨*Procedure*⟩
Output: *uniqueProcedures* : *List*⟨*Procedure*⟩

```
1  seenSignatures ← {} ;                                  // Map from signature to procedure
2  foreach proc ∈ procedures do
3      signature ← (proc.name, proc.parameters);
4      existing ← seenSignatures[signature];
       /* Keep first occurrence or implementation with longer text                    */
5      if existing = null then
6          seenSignatures[signature] ← proc;
7      end
8      else if |proc| > |existing| then
9          seenSignatures[signature] ← proc ;                  // Prefer fuller implementation
10     end
11 end
12 uniqueProcedures ← seenSignatures.values();
13 return uniqueProcedures;
```

Correspondence Between Semantic Rules and Imperative Algorithms Table A.1 shows how each semantic translation rule from Section 3.5.2 corresponds to specific sections of the imperative algorithms.

Semantic Rule	Imperative Algorithm Section
Rule 1 (SolutionRef Collection Translation)	Algorithm 46, lines 1–2, 55–61: Overall structure, input/output, and final assembly into slnDF structure
Library template lookup (`library.sr`)	Algorithm 46, within the main loop: template selection/instantiation ($tmpl \leftarrow$ library.$slnRef$)
Rule 2 (Datatype Aggregation)	Algorithm 46, within the main loop: $dtAcc \leftarrow dtAcc \cup \text{datatypes}(tmpl)$
Rule 3 (State Variable Deduplication)	Algorithm 46, within the main loop: iterate over $\text{stateVars}(tmpl)$ with name-based deduplication using `seenVarNames`
Rule 4 (Procedure Signature Deduplication)	Algorithm 46: $procAcc \leftarrow procAcc \cup \text{procedures}(tmpl)$ in the loop, then `deduplicateProcedures(procAcc);` detailed in Algorithm 47
Rule 5 (Function Signature Deduplication)	Algorithm 46: $funcAcc \leftarrow funcAcc \cup \text{functions}(tmpl)$ in the loop, then `deduplicateFunctions(funcAcc)` (analogous to Algorithm 47)
Rule 6 (Computation Sequence)	Algorithm 46, within the main loop: $compAcc \leftarrow \text{concat}([compAcc, \text{computation}(tmpl)])$ preserves the sequential computation order

Table A.1: Correspondence between semantic translation rules and imperative algorithm sections

A.2.3.2 Generated C++ excerpts for the Simple Arm

```cpp
// State variables (lines 2-7)
Eigen::MatrixXd tau;
Eigen::MatrixXd phi;
Eigen::MatrixXd theta;
Eigen::MatrixXd d_theta;
Eigen::MatrixXd dd_theta;
Eigen::MatrixXd M_mass;
// ... other state variables

// Procedures (lines 11-14)
void SKOm_set(Eigen::MatrixXd &modifier, int x, int y,
           Eigen::MatrixXd input) {
   modifier.block((6 * x), (6 * y), 6, 6) = input;
}
// ... other procedures

// Functions (lines 18-26)
Eigen::MatrixXd CalcPhi(int m, int n,
                  std::vector<Eigen::MatrixXd> Bk) {
   Eigen::MatrixXd result = Eigen::MatrixXd::Zero(6, 6);
   result.block(0, 0, 3, 3) = Eigen::MatrixXd::Identity(3, 3);
   result.block(0, 3, 3, 3) = lx(getFramePosition(m, Bk),
                           getFramePosition(n, Bk));
   result.block(3, 0, 3, 3) = Eigen::MatrixXd::Identity(3, 3);
   return result;
}
// ... other functions

// Computation: physics_update (lines 30-51)
void physics_update() {
   // Compute rigid-shift operator phi
   SKOm_set(phi, 0, 0, Eigen::MatrixXd::Identity(6, 6));
   SKOm_set(phi, 1, 0, CalcPhi(2, 1, Bk));
   SKOm_set(phi, 1, 1, Eigen::MatrixXd::Identity(6, 6));
   SKOm_set(phi, 2, 0, CalcPhi(3, 1, Bk));
   SKOm_set(phi, 2, 1, CalcPhi(3, 2, Bk));
   SKOm_set(phi, 2, 2, Eigen::MatrixXd::Identity(6, 6));

   // Compute inverse dynamics C
   // ... inverse dynamics computation

   // Compute mass matrix M via composite body algorithm
   // ... mass matrix computation

   // Compute forward dynamics dd_theta
   dd_theta = M_inv * (tau - C);

   // Euler integration
   d_theta = d_theta + (dt * dd_theta);
   theta = theta + (dt * d_theta);
}
```

Listing A.13: Excerpts from generated C++ platform engine (platform1_engine.cpp)

```cpp
// Stub: d-model main function declaration (line 2)
extern "C" int pickplace_main(int argc, char* argv[]);

// Communication state (lines 5-8)
// State variables shared via registerRead/registerWrite channels
static input_register_t inputRegister = {};
static engine_output_t engine_output = {};
```

```
8  static IPlatformEngine* platform_engine = nullptr;
9
10 // Evolve: continuous integration (lines 11-29)
11 bool Evolve(double& timer) {
12    // Update world-platform mapping sensors
13    EvolveWorldMapping();
14
15    Wait(platform_dt); // Wait for timestep
16
17    // Update platform physics
18    platform_engine->update();
19
20    // Integrate k (dk/dt = 1 from Mapping.pm)
21    if (operationActive) {
22       p_mapping.k += dt;
23       update_mapping_from_operations();
24    }
25
26    // Advance timer (dtimer/dt = 1)
27    timer += dt;
28
29    return timer >= cycle_duration;
30 }
31
32 // Stubs: Integration with d-modelGen and mappingGen (lines 33-52)
33 class DModelIOAdapter : public IDModelIO {
34 public:
35    // Stubs for registerRead/registerWrite channels
36    // To be filled based on platform mapping from mappingGen
37    bool registerRead(int* type, bool* value) override;
38    void registerWrite(int op, double param, double time) override;
39    void tock(int type) override;
40 };
41
42 int main(int argc, char* argv[]) {
43    // Init: Initialize physics runtime
44    engine_threads_t threads = start_engine_runtime();
45
46    // Install IO adapter and launch d-model
47    DModelIOAdapter io;
48    set_active_dmodel_io(&io);
49    int exit_code = pickplace_main(argc, argv);
50
51    shutdown_engine(threads);
52    return exit_code;
53 }
```

Listing A.14: Excerpts from generated orchestrator (orchestrator.cpp) showing integration stubs for d-modelGen and mappingGen

A.3 Pseudocode and Implementation Comparison

This appendix provides side-by-side comparisons of the pseudocode algorithms from Chapter 3 and Appendix A.2.1 with their corresponding Xtend implementation code. These comparisons demonstrate the direct correspondence between the theoretical algorithms and their practical implementation.

A.3.1 Technique 3: Equation Composition

A.3.1.1 Main Algorithm: eqnComp

```
Input: pmodel : PModel
     overrideFormulation (optional)
Output: pmodel_ : PModel+

/* Step 1: Calculate Formulations */
formulations = calculateFormulations(
  pmodel, overrideFormulation);
formulation_dyn = formulations.formulation_dyn;

/* Step 2: Localise reference components */
pmodel_ = localiseRefs(pmodel);

/* Step 3: Validate generator compatibility */
eqnCompValidator(pmodel_, formulation_dyn);

/* Step 4: Determine topology */
topology = calculateTopology(pmodel_);

/* Step 5: Assign numbering to Links and Joints */
pmodel_ = assignNumbering(pmodel_, formulation_dyn);

/* Step 6: Calculate equations for all components */
pmodel_ = calculateEquations(pmodel_, topology, formulations);

return pmodel_;
```

Listing A.15: Algorithm 1 (Pseudocode)

```
def CharSequence eqnComp(PModel pmodel,
   Resource resource,
   Formulation overrideFormulation) {
  // Ensure the model is valid before processing
  if (pmodel === null) {
    throw new IllegalArgumentException(
      "PModel cannot be null")
  }

  // Create working copy to avoid modifying the original resource
  val copyPair = createWorkingCopy(
    pmodel, resource)
  val pmodelCopy = copyPair.key
  val tempResource = copyPair.value

  if (pmodelCopy === null) {
    throw new IllegalStateException(
      "Failed to create working copy")
  }

  try {
    //Step 1) Calculate Formulations
    var formulations = calculateFormulations(
      pmodel, overrideFormulation)
    var dynamicsFormulation =
      formulations?.dynamicsFormulation

    //Step 2) Localise reference components
    var pmodel_ = localiseRefs(pmodelCopy)

    //Step 3) Validate generator compatibility
    eqnCompValidator(pmodel_, dynamicsFormulation)

    //Step 4) Determine topology
    var topology = calculateTopology(pmodel_)

    //Step 5) Assign numbering to Links And Joints
    pmodel_ = assignNumbering(
      pmodel_, dynamicsFormulation)
```

```
40
41    //Step 6) Calculate equations for all components
42    pmodel_ = calculateEquations(
43      pmodel_, topology,
44      formulations, tempResource)
45
46    // Serialize and return
47    state.idSanitizationMap.clear
48    val outputStream = new ByteArrayOutputStream()
49    tempResource.save(outputStream,
50      SaveOptions.newBuilder().getOptions().toOptionsMap())
51    var result = outputStream.toString("UTF-8")
52    result = result.replace(
53      "record Geom", "datatype Geom")
54    return result
55  } catch (IOException e) {
56    System.err.println(
57      "Warning: Could not save resource")
58    return serializer.serialize(
59      tempResource.contents.get(0))
60  } finally {
61    deleteTempResource(tempResource)
62  }
63 }
```

Listing A.16: eqnComp Implementation

Figure A.2: Side-by-side comparison of the Technique 3 main algorithm and its implementation.

Implementation Notes. The implementation follows the structure of the pseudocode, with each step mapped to a method call, but adds generator-specific concerns.
A working copy is created using `createWorkingCopy(pmodel, resource)`, and only that copy (`pmodelCopy`) is mutated. This realises the pass-by-value intent of the pseudocode while operating in EMF/Xtext's pass-by-reference model.
The expression `formulations?.dynamicsFormulation` uses Xtend's null-safe operator (`?.`). Since `calculateFormulations` can return `null` when no dynamics formulation is detected, this operator safely evaluates to `null` rather than throwing a `NullPointerException`.
To return a `CharSequence` for Xtext code generation, the implementation serialises the working `tempResource` using Xtext `SaveOptions`, and performs a small post-processing substitution for consistency (`record` to `datatype`). The `try/catch/finally` ensures fallback serialisation on I/O failure and cleanup of the temporary resource.

A.3.1.2 Step 1: Calculate Formulations

```
Input: pmodel : PModel
     overrideDynamicsFormulation (optional)
Output: result : Formulations

/* Step 1: Find dynamics formulation */
if overrideDynamicsFormulation ≠ null then
  formulation_dyn = overrideDynamicsFormulation;
else
  formulation_dyn = findDynamicsFormulation(pmodel);
end

/* Step 2: Find sensor formulations */
formulation_sen = findSensorFormulations(pmodel);

/* Step 3: Find actuator formulations */
formulation_act = findActuatorFormulations(pmodel);

/* Step 4: Return Formulations */
if formulation_dyn ≠ null then
  return Formulations(formulation_dyn, formulation_sen, formulation_act);
end
return null;
```

Listing A.17: Algorithm 2 (Pseudocode)

```
def Formulations calculateFormulations(
  PModel pmodel,
  Formulation overrideDynamicsFormulation) {
 // Ensure library references are resolved before inspection
 EcoreUtil.resolveAll(pmodel)

 // Step 1: Find dynamics formulation (or use override)
 val dynamicsFormulation =
  if (overrideDynamicsFormulation !== null)
   overrideDynamicsFormulation
  else
   eqnCompFormulationUtils
    .findDynamicsFormulation(pmodel)

 // Step 2: Find sensor formulations
 val sensorFormulationMap =
  eqnCompFormulationUtils
   .findSensorFormulations(pmodel)

 // Step 3: Find actuator formulations
 val actuatorFormulationMap =
  eqnCompFormulationUtils
   .findActuatorFormulations(pmodel)

 // Step 4: Return Formulations
 if (dynamicsFormulation !== null) {
  return new Formulations(
   dynamicsFormulation,
   sensorFormulationMap,
   actuatorFormulationMap)
 }

 return null
}
```

Listing A.18: calculateFormulations Implementation

Figure A.3: Side-by-side comparison of calculateFormulations algorithm and implementation.

Implementation Notes. The implementation uses Xtend's `val` keyword for immutable variable declarations, whereas the pseudocode uses direct assignment. The null check uses `!==` (Xtend's identity comparison operator) rather than the mathematical $\neq$ notation. When constructing the result, the implementation requires the `new` keyword for object instantiation (`new Formulations(...)`), which is standard Java/Xtend syntax for creating class instances, whereas the pseudocode omits this for clarity. Before inspection, the implementation calls `EcoreUtil.resolveAll(pmodel)` to ensure that cross-references to library definitions are resolved, and it delegates formulation detection to `eqnCompFormulationUtils`.

A.3.1.3 Step 1a: Find Dynamics Formulation

```
Input: pmodel : PModel
Output: formulation_dyn : Formulation

/* Check for TEST formulation first */
if hasTestConstant(pmodel) then
  return TEST;
```

```
end

/* Step 1: Collect all joints from links */
joints = {};
foreach link ∈ pmodel.links do
 foreach joint ∈ link.joints do
  if joint ≠ null then
   joints = joints ∪ {joint};
  end
 end
end

/* Step 2: Classify joints */
skoCount = 0; featherstoneCount = 0;
hasJoints = false;
foreach j ∈ joints do
 hasJoints = true;
 isSKOJoint = false;
 isFeatherstoneJoint = false;
 if j instanceof LocalJoint then
  isSKOJoint = checkLocalJointForH(j);
  isFeatherstoneJoint = checkLocalJointForFeatherstone(j);
 else if j instanceof ReferenceJoint then
  isSKOJoint = checkReferenceJointForSKO(j);
  isFeatherstoneJoint = checkReferenceJointForFeatherstone(j);
 end
 if isSKOJoint then
  skoCount = skoCount + 1;
 else if isFeatherstoneJoint then
  featherstoneCount = featherstoneCount + 1;
 end
end

/* Step 3: Return formulation */
if ¬hasJoints then
 return null;
end
if skoCount = |joints| then
 return SKO;
end
if featherstoneCount = |joints| then
 return FEATHERSTONE;
end
return null;
```

Listing A.19: Algorithm 3 (Pseudocode)

```
def static eqnCompGenerator.Formulation
   findDynamicsFormulation(PModel pmodel) {
 // Check for TEST formulation first
 if (hasTestConstant(pmodel)) {
  return eqnCompGenerator.Formulation::TEST
 }

 // Step 1: Collect all joints from links
 val joints = new LinkedHashSet<Joint>()
 for (link : pmodel.links) {
  for (joint : link.joints) {
   if (joint !== null) {
    joints.add(joint)
   }
  }
 }

 // Step 2: Classify each joint as SKO, Featherstone, or unknown
 var int skoCount = 0
 var int featherstoneCount = 0
 var boolean hasJoints = false

 for (j : joints) {
  hasJoints = true
  var boolean isSKOJoint = false
  var boolean isFeatherstoneJoint = false
```

```
27
28    if (j instanceof LocalJoint) {
29      isSKOJoint = checkLocalJointForH(j as LocalJoint)
30      isFeatherstoneJoint = checkLocalJointForFeatherstone(j as LocalJoint)
31    } else if (j instanceof ReferenceJoint) {
32      isSKOJoint = checkReferenceJointForSKO(j as ReferenceJoint)
33      isFeatherstoneJoint = checkReferenceJointForFeatherstone(j as ReferenceJoint)
34    }
35
36    if (isSKOJoint) {
37      skoCount++
38    } else if (isFeatherstoneJoint) {
39      featherstoneCount++
40    }
41  }
42
43  // Step 3: Return formulation based on joint classification
44  if (!hasJoints) {
45    return null
46  }
47  if (skoCount == joints.size) {
48    return eqnCompGenerator.Formulation::SKO
49  }
50  if (featherstoneCount == joints.size) {
51    return eqnCompGenerator.Formulation::FEATHERSTONE
52  }
53  return null
54 }
```

Listing A.20: findDynamicsFormulation Implementation

Figure A.4: Side-by-side comparison of findDynamicsFormulation algorithm and implementation.

Implementation Notes. As defined in Section 3.2.2, the pseudocode collection notation `{}` specifies ordered collections that preserve insertion order. The implementation realises this semantics using `LinkedHashSet` rather than `HashSet`. Standard `HashSet` does not preserve insertion order, which would violate the pseudocode semantics and cause joints to be processed in an unpredictable sequence across runs. The implementation also checks for a `TEST` override condition (via `hasTestConstant(pmodel)`) before classifying joints as `SKO`, `FEATHERSTONE`, or `unknown`.

A.3.1.4 Step 1b: Find Sensor Formulations

```
1 Input: pmodel : PModel
2 Output: formulation_sen : Map<String, String>
3
4 formulation_sen = {};
5
6 /* Step 1: Iterate through all sensors */
7 foreach link ∈ pmodel.links do
8   /* Link-scoped sensors */
9   foreach sensor ∈ link.sensors do
10    sensorFQN = link.name + "." + sensor.
          name;
11    formulation_sen[sensorFQN] =
12      detectSensorFormulation(sensor);
13  end
14
15  /* Joint-scoped sensors (nested inside
        joints) */
16  foreach joint ∈ link.joints do
17    foreach sensor ∈ joint.sensors do
18      sensorFQN = link.name + "." + sensor.
            name;
19      if sensorFQN ∉ dom(formulation_sen) then
20        formulation_sen[sensorFQN] =
21          detectSensorFormulation(sensor);
22      end
23    end
24  end
25 end
26
27 return formulation_sen;
```

Listing A.21: `findSensorFormulations` (Pseudocode)

```
1 def static Map<String, String>
2   findSensorFormulations(PModel pmodel) {
3  val formulation_sens =
4   new LinkedHashMap<String, String>()
5
6  for (link : pmodel.links) {
7   // Link-scoped sensors
8   for (sensor : link.sensors) {
9    val sensorFQN = link.name + "." +
         sensor.name
10   val formulation =
         detectSensorFormulation(sensor)
11   formulation_sens.put(sensorFQN,
         formulation)
12  }
13
14  // Joint-scoped sensors (nested inside
       joints)
15  for (joint : link.joints) {
16   for (sensor : joint.sensors) {
17    val sensorFQN = link.name + "." +
         sensor.name
18    if (!formulation_sens.containsKey(
         sensorFQN)) {
19     val formulation =
         detectSensorFormulation(sensor
         )
20     formulation_sens.put(sensorFQN,
         formulation)
21    }
22   }
23  }
24 }
25
26 return formulation_sens
27 }
```

Listing A.22: findSensorFormulations Implementation

Figure A.5: Side-by-side comparison of findSensorFormulations algorithm and implementation.

Implementation Notes. Following the collection semantics defined in Section 3.2.2, the implementation uses `LinkedHashMap` rather than `HashMap` to preserve insertion order when iterating over the formulation map. This ensures sensors are processed in the order they were added during equation generation (Step 6). The implementation also searches both link-scoped sensors (`link.sensors`) and joint-scoped sensors (`joint.sensors`); when a joint-scoped sensor shares the same fully-qualified name as a link-scoped sensor, the link-scoped entry takes precedence.

A.3.1.5 Step 1c: Find Actuator Formulations

```
Input: pmodel : PModel
Output: formulation_act : Map<String, String>

formulation_act = {};

foreach link ∈ pmodel.links do
  /* Link-scoped actuators */
  foreach actuator ∈ link.actuators do
    actuatorFQN = link.name + "." +
        actuator.name;
    formulation_act[actuatorFQN] =
      detectActuatorFormulation(actuator);
  end

  /* Joint-scoped actuators (nested inside
      joints) */
  foreach joint ∈ link.joints do
    foreach actuator ∈ joint.actuators do
      actuatorFQN = link.name + "." +
          actuator.name;
      if actuatorFQN ∉ dom(formulation_act)
          then
        formulation_act[actuatorFQN] =
          detectActuatorFormulation(actuator
              );
      end
    end
  end
end

return formulation_act;
```

Listing A.23: `findActuatorFormulations` (Pseudocode)

```
def static Map<String, String>
    findActuatorFormulations(PModel pmodel)
        {
  val formulation_act =
    new LinkedHashMap<String, String>()

  for (link : pmodel.links) {
    // Link-scoped actuators
    for (actuator : link.actuators) {
      val actuatorFQN = link.name + "." +
          actuator.name
      val formulation =
          detectActuatorFormulation(
          actuator)
      formulation_act.put(actuatorFQN,
          formulation)
    }

    // Joint-scoped actuators (nested
        inside joints)
    for (joint : link.joints) {
      for (actuator : joint.actuators) {
        val actuatorFQN = link.name + "." +
            actuator.name
        if (!formulation_act.containsKey(
            actuatorFQN)) {
          val formulation =
              detectActuatorFormulation(
              actuator)
          formulation_act.put(actuatorFQN,
              formulation)
        }
      }
    }
  }

  return formulation_act
}
```

Listing A.24: findActuatorFormulations Implementation

Figure A.6: Side-by-side comparison of findActuatorFormulations algorithm and implementation.

Implementation Notes. As with sensor formulations, the implementation uses `LinkedHashMap` to realise the ordered collection semantics, ensuring actuators are processed in their insertion order during equation generation. The implementation also searches both link-scoped actuators (`link.actuators`) and joint-scoped actuators (`joint.actuators`); when a joint-scoped actuator shares the same fully-qualified name as a link-scoped actuator, the link-scoped entry takes precedence.

A.3.1.6 Step 2: Localise References

```
Input: pmodel : PModel
Output: pmodel_ : PModel

pmodel_ = pmodel;
expandParts(pmodel_);
convertReferenceJointsToLocal(pmodel_);
convertReferenceLinksToLocal(pmodel_);
convertReferenceSensorsToLocal(pmodel_);
convertReferenceActuatorsToLocal(pmodel_);
convertReferenceBodiesToLocal(pmodel_);

return pmodel_;
```

Listing A.25: Algorithm 4 (Pseudocode)

```
def PModel localiseRefs(PModel pmodel) {
  val pmodel_ = pmodel
  expandParts(pmodel_)
  convertReferenceJointsToLocal(pmodel_)
  convertReferenceLinksToLocal(pmodel_)
  convertReferenceSensorsToLocal(pmodel_)
  convertReferenceActuatorsToLocal(pmodel_
      )
  convertReferenceBodiesToLocal(pmodel_)
  return pmodel_
}
```

Listing A.26: localiseRefs Implementation

Figure A.7: Side-by-side comparison of localiseRefs algorithm and implementation.

A.3.1.7 Step 2a: Convert Reference Joints to Local

```
Input: pmodel : PModel
Output: pmodel with all reference
      joints converted to local

foreach link ∈ pmodel.links do
  jointsToReplace = {};
  newJoints = {};

  foreach joint ∈ link.joints do
    if joint instanceof ReferenceJoint then
      localJoint = createLocalJoint(joint);
      jointsToReplace = jointsToReplace ∪ {
          joint};
      newJoints = newJoints ∪ {localJoint};
    end
  end

  /* Replace reference joints */
  foreach i ∈ [0, size(jointsToReplace))
      do
    index = link.joints.indexOf(
        jointsToReplace[i]);
    if index ≥ 0 then
      link.joints[index] = newJoints[i];
    end
  end
end

return pmodel;
```

Listing A.27: Algorithm 5 (Pseudocode)

```
def PModel convertReferenceJointsToLocal(
    PModel pmodel) {
  for (link : pmodel.links) {
    val jointsToReplace = newArrayList()
    val newJoints = newArrayList()

    for (joint : link.joints) {
      if (joint instanceof ReferenceJoint)
          {
        val localJoint = createLocalJoint(
            joint)
        jointsToReplace.add(joint)
        newJoints.add(localJoint)
      }
    }

    // Replace reference joints
    for (i : 0 ..< jointsToReplace.size) {
      val index = link.joints.indexOf(
        jointsToReplace.get(i))
      if (index >= 0) {
        link.joints.set(index,
          newJoints.get(i))
      }
    }
  }
  return pmodel
}
```

Listing A.28: convertReferenceJointsToLocal Implementation

Figure A.8: Side-by-side comparison of convertReferenceJointsToLocal algorithm and implementation.

A.3.1.8 Step 3: Validate

```
Input: pmodel : PModel
Output: success or validation error

# SKO-specific inertia requirements
foreach link ∈ pmodel.links do
  hasInertia = (link.def.inertial ≠ null)
    or ∃ b ∈ link.bodies :
      b.def.inertial ≠ null
  if ¬hasInertia then
    error "Link link.name has no
          valid inertia definition"
  end
end

return success
```

Listing A.29: Algorithm 7 (Pseudocode)

```
def private static void
    validateInertiaAndBodyRequirements(
      PModel pmodel,
      List<String> validationErrors) {
  if (pmodel.links === null ||
      pmodel.links.empty) {
    return
  }

  for (link : pmodel.links) {
    if (link !== null &&
        link.name !== null) {
      val hasValidInertia =
        validateLinkInertia(link)
      if (!hasValidInertia) {
        validationErrors.add(
          "Link '" + link.name +
          "' has no valid inertia definition
               (neither direct inertia nor
              body with inertia)")
      }
    }
  }
}

def private static boolean
      validateLinkInertia(Link link) {
  // Check link-level inertia (LocalLink)
  if (link instanceof LocalLink) {
    val localLink = link as LocalLink
    if (localLink.def instanceof LinkDef) {
      val linkDef = localLink.def as
          LinkDef
      if (linkDef.inertial !== null) {
        return true
      }
    }
  }

  // Check body-level inertia (LocalBody)
  if (link.bodies !== null &&
      !link.bodies.empty) {
    for (body : link.bodies) {
      if (body !== null &&
          body instanceof LocalBody) {
        val localBody = body as LocalBody
        if (localBody.def instanceof BodyDef
            ) {
          val bodyDef = localBody.def as
              BodyDef
          if (bodyDef.inertial !== null) {
            return true
          }
        }
      }
    }
  }

  return false
}
```

Listing A.30: validateInertiaAndBodyRequirements Implementation

Figure A.9: Side-by-side comparison of the validate algorithm (SKO formulation) and its implementation.

The implementation directly corresponds to the pseudocode structure. The outer loop iterates through all links, and for each link the `validateLinkInertia` helper checks whether the link has direct inertia (via its definition) or a body with inertia. The existential quantifier from the pseudocode ($\exists\, b \in \mathit{link.bodies}$) is implemented as a loop that returns `true` as soon as a body with valid inertial properties is found. The implementation collects all validation errors into a list rather than failing on the first error, providing more comprehensive feedback to users. In the complete generator, this inertia check is one of several checks performed by `validateSKOFormulation`, which also validates basic model structure and pose availability.

A.3.1.9 Step 4: Calculate Topology

```
Input: pmodel : PModel
Output: Topology : {SerialChain,
      KinematicTree, ClosedChain}

if |pmodel.links| = |getJoints(pmodel)| +
     1 then
  if ∃ link ∈ pmodel.links : |link.joints|
       > 1 then
    Topology = KinematicTree;
  else
    Topology = SerialChain;
  end
else
  Topology = ClosedChain;
end

return Topology;
```

Listing A.31: Algorithm from Section 3.2.5 (Pseudocode)

```
def Topology calculateTopology(PModel
     pmodel) {
  if (pmodel.links.size ==
     getJoints(pmodel).size + 1) {
    for (link : pmodel.links) {
      if (link.joints.size > 1) {
        return Topology::treeTopology
      }
    }
    return Topology::serialChain
  }
  return Topology::closedChain
}
```

Listing A.32: calculateTopology Implementation

Figure A.10: Side-by-side comparison of calculateTopology algorithm and implementation.

A.3.1.10 Step 5: Assign Numbering

```
Input: pmodel : PModel,
      formulation : Formulation
Output: pmodel' : PModel with
       numbering assigned

switch formulation do
  case Formulation::SKO:
    return orderPmodel(pmodel,
      Ordering::reverse);
  case Formulation::Featherstone:
    return orderPmodel(pmodel,
      Ordering::regular);
  case Formulation::Test:
    return pmodel;
  default:
    return null;
end
```

Listing A.33: Algorithm 26 (Pseudocode)

```
def assignNumbering(PModel pmodel,
   Formulation formulation) {
  switch formulation {
    case SKO:
      return orderPmodel(pmodel,
        Ordering::reverse)
    case FEATHERSTONE:
      return orderPmodel(pmodel,
        Ordering::regular)
    case TEST:
      return pmodel
    default:
      return null
  }
}
```

Listing A.34: assignNumbering Implementation

Figure A.11: Side-by-side comparison of assignNumbering algorithm and implementation.

A.3.1.11 Step 5a: Order PModel

```
Input: pmodel : PModel,
     ordering : Ordering
Output: pmodel' : PModel with
       sequential numbering

baseLink = getBaseLink(pmodel);
loopLinks = {baseLink};
spanningList = getChildLinks(
 pmodel, baseLink, loopLinks);
spanningList.addFirst(baseLink);

if ordering = Ordering::reverse then
 spanningList.reverse();
end

n = 0;
n1 = |pmodel.links|;
addNumber(baseLink, n1, pmodel);
addPoseVectorVariable(baseLink, pmodel);

/* Assign numbers to bodies in baseLink */
bodyNum = 0;
foreach body ∈ baseLink.bodies do
 addNumber(body, bodyNum, pmodel);
 bodyNum = bodyNum + 1;
end

/* Assign numbers to remaining links/joints */
loopLinks = {};
loopJoints = {};
foreach i ∈ [0, size(spanningList) - 1) do
 /* Check first link for loop-closing joints */
 if i = 0 then
  foreach joint ∈ spanningList[i].joints do
   if joint.flexible.link = baseLink then
    loopJoints = loopJoints ∪ {joint};
   end
  end
 end
 foreach joint ∈ spanningList[i+1].joints do
  if joint.flexible.link ∉ loopLinks then
   n = n + 1;
   addNumber(joint, n, pmodel);
   addPoseVectorVariable(joint, pmodel);
   addNumber(
    joint.flexible.link, n, pmodel);
   addPoseVectorVariable(
    joint.flexible.link, pmodel);
   loopLinks = loopLinks ∪ {joint.flexible.link};

   /* Assign numbers to bodies */
   bodyNum = 0;
   foreach body ∈ joint.flexible.link.bodies do
    addNumber(body, bodyNum, pmodel);
    bodyNum = bodyNum + 1;
   end
  else
   loopJoints = loopJoints ∪ {joint};
  end
 end
end

/* Assign numbers to loop-closing joints */
foreach joint ∈ loopJoints do
 addNumber(joint, n, pmodel);
 addPoseVectorVariable(joint, pmodel);
end

return pmodel;
```

Listing A.35: Algorithm 27 (Pseudocode)

```
def orderPmodel(PModel pmodel,
   Ordering ordering) {
 var PModel pmodelOrdered
 switch ordering {
  case reverse: {
   pmodelOrdered = pmodel
   var Link baseLink =
    getBaseLink(pmodelOrdered)
   var EList<Link> loopLinks =
    new BasicEList<Link>()
   var EList<Link> spanningList =
    new BasicEList<Link>()
   loopLinks.add(baseLink)
   spanningList = getChildLinks(
    pmodelOrdered, baseLink, loopLinks)
   spanningList.add(0, baseLink)
   spanningList.reverse()

   var int n = 0
   var int n1 = pmodel.links.size()
   addNumber(baseLink, n1, pmodel)
   addPoseVectorVariable(baseLink, pmodel)

   // Assign numbers to bodies in baseLink
   var int bodyNum = 0
   for (body : baseLink.bodies) {
    addNumber(body, bodyNum, pmodel)
    bodyNum++
   }

   // Assign numbers to remaining links/joints
   loopLinks = new BasicEList<Link>()
   var EList<Joint> loopJoints =
    new BasicEList<Joint>()
   for (i : 0 ..< spanningList.size - 1) {
    // Check first link for loop-closing joints
    if (i == 0) {
     for (joint : spanningList.get(i).joints) {
      if (joint.flexible.link == baseLink) {
       loopJoints.add(joint)
      }
     }
    }
    for (joint : spanningList.get(i+1).joints) {
     if (!loopLinks.contains(
        joint.flexible.link)) {
      n = n + 1
      var int n2 = n
      addNumber(joint, n2, pmodel)
      addPoseVectorVariable(joint, pmodel)
      addNumber(
       joint.flexible.link, n2, pmodel)
      addPoseVectorVariable(
       joint.flexible.link, pmodel)

      // Assign numbers to bodies
      bodyNum = 0
      for (body :
       joint.flexible.link.bodies) {
       addNumber(body, bodyNum, pmodel)
       bodyNum++
      }

      loopLinks.add(joint.flexible.link)
     } else {
      loopJoints.add(joint)
     }
    }
   }

   // Assign numbers to loop-closing joints
   for (joint : loopJoints) {
    addPoseVectorVariable(joint, pmodel)
    addNumber(joint, n, pmodel)
   }
   return pmodelOrdered
  }
 }
}
```

Listing A.36: orderPmodel Implementation (reverse case)

Figure A.12: Side-by-side comparison of orderPmodel algorithm and implementation (reverse ordering case for SKO).

A.3.1.12 Step 6: Calculate Equations

```
Input: pmodel : PModel,
     topology : Topology,
     formulations : Formulations
Output: pmodel' : PModel

/* Step 6a: Calculate dynamics equations */
pmodel' = lib.calculateEquations(
  pmodel, topology,
  formulations.formulation_dyn);

/* Step 6b: Calculate sensor equations */
foreach sensorFQN ∈
    dom(formulations.formulation_sen) do
  formulation =
    formulations.formulation_sen[sensorFQN];
  sensor = findSensorByFQN(pmodel',
      sensorFQN);
  if sensor ≠ null then
    lib.calculateEquations(
      sensor, topology, formulation);
  end
end

/* Step 6c: Calculate actuator equations */
foreach actuatorFQN ∈
    dom(formulations.formulation_act) do
  formulation =
    formulations.formulation_act[actuatorFQN];
  actuator = findActuatorByFQN(pmodel',
      actuatorFQN);
  if actuator ≠ null then
    lib.calculateEquations(
      actuator, topology, formulation);
  end
end

return pmodel';
```

Listing A.37: Algorithm 9 (Pseudocode)

```
def PModel calculateEquations(
   PModel pmodel, Topology topology,
   Formulations formulations,
   Resource resource) {
  // Step 6a: Calculate dynamics equations
       for the entire p-model
  val pmodel_out = lib.calculateEquations(
    pmodel, topology, formulations.
        dynamicsFormulation,
    resource)

  // Step 6b: Calculate equations for
      sensors
  for (sensorFQN : formulations.
      sensorFormulation.keySet) {
    val sensorFormulation =
      formulations.sensorFormulation.get(
          sensorFQN)
    val sensor = findSensorByFQN(pmodel_out
        , sensorFQN)
    if (sensor !== null) {
      lib.calculateEquations(
        sensor, topology, sensorFormulation,
            resource)
    }
  }

  // Step 6c: Calculate equations for
      actuators
  for (actuatorFQN : formulations.
      actuatorFormulation.keySet) {
    val actuatorFormulation =
      formulations.actuatorFormulation.get(
          actuatorFQN)
    val actuator = findActuatorByFQN(
        pmodel_out, actuatorFQN)
    if (actuator !== null) {
      lib.calculateEquations(
        actuator, topology,
            actuatorFormulation, resource)
    }
  }

  return pmodel_out
}
```

Listing A.38: calculateEquations Implementation

Figure A.13: Side-by-side comparison of calculateEquations algorithm and implementation.

A.3.1.13 Helper Functions: Component Lookup

```
Input: pmodel : PModel,
     sensorFQN : String
Output: sensor : Sensor

parts = split(sensorFQN, ".");
if |parts| ≠ 2 then
  return null;
end

linkName = parts[0];
sensorName = parts[1];

foreach link ∈ pmodel.links do
  if link.name = linkName then
    foreach sensor ∈ link.sensors do
      if sensor.name = sensorName then
        return sensor;
      end
    end
    foreach joint ∈ link.joints do
      foreach sensor ∈ joint.sensors do
        if sensor.name = sensorName then
          return sensor;
        end
      end
    end
  end
end

return null;
```

Listing A.39: `findSensorByFQN` (Pseudocode)

```
def Sensor findSensorByFQN(
   PModel pmodel, String sensorFQN) {
  val parts = sensorFQN.split("\\.")
  if (parts.length != 2) {
    return null
  }

  val linkName = parts.get(0)
  val sensorName = parts.get(1)

  for (link : pmodel.links) {
    if (link.name.equals(linkName)) {
      for (sensor : link.sensors) {
        if (sensor.name.equals(sensorName))
            {
          return sensor
        }
      }
      for (joint : link.joints) {
        for (sensor : joint.sensors) {
          if (sensor.name.equals(sensorName)
              ) {
            return sensor
          }
        }
      }
    }
  }
  return null
}
```

Listing A.40: findSensorByFQN Implementation

Figure A.14: Side-by-side comparison of findSensorByFQN algorithm and implementation.

```
Input: pmodel : PModel,
     actuatorFQN : String
Output: actuator : Actuator

parts = split(actuatorFQN, ".");
if |parts| ≠ 2 then
  return null;
end

linkName = parts[0];
actuatorName = parts[1];

foreach link ∈ pmodel.links do
  if link.name = linkName then
    foreach actuator ∈ link.actuators do
      if actuator.name = actuatorName then
        return actuator;
      end
    end
    foreach joint ∈ link.joints do
      foreach actuator ∈ joint.actuators do
        if actuator.name = actuatorName then
          return actuator;
        end
      end
    end
  end
end

return null;
```

Listing A.41: `findActuatorByFQN` (Pseudocode)

```
def Actuator findActuatorByFQN(
   PModel pmodel, String actuatorFQN) {
  val parts = actuatorFQN.split("\\.")
  if (parts.length != 2) {
    return null
  }

  val linkName = parts.get(0)
  val actuatorName = parts.get(1)

  for (link : pmodel.links) {
    if (link.name.equals(linkName)) {
      for (actuator : link.actuators) {
        if (actuator.name.equals(
            actuatorName)) {
          return actuator
        }
      }
      for (joint : link.joints) {
        for (actuator : joint.actuators) {
          if (actuator.name.equals(
              actuatorName)) {
            return actuator
          }
        }
      }
    }
  }
  return null
}
```

Listing A.42: findActuatorByFQN Implementation

Figure A.15: Side-by-side comparison of findActuatorByFQN algorithm and implementation.

A.3.1.14 Step 5b: Get Base Link

```
Input: pmodel : PModel
Output: baseLink : Link

baseLink = pmodel.links[0];
maxContainedLinks = 0;

foreach link ∈ pmodel.links do
  loopLinks = {};
  containedLinks = getChildLinks(
    pmodel, link, loopLinks);
  if size(containedLinks) >
     maxContainedLinks then
    maxContainedLinks =
      size(containedLinks);
    baseLink = link;
  end
end

return baseLink;
```

Listing A.43: Algorithm 28 (Pseudocode)

```
def getBaseLink(PModel pmodel) {
  var Link baseLink = pmodel.links.get(0)
  var EList<Link> containedLinks =
    new BasicEList<Link>()
  var EList<Link> loopLinks =
    new BasicEList<Link>()
  var int maxContainedLinks = 0

  for (link : pmodel.links) {
    loopLinks = new BasicEList<Link>()
    containedLinks = getChildLinks(
      pmodel, link, loopLinks)
    if (containedLinks.size >
       maxContainedLinks) {
      maxContainedLinks = containedLinks.
          size
      baseLink = link
    }
  }
  return baseLink
}
```

Listing A.44: getBaseLink Implementation

Figure A.16: Side-by-side comparison of getBaseLink algorithm and implementation.

A.3.1.15 Step 5c: Get Child Links

```
Input: pmodel : PModel,
     link : Link,
     loopLinks : Set<Link>
Output: containedLinks : List<Link>

containedLinks = {};

if link.joints = null then
 return containedLinks;
end

foreach joint ∈ link.joints do
 if joint.flexible ≠ null then
  childLink = joint.flexible.link;
  if childLink ∉ loopLinks then
   containedLinks = containedLinks ∪ {
       childLink};
   loopLinks = loopLinks ∪ {childLink};
   /* Recursively add descendants */
   descendants = getChildLinks(
    pmodel, childLink, loopLinks);
   containedLinks = containedLinks ∪
       descendants;
  end
 end
end

return containedLinks;
```

Listing A.45: Algorithm 29 (Pseudocode)

```
def EList<Link> getChildLinks(
  PModel pmodel, Link link,
  EList<Link> loopLinks) {
 var EList<Link> containedLinks =
  new BasicEList<Link>()
 var Link childLink
 var EList<Joint> childJoints = link.
     joints

 if (childJoints === null) {
  return containedLinks
 }

 for (joint : childJoints) {
  if (joint.flexible !== null) {
   childLink = joint.flexible.link
   if (!loopLinks.contains(childLink)) {
    containedLinks.add(childLink)
    loopLinks.add(childLink)
    // Recursively add descendants
    containedLinks.addAll(
     getChildLinks(pmodel, childLink,
      loopLinks))
   }
  }
 }

 return containedLinks
}
```

Listing A.46: getChildLinks Implementation

Figure A.17: Side-by-side comparison of getChildLinks algorithm and implementation.

A.3.2 Technique 4: Guided Choice

A.3.2.1 Main Algorithm: Technique4

```
Input: pmodel : pmodel+ with
     user-annotated SolutionRefs
Output: pmodel' : pmodel+,
      slnRefs : List<SolutionRef>

/* Step 1: Initialise sidecar and add Null
      type */
initialiseSidecar(pmodel);
addNullDatatype(pmodel);

/* Step 2: Calculate formulation */
formulation = calculateFormulation(pmodel)
     ;

/* Step 3: Unfold user-annotated solutions
   into sub-solutions (first pass) */
solution_ref = resolveSolutionFirstPass(
  pmodel, formulation);

/* Step 4: Reorder solutions by order
      property */
pmodel.solutions.sortBy(
  solution → solution.order);
solution_ref.sortBy(
  element → element.order);

/* Step 5: Resolve each solution in order
      */
foreach solution in pmodel.solutions do
  sln_ref = first element in solution_ref
    with matching expression name;
  for i = 0 to sln_ref.iterations do
    resolveSolution(
      pmodel, formulation, sln_ref);
  end
end

return pmodel, solution_ref;
```

Listing A.47: Algorithm 10 (Pseudocode)

```
def String Technique4(
   PModel pmodel, Resource resource) {

 // Step 1: Initialise sidecar
 initialiseSidecar(pmodel, sidecar)
 GeneratorServices.initialize()
 val Null = RoboChartFactory.eINSTANCE
  .createPrimitiveType
 Null.name = "Null"
 pmodel.addDatatype(Null)

 // Step 2: Calculate formulation
 var formulation =
  calculateFormulation(pmodel)
 var solution_ref =
  new BasicEList<SolutionRef>
 resource.save(...)

 // Step 3: First pass resolution
 solution_ref = resolveSolutionFirstPass(
  pmodel, formulation, resource)
 resource.save(...)

 // Step 4: Reorder solutions
 pmodel.solutions.sortBy[order]
 solution_ref.sortBy[order]

 // Step 5: Iterative resolution
 var slns_PM = new BasicEList<Solution>
 for (sln_PM : pmodel.solutions) {
  var sln_ref = solution_ref.findFirst[
   expression.name ==
    sln_PM.solutionExpr.name]
  slns_PM = resolveSolution(
   pmodel, formulation,
   sln_ref, resource)
 }

 // Remove Null type before output
 container.types.remove(nullType)
 rebindSolutionConstraintSNames(pmodel)
 return serializer.serialize(resource.
     contents.get(0))
}
```

Listing A.48: Technique4 Implementation

Figure A.18: Side-by-side comparison of the Technique 4 main algorithm and its implementation.

Implementation Notes. The implementation includes several resource save operations (`resource.save(...)`) after major steps. These are necessary to mitigate Eclipse's dirty resource state issues, where the EMF model becomes inconsistent with the underlying file representation. While these saves ensure correctness, they significantly slow down the generator. The implementation explicitly removes the temporary Null type before serialisation and calls `rebindSolutionConstraintsNames` to normalise qualified-name references in constraints. Finally, although Algorithm 10 expresses iteration in Step 5 as an explicit inner loop, the implementation performs this loop inside `resolveSolution`; the top-level method invokes `resolveSolution` once per solution, and the `iterations` field controls the number of refinement passes.

A.3.2.2 Step 1: Initialise Sidecar

```
Input: pmodel : PModel,
     sidecar : Map<DynamicDevice,
                Map<String, VarInfo>>
Output: sidecar (populated)

/* Collect devices: root + all contained
    */
devices = {pmodel} ∪
  pmodel.dynamicDevices;

/* Seed per-device variable maps */
foreach device ∈ devices do
  vars = {};
  foreach v ∈ device.locals do
    vars[v.name] = VarInfo(null, false,
                      {}, {});
  end
  foreach v ∈ device.inOuts ∪
           device.inputs ∪
           device.outputs ∪
           device.constants do
    if v.initial ≠ null then
      vars[v.name] = VarInfo(
        v.initial,
        isAllLiterals(v.initial),
        {}, {});
    else
      vars[v.name] = VarInfo(null, false,
                        {}, {});
    end
  end
  sidecar[device] = vars;
  populateSubExprs(device, sidecar);
end

return sidecar;
```

Listing A.49: Algorithm 11 (Pseudocode)

```
def initialiseSidecar(PModel pmodel,
   Map<DynamicDevice,
      Map<String, VarInfo>> sidecar) {
  sidecar.clear

  // Collect devices: root + all contained
  val List<DynamicDevice> devices =
    new ArrayList<DynamicDevice>()
  devices.add(pmodel)
  for (d : EcoreUtil2.getAllContentsOfType
      (
     pmodel, DynamicDevice)) {
    devices.add(d)
  }

  // Seed per-device variable maps
  for (device : devices) {
    val Map<String, VarInfo> varMap =
      Maps.newHashMap

    // Locals (always unresolved)
    for (vl : device.getLocals()) {
      for (v : vl.getVars()) {
        varMap.put(v.getName,
          new VarInfo(null, false,
            new ArrayList<String>(),
            new ArrayList<Expression>()))
      }
    }

    // InOuts, Inputs, Outputs, Constants
    for (list : #[device.getInOuts,
       device.getInputs, device.getOutputs,
       device.getConstants]) {
      for (vl : list) {
        for (v : vl.getVars()) {
          if (v.initial !== null) {
            varMap.put(v.getName,
              new VarInfo(
                EcoreUtil.copy(v.initial),
                isAllLiterals(v.initial),
                new ArrayList<String>(),
                new ArrayList<Expression>()))
          } else {
            varMap.put(v.getName,
              new VarInfo(null, false,
                new ArrayList<String>(),
                new ArrayList<Expression>()))
          }
        }
      }
    }
    sidecar.put(device, varMap)
    populateSubExprs(device, sidecar)
  }
}
```

Listing A.50: initialiseSidecar Implementation

Figure A.19: Side-by-side comparison of initialiseSidecar algorithm and implementation.

Implementation Notes. The implementation uses `EcoreUtil2.getAllContentsOfType` to collect all dynamic devices from the model hierarchy, which is functionally equivalent to the pseudocode's set union. The implementation explicitly clears the sidecar at the start to handle

cases where the generator is invoked multiple times. Variable initial values are deep-copied using `EcoreUtil.copy` to avoid sharing mutable EMF objects between the original model and the sidecar state.

A.3.2.3 Step 3: Resolve Solution First Pass

```
Input: pmodel : PModel, formulation : Formulation
Output: slnRefList : List<SolutionRef>

ctx = getFullyQualifiedName(pmodel);
slnRefList = [];
slnsPM = [];
pm2 = copy(pmodel);

/* Step 1: Convert each p-model solution to SolutionRef
    */
foreach slnPM ∈ pm2.solutions do
  slnRef = new SolutionRef();
  slnRef.expression.name = slnPM.solutionExpr.name;
  slnRef.expression.type =
    printType(slnPM.solutionExpr.type);
  slnRef.method = slnPM.method;
  slnRef.order = slnPM.order;
  /* Convert constraints and inputs to strings */
  if slnPM.constraints ≠ null then
    foreach constraint ∈ slnPM.constraints do
      slnRef.constraints.add(
        printExpression(constraint));
    end
  end
  if slnPM.inputs ≠ null then
    foreach input ∈ slnPM.inputs do
      inputRef.name = input.name;
      inputRef.type = printType(input.type);
      slnRef.inputs.add(inputRef);
    end
  end

  /* Step 2: Unfold via solution library */
  resolvedRef = SolutionLib.resolveSolution(
    formulation, slnRef);

  /* Step 3: Remove original solution */
  pmodel.solutions.remove(slnPM);

  /* Step 4: Process each sub-solution */
  foreach ref ∈ ensureList(resolvedRef) do
    ref.expression = updateVariable(
      pmodel, ref.expression);
    foreach input ∈ ref.inputs do
      input = updateVariable(pmodel, input);
    end
    ref = calculateInitialConditions(pmodel, ref);
    ref = processAlgebraicConstraints(pmodel, ref);
    sln = createSolution(pmodel, ctx, ref);
    slnsPM.add(sln);
    slnRefList.add(ref);
  end
end

/* Step 5: Add unfolded solutions to p-model */
foreach sln ∈ slnsPM do
  pmodel.solutions.add(sln);
end

return slnRefList;
```

Listing A.51: Algorithm 12 (Pseudocode)

```
def BasicEList<SolutionRef>
    resolveSolutionFirstPass(PModel pmodel,
    Formulation formulation, Resource resource) {
  var ctx = qnp.getFullyQualifiedName(pmodel)
    .toString("::")
  var sln_ref_out = new BasicEList<SolutionRef>
  var slns_PM = new BasicEList<Solution>
  var pm2 = EcoreUtil.copy(pmodel)

  for (sln_PM2 : pm2.solutions) {
    var solution = new SolutionRef
    solution.expression = new Local()
    solution.expression.name =
      sln_PM2.solutionExpr.name
    solution.expression.type =
      HeadlessPrintingServices
        .printTypeStatic(sln_PM2.solutionExpr.type)
    solution.method = sln_PM2.method
    solution.order = sln_PM2.order

    if (sln_PM2.constraints !== null) {
      solution.constraints = new BasicEList<String>
      for (constraint : sln_PM2.constraints) {
        solution.constraints.add(
          HeadlessPrintingServices
            .printExpression(constraint))
      }
    }
    if (sln_PM2.inputs !== null) {
      solution.inputs = new BasicEList<Local>
      for (input : sln_PM2.inputs) {
        var input_ref = new Local()
        input_ref.name = input.name
        input_ref.type = HeadlessPrintingServices
          .printTypeStatic(input.type)
        solution.inputs.add(input_ref)
      }
    }

    var solution_ref = SolutionLib
      .resolveSolution(formulation, solution)
    pmodel.solutions.removeIf[
      element.solutionExpr.name ==
        sln_PM2.solutionExpr.name]

    // Handle single or list return
    if (solution_ref instanceof SolutionRef) {
      solution_ref.expression = updateVariable(
        pmodel, solution_ref.expression, resource)
      // ... update inputs, calculate ICs,
      // process algebraic constraints ...
      slns_PM.add(createSolution(
        pmodel, ctx, solution_ref))
      sln_ref_out.add(solution_ref)
    } else if (solution_ref instanceof BasicEList) {
      for (sln : solution_ref) {
        // ... same processing for each ...
      }
    }
  }

  for (sln_PM : slns_PM) {
    pmodel.solutions.add(sln_PM)
  }
  return sln_ref_out
}
```

Listing A.52: resolveSolutionFirstPass Implementation

Figure A.20: Side-by-side comparison of resolveSolutionFirstPass algorithm and implementation.

Implementation Notes. The implementation handles two return types from `SolutionLib.resolveSolution`: a single `SolutionRef` or a `BasicEList<SolutionRef>`. This polymorphism allows the library to return either a single resolved solution or multiple sub-solutions

depending on the method being unfolded. The implementation uses `removeIf` with a lambda predicate rather than explicit removal by reference. The solution copy (`pm2 = EcoreUtil.copy(pmodel)`) is necessary because iterating over a list while modifying it causes concurrent modification issues.

A.3.2.4 Step 3a: Update Variable

```
Input: pmodel : PModel,
     varRef : Local
Output: Updated varRef

if varRef.type = null then
  varRef.type = "Null";
end

varPM = pmodel.getVariable(varRef.name);

if varPM = null and varRef.type ≠ null
     then
  /* Variable doesn't exist, add it */
  pmodel.addLocal(varRef.name, varRef.type
       );
  varPM = pmodel.getVariable(varRef.name);
else if varPM.type = "Null" then
  /* Variable has placeholder type */
  varPM.type = pmodel.createType(varRef.
       type);
end

if varPM ≠ null then
  varRef.type = printType(varPM.type);
end

return varRef;
```

Listing A.53: Algorithm 13 (Pseudocode)

```
def Local updateVariable(PModel pmodel,
   Local var_ref, Resource resource) {
  var sln_out = var_ref
  if (var_ref.type === null) {
    var_ref.type = "Null"
  }

  val varPM = pmodel.locals
    .flatMap[vars]
    .findFirst[name == var_ref.name]

  if (varPM === null &&
     var_ref.type !== null) {
    // Add new variable to p-model
    circus.robocalc.robosim.physmod.
        generator
      .eqnComp.GeneratorServices.initialize
          ()
    val newVar = GeneratorServices
      .readVariable(pmodel,
       "var " + var_ref.name + ":" +
         var_ref.type,
       pmodel?.eResource)
    if (newVar?.object?.vars?.size > 0) {
      pmodel.locals.add(newVar.object)
    }
  } else if (varPM !== null &&
      HeadlessPrintingServices
        .printTypeStatic(varPM.type) == "
            Null") {
    // Update placeholder type
    varPM.type = createTypeFromString(
      pmodel, var_ref.type, resource)
  }

  if (varPM !== null) {
    var_ref.type = HeadlessPrintingServices
      .printTypeStatic(varPM.type)
  }

  return sln_out
}
```

Listing A.54: updateVariable Implementation

Figure A.21: Side-by-side comparison of updateVariable algorithm and implementation.

Implementation Notes. The implementation uses `GeneratorServices.readVariable` to parse and create new variable declarations from string specifications. This leverages the existing Xtext parsing infrastructure rather than manually constructing EMF objects. The nested `flatMap` and `findFirst` operations navigate the EMF model structure to locate variables within `VariableList` containers.

A.3.2.5 Step 4: Reorder Solutions

```
Input: pmodel : PModel+,
     slnRefList : List<SolutionRef>
Output: pmodel and slnRefList
      with solutions sorted by order

/* Sort p-model solutions */
pmodel.solutions.sortBy(
  sln → sln.order);

/* Sort solution references */
slnRefList.sortBy(
  ref → ref.order);
```

Listing A.55: Algorithm 43 (Pseudocode)

```
// In Technique4 method:

// Reorder solutions to correct order
pmodel.solutions.sortBy(
  solution | (solution.order))
solution_ref.sortBy(
  element | element.order)
```

Listing A.56: Reorder Solutions Implementation

Figure A.22: Side-by-side comparison of the reorder solutions step and its implementation.

Implementation Notes. The implementation uses Xtend's lambda syntax (`element | element.order`) which is equivalent to the pseudocode's arrow notation. The parentheses around `solution.order` in the first sort are optional but included for clarity.

A.3.2.6 Step 5: Resolve Solution (Iterative)

```
Input: pmodel : PModel+,
     formulation : Formulation,
     slnRef : SolutionRef
Output: slnList : List<Solution>

slnList = [];
ctx = getFullyQualifiedName(pmodel);

/* Perform iterations */
for i = 0 to slnRef.iterations do
  /* Step 1: Resolve via library */
  slnRef = SolutionLib.resolveSolutionRef(
    formulation, slnRef);

  /* Step 2: Update expression variable */
  slnRef.expression = updateVariable(
    pmodel, slnRef.expression);

  /* Step 3: Update input variables */
  foreach input ∈ slnRef.inputVariables do
    input = updateInputVariable(pmodel,
        input);
  end

  /* Step 4: Calculate initial conditions
      */
  slnRef = calculateInitialConditions(
    pmodel, slnRef);

  /* Step 5: Process algebraic constraints
      */
  slnRef = processAlgebraicConstraints(
    pmodel, slnRef);

  /* Step 6: Create and add solution */
  sln = createSolution(pmodel, ctx, slnRef
      );
  pmodel.solutions.add(sln);
  slnList.add(sln);
end

return slnList;
```

Listing A.57: Algorithm 14 (Pseudocode)

```
def BasicEList<Solution> resolveSolution(
   PModel pmodel, Formulation formulation,
   SolutionRef sln_ref, Resource resource)
       {
  var ctx = qnp.getFullyQualifiedName(
      pmodel)
    .toString("::")
  var solution_ref = sln_ref
  var sln_PM = PhysModFactory.eINSTANCE
    .createSolution
  var PM_out = EcoreUtil.copy(pmodel)

  for (i : 0 ..< sln_ref.iterations) {
    sln_PM = PhysModFactory.eINSTANCE
      .createSolution
    solution_ref = SolutionLib.
        resolveSolution(
      formulation, solution_ref) as
          SolutionRef

    solution_ref.expression =
        updateVariable(
      pmodel, solution_ref.expression,
          resource)

    if (solution_ref.inputs !== null) {
      for (input : solution_ref.inputs) {
        val index = solution_ref.inputs
          .indexOf(input)
        solution_ref.inputs.set(index,
          updateVariable(pmodel, input,
              resource))
      }
    }

    solution_ref =
        calculateInitialConditions(
      pmodel, solution_ref) as SolutionRef

    try {
      solution_ref =
          processAlgebraicConstraints(
        pmodel, solution_ref, this.sidecar)
    } catch (Exception e) {
      logger.error("Failed: " + e.
          getMessage, e)
    }

    val sln_PM_ = createSolution(
      pmodel, ctx, solution_ref)
    var result = pmodel.solutions.findFirst
        [
      solutionExpr.name ==
        sln_PM_.solutionExpr.name]
    val index = pmodel.solutions.indexOf(
        result)
    pmodel.solutions.set(index, sln_PM)
  }

  return PM_out.solutions as BasicEList<
      Solution>
}
```

Listing A.58: resolveSolution Implementation

Figure A.23: Side-by-side comparison of resolveSolution algorithm and implementation.

Implementation Notes. The implementation wraps `processAlgebraicConstraints` in a try-catch block to handle potential resolution failures without aborting the entire generation process. It updates existing solutions in-place using `pmodel.solutions.set(index, sln_PM)` rather than adding new solutions, preventing duplicate entries. Resource saves are performed in `Technique4` after major phases (e.g., after the first pass and after all solutions have been resolved), rather than on each iteration, to avoid the performance cost of frequent serialisation.

A.3.2.7 Step 5 Main Loop

```
Input: pmodel : PModel,
     formulation : Formulation,
     slnRefList : List<SolutionRef>
Output: Updated pmodel with all
      solutions resolved

/* Process each solution in order */
foreach slnPM ∈ pmodel.solutions do
 /* Find matching solution reference */
 slnRef = findMatchingSolutionRef(
  slnRefList, slnPM);
 if slnRef ≠ null then
  /* Perform iterative resolution */
  resolveSolution(
   pmodel, formulation, slnRef);
 end
end
```

Listing A.59: Algorithm 15 (Pseudocode)

```
// In Technique4 method:

var slns_PM = new BasicEList<Solution>
for (sln_PM : pmodel.solutions) {
 var sln_ref = solution_ref.findFirst(
  element | element.expression.name ==
   sln_PM.solutionExpr.name)
 slns_PM = resolveSolution(
  pmodel, formulation, sln_ref, resource)
}
```

Listing A.60: Step 5 Main Loop Implementation

Figure A.24: Side-by-side comparison of Step 5 main loop and its implementation.

Implementation Notes. The `findMatchingSolutionRef` function from the pseudocode is implemented inline using `findFirst` with a lambda that compares expression names. In normal execution a match is guaranteed because Step 3 constructs a `SolutionRef` for each p-model solution, and Step 4 sorts both structures consistently before Step 5.

A.3.2.8 Calculate Initial Conditions (Detailed)

```
Input: pmodel : PModel, slnRef : SolutionRef
Output: Updated slnRef with resolved constraints

foreach constraint ∈ slnRef.constraints do
 /* Skip algebraic constraints [t==t] */
 if constraint matches "[t == t]" then
  continue;
 end

 if constraint matches "[t == 0]" then
  expr = extractLeftExpression(constraint);
  varName = printExpression(expr);

  /* Evaluate variable using sidecar */
  evaluateVariable(pmodel, varName, sidecar);
  varInfo = getSidecarInfo(
   sidecar, pmodel, varName);

  if varInfo.resolved then
   resolvedValue = printExpression(varInfo.value);
   /* Clean artefacts and handle placeholders */
   resolvedValue = cleanEquationArtifacts(
    resolvedValue);
   if resolvedValue = "zeroMat(0,0)" then
    dims = extractDimensions(pmodel, expr);
    resolvedValue = "zeroMat(" +
     dims[0] + ", " + dims[1] + ")";
   end
   constraint = "(" + varName + ") [t == 0] ==" +
    resolvedValue;
  end
 end
end

return slnRef;
```

Listing A.61: Algorithm 40 (Pseudocode)

```
def SolutionRef calculateInitialConditions(
   PModel pmodel, SolutionRef solution) {
 val pmQN = qnp.getFullyQualifiedName(pmodel)
 var ctx = pmQN?.toString("::")
 circus.robocalc.robosim.physmod.generator
  .eqnComp.GeneratorServices.initialize()

 for (constraint : solution.constraints) {
  // Skip [t==t] algebraic constraints
  val hasAlgebraicTime = constraint
   .matches(".*\\[\\s*t\\s*==\\s*t\\s*\\].*")
  if (hasAlgebraicTime) {
   // Skip - handled by processAlgebraicConstraints
  } else {
   // Process [t==0] constraints
   val constr = GeneratorServices
    .readPMExpression(constraint, ctx,
     pmodel?.eResource)
   if (constr instanceof Equals) {
    val eq = constr as Equals
    if (eq.left instanceof IBCondition) {
     val c = eq.left as IBCondition
     val expr = c.exp
     val tempV = HeadlessPrintingServices
      .printExpression(expr)
     if (tempV !== null) {
      evaluateVariable(pmodel,
       pmodel as EObject, tempV, sidecar)
      val devMap = sidecar.get(pmodel)
      val varInfo = devMap?.get(tempV)
      if (varInfo?.resolved) {
       var rv = GuidedChoiceUtils
        .cleanEquationArtifacts(
         resolvedValue)
       rv = GuidedChoiceUtils
        .normalizeWhitespace(rv)
       if (rv == "zeroMat(0,0)") {
        val tStr = getTypeStrForLeftExpr(
         pmodel, expr)
        val dims = extractMatrixDims(tStr)
        if (dims !== null) {
         rv = "zeroMat(" + dims.get(0) +
          ", " + dims.get(1) + ")"
        }
       }
       // Similar handling for zeroVec(0)
       solution.constraints.set(index,
        "(" + exprStr + ") [ t == 0] ==" + rv)
      }
     }
    }
   }
  }
 }
 return solution
}
```

Listing A.62: calculateInitialConditions Implementation

Figure A.25: Side-by-side comparison of calculateInitialConditions algorithm and implementation.

Implementation Notes. The implementation uses regular expressions to detect algebraic constraints with various whitespace formats. The `cleanEquationArtifacts` and `normalizeWhitespace` utility functions remove parser artefacts and standardise formatting. The implementation also handles `zeroVec(0)` placeholders similarly to `zeroMat(0,0)`, extracting dimensions from the variable's declared type.

A.3.2.9 Process Algebraic Constraints (Detailed)

```
Input: pmodel : PModel, slnRef : SolutionRef,
     sidecar : Map<String, VarInfo>
Output: Updated slnRef with resolved constraints

/* Stage 1: Propagate system correspondence */
propagateSystemCorrespondence(pmodel, sidecar);

/* Stage 2: Detect variables needing constraints */
if slnRef.inputs ≠ null then
 foreach input ∈ slnRef.inputs do
   varName = input.name;
   hasInitialCondition = ∃ c ∈ slnRef.constraints :
    c.matches("(varName)[t == 0]");
   if ¬hasInitialCondition then
    hasAlgebraicConstraint = ∃ c ∈ slnRef.constraints :
     c.matches("(varName)[t == t]");
    if ¬hasAlgebraicConstraint then
     /* Stage 3: Generate algebraic constraint */
     equation = getEquation(
      pmodel, varName, sidecar);
     if equation ≠ null and equation ≠ "" then
      algebraicConstraint =
       "(varName) [t == t] ==" + equation;
      slnRef.constraints.add(algebraicConstraint);
     end
    end
   end
 end
end

/* Process existing [t==t] constraints */
foreach constraint ∈ slnRef.constraints do
 if constraint matches "[t == t]" then
   expr = extractLeftExpression(constraint);
   varName = printExpression(expr);
   equation = getEquation(pmodel, varName, sidecar);
   if equation = null then
    resolvedExpr = evaluateVariableAlgebraic(
     pmodel, varName, sidecar);
    equation = printExpression(resolvedExpr);
   end
   if equation ≠ null then
    equation = cleanEquationArtifacts(equation);
    constraint = "(varName) [t==t] ==" + equation;
   end
 end
end

return slnRef;
```

Listing A.63: Algorithm 41 (Pseudocode)

```
def SolutionRef processAlgebraicConstraints(
   PModel pmodel, SolutionRef solution,
   Map<DynamicDevice,
      Map<String, VarInfo>> sidecar) {

 // Stage 1: Propagate system correspondence
 // (handled during sidecar initialization)

 // Stage 2: Detect and generate missing constraints
 if (solution.inputs !== null) {
  for (input : solution.inputs) {
   val varName = input.name
   val hasInitialCondition =
    solution.constraints.exists[
     matches(".*\\(" + Pattern.quote(varName) +
      "\\).*\\[\\s*t\\s*==\\s*0\\s*\\].*")]
   if (!hasInitialCondition) {
    val hasAlgebraicConstraint =
     solution.constraints.exists[
      matches(".*\\(" + Pattern.quote(varName) +
       "\\).*\\[\\s*t\\s*==\\s*t\\s*\\].*")]
    if (!hasAlgebraicConstraint) {
     // Stage 3: Generate constraint
     val equation = utils.getEquation(
      pmodel, varName, sidecar)
     if (equation !== null &&
        !equation.isEmpty) {
      solution.constraints.add(
       "(" + varName + ") [t == t] ==" +
        equation)
     }
    }
   }
  }
 }

 // Process existing [t==t] constraints
 for (var i = 0; i < solution.constraints.size;
     i++) {
  val constraint = solution.constraints.get(i)
  if (constraint.matches(
     ".*\\[\\s*t\\s*==\\s*t\\s*\\].*")) {
   val varName = extractVarName(constraint)
   var equation = utils.getEquation(
    pmodel, varName, sidecar)
   if (equation === null) {
    val resolvedExpr = utils
     .evaluateVariableAlgebraic(
      pmodel, varName, sidecar)
    equation = HeadlessPrintingServices
     .printExpression(resolvedExpr)
   }
   if (equation !== null) {
    equation = GuidedChoiceUtils
     .cleanEquationArtifacts(equation)
    solution.constraints.set(i,
     "(" + varName + ") [t == t] ==" +
      equation)
   }
  }
 }

 return solution
}
```

Listing A.64: processAlgebraicConstraints Implementation

Figure A.26: Side-by-side comparison of processAlgebraicConstraints algorithm and implementation.

Implementation Notes. The implementation uses `Pattern.quote` to escape variable names in regular expressions, preventing special characters from being interpreted as regex metacharacters. The three-stage process (propagate system correspondence, detect missing constraints, generate constraints) is preserved in the implementation, though Stage 1 may be performed during sidecar initialisation rather than inline. The loop uses index-based iteration (`for (var i = 0; i < size; i++)`) rather than foreach when modifying the collection to avoid concurrent modification issues.

A.3.2.10 evaluateVariable (DFS-based Resolution)

```
Input: pmodel, object, varName, sidecar
Output: Expression (resolved value)

# Phase 1: Identify device and check cache
device = identifyDevice(object)
deviceVars = sidecar.get(device)
if deviceVars ≠ null and
  deviceVars.get(varName).resolved:
 return deviceVars.get(varName).value

# Phase 2: Check IB constraints [t==0]
foreach constraint ∈ device.constraints:
 if constraint is IBCondition
   with [t==0] matching varName:
   rhs = constraint.right
   constants = buildConstantsMap(
     sidecar, device)
   resolved = resolveExpression(
     rhs, constants)
   updateSidecarWithValue(
     device, varName, resolved, sidecar)
   if sidecar[device][varName].resolved:
     return sidecar[device][varName].value

# Phase 3: Resolve constraints via DFS
deps = evaluateVariableWRTConstraints(
 varName, device, sidecar, {})
if sidecar[device][varName].resolved:
 return sidecar[device][varName].value

# Phase 4: Try direct relation
direct = tryDirectRelation(
 pmodel, object, varName, sidecar)
if isAllLiterals(direct):
 return direct

# Phase 5: Try relations for dependencies
foreach (depDevice, depVars) ∈ deps:
 foreach depVar ∈ depVars.keySet():
   tryDirectRelation(
     pmodel, object, depVar, sidecar)

# Phase 6: Final constraint folding
evaluateVariableWRTConstraints(
 varName, device, sidecar, {})
return sidecar[device][varName].value
```

Listing A.65: Algorithm 39 (Pseudocode)

```
def Expression evaluateVariable(
   PModel pmodel, EObject object,
   String varName,
   Map<DynamicDevice,
      Map<String, VarInfo>> sidecar) {

 // Phase 1: Identify device and check cache
 val DynamicDevice device =
  if (object instanceof PModel)
    object as DynamicDevice
  else if (object instanceof LocalRealisation)
    (object as LocalRealisation).def
  else throw new IllegalArgumentException(...)

 val deviceVars = sidecar.get(device)
 if (deviceVars !== null) {
  val info = deviceVars.get(varName)
  if (info !== null && info.resolved)
   return info.value
 }

 // Phase 2: Check IB constraints [t==0]
 for (constraint : device.constraints) {
  if (constraint instanceof Equals) {
   val eq = constraint as Equals
   if (eq.left instanceof IBCondition) {
    val ibCond = eq.left as IBCondition
    if (isInitialCondition(ibCond) &&
       matchesVariableExpression(
         ibCond.exp, varName)) {
      val rhs = eq.right
      val constants = Maps.newHashMap
      // ... build constants map ...
      val resolvedPair =
       resolveExpression(rhs, constants, true)
      updateSidecarWithValue(
       device, varName,
       resolvedPair.key, sidecar, null)
      val finalInfo =
       sidecar.get(device)?.get(varName)
      if (finalInfo?.resolved)
       return finalInfo.value
    }
   }
  }
 }

 // Phase 3: First pass via DFS
 val deps = evaluateVariableWRTConstraints(
  varName, device, sidecar,
  new HashSet<Pair<DynamicDevice,String>>())
 val postInfo = sidecar.get(device)?.get(varName)
 if (postInfo !== null && postInfo.resolved)
  return postInfo.value

 // Phase 4-6: Relations and final fold
 val direct = tryDirectRelation(
  pmodel, object, varName, sidecar)
 if (isAllLiterals(direct)) return direct
 for (devEntry : deps.entrySet) { /* ... */ }
 evaluateVariableWRTConstraints(...)
 return sidecar.get(device)?.get(varName)?.value
}
```

Listing A.66: evaluateVariable Implementation

Figure A.27: Side-by-side comparison of evaluateVariable algorithm and implementation.

Implementation Notes. The implementation uses Xtend's type dispatch pattern to identify the device type, throwing `IllegalArgumentException` for unsupported types. The `matchesVariableExpression` helper checks whether an expression references the target variable name. The null-safe navigation operator (`?.`) is used extensively when accessing sidecar entries that may not exist. The `HashSet<Pair<DynamicDevice,String>>` passed to `evaluateVariableWRTConstraints` serves as a cycle guard to prevent infinite recursion in the DFS traversal.

A.3.2.11 getEquation (Hierarchical Equation Lookup)

```
Input: pmodel, varName, sidecar
Output: String (equation) or null

# Step 1: Search global model equations
equation = findEquationInGlobalModel(
  pmodel, varName)
if equation ≠ null:
  return equation

# Step 2: Parse variable name for index
(baseVarName, varIndex) =
  parseIndexedVariable(varName)

# Step 3: Search numbered elements
if varIndex > 0:
  equation = findEquationInNumberedElement(
    pmodel, baseVarName, varIndex)
  if equation ≠ null:
    return equation

# Step 4: Try sidecar data
if sidecar ≠ null:
  equation = resolveFromSidecar(
    pmodel, varName, sidecar)
  if equation ≠ null and
     not isIdentityMatrix(equation):
    return equation

# Step 5: For X_J_i, use algebraic eval
if varName matches "X_J_\d+":
  expr = evaluateVariableAlgebraic(
    pmodel, pmodel, varName, sidecar)
  if expr ≠ null:
    return printExpression(expr)

# Step 6: Derive from constraints
equation = deriveEquationFromConstraints(
  pmodel, varName, baseVarName, varIndex)
return equation
```

Listing A.67: Algorithm 42 (Pseudocode)

```
def private String getEquation(
    PModel pmodel, String varName,
    Map<DynamicDevice,
        Map<String, VarInfo>> sidecar) {
  if (varName === null || pmodel === null)
    return null

  // Step 1: Search global equations
  var foundEquation =
    findEquationInGlobalModel(pmodel, varName)
  if (foundEquation !== null)
    return foundEquation

  // Step 2: Parse indexed variable
  val indexPattern = "(.+?)_(\\d+)$"
  val indexMatcher = java.util.regex.Pattern
    .compile(indexPattern).matcher(varName)
  var baseVarName = varName
  var varIndex = -1
  if (indexMatcher.find()) {
    baseVarName = indexMatcher.group(1)
    try {
      varIndex = Integer
        .parseInt(indexMatcher.group(2))
    } catch (NumberFormatException e) { }
  }

  // Step 3: Search numbered elements
  if (varIndex > 0) {
    foundEquation = findEquationInNumberedElement(
      pmodel, baseVarName, varIndex)
    if (foundEquation !== null)
      return foundEquation
  }

  // Step 4: Try sidecar
  if (sidecar !== null) {
    foundEquation = resolveFromSidecar(
      pmodel, varName, sidecar)
    if (foundEquation !== null) {
      if (isIdentityMatrix(foundEquation)) {
        val derived = deriveEquationFromConstraints(
          pmodel, varName, baseVarName, varIndex)
        if (derived !== null &&
            !isIdentityMatrix(derived))
          return derived
      }
      return foundEquation
    }
  }

  // Step 5: X_J_i pattern
  if (varName.matches("X_J_\\d+")) {
    val expr = utils.evaluateVariableAlgebraic(
      pmodel, pmodel, varName, sidecar)
    if (expr !== null) {
      val equation = HeadlessPrintingServices
        .printExpression(expr)
      if (equation !== null && !equation.trim.empty)
        return equation
    }
  }

  // Step 6: Derive from constraints
  return deriveEquationFromConstraints(
    pmodel, varName, baseVarName, varIndex)
}
```

Listing A.68: getEquation Implementation

Figure A.28: Side-by-side comparison of getEquation algorithm and implementation.

Implementation Notes. The implementation uses Java regex (`Pattern.compile`) to parse indexed variable names like `M_1` or `X_J_2` into base name and index components. The pattern `(.+?)_(\d+)$` captures the base name non-greedily and the trailing numeric index. The `isIdentityMatrix` check handles a special case where the sidecar returns an identity matrix placeholder that should be replaced with the actual derived equation. The `findEquationInNumberedElement` helper searches through links and joints by their `number` field to locate equations for indexed system variables. Trace logging (visible in the implementation) aids debugging but is omitted from the pseudocode.

A.3.3 Technique 5: Generation for a Platform Engine

A.3.3.1 Main Algorithm: sourceCodeGen Pipeline

```
Input: slnRefFile, resource
Output: generated files

slnRefs = parseSlnRefs(resource)
solution = compile(slnRefs)
write(solution, "*.sln")

mode = getGenerationMode()
format = getOutputFormat()
if format == "latex":
  generateLatex(solution)
else if format == "isabelle":
  generateIsabelle(solution)
else:
  generateCppMultiFile(solution, mode)
```

Listing A.69: sourceCodeGen Pipeline (Pseudocode)

```
def void doGenerate(
    Resource resource,
    IFileSystemAccess2 fsa,
    IGeneratorContext context) {

  val slnRefs = extractSlnRefs(resource)
  val solution = compile(slnRefs)
  writeSln(solution, fsa)

  val mode = getGenerationMode()
  val format = getOutputFormat()
  switch format {
    case "latex":
      latexGen.generate(solution, fsa)
    case "isabelle":
      isabelleGen.generate(solution, fsa)
    default:
      multiFileGen.generate(solution, fsa,
          mode)
  }
}
```

Listing A.70: SolutionRefGenerator Implementation

Figure A.29: Side-by-side comparison of sourceCodeGen pipeline and implementation.

Implementation Notes. The implementation delegates the slnRef-to-slnDF assembly to `compile(slnRefs)`, which corresponds to the imperative translation algorithm in Appendix A.2.3.1 (Algorithm 46). The `slnRefs` list is already ordered by Technique 4, so the compile step preserves solution ordering when concatenating computation blocks. The `getOutputFormat` method reads the `physmod.output.format` system property and defaults to C++ output when unset; LaTeX and Isabelle targets are dispatched explicitly.

A.4 Physics Engine Evaluation Reference Locations

This appendix provides the documentation references supporting the physics engine evaluation tables (Tables 2.3–2.5). Each lettered sub-reference in the evaluation tables corresponds to a row below, which records where (in the engine documentation and/or source tree) the relevant capability is specified or implemented.

Table A.2: MuJoCo Documentation locations for evaluation references.

Reference	Exact location
[4]a	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: pose/orientation specified in MJCF via `body`/`geom` orientation attributes (`quat`, `axisangle`, `euler`, `xyaxes`, `zaxis`).

Continued on next page

Table A.2 – *Continued from previous page*

Reference	Exact location
[4]b	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: kinematics computed from state in `mjData.qpos` (and `qvel` where needed); see `mj_forward` / `mj_kinematics`.
[4]c	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: forward-kinematics outputs stored in `mjData` (e.g., `xpos`, `xquat`, `xmat`).
[4]d	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: dynamics parameterisation in MJCF via `inertial`, `joint`, `body`, and global `option` settings (e.g., gravity).
[4]e	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: dynamics outputs in `mjData` (e.g., `qacc`, `qfrc_inverse` via `mj_inverse`).
[4]f	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: collision/contact parameters on `geom` (e.g., `contype`, `conaffinity`, `condim`, `friction`, `solref`, `solimp`).
[4]g	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: time-step and integrator selection via `option timestep=...` and `integrator=...`.
[4]h	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: actuator definitions under `actuator` (e.g., `motor`) with `gear` and actuator gains.
[4]i	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: sensor definitions under `sensor` (sensor type, referenced object, `noise`, `cutoff`).
[4]j	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: sensor outputs stored in `mjData.sensordata`.
[4]k	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: `mjModel.sensor_*` arrays define the ordering and type of entries in `mjData.sensordata`.
[4]l	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: Jacobian routines provide angular and linear parts (6D motion representation); see `mj_jacBody` / related Jacobian APIs.
[4]m	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: Euler-angle pose specification via `euler` attributes (e.g., on `body`/`geom`).
[4]n	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: axis–angle pose specification via `axisangle` attributes.

Continued on next page

Table A.2 – *Continued from previous page*

Reference	Exact location
[4]o	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: quaternion pose specification via `quat` attributes; runtime orientation in `mjData.xquat`.
[4]p	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: rotation matrices available via `mjData.xmat` and conversion helpers (`mju_*`).
[4]q	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: `mj_forward` / `mj_kinematics` compute FK and populate `mjData` pose fields.
[4]r	MuJoCo paper [4]: describes the articulated-body dynamics core used in MuJoCo (Featherstone-style ABA).
[4]s	MuJoCo API https://mujoco.readthedocs.io/en/stable/APIreference/APIfunctions.html: `mj_inverse` computes inverse dynamics; output in `mjData.qfrc_inverse`.
[4]t	MuJoCo computation docs https://mujoco.readthedocs.io/en/stable/computation/index.html: Projected Gauss–Seidel solver (`solver="PGS"`) and related constraint-solver settings.
[4]u	MuJoCo computation docs https://mujoco.readthedocs.io/en/stable/computation/index.html: additional solvers (e.g., Conjugate Gradient `CG` and `Newton` solver types).
[4]v	MuJoCo overview/computation docs https://mujoco.readthedocs.io/en/stable/overview.html: friction-cone/contact handling and constraint impedance parameters (convex formulations for friction cones).
[4]w	MuJoCo computation docs https://mujoco.readthedocs.io/en/stable/computation/index.html: additional constraint behaviours configured via contact parameters (`solref`, `solimp`, `cone`).
[4]x	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: `integrator="Euler"` documented as semi-implicit Euler (symplectic).
[4]y	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: implicit integrators `integrator="implicit"` and `"implicitfast"`.
[4]z	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: `integrator="RK4"`.
[4]aa	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: `actuator gear` attribute provides gear ratios/gains.

Continued on next page

Table A.2 – *Continued from previous page*

Reference	Exact location
[4]ab	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: built-in sensor types and outputs; see the `sensor` section.
[4]ac	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: rigid links via `body`/`geom` definitions.
[4]ad	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: `joint` definitions.
[4]ae	MuJoCo XML Reference https://mujoco.readthedocs.io/en/stable/XMLreference.html: deformable components via `flexcomp` and related elements.
[4]af	MuJoCo open-source codebase: pose conversions implemented in utility functions (`mju_*`) and MJCF parsing; extensions require source-level changes.
[4]ag	MuJoCo open-source contact/collision implementation (engine collision modules) can be modified; model-level tuning available via MJCF contact parameters in the XML reference https://mujoco.readthedocs.io/en/stable/XMLreference.html.

Table A.3: PyBullet Documentation locations for evaluation references.

Reference	Exact location
[84]a	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: pose I/O uses quaternions; conversion helpers (`getQuaternionFromEuler`, `getEulerFromQuaternion`, `getMatrixFromQuaternion`).
[84]b	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: IK/FK APIs take joint state as input (e.g., `calculateInverseKinematics`, `getLinkState`, `calculateJacobian`).
[84]c	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getLinkState` returns link/world pose and optional velocities from forward kinematics.
[84]d	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `calculateInverseKinematics` exposes parameters (e.g., damping, residual threshold, joint limits) enabling alternative IK behaviours.

Continued on next page

Table A.3 – *Continued from previous page*

Reference	Exact location
[84]e	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `calculateInverseDynamics` inputs (q, qdot, qddot) and other state-dependent dynamics queries.
[84]f	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: inverse dynamics torques via `calculateInverseDynamics`; mass matrix via `calculateMassMatrix`.
[84]g	Bullet source https://github.com/bulletphysics/bullet3: multiple multibody back-ends (rigid bodies vs `btMultiBody`) and solver settings can change dynamics behaviour.
[84]h	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: collision geometry defined in model files (URDF/SDF/MJCF) or via `createCollisionShape` / `createMultiBody`.
[84]i	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: contact outputs via `getContactPoints` and ray results via `rayTest`.
[84]j	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: step size and solver parameters set via `setTimeStep` and `setPhysicsEngineParameter`; stepping via `stepSimulation`.
[84]k	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: stepping updates simulator state queried by `getJointState` / `getBasePositionAndOrientation`.
[84]l	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: actuation via `setJointMotorControl2`; gear constraints via `createConstraint` (gear ratio).
[84]m	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: camera parameters for `getCameraImage`; ray tests via `rayTest`; logging via state/log APIs.
[84]n	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getCameraImage` outputs RGB/depth/segmentation; `rayTest` outputs hit results.
[84]o	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getCameraImage` outputs RGB/depth/segmentation; `rayTest` outputs hit results.

Continued on next page

Table A.3 – *Continued from previous page*

Reference	Exact location
[84]p	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: image output tuple/buffer layout and ray-test result tuples define output structure.
[84]q	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `calculateJacobian` returns angular + linear Jacobians (6D stacked form).
[84]r	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getEulerFromQuaternion` / `getQuaternionFromEuler`.
[84]s	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: axis–angle conversion helpers (e.g., `getAxisAngleFromQuaternion`).
[84]t	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: quaternions used for base/link orientations in API I/O.
[84]u	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getMatrixFromQuaternion` converts quaternions to rotation matrices.
[84]v	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `getLinkState` (with forward-kinematics computation enabled) provides FK results.
[84]w	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `calculateInverseKinematics` provides IK with configurable damping/solver parameters.
[84]x	Bullet source https://github.com/bulletphysics/bullet3: articulated-body dynamics for multibodies implemented in `btMultiBody` (Featherstone-style).
[84]y	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: `calculateInverseDynamics`.
[84]z	Bullet MLCP solvers https://github.com/bulletphysics/bullet3/tree/master/src/BulletDynamics/MLCPSolvers: `btLemkeSolver`.
[84]aa	Bullet source https://github.com/bulletphysics/bullet3: PGS-style iterative solvers (e.g., `btSequentialImpulseConstraintSolver`).
[84]ab	Bullet MLCP solvers https://github.com/bulletphysics/bullet3/tree/master/src/BulletDynamics/MLCPSolvers: `btDantzigSolver`.

Continued on next page

Table A.3 – *Continued from previous page*

Reference	Exact location
[84]ac	Bullet MLCP solvers https://github.com/bulletphysics/bullet3/tree/master/src/BulletDynamics/MLCPSolvers: `btMLCPSolver` wrapper and solver variants.
[84]ad	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: generic constraints/joints via `createConstraint` and Bullet constraint classes.
[84]ae	Bullet source https://github.com/bulletphysics/bullet3: semi-implicit (symplectic) Euler integration used in `stepSimulation`.
[84]af	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: gear constraint type (`JOINT_GEAR`) exposes `gearRatio`.
[84]ag	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: rigid links represented by URDF/SDF/MJCF bodies or Bullet multibody links.
[84]ah	PyBullet Quickstart https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf: joints defined in URDF/SDF/MJCF and accessible via joint APIs.
[84]ai	PyBullet/Bullet is open-source https://github.com/bulletphysics/bullet3; additional pose utilities, model loaders, or bindings can be added in the Python API or C++ core.
[84]aj	PyBullet is open-source https://github.com/bulletphysics/bullet3; new kinematics/IK routines can be added in Python or by extending the C++ bindings.
[84]ak	Bullet dynamics core is open-source https://github.com/bulletphysics/bullet3; alternative dynamics methods can be implemented by modifying `btMultiBody` and solver modules.
[84]al	PyBullet supports custom collision shapes and constraints at model level via `createCollisionShape/createMultiBody` (https://raw.githubusercontent.com/bulletphysics/bullet3/master/docs/pybullet_quickstartguide.pdf).
[84]am	Bullet collision pipeline is open-source https://github.com/bulletphysics/bullet3; extending broadphase/narrowphase requires source changes.
[84]an	PyBullet motor/constraint APIs can be extended; new actuator models require extending Bullet/PyBullet source https://github.com/bulletphysics/bullet3.
[83]a	Bullet GPU acceleration: Bullet3 provides GPU rigid-body pipelines and demos under `examples/OpenCL` and related modules in the Bullet3 source tree (https://github.com/bulletphysics/bullet3).

Table A.4: Drake Documentation locations for evaluation references.

Reference	Exact location
[85]a	Drake https://drake.mit.edu/doxygen_cxx/: pose/rotation types in `drake::math` (e.g., `RigidTransform`, `RotationMatrix`, `RollPitchYaw`, `Quaternion`, `AngleAxis`) and model parsing via `drake::multibody::Parser`.
[85]b	Drake https://drake.mit.edu/doxygen_cxx/: forward kinematics in `MultibodyPlant` uses configuration `q` from the system `Context` as input.
[85]c	Drake https://drake.mit.edu/doxygen_cxx/: FK output via frame poses (e.g., `MultibodyPlant::CalcFramePoseInWorld`) and Jacobians.
[85]d	Drake https://drake.mit.edu/doxygen_cxx/: `drake::multibody::InverseKinematics` supports different constraints/costs, enabling alternative IK problem formulations.
[85]e	Drake https://drake.mit.edu/doxygen_cxx/: dynamics calculations take state from the `Context` (`q`, `v`) and inputs (actuation, external forces) in the multibody plant implementation.
[85]f	Drake https://drake.mit.edu/doxygen_cxx/: dynamics outputs via `MultibodyPlant` APIs such as `CalcMassMatrix`, `CalcBiasTerm`, `CalcTimeDerivatives`, and inverse dynamics calls.
[85]g	Drake https://drake.mit.edu/doxygen_cxx/: collision geometry and proximity properties specified via URDF/SDF or through `drake::geometry::SceneGraph`.
[85]h	Drake https://drake.mit.edu/doxygen_cxx/: integrators read the system state from the `Context`; integration is implemented in `drake::systems::analysis`.
[85]i	Drake https://drake.mit.edu/doxygen_cxx/: integration advances the `Context` (continuous state); outputs retrieved from system output ports.
[85]j	Drake https://drake.mit.edu/doxygen_cxx/: multiple integrators available under `drake::systems::analysis` (explicit/implicit Euler, Runge–Kutta family, Radau, etc.).
[85]k	Drake https://drake.mit.edu/doxygen_cxx/: actuation defined via `drake::multibody::JointActuator` (URDF/SDF parsing or `MultibodyPlant::AddJointActuator`), including actuator gains/limits.
[85]l	Drake https://drake.mit.edu/doxygen_cxx/: actuator behaviour implemented in `drake::multibody` (`JointActuator`) and controller systems in `drake::systems`.
[85]m	Drake https://drake.mit.edu/doxygen_cxx/: sensors configured as `Systems` (e.g., `RgbdSensor`) with parameters at construction time.

Continued on next page

Table A.4 – *Continued from previous page*

Reference	Exact location
[85]n	Drake https://drake.mit.edu/doxygen_cxx/: sensor systems read the `Context` and input ports (e.g., pose, scene data) to generate outputs.
[85]o	Drake https://drake.mit.edu/doxygen_cxx/: sensor outputs provided as output ports (images, point clouds, contact results).
[85]p	Drake https://drake.mit.edu/doxygen_cxx/: sensor outputs provided as output ports (images, point clouds, contact results).
[85]q	Drake https://drake.mit.edu/doxygen_cxx/: output port type defines sensor output structure (e.g., image buffer/message types).
[85]r	Drake https://drake.mit.edu/doxygen_cxx/: alternative sensors and rendering back-ends available via different sensor system implementations (RGB-D, lidar, contact-force reporting).
[85]s	Drake https://drake.mit.edu/doxygen_cxx/: spatial algebra types (`SpatialVelocity`, `SpatialAcceleration`, `SpatialForce`) in `drake::multibody`.
[85]t	Drake https://drake.mit.edu/doxygen_cxx/: Euler-angle representation via `drake::math::RollPitchYaw`.
[85]u	Drake https://drake.mit.edu/doxygen_cxx/: axis–angle via `drake::math::AngleAxis` and conversion utilities.
[85]v	Drake https://drake.mit.edu/doxygen_cxx/: quaternion via `drake::math::Quaternion` / Eigen quaternions.
[85]w	Drake https://drake.mit.edu/doxygen_cxx/: rotation matrices via `drake::math::RotationMatrix`.
[85]x	Drake https://drake.mit.edu/doxygen_cxx/: spatial velocity/acceleration outputs for bodies/frames available via `MultibodyPlant` methods (spatial vector types).
[85]y	Drake https://drake.mit.edu/doxygen_cxx/: general optimisation-based IK via `drake::multibody::InverseKinematics` built on `drake::solvers::MathematicalProgram`.
[85]z	Drake https://drake.mit.edu/doxygen_cxx/: articulated-body dynamics implemented in `drake::multibody` (tree/plant dynamics core).
[85]aa	Drake https://drake.mit.edu/doxygen_cxx/: Coriolis/centrifugal effects included in bias terms; see `MultibodyPlant::CalcBiasTerm` and related APIs.
[85]ab	Drake https://drake.mit.edu/doxygen_cxx/: inverse dynamics via `MultibodyPlant::CalcInverseDynamics`.
[85]ac	Drake https://drake.mit.edu/doxygen_cxx/: contact solvers (e.g., SAP) solve complementarity-style contact constraints; see `drake::multibody::contact_solvers`.
[85]ad	Drake https://drake.mit.edu/doxygen_cxx/: convex optimisation via `drake::solvers` (mathematical programming) for constraint-based formulations.

Continued on next page

Table A.4 – *Continued from previous page*

Reference	Exact location
[85]ae	Drake https://drake.mit.edu/doxygen_cxx/: additional constraints via contact/joint constraints and mathematical programming tools.
[85]af	Drake https://drake.mit.edu/doxygen_cxx/: `ExplicitEulerIntegrator` (`drake::systems::analysis`).
[85]ag	Drake https://drake.mit.edu/doxygen_cxx/: semi-implicit/symplectic Euler via `SemiExplicitEulerIntegrator` (and related variants) in `drake::systems::analysis`.
[85]ah	Drake https://drake.mit.edu/doxygen_cxx/: `ImplicitEulerIntegrator` (`drake::systems::analysis`).
[85]ai	Drake https://drake.mit.edu/doxygen_cxx/: Bogacki–Shampine method via `BogackiShampine3Integrator` in `drake::systems::analysis`.
[85]aj	Drake https://drake.mit.edu/doxygen_cxx/: implicit Runge–Kutta Radau method via `RadauIntegrator`.
[85]ak	Drake https://drake.mit.edu/doxygen_cxx/: `RungeKutta2Integrator`.
[85]al	Drake https://drake.mit.edu/doxygen_cxx/: `RungeKutta3Integrator`.
[85]am	Drake https://drake.mit.edu/doxygen_cxx/: Runge–Kutta integrator suite under `drake::systems::analysis` (see `RungeKutta*Integrator` classes).
[85]an	Drake https://drake.mit.edu/doxygen_cxx/: `RungeKutta5Integrator`.
[85]ao	Drake https://drake.mit.edu/doxygen_cxx/: gear coupling via `drake::multibody::Gearbox` / coupler constraints (gear ratio parameterisation) in the multibody tree.
[85]ap	Drake https://drake.mit.edu/doxygen_cxx/: sensor systems and output ports (RGB-D, lidar, contact reporting).
[85]aq	Drake https://drake.mit.edu/doxygen_cxx/: rigid links as `RigidBody` elements in `MultibodyPlant`.
[85]ar	Drake https://drake.mit.edu/doxygen_cxx/: joints via `drake::multibody::Joint` subclasses.
[85]as	Drake https://drake.mit.edu/doxygen_cxx/: deformable/soft-body support via deformable body models and hydroelastic contact in `drake::multibody` / `drake::geometry`.
[85]at	Drake open-source: extend kinematics by adding new methods in `drake/multibody` or by composing new `Systems` (https://drake.mit.edu/doxygen_cxx/).
[85]au	Drake open-source multibody dynamics core (`drake/multibody`) can be extended to add new dynamics algorithms (https://drake.mit.edu/doxygen_cxx/).

Continued on next page

Table A.4 – *Continued from previous page*

Reference	Exact location
[85]av	Drake uses `SceneGraph` for geometry; extend at model level via new geometries/proximity properties and contact-material parameters (https://drake.mit.edu/doxygen_cxx/).
[85]aw	Drake collision/contact code is open-source (`drake/geometry` and `drake/multibody`) and can be extended at source level (https://drake.mit.edu/doxygen_cxx/).
[85]ax	Drake integrators live under `drake/systems/analysis`; add new integrators by extending `IntegratorBase` (https://drake.mit.edu/doxygen_cxx/).
[85]ay	Drake contact solvers are modular; add new solvers under `drake/multibody/contact_solvers` (https://drake.mit.edu/doxygen_cxx/).
[85]az	Extend actuation/transmission models by adding new actuator types, coupler/gear constraints, or controller systems in `drake/systems` / `drake/multibody` (https://drake.mit.edu/doxygen_cxx/).
[85]ba	Drake sensor models are `Systems`; add new sensors by implementing new `LeafSystem` classes (https://drake.mit.edu/doxygen_cxx/).

Table A.5: RBDL Documentation locations for evaluation references.

Reference	Exact location
[87]a	RBDL docs https://rbdl.github.io/: model construction uses joint frames/transforms and rotation matrices (Eigen-based math types).
[87]b	RBDL API https://rbdl.github.io/: kinematics functions take `q` as input (e.g., `CalcBodyToBaseCoordinates`, `CalcBodyWorldOrientation`).
[87]c	RBDL API https://rbdl.github.io/: kinematics outputs returned as vectors/matrices or written to output arguments (positions/orientations/Jacobians).
[87]d	RBDL API https://rbdl.github.io/: dynamics functions take state `(q, qdot, qddot)` and external forces as inputs (e.g., `InverseDynamics`, `ForwardDynamics`).
[87]e	RBDL API https://rbdl.github.io/: dynamics outputs include joint torques (inverse dynamics) and accelerations (forward dynamics).
[87]f	RBDL math module https://rbdl.github.io/: `SpatialVector` and `SpatialMatrix` types.
[87]g	RBDL uses Eigen rotations; Euler-angle representations available via Eigen utilities used throughout RBDL (see math module in https://rbdl.github.io/).
[87]h	RBDL uses Eigen quaternions for rotations where needed; see Eigen-based math utilities in https://rbdl.github.io/.

Continued on next page

Table A.5 – *Continued from previous page*

Reference	Exact location
[87]i	RBDL uses Eigen `Matrix3d` rotation matrices for model transforms (math module in https://rbdl.github.io/).
[87]j	RBDL API https://rbdl.github.io/: forward kinematics and spatial quantities via functions such as `CalcPointVelocity` and body pose/orientation routines.
[87]k	RBDL add-ons/examples https://rbdl.github.io/: IK implementations include damped least squares (DLS) Jacobian pseudo-inverse.
[87]l	RBDL paper [87]: implements Featherstone-style articulated-body algorithm (ABA) for forward dynamics.
[87]m	RBDL API https://rbdl.github.io/: Coriolis/centrifugal contributions available via bias term computations (e.g., `NonlinearEffects`).
[87]n	RBDL API https://rbdl.github.io/: `InverseDynamics` routine.
[87]o	RBDL https://rbdl.github.io/: rigid links as bodies with inertial parameters.
[87]p	RBDL https://rbdl.github.io/: joints defined by `Joint` objects (revolute/prismatic/floating).
[87]q	RBDL is open-source; extend kinematics by adding new routines to the core library or add-ons (source in the RBDL repository; docs at https://rbdl.github.io/).
[87]r	RBDL is open-source; extend dynamics algorithms by modifying/adding implementations in the core library (repository source; docs at https://rbdl.github.io/).

Table A.6: PhysX Articulations Documentation locations for evaluation references.

Reference	Exact location
[86]a	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: kinematic targets and joint coordinates set via actor/articulation APIs (e.g., `PxRigidDynamic::setKinematicTarget`, articulation joint position setters).
[86]b	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: world poses read via `PxRigidActor::getGlobalPose` (and articulation link pose queries).
[86]c	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: simulation reads state from actors/links and applied forces/torques each step (e.g., `addForce`, `addTorque`) and evolves it in `PxScene::simulate`.
[86]d	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: dynamics outputs available via actor velocities/forces and articulation cache fields (joint forces/accelerations).

Continued on next page

Table A.6 – *Continued from previous page*

Reference	Exact location
[86]e	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: contact output via `PxSimulationEventCallback::onContact` and contact stream APIs.
[86]f	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: time step specified by the caller when invoking `PxScene::simulate(dt);` solver iterations and flags configured on the scene.
[86]g	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: results retrieved via `PxScene::fetchResults` and actor/link pose queries.
[86]h	PhysX articulations https://nvidia-omniverse.github.io/PhysX/physx/: actuation via joint drives (e.g., `PxArticulationJointReducedCoordinate` drive parameters) and constraints.
[86]i	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: scene queries (raycast/sweep/overlap) and simulation event callbacks provide sensor-like primitives and outputs.
[86]j	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: scene queries (raycast/sweep/overlap) and simulation event callbacks provide sensor-like primitives and outputs.
[86]k	PhysX articulations https://nvidia-omniverse.github.io/PhysX/physx/: spatial types (e.g., `PxSpatialForce`, `PxSpatialVelocity`) used in reduced-coordinate articulation APIs.
[86]l	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: axis–angle represented via `PxQuat(angle, axis)` constructors.
[86]m	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: `PxQuat` and `PxTransform`.
[86]n	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: matrix types `PxMat33`/`PxMat44` (rotation matrices and transforms).
[86]o	PhysX articulations https://nvidia-omniverse.github.io/PhysX/physx/: reduced-coordinate articulation cache exposes bias terms for articulated dynamics, which include Coriolis/centrifugal effects.
[86]p	PhysX articulations https://nvidia-omniverse.github.io/PhysX/physx/: articulation cache and joint-force computation provide inverse-dynamics-related quantities for reduced-coordinate articulations.
[86]q	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: solver supports Gauss–Seidel variants (PGS/TGS) for contact and joint constraints.

Continued on next page

Table A.6 – *Continued from previous page*

Reference	Exact location
[86]r	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: additional solver variants/settings (e.g., Temporal Gauss–Seidel, TGS).
[86]s	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: general constraints/joints (e.g., D6 joints, articulation joints, limits, drives) beyond basic contact complementarity.
[86]t	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: rigid-body integration uses a semi-implicit (symplectic) Euler-style update; configured through timestep and solver parameters.
[86]u	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: scene query subsystem provides raycast/overlap results used as sensor primitives.
[86]v	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: rigid links via `PxRigidDynamic` and articulation links (`PxArticulationLink`).
[86]w	PhysX https://nvidia-omniverse.github.io/PhysX/physx/: joints via `PxJoint` types and `PxArticulationJointReducedCoordinate` for articulations.
[116]a	PhysX GPU rigid bodies: PhysX SDK documentation https://nvidia-omniverse.github.io/PhysX/physx/ section on GPU Rigid Bodies (CUDA pipeline and scene configuration).

Table A.7: Pinocchio Documentation locations for evaluation references.

Reference	Exact location
[88]a	Pinocchio https://stack-of-tasks.github.io/pinocchio/: rigid-body poses represented by `pinocchio::SE3` (rotation matrix + translation); model built from URDF parsing.
[88]b	Pinocchio https://stack-of-tasks.github.io/pinocchio/: kinematics functions take `q` (and `v`) as inputs, e.g., `forwardKinematics(model,data,q)`.
[88]c	Pinocchio https://stack-of-tasks.github.io/pinocchio/: kinematics results written into `Data` (joint/frame placements, Jacobians) after calling `forwardKinematics` and `updateFramePlacements`.
[88]d	Pinocchio https://stack-of-tasks.github.io/pinocchio/: dynamics routines take `(q,v,a)` and model parameters as inputs (e.g., `rnea`, `aba`).
[88]e	Pinocchio https://stack-of-tasks.github.io/pinocchio/: dynamics outputs written to `Data` (accelerations, bias terms, joint forces) or returned by `rnea/aba`.
[88]f	Pinocchio https://stack-of-tasks.github.io/pinocchio/: spatial types `pinocchio::Motion` and `pinocchio::Force` (6D motion/force vectors).

Continued on next page

Table A.7 – *Continued from previous page*

Reference	Exact location
[88]g	Pinocchio https://stack-of-tasks.github.io/pinocchio/: Euler-angle conversions available via Eigen rotation utilities used with `SE3` rotations.
[88]h	Pinocchio https://stack-of-tasks.github.io/pinocchio/: axis–angle supported via Eigen `AngleAxis` interop when constructing rotations for `SE3`.
[88]i	Pinocchio https://stack-of-tasks.github.io/pinocchio/: quaternion support via Eigen quaternions and conversion utilities when working with rotations.
[88]j	Pinocchio https://stack-of-tasks.github.io/pinocchio/: `SE3` stores rotation matrices (access via `SE3.rotation()`).
[88]k	Pinocchio https://stack-of-tasks.github.io/pinocchio/: forward kinematics and spatial quantities via `forwardKinematics` / `computeJointJacobians` / `getFrameVelocity`.
[88]l	Pinocchio https://stack-of-tasks.github.io/pinocchio/: forward dynamics via ABA (`pinocchio::aba`), a Featherstone-style articulated-body algorithm.
[88]m	Pinocchio https://stack-of-tasks.github.io/pinocchio/: Coriolis/centrifugal terms via bias computations (e.g., `computeCoriolisMatrix` / `nonLinearEffects` depending on API).
[88]n	Pinocchio https://stack-of-tasks.github.io/pinocchio/: inverse dynamics via RNEA (`pinocchio::rnea`).
[88]o	Pinocchio https://stack-of-tasks.github.io/pinocchio/: rigid links represented by bodies with inertias in `Model`.
[88]p	Pinocchio https://stack-of-tasks.github.io/pinocchio/: joints represented by `JointModel` types in `Model`.
[88]q	Pinocchio is open-source; extend kinematics algorithms by adding new routines/classes in the core library (see repository and docs at https://stack-of-tasks.github.io/pinocchio/).
[88]r	Pinocchio is open-source; extend dynamics algorithms by adding new routines in the core library (see docs at https://stack-of-tasks.github.io/pinocchio/).

Table A.8: DQRobotics Documentation locations for evaluation references.

Reference	Exact location
[89]a	DQRobotics docs https://dqrobotics.com/: kinematics/control routines take joint variables as input and operate in dual-quaternion space (see API/reference sections).
[89]b	DQRobotics docs https://dqrobotics.com/: kinematics outputs returned as dual quaternions (DQ objects) representing pose, plus Jacobians in matrix form.

Continued on next page

Table A.8 – *Continued from previous page*

Reference	Exact location
[89]c	DQRobotics docs https://dqrobotics.com/: quaternion primitives are supported as part of the dual-quaternion (DQ) algebra.
[89]d	DQRobotics docs https://dqrobotics.com/: primary pose representation is the dual quaternion (rigid motion as a dual quaternion).
[89]e	DQRobotics docs https://dqrobotics.com/: IK/control examples use Jacobian pseudo-inverse with damping (DLS) in dual-quaternion formulation.
[89]f	DQRobotics docs https://dqrobotics.com/: robot kinematics modelled as rigid-link chains whose pose is computed as a dual quaternion.
[89]g	DQRobotics docs https://dqrobotics.com/: joint variables parameterise the kinematic chain (revolute/prismatic joints in serial robot models).
[89]h	DQRobotics is open-source; extend kinematics algorithms by adding new formulations and mappings in the library source (see project documentation at https://dqrobotics.com/).

Table A.9: GAFRO Documentation locations for evaluation references.

Reference	Exact location
[90]a	GAFRO repository https://github.com/idiap/gafro: kinematics modules accept joint variables and robot model parameters as inputs (see source under kinematics-related modules).
[90]b	GAFRO repository https://github.com/idiap/gafro: kinematics outputs returned as CGA entities (motors/rotors) representing pose.
[90]c	GAFRO repository https://github.com/idiap/gafro: dynamics-related computations take state/model parameters as input and represent quantities using CGA types.
[90]d	GAFRO repository https://github.com/idiap/gafro: dynamics outputs expressed as CGA entities/vectors; see dynamics-related modules and examples.
[90]e	GAFRO repository https://github.com/idiap/gafro: pose represented using conformal geometric algebra (CGA) entities (motors/rotors); see core CGA types and documentation.
[90]f	GAFRO https://github.com/idiap/gafro: robot models include rigid-link representations whose pose is computed via CGA operations.
[90]g	GAFRO https://github.com/idiap/gafro: joints parameterise the kinematic chain; see robot model and kinematics modules.
[90]h	GAFRO is open-source https://github.com/idiap/gafro; extend kinematics by adding new CGA-based formulations to the kinematics modules.

Continued on next page

Table A.9 – *Continued from previous page*

Reference	Exact location
[90]i	GAFRO is open-source https://github.com/idiap/gafro; extend dynamics modules by adding new CGA-based dynamics formulations and operators.

A.5 Robot Modelling Languages and File Formats: Documentation References

This appendix collects the documentation sources used to support the claims and table entries in Section 2.3.3.

[41] *Gazebo tutorial: Make a model*, Open Source Robotics Foundation (Gazebo Classic). Accessed: 29th Dec. 2025. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=build_model

[43] *URDF (Unified Robot Description Format) — ROS Wiki*, Open Robotics / ROS, 2023. Accessed: 29th Dec. 2025. [Online]. Available: https://wiki.ros.org/urdf

[44] *simmechanics_to_urdf — ROS Wiki*, Open Robotics / ROS, 2011. Accessed: 29th Dec. 2025. [Online]. Available: https://wiki.ros.org/simmechanics_to_urdf

[45] *Gazebo tutorial: Model a 4-bar linkage in SDFormat and URDF*, Open Source Robotics Foundation (Gazebo Classic). Accessed: 29th Dec. 2025. [Online]. Available: https://classic.gazebosim.org/tutorials?cat=&tut=kinematic_loop

[46] *urdf/XML/Gazebo — ROS Wiki*, Open Robotics / ROS, 2023. Accessed: 29th Dec. 2025. [Online]. Available: https://wiki.ros.org/urdf/XML/Gazebo

[47] *Gazebo tutorial: Using a URDF in Gazebo*, Open Source Robotics Foundation (Gazebo Classic). Accessed: 29th Dec. 2025. [Online]. Available: https://classic.gazebosim.org/tutorials/?tut=ros_urdf

[48] *SDFormat: <world> element (API documentation)*, Open Source Robotics Foundation (SDFormat). Accessed: 29th Dec. 2025. [Online]. Available: https://osrf-distributions.s3.amazonaws.com/sdformat/api/dev.html

[49] *SDFormat — Gazebo documentation*, Open Robotics (Gazebo). Accessed: 29th Dec. 2025. [Online]. Available: https://gazebosim.org/libs/sdformat/

[50] *Pose frame semantics — SDFormat specification tutorial*, Open Source Robotics Foundation (SDFormat). Accessed: 29th Dec. 2025. [Online]. Available: https://sdformat.org/tutorials/specification/pose_frame_semantics/1.5/

[51] *SDFormat specification (plugin description)*, Open Source Robotics Foundation (SDFormat), 2020. Accessed: 29th Dec. 2025. [Online]. Available: https://sdformat.org/spec?elem=model&ver=1.7

[52] *Drake: Multibody parsing (SDFormat deformable bodies via custom tags)*, Toyota Research Institute / RobotLocomotion (Drake). Accessed: 29th Dec. 2025. [Online]. Available: https://drake.mit.edu/doxygen_cxx/group__multibody__parsing.html

[53] *MuJoCo documentation: XML reference (MJCF)*, Google DeepMind. Accessed: 29th Dec. 2025. [Online]. Available: https://mujoco.readthedocs.io/en/stable/XMLreference.html

[54] *MuJoCo documentation (overview)*, Google DeepMind. Accessed: 29th Dec. 2025. [Online]. Available: https://mujoco.readthedocs.io/

[55] *MuJoCo documentation: Computation (constraints and contact parameters)*, Google DeepMind. Accessed: 29th Dec. 2025. [Online]. Available: https://mujoco.readthedocs.io/en/stable/computation/index.html

[56] *MuJoCo documentation: Modeling (MJCF mechanisms, defaults/classes)*, Google DeepMind. Accessed: 29th Dec. 2025. [Online]. Available: https://mujoco.readthedocs.io/en/stable/modeling.html

[57] *MuJoCo documentation: Extensions (engine plugins and resource providers)*, Google DeepMind. Accessed: 29th Dec. 2025. [Online]. Available: https://mujoco.readthedocs.io/en/stable/programming/extension.html

[58] *Introduction to OpenUSD*, The Alliance for OpenUSD / Pixar et al. Accessed: 29th Dec. 2025. [Online]. Available: https://openusd.org/release/intro.html

[59] *USD glossary: Composition arcs and layers*, OpenUSD. Accessed: 29th Dec. 2025. [Online]. Available: https://openusd.org/dev/glossary.html

[60] *UsdPhysics: USD physics schema (front page / design concerns)*, OpenUSD. Accessed: 29th Dec. 2025. [Online]. Available: https://openusd.org/dev/api/usd_physics_page_front.html

[61] *UsdPhysics module API documentation*, NVIDIA Omniverse (pxr-usd-api mirror), 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://docs.omniverse.nvidia.com/kit/docs/pxr-usd-api/latest/pxr/UsdPhysics.html

[62] *Omni Physics deformable schema (extension of USD physics schema)*, NVIDIA Omniverse. Accessed: 29th Dec. 2025. [Online]. Available: https://docs.omniverse.nvidia.com/kit/docs/omni_physics/107.3/dev_guide/deformables_beta/omniphysics_deformable_schema.html

[63] *Creating new schema classes with usdGenSchema*, OpenUSD. Accessed: 29th Dec. 2025. [Online]. Available: https://openusd.org/dev/api/_usd__page__generating_schemas.html

[65] *Multibody dynamics — Simscape Multibody*, MathWorks. Accessed: 29th Dec. 2025. [Online]. Available: https://www.mathworks.com/help/sm/multibody-dynamics.html

[66] *Model a closed-loop kinematic chain (four-bar example)*, MathWorks. Accessed: 29th Dec. 2025. [Online]. Available: https://www.mathworks.com/help/sm/ug/model-four-bar.html

[67] *Spatial Contact Force block (Simscape Multibody)*, MathWorks. Accessed: 29th Dec. 2025. [Online]. Available: https://www.mathworks.com/help/sm/ref/spatialcontactforce.html

[68] *Reduced Order Flexible Solid block (Simscape Multibody)*, MathWorks. Accessed: 29th Dec. 2025. [Online]. Available: https://www.mathworks.com/help/sm/ref/reducedorderflexiblesolid.html

[69] *Creating custom components (Simscape language)*, MathWorks. Accessed: 29th Dec. 2025. [Online]. Available: https://www.mathworks.com/help/simscape/creating-custom-components.html

[70] *STL (STereoLithography) file format family*, Library of Congress, 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000504.shtml

[71] *Wavefront OBJ file format*, Library of Congress, 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml

[72] *Wavefront Material Template Library (MTL) file format*, Library of Congress, 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000508.shtml

[73] *COLLADA (interchange file format for interactive 3D applications)*, Wikipedia, 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://en.wikipedia.org/wiki/COLLADA

[74] *glTF 2.0 specification*, Khronos Group, 2021. Accessed: 29th Dec. 2025. [Online]. Available: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html

[75] *Khronos glTF (GL Transmission Format) overview (PBR materials)*, Khronos Group. Accessed: 29th Dec. 2025. [Online]. Available: https://www.khronos.org/gltf

[76] *Universal Scene Description (USD)*, Library of Congress, 2025. Accessed: 29th Dec. 2025. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000561.shtml

[77] *OpenUSD file formats (USDA / USDC)*, NVIDIA Learn OpenUSD. Accessed: 29th Dec. 2025. [Online]. Available: https://docs.nvidia.com/learn-openusd/latest/stage-setting/usd-file-formats.html

[78] *Xacro documentation*, Open Robotics / ROS. Accessed: 29th Dec. 2025. [Online]. Available: https://docs.ros.org/en/jazzy/p/xacro/

[79] *SDFormat extensions to URDF (the <gazebo> tag)*, Open Source Robotics Foundation (SDFormat). Accessed: 29th Dec. 2025. [Online]. Available: https://sdformat.org/tutorials/specification/sdformat_urdf_extensions/1.6/

[80] *Collisions and visuals — SDFormat specification tutorial*, Open Source Robotics Foundation (SDFormat). Accessed: 19th Oct. 2026. [Online]. Available: https://sdformat.org/tutorials/?tut=spec_shapes

[81] *SDFormat: Meta-tags (<include>) (API documentation)*, Open Source Robotics Foundation (SDFormat). Accessed: 29th Dec. 2025. [Online]. Available: https://osrf-distributions.s3.amazonaws.com/sdformat/api/dev.html

[82] *USD physics schema (Omni Physics): Joints between bodies and world*, NVIDIA Omniverse. Accessed: 29th Dec. 2025. [Online]. Available: https://docs.omniverse.nvidia.com/kit/docs/omni_physics/latest/dev_guide/schemas/usdphysics.html

A.6 Coverage Tests

This appendix provides the test coverage traceability tables for the simulation framework. These tables map each algorithm from Chapter 3 to its corresponding test cases, enabling verification that the implementation correctly realises the theoretical framework. Coverage criteria, justification, and analysis are presented in Section 4.2.1.

A.6.1 eqnComp Coverage

Table A.11 provides a complete mapping from eqnComp algorithms to their corresponding test cases.

Integration test key. Integration test coverage uses the format `Category::Formulation::(Components)`, where: *All* indicates coverage by all integration tests in Table A.12; *All::(X,Y)* indicates all tests exercise components X and Y; *Category::Formulation::(X)* indicates specific tests with component X. Component abbreviations: L = local links, J = joints, S = sensors, A = actuators, B = bodies.

Table A.10: Test coverage for eqnComp algorithm steps.

Step	Algorithm	Tests	Coverage
Main	`eqnComp`	2	C1: testEqnCompMainSKO, testEqnCompMainNullModel C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest, MetamodelCoverageSKO-GeneratorTest
S1	`calculateFormulations`	14	C1: testMixedFormulations C2: SimpleArmSKOTest C3: testSKOLocalJoint, testFeatherstoneLocalJoint C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest
S1a	`findDynFormulation`	5	C1: testSKOLocalJoint C2: SimpleArmSKOTest C3: testSKOLocalJoint, testFeatherstoneLocalJoint C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest
S1b	`findSensorFormulations`	3	C1: testTrivialSensor C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest

Continued on next page

Table A.10 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S1c	`findActuatorFormulations`	3	C1: testTrivialActuator C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest
S2	`localiseRefs`	13	C1: testConvertRefJoints C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2a	`convertRefJoints`	3	C1: testConvertRefJoints C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2a	`convertRefLinks`	1	C1: testConvertRefLinks C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2a	`convertRefSensors`	2	C1: testConvertRefSensors C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest

Continued on next page

Table A.10 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S2a	`convertRefActuators`	2	C1: testConvertRefActuators C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2a	`convertRefBodies`	1	C1: testConvertRefBodies C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2b	`createLocalJoint`	3	C1: testCreateLocalJoint C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2b	`createLocalLink`	1	C1: testConvertRefLinks C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest
S2b	`createLocalSensor`	1	C1: testConvertRefSensors C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO-GeneratorTest

Continued on next page

Table A.10 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S2b	`createLocalActuator`	1	C1: testConvertRefActuators C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO- GeneratorTest
S2b	`createLocalBody`	1	C1: testConvertRefBodies C2: SimpleArmLibRefsTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverageSKO- GeneratorTest
S3	`validate`	7	C1: testValidPModel C2: SimpleArmSKOTest C3: testValidFeatherstoneModel C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage- GeneratorTest
S4	`calculateTopology`	4	C1: testSerialChainTwoLink C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: testSerialChainTwoLink, testTreeTopology, testClosedChain C5: MetamodelCoverage- GeneratorTest
S5	`assignNumbering`	17	C1: testTwoLinkSKO C2: SimpleArmSKOTest C3: testTwoLinkSKO, testTwoLinkFeatherstone C4: testTreeTopologySKO, testClosedChainSKO C5: MetamodelCoverage- GeneratorTest

Continued on next page

Table A.10 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S5a	`orderPmodel`	3	C1: testOrderedLinksBasic C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: testTreeTopologySKO, testClosedChainSKO C5: MetamodelCoverage-GeneratorTest
S5b	`getBaseLink`	4	C1: testGetBaseLinkSerialChain C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: testGetBaseLinkTreeTopology, testGetBaseLinkClosedChain C5: MetamodelCoverage-GeneratorTest
S5c	`getChildLinks`	4	C1: testGetChildLinksSerialChain C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: testGetChildLinksTreeTopology, testGetChildLinksClosedChain C5: MetamodelCoverage-GeneratorTest
S6	`calculateEquations`	24	C1: testCreateInertiaVariable-WithMixedTypes C2: SimpleArmSKOTest C3: AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest
S6	`calculateEquations` (SKO)	4	C1: testPhiEqnsN1 C2: SimpleArmSKOTest C3: SimpleArmSKOTest, AcrobotFeatherstoneTest C4: SimpleArmSKOTest (serial), MobileRobotSKOTest (tree), ClosedChainSKOTest (closed) C5: MetamodelCoverage-GeneratorTest
	Total	**126**	

Total counts distinct tests; some tests exercise multiple steps, so row totals are not additive.

Table A.11: eqnComp algorithm to test case traceability matrix.

Step	Algorithm	Coverage	Test Method(s)
Main Pipeline (Chapter 3)			
Main	`eqnComp`	Unit (2)	`testEqnCompMainSKO`, `testEqnCompMainNullModel`
		Integration (9)	All
S1	`calculateFormulations`	Unit (14)	`testMixedFormulations`, `testEmptyMaps` plus nested tests
		Integration (9)	All
S1a	`findDynFormulation`	Unit (5)	`testSKOLocalJoint`, `testSKOReferenceJoint`, `testFeatherstoneLocalJoint`, `testFeatherstoneReferenceJoint`, `testNoJoints`
		Integration (9)	All::(J)
S1b	`findSensorFormulations`	Unit (3)	`testTrivialSensor`, `testTrivialSensorLibraryImport`, `testUnknownSensor`
		Integration (3)	RobotExamples::SKO::(S) (`SimpleArmSKOTest`, `SimpleArmLibRefsTest`, `MobileRobotSKOTest`)
S1c	`findActuatorFormulations`	Unit (3)	`testTrivialActuator`, `testTrivialActuatorLibraryImport`, `testUnknownActuator`
		Integration (5)	RobotExamples::SKO::(A) (`SimpleArmSKOTest`, `SimpleArmLibRefsTest`, `AcrobotSKOTest`, `AcrobotControlledTest`, `MobileRobotSKOTest`)
S2	`localiseRefs`	Unit (13)	`testConvertRefJoints`, `testConvertRefLinks`, `testConvertRefSensors*` (2), `testConvertRefActuators*` (2), `testConvertRefBodies`, `testCreateLocalJoint*` (3)
		Integration (9)	All
S2a	`convertRefJoints`	Unit (3)	`testConvertRefJoints`, `testConvertSingleRefJoint`, `testConvertMixedJoints`
S2a	`convertRefLinks`	Unit	`testConvertRefLinks`
S2a	`convertRefSensors`	Unit (2)	`testConvertRefSensors`, `testConvertRefSensorsJointScope`
S2a	`convertRefActuators`	Unit (2)	`testConvertRefActuators`, `testConvertRefActuatorsLinkScope`
S2a	`convertRefBodies`	Unit	`testConvertRefBodies`

Continued on next page

Table A.11 – *Continued from previous page*

Step	Algorithm	Coverage	Test Method(s)
S2b	`createLocalJoint`	Unit (3)	`testCreateLocalJoint`, `testCreateLocalJointPreservesNestedActuators`, `testCreateLocalJointPreservesNestedSensors`
S2b	`createLocalLink`	Indirect	via `testConvertRefLinks`
S2b	`createLocalSensor`	Indirect	via `testConvertRefSensors`
S2b	`createLocalActuator`	Indirect	via `testConvertRefActuators`
S2b	`createLocalBody`	Indirect	via `testConvertRefBodies`
S3	`validate`	Unit (7)	`testValidPModel`, `testValidFeatherstoneModel`, `testMissingPose`, `testMissingInertia`, `testEmptyPModel`, `testMixedValidity`, `testNonSKO`
		Integration (9)	All::(L,B)
S4	`calculateTopology`	Unit (4)	`testSerialChainTwoLink`, `testSerialChainThreeLink`, `testTreeTopology`, `testClosedChain`
		Integration (9)	All::(L,J)
S5	`assignNumbering`	Unit (17)	`testTwoLinkSKO`, `testTwoLinkFeatherstone`, `testThreeLinkChainSKO`, `testThreeLinkChainFeatherstone`, `testTreeTopologySKO`, `testClosedChainSKO` plus nested tests
		Integration (9)	All::(L,J)
S5a	`orderPmodel`	Unit (3)	`testOrderedLinksBasic`, `testOrderedLinksFourLinks`, `testOrderedLinksSingleLink`
S5b	`getBaseLink`	Unit (4)	`testGetBaseLinkSerialChain`, `testGetBaseLinkThreeLinkChain`, `testGetBaseLinkTreeTopology`, `testGetBaseLinkClosedChain`
S5c	`getChildLinks`	Unit (4)	`testGetChildLinksSerialChain`, `testGetChildLinksLeafLink`, `testGetChildLinksTreeTopology`, `testGetChildLinksClosedChain`

Continued on next page

Table A.11 – *Continued from previous page*

Step	Algorithm	Coverage	Test Method(s)
S6	calculateEquations	Unit (24)	testCreateInertiaVariable* (2), testClassCastExceptionFix, testCreateGeometryVariable* (2), testFindSensor* (5), testFindActuator* (5), testTrivialSensorHasEquation, testMultipleSensors, testNoSensors, testTrivialActuatorHasEquation, testMultipleActuators, testNoActuators, testFullPipelineWithSensorsActuatorsDynamics, testDynamicsOnlyNoSensorsActuators, testSensorsActuatorsOnlyNoDynamics
		Integration (3)	RobotExamples::SKO::(S,A) (SimpleArmSKOTest, SimpleArmLibRefsTest, MobileRobotSKOTest)
S6	calculateEquations (SKO)	Unit (4)	testPhiEqnsN1, testPhiEqnsN2, testPhiEqnsN3, testPhiEqnsEdgeCases
Appendix Algorithms (SKO-specific)			
App	SKO-defineDataTypes	Indirect	via integration tests
App	SKO-defineSystemVars	Indirect	via integration tests
App	SKO-componentEqns	Indirect	via integration tests
App	SKO-systemEqns	Indirect	via testPhiEqnsN1--N3

Table A.12 documents the integration tests organised by category and formulation.

Category	Formulation	Test	Components
RobotExamples	SKO (serial)	SimpleArmSKOTest	L, J, B, S, A
RobotExamples	SKO (library refs)	SimpleArmLibRefsTest	L, J, B, S, A
RobotExamples	SKO (serial)	AcrobotSKOTest	L, J, B, A
RobotExamples	SKO (controlled)	AcrobotControlledTest	L, J, B, A
RobotExamples	SKO (tree)	MobileRobotSKOTest	L, J, B, S, A
RobotExamples	SKO (closed)	ClosedChainSKOTest	L, J, B
RobotExamples	Featherstone	AcrobotFeatherstoneTest	L, J, B

Table A.12: eqnComp integration tests organised by category and formulation.

Component key. L = local links, J = joints, S = sensors, A = actuators, B = bodies (inertia).

A.6.2 guidedChoice Coverage

The guidedChoice test suite contains 132 tests (101 unit, 31 integration). Table A.13 summarises the coverage for each step, and Table A.14 maps the guidedChoice algorithms to their covering tests.

Integration test key. Integration tests are grouped by formulation and generator mode. Table A.15 lists the primary focus for each category.

Table A.13: Test coverage for guidedChoice algorithm steps.

Step	Algorithm	Tests	Coverage
Main	`guidedChoice`	Integration suite (31 tests)	C1: `GuidedChoiceUtilsTest`, `HeadlessInitializationTest`, `SimpleArmSerialGuidedChoiceTest` C2: integration suite C3: SKO/Featherstone integration tests C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`
S1	`initialiseSidecar`, `addNullDatatype`	`GuidedChoiceUtilsTest` (72), `HeadlessInitializationTest` (1), `SimpleHeadlessTest` (1)	C1: `testInitialiseSidecar`, `testInitialiseSidecarSubExpressions` C2: integration suite C3: SKO/Featherstone integration tests C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`
S2	`calculateFormulation`	`SKO_integrationTest_LibRefs` (1), `SKO_integrationTest` (4), `FEATHERSTONE_integrationTest_AcrobotFromT3` (1), `AcrobotControlledProofTest` (6)	C1: `SKO_integrationTest`, `SKO_integrationTest_LibRefs` C2: integration suite C3: `FEATHERSTONE_integrationTest_AcrobotFromT3` C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`
S3	`resolveSolutionFirstPass`	`GuidedChoiceUtilsTest` (72), `AlgebraicConstraintsTest` (11), `HeadlessPrintingServicesTest` (14), `SimpleArmSerialGuidedChoiceTest` (1)	C1: `GuidedChoiceUtilsTest`, `AlgebraicConstraintsTest`, `HeadlessPrintingServicesTest` C2: integration suite C3: SKO/Featherstone integration tests C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`

Continued on next page

Table A.13 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S4	`reorderSolutions`	Integration suite (31 tests)	C1: `SKO_integrationTest` (ordering assertions) C2: integration suite C3: SKO/Featherstone integration tests C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`
S5	`resolveSolution`	Integration suite (31 tests)	C1: integration suite (end-to-end resolution) C2: integration suite C3: SKO/Featherstone integration tests C4: `ClosedChainSKOIntegrationTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`
	Total	**132 tests**	

Total counts distinct tests; some tests exercise multiple steps, so row totals are not additive.

Table A.14: guidedChoice algorithm to test case traceability matrix.

Step	Algorithm	Coverage	Test Method(s)
Main Pipeline (Chapter 3)			
Main	`guidedChoice`	Unit	—
		Integration	All
S1	`initialiseSidecar`	Unit	`testInitialiseSidecar`, `testInitialiseSidecarSubExpressions`, `HeadlessInitializationTest`, `SimpleHeadlessTest`
S1	`addNullDatatype`	Indirect	via integration tests
S2	`calculateFormulation`	Integration	`SKO_integrationTest_LibRefs`, `SKO_integrationTest`, `FEATHERSTONE_integrationTest_AcrobotFromT3`, `AcrobotControlledProofTest`
S3	`resolveSolutionFirstPass`	Unit	`GuidedChoiceUtilsTest` (variable evaluation, relations), `AlgebraicConstraintsTest` (stage-3 algebraic constraints), `HeadlessPrintingServicesTest`, `SimpleArmSerialGuidedChoiceTest`
S3a	`calculateInitialConditions`	Unit	`HeadlessPrintingServicesTest`, `GuidedChoiceUtilsTest` (constraint evaluation)
S3b	`processAlgebraicConstraints`	Unit/ Integration	`AlgebraicConstraintsTest`, `GuidedChoiceUtilsTest`, `Vis_AlgebraicConstraints_Test`
S4	`reorderSolutions`	Integration	All

Continued on next page

Table A.14 – *Continued from previous page*

Step	Algorithm	Coverage	Test Method(s)
S5	`resolveSolution`	Integration	All (includes mapping, proof, visualisation)

Table A.15 documents the integration tests organised by category and formulation.

Category	Formulation/Mode	Test	Focus
SKO	Base pipeline	`SKO_integrationTest`, `SKO_integrationTest_AcrobotFromT3`	End-to-end SKO pipeline on serial models
SKO	Library refs	`SKO_integrationTest_LibRefs`	Reference component imports and formulation detection
Featherstone	Alternative formulation	`FEATHERSTONE_integrationTest_AcrobotFromT3`	Featherstone formulation coverage
Closed chain	SKO	`ClosedChainSKOIntegrationTest`	Closed-chain topology handling
Visualisation	Vis	`Vis_integrationTest`, `Vis_AlgebraicConstraints_Test`	Visualisation slnRef generation and algebraic constraints
Mapping	Mapping/ MappingPM	`MappingPMIntegrationTest`, `MappingGeneratorTest`, `AcrobotControlledMappingTest`	Mapping p-model and generator integration
Proof	Proof mode	`AcrobotControlledProofTest`	Proof slnRef export and constraints

Table A.15: guidedChoice integration tests organised by category and formulation.

A.6.3 sourceCodeGen Coverage

The sourceCodeGen test suite contains 38 tests (12 unit, 26 integration). Table A.16 summarises the coverage for each step, and Table A.17 maps the sourceCodeGen algorithms to their covering tests.

Integration test key. Integration tests are grouped by target and formulation. Table A.18 lists the primary focus for each category.

Table A.16: Test coverage for sourceCodeGen algorithm steps.

Step	Algorithm	Tests	Coverage
Main	`sourceCodeGen`	Integration suite (26 tests)	C1: `SolutionToCppGeneratorTest` C2: integration suite C3: `TestAcrobot_Featherstone_T5Standalone_Visualisation` C4: `FourBarClosedChainSourceCodeGenTest` C5: `MetamodelCoverageValidationTest`, `MetamodelCoverageGeneratorTest`, `MetamodelCoverageSKOGeneratorTest`

Continued on next page

Table A.16 – *Continued from previous page*

Step	Algorithm	Tests	Coverage
S1	slnRefToSlnTranslation	TestSKO_T5Integration_LibRefs (1), TestSKO_T5Standalone (1), TestSKO_T5FullSimulation (1), TestSKO_T5Standalone_Visualisation (1), TestSKO_T5FullSimulation_Visualisation (1), TestVisual_T5Integration (1), TestIsabelle_T5Equations (1), GenerateProofSolution (1)	C1: TestSKO_T5Integration_LibRefs, TestSKO_T5Standalone C2: integration suite C3: TestAcrobot_Featherstone_T5Standalone_Visualisation C4: FourBarClosedChainSourceCodeGenTest, FourBarClosedChainStandaloneVisualisationTest C5: MetamodelCoverageValidationTest, MetamodelCoverageGeneratorTest, MetamodelCoverageSKOGeneratorTest
S2	slnDFToTargetTranslation	SolutionToCppGeneratorTest (12), TestSKO_T5Latex (1), TestSKO_Acrobot_Latex (1), TestIsabelle_SKO_WithProof (1), TestIsabelle_Acrobot_WithProof (1), TestIsabelle_AcrobotControlled_WithProof (1)	C1: SolutionToCppGeneratorTest C2: integration suite C3: TestAcrobot_FeatherstoneTrajectoryMatchesSKO C4: FourBarClosedChainSourceCodeGenTest, FourBarClosedChainStandaloneVisualisationTest C5: MetamodelCoverageValidationTest, MetamodelCoverageGeneratorTest, MetamodelCoverageSKOGeneratorTest
	Total	**38 tests**	

Total counts distinct tests; some tests exercise multiple steps, so row totals are not additive.

Table A.17: sourceCodeGen algorithm to test case traceability matrix.

Step	Algorithm	Coverage	Test Method(s)
Main Pipeline (Chapter 3)			
Main	sourceCodeGen	Unit	—
		Integration	All
S1	slnRefToSlnTranslation	Integration	TestSKO_T5Integration_LibRefs, TestSKO_T5Standalone, TestSKO_T5FullSimulation, TestSKO_T5Standalone_Visualisation, TestSKO_T5FullSimulation_Visualisation, TestVisual_T5Integration, TestIsabelle_T5Equations, GenerateProofSolution
S2	slnDFToTargetTranslation	Unit/Integration	SolutionToCppGeneratorTest, TestSKO_T5Latex, TestSKO_Acrobot_Latex, TestIsabelle_SKO_WithProof, TestIsabelle_Acrobot_WithProof, TestIsabelle_AcrobotControlled_WithProof, RobotExamples tests

Table A.18 documents the integration tests organised by category and formulation.

A.6.4 Metamodel Coverage

Table A.19 maps p-model metamodel components to their covering metamodel coverage tests. Tests are located in the tests/integrationTests/metamodel-coverage-tests / directory. The test suite contains 56 metamodel coverage tests providing systematic coverage of the core metamodel components. Each test exercises a specific metamodel element in isolation, with descriptive names indicating the component under test. Testing every pairwise combination of metamodel components is combinatorially infeasible; instead, individual

Category	Target/ Formulation	Test	Focus
C++	SKO pipeline	TestSKO_T5Integration_LibRefs, TestSKO_T5Standalone, TestSKO_T5FullSimulation	End-to-end SKO pipeline to C++
Visualisation	C++ visual output	TestSKO_T5Standalone_Visualisation, TestSKO_T5FullSimulation_Visualisation, TestAcrobot_T5Standalone_Visualisation, TestAcrobot_T5FullSimulation_Visualisation, TestSimpleArm_T5Standalone_Visualisation, TestAcrobotControlled_T5FullSimulation_Visualisation	Visualisation code paths and geometry variables
Mapping	C++ with mapping	TestAcrobotControlled_T5FullSimulation_Visualisation_Mapping	Mapping integration for controlled systems
LaTeX	SKO equations	TestSKO_T5Latex, TestSKO_Acrobot_Latex	LaTeX equation generation
Isabelle	Proof output	TestIsabelle_SKO_WithProof, TestIsabelle_Acrobot_WithProof, TestIsabelle_AcrobotControlled_WithProof, TestIsabelle_AcrobotControlled_RegularProof, TestIsabelle_MobileRobot_WithProof, TestIsabelle_T5Equations, GenerateProofSolution	Proof theory and equation export
Featherstone	Alternative formulation	TestAcrobot_Featherstone_T5Standalone_Visualisation, TestAcrobot_FeatherstoneTrajectoryMatchesSKO	Featherstone formulation and trajectory comparisons
Closed chain	Four Bar	FourBarClosedChainSourceCodeGenTest, FourBarClosedChainStandaloneVisualisationTest	Closed-chain artefact generation and compilation

Table A.18: sourceCodeGen integration tests organised by category and formulation.

components are tested separately, and the Complete_Order1 test exercises all components simultaneously in a single comprehensive model. SDF-specific sensor annotations (Ray, Camera, Contact, Sonar, Light) and complete robot models (marXbot, VacuumCleaner) are tested separately in the p_model_SDF/ directory for the SDF code generator and are not included here. The Annotation and AnnotationInstantiation tests demonstrate that annotations are parsed but ignored by the eqnComp generator. The metamodel coverage models are validated by MetamodelCoverageValidationTest (parser and typechecking) and exercised at the generator level by MetamodelCoverageGeneratorTest and MetamodelCoverageSKOGeneratorTest (eqnComp generation and output validation).

Table A.19: P-model metamodel component to test mapping.

Component	Covering Tests
Core Structure	
PModel	All 54 tests, Complete_Order1
Link	LocalLink, ReferenceLink, FixedConnection
LocalLink	LocalLink, LocalLink_Index, Link_Relation
ReferenceLink	ReferenceLink, ReferenceLink_Index
Body	LocalBody_Box, LocalBody_Cylinder, LocalBody_Sphere, LocalBody_Mesh, Body_Relation
LocalBody	LocalBody_Simple, LocalBody_Box, LocalBody_Sphere, LocalBody_Cylinder, LocalBody_Mesh, LocalBody_Index
ReferenceBody	ReferenceBody_Box, ReferenceBody_Cylinder, ReferenceBody_Sphere

Continued on next page

Table A.19 – *Continued from previous page*

Component	Covering Tests
Joint	LocalJoint, ReferenceJoint, LocalJoint_Equations
LocalJoint	LocalJoint, LocalJoint_Equations
ReferenceJoint	ReferenceJoint, Instantiation, ReferenceJoint_NestedActuator, ReferenceJoint_NestedSensor
Sensor	LocalSensor, ReferenceSensor
LocalSensor	LocalSensor, LocalSensor_Index
ReferenceSensor	ReferenceSensor, ReferenceSensor_Index
Actuator	LocalActuator, ReferenceActuator
LocalActuator	LocalActuator, LocalActuator_Index
ReferenceActuator	ReferenceActuator
Geometry	
Box	LocalBody_Box, ReferenceBody_Box
Cylinder	LocalBody_Cylinder, ReferenceBody_Cylinder
Sphere	LocalBody_Sphere, ReferenceBody_Sphere
Mesh	LocalBody_Mesh
Physics	
Pose	All component tests (pose is ubiquitous)
Inertial	Inertial
Inertia	Inertial
Connections	
FixedConnection	FixedConnection, FixedConnection_Part
FlexibleConnection	FlexibleConnection, FlexibleConnection_Part
Composition and Indexing	
Part	Part, FixedConnection_Part, FlexibleConnection_Part
Instantiation	Instantiation, ReferenceJoint
Index	Index, LocalLink_Index, ReferenceLink_Index, LocalActuator_Index, LocalBody_Index
DynamicDevice Properties	
inputs	DynamicDevice_Inputs, Complete_Order1
outputs	DynamicDevice_Outputs, Complete_Order1
InOuts	DynamicDevice_InOuts, Complete_Order1
locals	DynamicDevice_Locals, DynamicDevice_Variables
constants	Constants, DynamicDevice_Variables
equations	LocalJoint_Equations, DynamicDevice_Variables
constraints	DynamicDevice_Constraints, Complete_Order1
Relations	
Link.relation	Link_Relation, Complete_Order1
Link.relation_flexi	Link_RelationFlexi, Complete_Order1
Body.relation	Body_Relation, Complete_Order1

Continued on next page

Table A.19 – *Continued from previous page*

Component	**Covering Tests**
`Joint.relation`	`Joint_Relation`, `Complete_Order1`
`Sensor.relation`	`Sensor_Relation`, `Complete_Order1`
`Actuator.relation`	`Actuator_Relation`, `Complete_Order1`
Solution	
`Solution`	`Solution`, `Complete_Order1`
`IBCondition`	`IBCondition`, `Complete_Order1`
Expressions	
`DerivativeExp`	`DerivativeExp`, `LocalJoint_Equations`
`IntegralExp`	`IntegralExp`
`UnitExp`	`UnitExp`
Units	
`UnitDefinition`	`UnitDefinition`
Fragment	
`Fragment`	`Fragment`
Other	
`Annotation`	`Annotation` (demonstrates annotations are ignored)
`AnnotationInstantiation`	`AnnotationInstantiation` (demonstrates annotation instantiation is ignored)
Comprehensive (all components)	`Complete_Order1`

Bibliography

[1] A. Platzer, *Logical Foundations of Cyber-Physical Systems*. Cham: Springer International Publishing, 2018.

[2] D. Baraff, A. Witkin and M. Kass, "Untangling cloth," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 862–870, 2003.

[3] A. Sanders, *An Introduction to Unreal Engine 4*. AK Peters/CRC Press, 2016.

[4] E. Todorov, T. Erez and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033.

[5] T. Erez, Y. Tassa and E. Todorov, "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2015, pp. 4397–4404.

[6] T. Gibson-Robinson, P. Armstrong, A. Boulgakov and A. W. Roscoe, "FDR3 — A Modern Refinement Checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds., ser. Lecture Notes in Computer Science, vol. 8413, 2014, pp. 187–201.

[7] S. Foster, F. Zeyda and J. Woodcock, "Isabelle/UTP: A Mechanised Theory Engineering Framework," in *International Symposium on Unifying Theories of Programming*, Springer, 2014, pp. 21–41.

[8] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems," *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–41, Oct. 2019. DOI: 10.1145/3342355

[9] Y. Kouskoulas, D. W. Renshaw, A. Platzer and P. Kazanzides, "Certifying the Safe Design of a Virtual Fixture Control Algorithm for a Surgical Robot," in *Hybrid Systems: Computation and Control (Part of CPS Week 2013), HSCC'13, Philadelphia, PA, USA, April 8-13, 2013*, C. Belta and F. Ivancic, Eds., ACM, 2013, pp. 263–272.

[10] S. Mitsch, K. Ghorbal, D. Vogelbacher and A. Platzer, "Formal verification of obstacle avoidance and navigation of ground robots," *The International Journal of Robotics Research*, vol. 36, no. 12, pp. 1312–1340, Oct. 2017, ISSN: 0278-3649. DOI: 10.1177/0278364917733549

[11] A. Cavalcanti, "RoboStar Modelling Stack: Tackling the Reality Gap," in *Proceedings of the 1st International Workshop on Verification of Autonomous & Robotic Systems*, ser. VARS '21, New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–3, ISBN: 978-1-4503-8444-5. DOI: 10.1145/3459086.3459628

[12] D. Kortenkamp, R. Simmons and D. Brugali, "Robotic Systems Architectures and Programming," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Cham: Springer International Publishing, 2016, pp. 283–306.

[13] F. Michaud and M. Nicolescu, "Behavior-Based Systems," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Cham: Springer International Publishing, 2016, pp. 307–328.

[14] K. M. Lynch and F. C. Park, *Modern Robotics*. Cambridge University Press, 2017.

[15] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer Science & Business Media, 2010.

[16] A. Platzer, "The Complete Proof Theory of Hybrid Systems," in *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, ser. LICS '12, USA: IEEE Computer Society, 2012, pp. 541–550, ISBN: 978-0-7695-4769-5. DOI: 10.1109/LICS.2012.64

[17] S. Foster, J. J. Huerta y Munive and G. Struth, "Differential Hoare Logics and Refinement Calculi for Hybrid Systems with Isabelle/HOL," in *Relational and Algebraic Methods in Computer Science*, U. Fahrenberg, P. Jipsen and M. Winter, Eds., Cham: Springer International Publishing, 2020, pp. 169–186, ISBN: 978-3-030-43520-2. DOI: 10.1007/978-3-030-43520-2_11

[18] SDFormat, *SDFormat Home*, Apr. 2023. [Online]. Available: http://sdformat.org/

[19] S. Thrun, W. Burgard and D. Fox, *Probabilistic Robotics*. The MIT Press, 2005.

[20] A. Seth, M. Sherman, P. Eastman and S. Delp, "Minimal formulation of joint motion for biomechanisms," *Nonlinear dynamics*, vol. 62, no. 1, pp. 291–303, 2010.

[21] M. Sherman, "Simbody Theory Manual," Simbios, Stanford University, Tech. Rep., 2019. [Online]. Available: https://github.com/simbody/simbody/blob/master/Simbody/doc/SimbodyTheoryManual.pdf

[22] A. Miyazawa, A. Cavalcanti, S. Ahmadi, M. Post and J. Timmis, "RoboSim Physical Modelling Reference Manual," University of York, Technical Report, Nov. 2020.

[23] A. Miyazawa et al., "Diagrammatic physical robot models," *Softw Syst Model*, vol. 24, no. 5, pp. 1549–1593, Oct. 2025, ISSN: 1619-1374. DOI: 10.1007/s10270-025-01270-9

[24] A. Jain, *Robot and Multibody Dynamics: Analysis and Algorithms*. Springer Science & Business Media, 2010.

[25] A. Cavalcanti et al., "Verified simulation for robotics," *Science of Computer Programming*, vol. 174, pp. 1–37, Apr. 2019, ISSN: 0167-6423. DOI: 10.1016/j.scico.2019.01.004

[26] A. Cavalcanti, P. Ribeiro, A. Miyazawa, A. Sampaio, M. Conserva Filho and A. Didier, "RoboSim Reference Manual," University of York, Technical Report, 2019.

[27] S. Foster, "Hybrid Relations in Isabelle/UTP," in *Unifying Theories of Programming*, P. Ribeiro and A. Sampaio, Eds., Cham: Springer International Publishing, 2019, pp. 130–153, ISBN: 978-3-030-31038-7. DOI: 10.1007/978-3-030-31038-7_7

[28] F. Renda and L. Seneviratne, "A geometric and unified approach for modeling soft-rigid multi-body systems with lumped and distributed degrees of freedom," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 1567–1574.

[29] T. George Thuruthel, F. Renda and F. Iida, "First-order dynamic modeling and control of soft robots," *Frontiers in Robotics and AI*, vol. 7, p. 95, 2020.

[30] N. Bourbaki, *Éléments de mathématique*. Paris: Hermann, 1939.

[31] R. Featherstone and D. E. Orin, "Dynamics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Cham: Springer International Publishing, 2016, pp. 37–66, ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_3

[32] B. Siciliano and O. Khatib, Eds., *Springer Handbook of Robotics*, 2nd ed. Cham: Springer International Publishing, 2016, ISBN: 978-3-319-32550-7. DOI: 10.1007/978-3-319-32552-1

[33] K. Waldron and J. Schmiedeler, "Kinematics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 9–33, ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_2

[34] R. Featherstone and D. Orin, "Robot dynamics: Equations and algorithms," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 1, 2000, 826–834 vol.1. DOI: 10.1109/ROBOT.2000.844153

[35] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[36] I. Kao, K. M. Lynch and J. W. Burdick, "Contact Modeling and Manipulation," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Cham: Springer International Publishing, 2016, pp. 931–954, ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_37

[37] R. L. Burden and J. D. Faires, *Numerical Analysis*. Brooks Cole, 2004, ISBN: 978-0-534-39200-0.

[38] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge university press, 2004.

[39] B. Acosta, W. Yang and M. Posa, "Validating Robotics Simulators on Real World Impacts," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 6471–6478, Jul. 2022. DOI: 10.1109/LRA.2022.3174367

[40] R. Tedrake, *Robotic Manipulation: Perception, Planning, and Control*, Course Notes for MIT 6.421, 2021. [Online]. Available: https://manipulation.mit.edu

[42] L. M. G. Johannessen, M. H. Arbo and J. T. Gravdahl, "Robot Dynamics with URDF & CasADi," in *2019 7th International Conference on Control, Mechatronics and Automation (ICCMA)*, IEEE, Nov. 2019, pp. 1–6. DOI: 10.1109/ICCMA46720.2019.8988702

[64] J. Collins, S. Chand, A. Vanderkop and D. Howard, "A Review of Physics Simulators for Robotic Applications," *IEEE Access*, vol. 9, pp. 51 416–51 431, 2021, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3068769

[83] E. Coumans, *Bullet 2.83 Physics SDK Manual*, 2015. [Online]. Available: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

[84] E. Coumans and Y. Bai, *PyBullet, a Python module for physics simulation for games, robotics and machine learning*, 2016. [Online]. Available: http://pybullet.org

[85] R. Tedrake and the Drake Development Team, *Drake: Model-based design and verification for robotics*, 2019. [Online]. Available: https://drake.mit.edu

[86] NVIDIA. "NVIDIA PhysX SDK documentation: Articulations." Accessed: 2025-12-26. [Online]. Available: https://nvidia-omniverse.github.io/PhysX/physx/5.1.3/docs/Articulations.html

[87] M. L. Felis, "RBDL: An efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, vol. 41, pp. 495–511, 2017.

[88] J. Carpentier et al., "The Pinocchio C++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, IEEE, Jan. 2019, pp. 614–619. DOI: 10.1109/SII.2019.8700380

[89] F. F. A. Silva, J. J. Quiroz-Omaña and B. V. Adorno, *Dynamics of Mobile Manipulators using Dual Quaternion Algebra*, Oct. 2022. DOI: 10.48550/arXiv.2007.08444 arXiv: 2007.08444.

[90] T. Löw, P. Abbet and S. Calinon, "GAFRO: Geometric algebra for robotics," *IEEE Robotics & Automation Magazine*, vol. 32, no. 3, pp. 184–194, Sep. 2025. DOI: 10.1109/MRA.2024.3433109

[91] S. Ivaldi, J. Peters, V. Padois and F. Nori, "Tools for simulating humanoid robot dynamics: A survey based on user feedback," in *2014 IEEE-RAS International Conference on Humanoid Robots*, IEEE, Nov. 2014, pp. 842–849. DOI: 10.1109/HUMANOIDS.2014.7041462

[92] S.-J. Chung and N. Pollard, "Predictable behavior during contact simulation: A comparison of selected physics engines," *Computer animation and virtual worlds.*, vol. 27, no. 3-4, pp. 262–270, 2016, ISSN: 1546-4261.

[93] Nvidia, *PhysX GitHub issue #394*, Jun. 2022. [Online]. Available: https://github.com/NVIDIAGameWorks/PhysX/issues/394

[94] M. Sherman, *Simbody GitHub commits*, Jun. 2022. [Online]. Available: https://github.com/simbody/simbody/graphs/commit-activity

[95] M. Sherman, *Simbody Forum Statistics*, Jun. 2022. [Online]. Available: https://simtk.org/project/stats/forum_stats.php?group_id=47

[96] R. Tedrake, *Drake GitHub commits*, Jun. 2022. [Online]. Available: https://github.com/RobotLocomotion/drake/graphs/commit-activity

[97] R. Smith, *Open Dynamics Engine*, 2007. [Online]. Available: https://www.ode.org/

[98] M. Frigerio, J. Buchli, D. G. Caldwell and C. Semini, "RobCoGen: A code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 36–54, 2016. DOI: 10.6092/JOSER_2016_07_01_P36

[99] S. M. Neuman, T. Koolen, J. Drean, J. E. Miller and S. Devadas, "Benchmarking and Workload Analysis of Robot Dynamics Algorithms," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, Nov. 2019, pp. 5235–5242. DOI: 10.1109/IROS40897.2019.8967694

[100] E. Heiden, D. Millard, E. Coumans, Y. Sheng and G. S. Sukhatme, "NeuralSim: Augmenting Differentiable Simulators with Neural Networks," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2021, pp. 9474–9481. DOI: 10.1109/ICRA48506.2021.9560935

[101] A. Sabelhaus, P. Manovi, J. Eisen, P. Bhandal and S. Kotwicz, *NASA Tensegrity Robotics Toolkit*, Jun. 2022. [Online]. Available: https://github.com/NASA-Tensegrity-Robotics-Toolkit

[102] A. Zeng, S. Song, J. Lee, A. Rodriguez and T. Funkhouser, "TossingBot: Learning to Throw Arbitrary Objects With Residual Physics," *IEEE Transactions on Robotics*, vol. 36, no. 4, pp. 1307–1319, 2020. DOI: 10.1109/TRO.2020.2988642

[103] E. Coumans, *Bullet Forward Kinematics Forum Post*, Jun. 2022. [Online]. Available: https://github.com/bulletphysics/bullet3/issues/2603

[104] S. R. Buss and J.-S. Kim, "Selectively damped least squares for inverse kinematics," *Journal of Graphics tools*, vol. 10, no. 3, pp. 37–49, 2005.

[105] Bullet Physics Contributors, *Bullet `btMultiBody` source*, Jun. 2022. [Online]. Available: https://github.com/bulletphysics/bullet3/blob/master/src/BulletDynamics/Featherstone/btMultiBody.h

[106] B. V. Mirtich, "Impulse-based dynamic simulation of rigid body systems," Ph.D. dissertation, University of California, Berkeley, 1996.

[107] J. Y. S. Luh, M. W. Walker and R. P. C. Paul, "On-Line Computational Scheme for Mechanical Manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, no. 2, pp. 69–76, Jun. 1980, ISSN: 0022-0434. DOI: 10.1115/1.3149599

[108] M. W. Walker and D. E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," *Journal of Dynamic Systems Measurement and Control-transactions of The Asme*, vol. 104, pp. 205–211, 1982.

[109] E. Coumans, *Bullet Inverse Dynamics Tests GitHub*, Jun. 2022. [Online]. Available: https://github.com/bulletphysics/bullet3/tree/master/test/InverseDynamics

[110] R. Featherstone, "A Divide-and-Conquer Articulated-Body Algorithm for Parallel O(log(n)) Calculation of Rigid-Body Dynamics. Part 2: Trees, Loops, and Accuracy," *The International Journal of Robotics Research*, vol. 18, no. 9, pp. 876–892, 1999. DOI: 10.1177/02783649922066628

[111] R. Featherstone, "A Divide-and-Conquer Articulated-Body Algorithm for Parallel O(log(n)) Calculation of Rigid-Body Dynamics. Part 1: Basic Algorithm," *The International Journal of Robotics Research*, vol. 18, no. 9, pp. 867–875, 1999. DOI: 10.1177/02783649922066619

[112] J. Lee et al., "DART: Dynamic Animation and Robotics Toolkit," *Journal of Open Source Software*, vol. 3, no. 22, p. 500, Feb. 2018. DOI: 10.21105/joss.00500

[113] C. E. Lemke, "Bimatrix equilibrium points and mathematical programming," *Management science*, vol. 11, no. 7, pp. 681–689, 1965.

[114] R. W. Cottle and G. B. Dantzig, "Complementary pivot theory of mathematical programming," *Linear Algebra and its Applications*, vol. 1, no. 1, pp. 103–125, 1968, ISSN: 0024-3795. DOI: 10.1016/0024-3795(68)90052-9

[115] E. Coumans, *Bullet3 GitHub Issue 2759*, Jun. 2022. [Online]. Available: https://github.com/bulletphysics/bullet3/issues/2759

[116] NVIDIA. "NVIDIA PhysX SDK documentation: GPU simulation." Accessed: 2025-12-26. [Online]. Available: https://nvidia-omniverse.github.io/PhysX/physx/5.5.0/docs/GPURigidBodies.html

[117] Toyota Research Institute and Drake Contributors. "Drake documentation (Doxygen C++ API)." Accessed: 2025-12-26. [Online]. Available: https://drake.mit.edu/doxygen_cxx/

[118] NVIDIA. "Physics — Isaac Sim documentation." Accessed: 2025-12-30. [Online]. Available: https://docs.isaacsim.omniverse.nvidia.com/5.1.0/physics/index.html

[119] Isaac Lab Contributors. "Overview — Isaac Lab documentation." Accessed: 2025-12-30. [Online]. Available: https://isaac-sim.github.io/IsaacLab/v2.0.0/index.html

[120] RBDL Contributors. "Rigid Body Dynamics Library (RBDL) documentation." Accessed: 2025-12-26. [Online]. Available: https://rbdl.github.io/

[121] C. Lavor, S. Xambó-Descamps and I. Zaplana, *A Geometric Algebra Invitation to Space-Time Physics, Robotics and Molecular Geometry*. Springer, 2018.

[122] S. Gil, A. Badyal, A. Miyazawa, P. G. Larsen and A. Cavalcanti, "A Model-Based Approach for Co-Simulation-Driven Digital Twins in Robotics," *SSRN preprint 5050127*, 2024. DOI: 10.2139/ssrn.5050127

[123] Meshcat Contributors, *Meshcat*, GitHub repository, Jun. 2025. Accessed: 30th Jun. 2025. [Online]. Available: https://github.com/meshcat-dev/meshcat

[124] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall, 1998.

[125] S. Efftinge, *Xtext - Documentation*. Accessed: 26th Nov. 2025. [Online]. Available: https://eclipse.dev/Xtext/documentation/

[126] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, Jul. 2016, ISBN: 978-1-78646-496-5.

[127] M. Spong, "The swing up control problem for the Acrobot," *IEEE Control Systems Magazine*, vol. 15, no. 1, pp. 49–55, Feb. 1995, ISSN: 1941-000X. DOI: 10.1109/37.341864