Improving the workflow for hardware engineers at Philips with a domain-specific language and graphical feedback

Master Thesis
Twan Bolwerk (s1061234)

dr. Mathijs Schuts, Radboud University supervisor
ir. Marco Alonso, Philips Company supervisor
dr. Mairieli Wessel, Radboud University second reader

October 19, 2023
Abstract

This thesis presents the development of an intuitive Domain-Specific Language (DSL) that support bi-directional navigation with immediate graphical feedback specifically for the hardware engineers at Philips. The DSL encompasses the description of robot properties as well as visual semantics, allowing engineers to visualize the robot and also express their visual preferences by specifying how changes are visually represented. This approach acknowledges the subjective nature of graphics and addresses it to some extent. The primary objective is to effectively document and manage changes to robot properties within a multidisciplinary team using version control. The immediate graphical feedback provided by the DSL facilitates easy identification of the specific properties being modified, enhancing collaboration and understanding among users.

The link-origin tracing technique is introduced to enable interactive bi-directional navigation between the textual and graphical representations, along with providing live visual feedback when hovering over textual elements. By incorporating this technique, the thesis proposes an approach to enhance the overall user experience, promote cohesion, improve usability, and deepen the understanding of the connections between textual and graphical links in the DSL’s description of robots.

Keywords: graphical, DSL, URDF, bi-directional, live feedback, visual semantics, collaboration, link-origin tracing
Acknowledgements

I would like to express my gratitude to dr. Mathijs Schuts, my university supervisor, for his invaluable guidance and support throughout my thesis. His expertise and availability have played a significant role in shaping my research and personal growth. I am also grateful to ir. Marco Alonso, my supervisor at Philips, for his inspiring mentorship and innovative research directions. I extend my thanks to the Atlas team for their helpful feedback and to Philips for the opportunity given. Special appreciation goes to the Centrum Wiskunde & Informatica (CWI) for their valuable advice and support. Lastly, I am grateful to my family for their support in making this long-distance arrangement possible. By providing a car and offering a place to stay near Eindhoven, allowing me to pursue this opportunity at Philips. Additionally, I want to express my thanks to my girlfriend for her creative contributions by drawing the robots seen in the mock-ups and her critical proofreading of my thesis. Their invaluable contributions have been instrumental in my research journey.
Contents

Acronyms ................................................................. 6
Glossary ................................................................. 7

1 Introduction ......................................................... 12
  1.1 Context of research ............................................. 12
  1.2 Problem statement ............................................. 13
  1.3 Relevance ...................................................... 13
  1.4 Case study ...................................................... 14
  1.5 Research goal .................................................. 14
  1.6 Overview ....................................................... 15

2 Related work ....................................................... 16
  2.1 Live, Rich, and Composable: Qualities for Programming Beyond Static Text ............................................. 16
  2.2 Toward live domain-specific languages .................................................. 16
  2.3 Towards Semantic Robot Description Languages ............................................. 17
  2.4 Cascade: a Meta-Language for Change, Cause and Effect ............................................. 17

3 Methodology ......................................................... 18
  3.1 Research framework ............................................ 18
  3.2 Research process ............................................... 19
  3.3 Evaluation method ............................................. 20
  3.4 Evaluation plan ................................................ 21

4 Analysis ............................................................. 22
  4.1 Exploring DSLs with graphical feedback .................................................. 22
    4.1.1 A hybrid Editor for Fast Robot Mission Prototyping ............................................. 22
    4.1.2 Graphical simulation of the execution of DSL models ............................................. 22
    4.1.3 Adapting Game Mechanics with Micro-Machinations ............................................. 23
    4.1.4 Combining Simulation-based Engineering and Graph-based Design Languages ............................................. 23
    4.1.5 Domain-Specific Languages with Graphical and Textual Views ............................................. 24
    4.1.6 Modeling and Simulation of Physical Systems in a Mechatronic Context ............................................. 24
    4.1.7 A Visual Environment for Rapid Behavior Definition ............................................. 24
    4.1.8 Visual Support for Learning Monads .................................................. 24
    4.1.9 Engineering hybrid graphical-textual languages with Sirius and Xtext: requirements and challenges ............................................. 25
    4.1.10 Towards seamless hybrid graphical-textual modelling for UML and profiles ............................................. 25
    4.1.11 Conclusion .................................................. 26
  4.2 Workflow and Actors ............................................ 26
  4.3 Proposed workflow ............................................. 27
  4.4 Use cases ......................................................... 28
    4.4.1 Document ................................................... 30
    4.4.2 Present ..................................................... 32
    4.4.3 Calculate .................................................. 34
    4.4.4 Compare .................................................... 35
## 4.4.5 Highlight

### 5 Approach

- **5.1 Features**: 39
- **5.2 Stakeholders**: 41
- **5.3 Functional Requirements**: 41
- **5.4 Non-functional Requirements**: 42
- **5.5 Solution choices**
  - 5.5.1 Language Workbench: 42
  - 5.5.2 Standardized format: 43
  - 5.5.3 Xacro: 43
  - 5.5.4 DesignerSL: 43
  - 5.5.5 Conversion tool: 43
  - 5.5.6 Visualize change of (invisible) properties: 44
- **5.6 Feasibility**
  - 5.6.1 Visualization: 44
  - 5.6.2 Bi-directional navigation: 45
  - 5.6.3 Immediate feedback: 45
  - 5.6.4 Automatically update and focus on hover: 45
- **5.7 Visual semantics**
  - 5.7.1 Compare robot: 47
  - 5.7.2 Highlight difference: 47

### 6 Implementation

- **6.1 Software design**
  - 6.1.1 Component diagram: 48
  - 6.1.2 Class diagram: 49
  - 6.1.3 Sequence diagrams
    - 6.1.3.1 Document: 50
    - 6.1.3.2 Present: 53
    - 6.1.3.3 Compare and Highlight: 54
  - 6.1.4 Link origin tracing: 57
  - 6.1.5 Focus on hover: 59
  - 6.1.6 Highlight differences: 59
- **6.2 Language design**
  - 6.2.1 Context free grammar: 60
  - 6.2.2 Syntax: 61

### 7 Evaluation

- **7.1 Deployment**: 64
- **7.2 Alpha**: 64
- **7.3 Beta**: 64

### 8 Results

- **8.1 Research answers**: 69
- **8.2 Requirements Fulfillment**: 72
- **8.3 Discussion**
  - 8.3.1 Implications
    - 8.3.1.1 Improved Usability: 74
    - 8.3.1.2 Pre-existing Knowledge: 74
    - 8.3.1.3 Conclusion: 74
  - 8.3.2 Limitations: 74
| 9  Conclusion                              | 75 |
| Bibliography                              | 76 |
| A  UrdfSI Abstract Data Structure         | 81 |
| B  Evaluation tasks                       | 82 |
| C  Questionnaire                          | 89 |
| D  Questionnaire responses questions      | 90 |
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree.</td>
<td>24</td>
</tr>
<tr>
<td>BTN</td>
<td>Behavior Transition Network.</td>
<td>24</td>
</tr>
<tr>
<td>DSR</td>
<td>Design Science Research.</td>
<td>18, 19</td>
</tr>
<tr>
<td>DSVL</td>
<td>Domain Specific Visual Language.</td>
<td>24</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus–Naur Form.</td>
<td>24</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine.</td>
<td>24, 69</td>
</tr>
<tr>
<td>GBDL</td>
<td>Graph-Based Design Language.</td>
<td>23</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment.</td>
<td>7, 8, 28, 50, 70</td>
</tr>
<tr>
<td>IGT</td>
<td>Image Guided Therapy.</td>
<td>7, 12–15, 18, 69, 70, 75</td>
</tr>
<tr>
<td>MM</td>
<td>Micro-Machinations.</td>
<td>23</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller.</td>
<td>8</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group.</td>
<td>23</td>
</tr>
<tr>
<td>PPM</td>
<td>Points Per Minute.</td>
<td>20, 21</td>
</tr>
<tr>
<td>RCS</td>
<td>Rate Correct Score.</td>
<td>20, 65</td>
</tr>
<tr>
<td>RMPATCH</td>
<td>Generic Run-Time Model Patching.</td>
<td>16</td>
</tr>
<tr>
<td>ROS</td>
<td>Robot Operating System.</td>
<td>9, 17, 22</td>
</tr>
<tr>
<td>TMDIFF</td>
<td>Textual Model Diff.</td>
<td>16, 17</td>
</tr>
<tr>
<td>UCD</td>
<td>User-Centered Design.</td>
<td>18</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language.</td>
<td>48, 69</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language.</td>
<td>17, 73</td>
</tr>
</tbody>
</table>
Glossary

CAD
Computer Aided Design or CAD, is a technology used in various fields, including engineering, architecture, industrial design, and manufacturing, to create, modify, analyze models. CAD software allows designers and engineers to create 2D and 3D models of products or structures using computer systems. page 13, 27, 28, 30, 40, 41, 43, 64, 69–73, 75

Calibrations
Calibrations in context of IGT systems refer to the process of adjusting and verifying the system’s settings and components to ensure accurate and consistent image quality, geometry alignment, and radiation dose delivery. page 14

Centers of Gravity (COG)
The point at which the entire weight of an object is considered to act, causing the object to behave as if all its mass were concentrated at that point. page 7, 41

CPS
A Cyber Physical System (CPS) is a system of collaborating computational elements controlling physical entities. CPS integrates sensing, computation, control, and networking into physical objects and infrastructure, connecting the physical world with the virtual one. page 13, 75

DSL
A domain-specific language is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language, which is broadly applicable across domains. page 1, 3, 8–10, 13–15, 17, 18, 20, 22–25, 27–31, 34, 36, 38, 40, 45–48, 50, 54, 55, 69–71, 73–75

Eclipse IDE
Eclipse\(^1\) is a versatile software platform known primarily as a Java IDE. However, it goes beyond that, serving as an IDE framework for multiple programming languages, a tools framework for various non-development purposes, and an application framework for building applications. page 28, 42, 70

Hardware engineers
Hardware engineers consists of the mechanical and mechatronic engineers. page 1, 13–16, 18, 40, 65, 69–71, 73–75

Hardware properties
Hardware properties such as Tolerances represent permissible deviations from specified values for dimensions or parameters, ensuring proper component fit. Masses quantify the amount of matter in an object, influencing its Inertia and response to external forces. Centers of Gravity (COG) pinpoint the point of perfect balance and stability within an object. Motion of Inertia (MOI) gauge resistance to rotational motion around an axis, determined by mass distribution. page 12–14, 33, 41, 72

Inertia
The resistance of an object to changes in its state of motion. page 7

LWB
A Language WorkBench (LWB) is a development environment used for the creation, editing, and management of domain-specific languages. Language WorkBenches provide tools and interfaces for defining language syntax, semantics, and transformations, often facilitating model-driven engineering. page 42, 70

Motion of Inertia (MOI)
The behavior of an object when it resists changes in its state of motion due to its inertia. page 7, 41

\(^1\)https://www.eclipse.org/
Rascal  a meta-programming language with integrated syntax definitions, powerful data types, and analysis and transformation constructs [1], [2]. It is used in DSL development, IDE, reverse engineering, and software repository mining. Rascal's features, flexible data types, and strategic syntax tree traversal. Its library ecosystem further enhances functionalities like visualization and analysis of existing programming languages. page 8, 42, 43, 45, 59, 64, 70

Salix  is a Rascal library that facilitates the development of Web-based GUI programs. Salix runs user code on the server side, opposed to client side execution. This architecture is chosen since Rascal does not fully run in the browser (yet). The library uses the Model View Controller (MVC) pattern by sending HTML patches to the browser and interpreting messages from the browser on the server to update the view accordingly. page 45, 49, 54

Stand  A "stand" refers to a support structure or framework that holds the X-ray equipment and allows for adjustable positioning and movement. The stand provides stability and flexibility, enabling the X-ray system to be positioned and adjusted according to the specific requirements of the medical procedure or examination. It allows for precise positioning of the X-ray source and detector, ensuring optimal image quality and patient comfort during imaging procedures. The stand may have various adjustable components, such as height, rotation, tilt, and lateral movement, to facilitate efficient and accurate positioning of the X-ray equipment. page 14

STL  Standard Triangle Language is a file format used by CAD software. It originates from the stereolithography (SLA) a form of 3D printing technologies for creating models, prototypes or patterns. The STL files describe only the surface geometry of a three-dimensional object without any representation of color, texture or other common CAD model attributes. page 28, 31, 36, 38, 44

Tolerances  The permissible limit of variation in a specified dimension, typically related to manufacturing and design. page 7, 41

TypePal  Rascal TypePal is a type checker and validator for programming languages implemented in Rascal. It is designed to analyze and enforce type constraints, ensuring correct usage of types and detecting potential type-related errors in DSLs. TypePal provides static type checking capabilities and can be used to improve the reliability and correctness of software code. TypePal also integrates with other Rascal libraries, including the Rascal language server, enabling additional features like definition navigation that allows users to jump to the definition of entities within the model [3] [4]. page 49, 50, 58

URDF  (Unified Robot Description Format) is an XML-based file format used for describing the structure, kinematics, and dynamics of robot systems. It provides a standardized way to represent the geometric, inertial, and visual properties of robot links and joints. URDF is commonly used in robot simulation and visualization tools, to create accurate robot models that can be used for motion planning, control, and simulation. It allows users to define the robot's joint types, limits, joint connections, visual and collision properties, as well as its sensor configurations. URDF files are human-readable and can be easily shared between different software tools and platforms, making it a popular choice for modeling and simulating robot systems [5] [6]. page 1, 9, 17, 28, 30, 31, 34, 40–44, 49–51, 56–61, 64, 65, 70–75

---

2https://www.rascal-mpl.org/
3https://github.com/usethesource/salix
4https://www.rascal-mpl.org/docs/Packages/typepal/
5http://wiki.ros.org/urdf
### Glossary

**VS Code**
Visual Studio Code, also commonly referred to as VS Code, is a source-code editor made by Microsoft with the Electron Framework, for Windows, Linux and MacOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. page 28, 42, 45, 50, 54, 64, 70, 72

**Xacro**
XML Macros, short for Xacro, is an XML macro language developed by the creators of ROS, the same team behind URDF. With Xacro, you can construct shorter and more readable URDF files by using macros that expand to larger URDF descriptions [7]. page 43, 49, 50, 56, 57, 60–62, 64, 74

**Xtext**
Xtext is an open-source software framework for developing DSL. Unlike standard parser generators, Xtext generates not only a parser, but also a class model for the abstract syntax tree. Besides that it provides a fully featured, customizable Eclipse-based IDE. page 16, 42, 70
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Azurion</td>
<td>12</td>
</tr>
<tr>
<td>1.2</td>
<td>Literature overview DSL [12]</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Design science research process</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Workflow and actors</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Tools and actors</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Actors and tools workflow with DSL</td>
<td>27</td>
</tr>
<tr>
<td>4.4</td>
<td>High level use cases</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Wireframe: Context menu</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Wireframe: Bi-directional navigation</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>Wireframe: Focus on hover link</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Wireframe: Compare robot</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>Wireframe: Highlight robot</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Feature model</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Component diagram DSL</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Class diagram of languages</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>Sequence diagram: Document robot properties</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Sequence diagram: Automatically update robot properties</td>
<td>51</td>
</tr>
<tr>
<td>6.5</td>
<td>Sequence diagram: Click on robot link</td>
<td>52</td>
</tr>
<tr>
<td>6.6</td>
<td>Result click</td>
<td>52</td>
</tr>
<tr>
<td>6.7</td>
<td>Sequence diagram: Compile robot properties</td>
<td>53</td>
</tr>
<tr>
<td>6.8</td>
<td>Sequence diagram: View robot properties</td>
<td>53</td>
</tr>
<tr>
<td>6.9</td>
<td>Sequence diagram: Focus on robot link</td>
<td>54</td>
</tr>
<tr>
<td>6.10</td>
<td>Result hover</td>
<td>54</td>
</tr>
<tr>
<td>6.11</td>
<td>Result compare</td>
<td>55</td>
</tr>
<tr>
<td>6.12</td>
<td>Sequence diagram: Compare robot properties</td>
<td>56</td>
</tr>
<tr>
<td>6.13</td>
<td>Result highlight</td>
<td>57</td>
</tr>
<tr>
<td>6.14</td>
<td>Link origin tracing</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>Boxplot: Questionnaire responses</td>
<td>66</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Feature description</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Functional Requirements of the Tool</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Non-Functional Requirements of the Tool</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Responsibility per component</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>Class Responsibilities</td>
<td>50</td>
</tr>
<tr>
<td>7.1</td>
<td>Evaluation Results</td>
<td>65</td>
</tr>
<tr>
<td>7.2</td>
<td>Survey Multiple Choice Questions</td>
<td>66</td>
</tr>
<tr>
<td>8.1</td>
<td>Fulfillment of Functional Requirements</td>
<td>72</td>
</tr>
<tr>
<td>8.2</td>
<td>Fulfillment of Non-Functional Requirements</td>
<td>73</td>
</tr>
<tr>
<td>C.1</td>
<td>User feedback and ratings questionnaire</td>
<td>89</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context of research

Image Guided Therapy (IGT) consists of employing imaging techniques for precise clinical procedures. Philips offers IGT systems that assist medical professionals in executing minimally invasive procedures with enhanced precision and efficiency.

The Azurion system Figure 1.1, created by Philips IGT\(^1\), is such a system designed to assist users in navigating medical equipment (e.g. catheters) within the human body's vascular system. The system employs X-ray imaging technology to generate three-dimensional visualizations of veins and arteries.

![Figure 1.1: Azurion.](https://www.usa.philips.com/healthcare/e/image-guided-therapy)

The key components of the Azurion system are the X-ray generator and detector. The accurate positioning and orientation of these components are crucial to the system's performance. To achieve the desired system's performance and ensure that each manufactured component moves as intended, complex calculations and simulations are performed.

Each manufactured component undergoes testing and verification. Mechanical engineers, for instance, measure the produced components to verify alignment with the given hardware properties. In sequence, mechatronic engineers calculate necessary adjustments, such as power bounds applied to an axis, to ensure each component operates within

\(^1\)https://www.usa.philips.com/healthcare/e/image-guided-therapy
the same tolerance and moves as intended.

The development and maintenance of these components requires the collaboration of multiple disciplines within the CPS, such as software engineers and hardware engineers, which include mechanical and mechatronic engineers. Graphics, also referred to as visuals, play a significant role in this domain as established by numerous research studies [8], [9].

García et al. highlight the challenge of interoperability between different tools in the multi-disciplinary robotic engineering domain [10], where various tools are utilized including Gazebo for simulation and more general tools like the Office suite. At Philips, mechanical engineers heavily depend on CAD software to accurately track and manage component measurements, while also utilizing spreadsheet applications to document changes in hardware properties. Mechatronic engineers similarly employ computational software tools to perform calculations and simulations.

This research focuses on the hardware engineers, who are less experienced with programming. DSLs can be divided in either horizontal DSLs, used commonly by developers and specialized for technical domains, or vertical DSLs, specifically designed for a domain and often used by domain experts [11], [12]. It is noteworthy that vertical DSLs align more closely with the requirements, making them particularly suitable for domain experts, such as hardware engineers in the CPS.

Exploring the combination of model-based engineering and live graphical feedback presents an opportunity to improve the workflow of the hardware engineers at Philips. By leveraging the benefits of live graphical feedback and its familiarity among hardware engineers, this research has the potential to improve collaboration and streamline processes within Philips. It serves as an exemplary approach to address the challenges of multidisciplinary collaboration in the CPS through the utilization of model-based engineering techniques.

1.2 Problem statement

Within Philips IGT, hardware engineers employ a range of tools to track hardware properties like tolerances, masses, centers of gravity, and moments of inertia. These individual tools, each with its own data formats, conventions, and workflows, are not naturally compatible with each other. This situation is not unique to Philips and is common in organizations with diverse engineering disciplines, it can lead to inconsistencies and communication gaps that potentially impact workflow efficiency.

1.3 Relevance

In the survey conducted by Shen et al. [12], the authors adopt a novel approach to categorize the latest research study in the field of DSLs based on three fundamental implementation concerns: abstract syntax, concrete syntax, and semantics. The authors analyze and characterize these study by examining their parsing and mapping strategies. The categorizations starts between the abstract syntax and concrete syntax (external/internal), the mapping results (textual/graphical) symbols, and the specific functions they prioritize (modeling/visualizing/embedding). This summarized overview of existing DSL categories highlights research gaps that this study aims to address and contribute to. Figure 1.2 horizontal axis contain categories; External (Ext), internal (Int) and textual (Tex), graphical (Gra) and domain specific visual language (DSVL), domain specific modelling language (DSML), domain specific embedded language (DSEL). In the vertical axis the literature references are listed.
CHAPTER 1. INTRODUCTION

Figure 1.2: Literature overview DSL [12]

Figure 1.2 visually represents the extent of research conducted for each specific category of DSLs by colorizing the intersection of these axes. It indicates that there is limited research on the combination of graphical and textual DSLs.

Enabling the language with live graphical feedback aims to minimize cognitive load, enhance collaboration, and improve communication. Hence this research not only addresses the technical aspects of creating a DSL, but also enters the field of behavioral science to develop a suitable graphical representation that effectively conveys changes of hardware properties within the engineering team. By bridging the computing science and behavioral research domains, our study contributes to a comprehensive understanding of how to design and implement a DSL with live graphical feedback that improves communication and collaboration in complex engineering projects.

1.4 Case study

This case study addresses the challenges faced by the hardware engineers working on the Azurion systems. For example, when developing a new stand the mechanic might do some last calibrations without communicating this with the mechatronic engineer and the work of the mechatronic engineer becomes obsolete. With the development of current and future systems a streamlined development process potentially results in faster time to market, ultimately saving on costs for Philips.

The study explores and applies model-based engineering techniques that are well-suited to the hardware engineers at Philips IGT. The hardware engineers are familiar with 3D graphical tools. The creation of such a DSL, establishes a common language that accurately describes system performance and serves as a single source of truth. Furthermore, the DSL should provide live graphical feedback to hardware engineers, enabling them to intuitively comprehend the impact of changes on robot properties early in the development cycle. This early understanding helps identify potential issues and mitigate them before they escalate, ultimately reducing costs and avoiding delays in later stages of the development. By offering an intuitive graphical representation of system properties, the DSL empowers hardware engineers to make informed decisions and optimize the design process. By designing and creating an external textual DSL with live graphical feedback this case study aims to improve the workflow of the hardware engineers at Philips.

1.5 Research goal

By investigating DSLs with graphical feedback that currently exist, along with the technologies and tools used for their creation and real-world examples of their (industrial) applications, we aim to expand our understanding of the available landscape and identify potential solutions that can be adapted to the specific context of the case study. Furthermore, the exploration of how graphical DSL can be applied at Philips, considering their existing technologies and tools, enables us to assess the feasibility and compatibility of incorporating such technologies at Philips. This includes understanding how editing changes can be visualized, when live editing changes propagate to the visual presentation, and how these technologies can be seamlessly integrated. Lastly, the investigation into making a DSL intuitive for mechanical and mechatronic engineers involves examining their current workflow, determining their desired workflow when utilizing a DSL, and assessing the alignment between their needs and the capabilities of the DSL. By addressing these research questions, our objective is to offer valuable insights for the development and implementation of a DSL.
with live graphical feedback. This case study has the potential to improve the workflow of hardware engineers at Philips IGT and may serve as a guiding example.

1. **What domain-specific programming languages with graphical feedback do exist?**
   - a What are the industry examples of domain-specific languages that incorporate graphical feedback?
   - b What technologies and tools are used at Philips IGT by the hardware engineers?

2. **How can graphical domain-specific language technologies be applied at Philips IGT given their existing technologies?**
   - a What technologies and tools are used at Philips IGT?
   - b How to visualize editing changes with the given technologies?
   - c When does a live editing change propagate to the visual presentation?

3. **How can a domain-specific language be intuitive to use for mechanic and mechatronic engineers?**
   - a What is the current workflow of the hardware engineers?
   - b What is the desired workflow for the hardware engineers using a domain-specific language?
   - c Does the domain-specific language fit the hardware engineers?

### 1.6 Overview

In Chapter 2, Related work conducts a review of existing literature and technologies aligned with our research goals. It specifically identifies the areas where our study aims to bridge the existing knowledge gap. Chapter 3, concerning Methodology, describes the research methodology, plan, and evaluation of the case study. Subsequently, Chapter 4, dealing with Analysis, explores DSLs with graphical feedback and identifies the pre-existing workflow of the hardware engineers at Philips IGT, establishing an understanding of the needs for the hardware engineers and their respective use cases of the tool. These needs and problems are then extracted as a feature model in Chapter 5, dedicated to Approach; from these features functional and non-functional requirements follow. The feasibility is then tested with prototypes. Chapter 6, covering Implementation, describes the architecture of the tool including the DSL as a whole, presenting details of both conceptual prototypes and final designs. In Chapter 7, devoted to Evaluation, the effectiveness and usefulness of the case study are assessed through user testing and statistical analysis based on logging. Chapter 8, addressing Results, answers the research questions and validates the fulfillment of each requirement. Finally, Chapter 9, summarizing Conclusion, concludes the entirety of the master thesis. It acknowledges the limitations of the study and proposes potential directions for future research.
Chapter 2

Related work

2.1 Live, Rich, and Composable: Qualities for Programming Beyond Static Text

Josuha Horowitz and Jeffrey Heer define distinct qualities of programming in their article [13]. The article focuses on the qualities of programming, with an emphasis on immediate feedback (liveness), domain-specific editing (richness), and the ability to freely combine language capabilities (composability). Composability in conventional programming enables the inclusion of external libraries or components, separating the responsibilities over multiple sources.

Often programming tasks require the use of multiple composed tools, work in one tool may produce effects that affect another tool. These effects should be visible as soon and with as little distraction and effort as possible.

Joshua Horowitz and Jeffrey Heer identified a trend where both liveness and richness often fail to retain their composability. Their conclusion highlights that the interfaces, which are live and rich, currently offer limited utility in practice as they are realized as standalone applications lacking composability. The exploration of the intersection between the qualities of liveness, richness, and composability remains an open field to be explored.

This research explores the field by striving to adhere to these three qualities, using the familiarity of hardware engineers with 3D graphical tools. Improving the intuitiveness of the tool and ensuring that it fits in the hardware engineers workflow.

2.2 Toward live domain-specific languages

Authors van Rozen and van der Storm [14] recognized the need for program execution to observe its behavior. In traditional programming, changes to a program require re-execution, involving the compilation and execution of updated source code from the beginning. This process can be time-consuming and distracting, particularly when valuable states are lost and difficult to reproduce. To address these challenges, van Rozen and van der Storm propose a more fluid and live experience in programming.

In order to check for textual differences or deltas, between updated source code, van Rozen and van der Storm used the Textual Model Diff (TMDIFF) algorithm [15]. This algorithm is based on two key techniques:

1. **Origin tracking.** Each semantic model element can be traced back to its defining name in the source code.
2. **Text differencing.** When two names in the source code are aligned and the origin relation is the same TMDIFF will identify the corresponding model elements to be the same.

The deltas found with TMDIFF are then converted to run-time edit operations. These edit operations can be applied atomically to the run-time instance using the Generic Run-Time Model Patching (RMPATCH) algorithm. The edit operations are unaware of any additional state maintained in the run-time models. To avoid that any information is lost or the state becomes in an invalid run-time state, RMPATCH can be extended with custom state migrations. All events such as user interactions and changes in the source code are recorded. This history is than used to implement e.g. "undo", allows for persistent application state and back-in-time debugging.

The authors also evaluate existing approaches to comparing textual model deltas, like Xtext and EMFCompare.1

---

1https://www.eclipse.org/emf/compare/overview.html
They highlight the limitations of these methods and suggest that TMDIFF, due to its scope-handling ability, offers a more flexible solution.

In their article the aim is to minimize distractions and preserve intermediate visual state to provide a smoother and more seamless programming experience. Hence it closely aligns with this research goal by giving insights in how and when to visualize changes.

2.3 Towards Semantic Robot Description Languages

The aim of the case study in this research is to create an intuitive language for describing robot properties and visualizing changed hardware components. A commonly used technique to visualize and describe robots is via the Robot Operating System (ROS) URDF. URDF is Extensible Markup Language (XML) based and its data is static. Kunze, Roehm, and Beetz [16] delve into the semantic disparity between high-level actions and low-level capabilities. Their proposal to enhance the URDF with a Semantic Robot Description Language (SRDL) resonates with this research.

Utilizing established formats such as the URDF can significantly enhance the intuitiveness of the DSL this research aims to design and implement. It offers a human-readable and machine-interpretable framework for defining robotic properties and capabilities. By adopting or modifying these standards, as done in this article, it can provide an intuitive structure that reduces the learning curve for users, facilitates interoperability with other systems, and potentially simplifies the process of validating syntax and semantics. However, customization or a completely new format may be required depending on the specifics of the DSL.

2.4 Cascade: a Meta-Language for Change, Cause and Effect

Cascade, a meta-programming language [17], supports bidirectional transformations, providing a mechanism for undoing, redoing changes and view effect that displays a cause-effect-chain textually. This aligns well with this research as it requires similar features. Different from Cascade this research tries to visualize changes graphically, the DSL designed and implemented in this research, could support reduced cognitive load and improved understanding of complex robotic language descriptions. Cascade serves as an inspiration for our change-driven language, highlighting the benefits of live, change-driven development.

21http://www.ros.org/wiki/urdf
Chapter 3

Methodology

3.1 Research framework

In order to develop an intuitive tool that improves the workflow of hardware engineers at Philips, the research employs a case study approach. The analysis incorporates a literature study that specifically examines DSLs that provide graphical feedback. The approach is evaluated by a task based approach followed by a questionnaire.

To improve the workflow of hardware engineers at Philips, an appropriate research framework is chosen that aligns best with the research objectives. User-Centered Design (UCD), Design Thinking and Design Science Research (DSR) are well-established frameworks that have been widely used and documented in the fields of design, human-computer interaction, and innovation. Each of these provides unique benefits and considerations.

UCD involves actively engaging end-users throughout the design process, ensuring that the with live graphical feedback aligns seamlessly with their requirements [18]. However, the reliance on user feedback in UCD may be time-consuming and not feasible due to time constraints and limited availability of the users’ schedules.

Design Thinking emphasizes empathy [19], collaboration, and iterative prototyping [20] to develop intuitive DSLs with live graphical feedback. However, this approach might not place adequate emphasis on technical constraints and feasibility, which are important factors with regard to the existing environment and tools used at Philips.

Design Science Research (DSR), as defined by Iivari [21], is commonly employed in studies involving complex information systems of which some incorporate user interfaces [22]. To effectively provide live graphical feedback alongside textual representation, it is crucial to have an intuitive user interface that facilitates user interaction. DSR focuses on creating novel prototypes. These prototypes will give confidence in the feasibility and understanding of creating intuitive live graphical feedback mechanisms, for which insights from behavioral science and computing science may be used. By combining these research fields, it gives a comprehensive understanding of the cognitive and perceptual aspects of human interaction with the DSL [23]. This knowledge may aid the creation of intuitive live graphical feedback mechanisms that align with users’ mental models and enable more intuitive interaction. Moreover, DSR ensures that our solutions are not only user-centric but also technically feasible.

The Design Science Research (DSR) framework aligns best with our research goal considering the above mentioned advantages. Additionally it puts its emphasis on prototyping of novel solutions rather than mainly utilizing existing ones [24]. Prototyping may be crucial in the development of unique live graphical feedback mechanisms that could effectively and intuitively communicate changes in an improved workflow for the hardware engineers at Philips IGT.
3.2 Research process

This section elaborates on the DSR process. The development phase involves solving a variety of problems through the creation and evaluation of prototypes. Alta van der Merwe, Aurona Gerber and Hanlie Smuts [25] incorporate multiple cycles during the development phase, as illustrated in Figure 3.1. This is an alteration of the original process model, in which this cycle is not present.

Figure 3.1: Design science research process

1. **Awareness of Problem**: Identify and become aware of a specific problem or opportunity that requires a design solution. This phase involves understanding the problem context, gathering relevant information, and recognizing the need for a design intervention.

2. **Suggestion**: Generate and propose potential design solutions or artifacts that have the potential to address the identified problem. This phase involves brainstorming, exploring various options, and formulating initial ideas for the design intervention.

3. **Development**: Design and develop the chosen artifact or intervention based on the suggestions phase. This phase involves creating a tangible design solution and building prototypes.

4. **Evaluation**: Assess and evaluate the effectiveness, efficiency, and usefulness of the developed artifact. This phase involves conducting evaluations through weekly demos, usability testing with alpha and beta testers, conducting experiments, or executing the artifact in order to gather data and analyze the impact of the design solution.

5. **Conclusion**: Draw conclusions and reflect on the findings from the evaluation phase. This phase involves interpreting the evaluation results, assessing the success of the design intervention, and reflecting on the broader implications and potential improvements for future iterations.

The research process can be traced back in the thesis's structure. The Introduction and Analysis chapters address problem awareness and suggestions. The development of prototypes and final product is covered in Approach and Implementation chapters respectively. Then our final product is evaluated in the Evaluation chapter. Based on the evaluation the research will conclude in the Conclusion chapter.
3.3 Evaluation method

Various articles on evaluation, such as [26]–[28], rely on questionnaires to measure usability. Despite the existence of alternative approaches like recording user interactions or analyzing user reactions [29], privacy concerns discourage their use.

A task-based evaluation approach was chosen by Benjamin Hoffman, Neil Urquhart, Kevin Chalmers and Michael Guckert [30] to assess the efficiency of the DSL. Prior to the tasks, the test group received learning materials providing instructions on using the language. After performing the tasks a questionnaire was used to gather initial impressions and obtain feedback on the learning materials. Participants’ programming experience was considered in the result section with regard to the performance measure.

The task based approach used questions and time as an indicator of efficiency. Answers to validate the accuracy of the users and time to measure the efficiency of each user. By keeping track of these two factors the authors were able to mathematically measure efficiency, the researchers employed the Rate Correct Score (RCS) formula [31], defined as:

\[
RCS = \frac{c}{\sum RT} \quad (3.1)
\]

Where \(c\) represents the number of correct responses and \(\sum RT\) is the total reaction time for a trial [32]. This formula originates from the behavioral science field and many adaptations of it exists, such as Rate Correct Score (RCS)

Points Per Minute (PPM) introduced by Hoffmann et al. [30], similar to another measure called throughput [33], reflects correct responses per qualified time unit. In the case of PPM the time unit is minute. Higher RCS, PPM, and throughput values suggest faster, more efficient task completion. However, a high score in less time might not always surpass a higher overall score achieved over longer periods. The balance between speed and accuracy should be carefully considered, keeping in mind the task or domain requirements.

\[
PPM = \frac{60 \cdot c}{\sum RT} \quad (3.2)
\]

As noted by Gökhan Kahraman and Semih Bilgen [28], in addition to questionnaires, which may be highly subjective, the construct validity framework incorporates assessment evidence. In this evaluation this evidence could include log files collected during the participant’s use of the tool, providing a more objective measure.

The RCS and PPM serve as benchmarks for evaluating user efficiency. These benchmarks are utilized to assess future enhancements and determine whether they have effectively improved user efficiency.
3.4 Evaluation plan

The evaluation of the case study involves designing tasks based on the tool’s use cases. Each task includes a question that participants need to answer. The accuracy of participants’ responses is measured using the PPM formula, as shown in Equation 3.2, to establish a baseline for the efficiency of the application. This baseline will be useful for future development to measure the actual efficiency improvements compared to this baseline.

The time taken by participants to complete the tasks is tracked using logging as a quantitative measure. This allows for the gathering of data on task completion times, which can provide insights into the efficiency of the application. Additionally, a questionnaire, see Table C.1, will be administered to collect subjective feedback on participants’ perceptions and programming experience. This will aid in assessing the intuitiveness of the language.

Quantitative measures and questionnaire feedback help assess the language’s intuitiveness. Analysis of logged data can validate user feedback, like slow application performance. This strategy guides the tool’s user experience evaluation and sets an efficiency standard for its future development.
Chapter 4

Analysis

In this chapter, an analysis of existing solutions is conducted for DSLs with graphical feedback some in industrial contexts, extracting valuable insights. We proceed by identifying the workflow and identifying actors to determine specific use cases for the tool. Furthermore, the provided use case descriptions serve as the tasks for the task-based approach discussed in the Methodology chapter.

4.1 Exploring DSLs with graphical feedback

This section explores various Domain-specific Language (DSL) with graphical feedback implementations, including those in industrial settings. We examine relevant tools and techniques to enhance our visualization approach and gather additional requirements for our implementation.

4.1.1 A hybrid Editor for Fast Robot Mission Prototyping

An interactive robot programming editor is presented by Thomas Witte and Matthias Tichy [34]. The textual editor provides an instant visual feedback preview that allows the user to move robot components. In the visual preview that is based on the textual editor, each robot component has a marker. This marker can be used to change the position of that component. The values in the textual editor are changed accordingly. Each character change made in the textual editor are immediately changed in the visual preview. A bi-directional mapping between the code and visualization is created by keeping track of source location information at run time.

They used the Robot Operating System (ROS) standard to display 3D visualizations. Using the ROS standard the authors were able to visualize the messages and shapes using the RViz tool. When a certain movement was not possible an error would be shown in the console of the visual preview window. An interactive preview is created by annotating each value during execution. Any change in the visualization is highlighted in the editor.

The instant visual feedback helps novice and advanced programmers by stimulating experimentation. The authors remarked that bi-directional editing, running the visualization and text editor on separate hosts, even allows for collaborative work.

The focus of this research is on the change of measurements for each component, opposed to Thomas Witte and Matthias Tichy solution were only position of the robots components can be changed. To represent the robot, 3D visualization is utilized, offering instant visual feedback. This feature benefits both novice and advanced programmers by stimulating experimentation. Moreover, the inclusion of a bi-directional editing feature enables collaborative work.

4.1.2 Graphical simulation of the execution of DSL models

Master Thesis of R.C. Boudewijns focused on how the semantics of the industrially used DSL of ASML\textsuperscript{2} can be visualized and how this can be achieved using the B-Motion studio visualization tool [35].

\footnotesize
\begin{itemize}
\item \textsuperscript{1}http://wiki.ros.org/rviz
\item \textsuperscript{2}https://www.asml.com/en/
\end{itemize}
CHAPTER 4. ANALYSIS

The DSL is created with the Eclipse EMF-based implementation. It uses the QVT meta-model standard specified by Object Management Group (OMG). By using this standard the QVTo model transformation tool is able to transform the DSL to an intermediate meta-model, greatly improving re-usability for other DSLs that want similar visualization capabilities. This intermediate meta-model can then be transformed into a specific graphical simulation. The BMotion Studio\(^3\) plugin for Rodin\(^4\) is used to visualize this final model.

The research did not provide any evidence that shows how effective the new method is. Although the user study showed that the visualizations improved usability by making behavior clear while reducing effort during debugging.

4.1.3 Adapting Game Mechanics with Micro-Machinations

In game development, game rules are adjusted in a rapid, iterative and flexible way. Game developers can use tools such as Machinations\(^5\) to balance their game. R.A. van Rozen and Joris Dormans [36] created a DSL using the tool Rascal\(^6\). Micro-Machinations allow the game designer to edit game rules and see the change directly in a visualization model, shortening the feedback loop and significantly reducing design iteration times. This is achieved by improving flexibility and adaptability.

The Rascal (MM AiR) framework with the SPIN model-checker is used for analyzing Micro-Machinations (MM). With this framework comes an IDE that reads textual MM and displays a visual model of the MM interactively.

The MM Lib is used in the game itself. This library tackles technical challenges such as interoperability, traceability and debugging. It enables the embedding of the MM models in the actual game. As the textual MM changes the changes are reflected in the game at run-time [37], [38].

Similar to the engineering domain, the gaming domain has multiple disciplines. van Rozen and Joris Dormans found that an immediate feedback loop greatly improves the game development process in a multi-disciplinary team.

4.1.4 Combining Simulation-based Engineering and Graph-based Design Languages

In the production system domain, engineers are dealing with changing requirements [9]. Karl Kübler, Dominik Schopper, Olivier Riedel and Stephan Rudolph identified that decision-making is often discipline-specific (mechanic, electronics, software) and usually involves compromises. They propose to incorporate simulations early in the development process. A Graph-Based Design Language (GBDL) is used which then can be compiled to different formats using a model to model transformation. A digital twin as an output format is to be used as it provides two advantages during development;

1. It verifies the system in terms of scientific laws.
2. It verifies that all requirements are met.

A digital twin uses real-time data from the sensors of the real-world object it mimics. The digital twin would then be able to simulate responses and choose the best response.

The combination of graphical simulations and GBDL offers several benefits, including improved design optimization, enhanced interoperability, improved visualization, efficient design iterations, and cost and time savings. By leveraging the strengths of both simulations and GBDL.

In the context of a static representation of changing robot properties, a 3D representation serves the purpose of simulation, providing the same benefits as graphical simulations. By utilizing a 3D representation, designers can achieve enhanced visualization, improved design optimization, efficient design iterations, and accurate analysis of static robot properties. This approach offers the advantages of simulations without involving dynamic movements, allowing for an efficient and effective design process.

\(^3\)https://prob.hhu.de/w/index.php?title=BMotion_Studio
\(^4\)http://www.event-b.org/index.html
\(^5\)https://machinations.io/
\(^6\)https://www.rascal-mpl.org/
4.1.5 Domain-Specific Languages with Graphical and Textual Views

Francisco Perez Andres, Juan de Lara and Esther Guerra created a Domain Specific Visual Language (DSVL) using both textual and graphical views. The views are created with the AToM³ tool\(^7\) [39]. They followed a meta-model centric approach where the EBNF grammar is automatically generated based on the meta-model. The argument that is given for this approach is that the decision can be made later whether a graphical or textual concrete syntax or both is chosen. Another argument given is that the produced Abstract Syntax Tree (AST) from parsing is usually not formally defined it causes problems with integration with the multi-view approach proposed by DSL Francisco Perez Andres, Juan de Lara and Esther Guerra. Francisco Perez Andres, Juan de Lara and Esther Guerra noticed that in order to describe an equation it is much more natural to do this in a textual notation.

4.1.6 Modeling and Simulation of Physical Systems in a Mechatronic Context

The thesis of Carl-Johan Sjöstedt describes the mechatronic domain as the domain where electric and mechanical domains meet [40]. It compares tools such as SysML, Simulink and Modelica commonly used in the mechatronic domain. Two cases studies conducted during the thesis try to answer what suitable abstraction levels of a physical system are. Its conclusion is that it very much depends on the case. In the compressor case the Simulink abstraction was a too high abstraction and an object-oriented language like C++ would have been more convenient. As were for the fuel-cell environment the Simulink was a too-low abstraction level and it would have benefited from a higher abstraction level with Modelica. The thesis of Carl-Johan Sjöstedt concludes that to enable model-based development for the mechatronics domain, there is a need for a language that captures the important aspects of the system at the right level of detail.

4.1.7 A Visual Environment for Rapid Behavior Definition

Ryan Houlette and Randy Jensen describe a visual framework that simplifies describing simulated behavior. The framework uses a generalized Finite State Machine (FSM) graph to depict behavior [41]. A so called Behavior Transition Network (BTN). These have current states and transitions like FSM but also hierarchically decompose.

A usability study was conducted that reported that it made them more productive and even non-programmers were able to modify simulated behavior without the help of programmers. The authors themselves notice an improved efficiency of 70 percent. This indicates that even programmers can benefit from the use of visuals.

The visual representation gives the reader insight in whether certain control elements have direct affects the entity's behavior.

4.1.8 Visual Support for Learning Monads

The article by Tim Steenvoorden, Jurriën Stutterheim, Erik Barendsen, and Rinus Plasmeijer on the topic of visual support for learning Monads holds particular relevance to our project. As it uses graphical feedback to make monads more intuitive and understandable. This approach aligns closely with the objectives of our project.

The innovative tool Tonic (Task-Oriented Notation Inferred from Code) incorporates graphical feedback to assist bachelor students in comprehending monads - a concept they often find challenging. Tonic produces a visual representation of the monadic structure in Clean programs, visualized as a flow diagram.

The effectiveness of Tonic was appraised through a three-phase data collection process. Initially, the lectures and assignments were crafted. Subsequently, in the second phase, the students' work on assignments was monitored. Lastly, the students participated in filling out questionnaires and undertaking final exams.

The collated data includes audio and video recordings from practical sessions, screen-casts, handwritten session notes, interview audio recordings, questionnaire responses, and final exam grades.

The evaluation method chosen in this article not only incorporates questionnaires it also uses other means of capturing user responses, such as recordings of audio and video. A similar approach will be used in this research using log files.

\(^7\)http://atom3.cs.mcgill.ca/
4.1.9 Engineering hybrid graphical-textual languages with Sirius and Xtext: requirements and challenges

Justin C. Cooper and Dimitiris Kolovos present [42] requirements, existing approaches and open challenges for integrating graphical editors implemented using the Sirius graphical modelling framework within Eclipse Modeling eco-system.

Sirius is a graphical modeling framework that allows for the creation of custom graphical editors within the Eclipse Modeling ecosystem. It provides a visual representation of models and supports the creation of domain-specific modeling languages (DSLs) with graphical concrete syntaxes. The advantages of using Sirius include the ability to create intuitive and visually appealing modeling editors, as well as the support for model validation and transformation. However, one potential disadvantage is the complexity of learning and using the Sirius framework.

The authors note that a hybrid graphical-textual editor has advantages e.g. a state machine language where states and transitions are specified graphically, but guards in transitions and actions in state are specified using textual expressions.

To address the shortcomings associated with the limitations of Sirius, where users are left with limited guidance because of graphical interface limitations the authors introduced the following requirements.

1. Syntax-aware text editing (error detection, syntax highlighting)
2. Scoping and referencing
3. Rename refactoring
4. Displaying of error/warning markers
5. Accessibility of textual model

The purpose of these requirements is to ensure that the hybrid textual and graphical modeling workbench can provide the best of both worlds, allowing for the creation of models using both textual and graphical syntaxes and providing a seamless integration between them. By meeting these requirements, the workbench can improve productivity, reduce errors, and facilitate collaboration between users.

The defined requirements for a modelling workbench align with the case study as well.

4.1.10 Towards seamless hybrid graphical-textual modelling for UML and profiles

In their work, Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse created a hybrid domain-specific modelling language (DSML) that uses graphical and textual views [43]. This language is designed for creating UML profiles and is built on the foundations of Papyrus and Xtext. The unique feature of their approach is the shared storage base for both graphical and textual representations, which ensures minimal synchronization efforts. Thus, any alterations made in one view - be it textual or graphical - are effortlessly mirrored and visible in the other.

The authors put their creation to the test through a series of experiments consisting of four distinct scenarios: Create1, Modify1, Create2, and Modify2. These modeling tasks were undertaken by developers having comparable experience with the UML language. The time taken to complete these tasks, measured in minutes, provided a basis for comparison.

From the experimental results, it was observed that creating UML elements and setting properties (Create1), as well as modifying a model element by introducing a new element (Modify1), were faster when done through textual notation. On the other hand, the construction of state machines (Create2) proved to be quicker in graphical notation due to the creation of transitions between states using the graphical view. Renaming operations were also significantly faster in textual notation (Modify2) because of the efficiency of the textual regex search & replace function.

In conclusion, the authors demonstrated that their hybrid solution offers superior efficiency - more than doubling the speed in the above mentioned scenarios. This showcases the potential of a combined graphical-textual approach when it comes to domain specific languages.
4.1.11 Conclusion

Based on this literature study, several key insights can be drawn. First, **visual feedback** proves to be beneficial for both novice and advanced programmers. Second, **bi-directional editing** enhances collaborative work. Third, Visualizations help **reduce debugging effort**. Fourth, the implementation of an **immediate feedback** loop significantly improves the development process in multi-disciplinary teams proven in the game domain. Lastly, it is found that writing **equations in textual notation** is more natural compared to graphical notation. These findings provide valuable insights for practitioners and researchers in the field.

4.2 Workflow and Actors

The workflow for both **mechanical engineers** and **mechatronic engineers** in relation to measuring real system properties is illustrated in Figure 4.1.

![Figure 4.1: Workflow and actors.](image1)

The tools used by these actors are depicted in Figure 4.2, which illustrates their interactions. The interactions between the tools can occur either automatically, with data being stored or transferred automatically in the tool, or manually inputted by the user.

![Figure 4.2: Tools and actors.](image2)
• **Mechanical engineer**: The mechanic is responsible for measuring the system real-world properties. The mechanic keeps track of these properties using **CREO**\(^8\) for opening 3D CAD files and **Excel**\(^9\). Some calculations based on these properties are performed in **Excel**. Any change compared to the original **CAD** model are documented in **Word**\(^10\) and presented via **PowerPoint**\(^11\).

• **Mechatronic engineer**: This mechatronic engineer compares previous measurements, saved in a Matlab instance, with the changed properties recorded in **CAD** and **Excel** in order to calculate if the system still conforms to the specifications. One example of this is ensuring the system adheres to the predefined tolerances for each link. These assessments are integral to maintain the system's performance and reliability and is calculated in **Matlab**\(^12\) and **Simulink**\(^13\) using complex calculations and simulations.

### 4.3 Proposed workflow

In the proposed workflow all office tools are replaced by a single tool, being the DSL with live graphical feedback. It composes these tools into a unified language that is capable of presenting changes, serves as documentation, capable of evaluating mathematical expressions that can be used to describe properties.

![Figure 4.3: Actors and tools workflow with DSL.](image)

By removing the office tool-stack the complexity of using multiple tools and multiple source of truth is reduced, this can be observed in Figure 4.3 compared to Figure 4.2. Additionally, no individual licensing is required and the tool is archivable within the Git versioning control system.

A CAD file can be automatically converted to the DSL. The mechanical engineer is able to manual input the measurements, in the expressive manner similar to the Excel spreadsheet. The mechatronic engineer is able to compare previous measurements, which are automatically stored in the single source DSL, that are archived in the version control system of Philips.

---

\(^8\)https://www.ptc.com/en/products/creo/


\(^10\)https://www.microsoft.com/en-us/microsoft-365/word


\(^12\)https://nl.mathworks.com/products/matlab.html

\(^13\)https://nl.mathworks.com/products/simulink.html
4.4 Use cases

At a high abstract level, the use cases are extracted from the existing workflow and incorporated into the workflow designed for the DSL.

The URDF file originates from a CAD drawing. However, this exportation process is outside the scope of these use case descriptions. Hence the exported URDF and STL files will be present and accessible to the engineers.

Robots are composed of numerous links described in the URDF files, referred to as 'links'. For the use cases, it is presumed that there is access to an URDF file that contains a minimum of three links. This assumption is pivotal for the successful execution and visualization of the scenarios in question.

The DSL could potentially be incorporated with an Integrated Development Environment (IDE) or text editor, such as Eclipse IDE or VS Code. For simplicity and clarity in the following use case descriptions, the DSL will be referred to as a 'tool' meaning that it could be part of either an text editor or IDE.

![High level use cases diagram]

Figure 4.4: High level use cases.

Each use case description is paired with a wireframe design tailored for the VS Code editor. This design visually reinforces and complements the details provided in the use case description.
In the wireframe design depicted in Figure 4.5, a viewer is located left side, while a text editor is positioned to the right. However, this can be changed by the user. This layout demonstrates a user's interaction with the link. The interaction is prompted by a right-click on a robot link within the viewer, bringing up a context menu that displays the selected link's name. The viewer is interactive, equipped with adjustable sliders for precise control over axis rotation. In the text editor a DSL instance specification of the robot recites. It reflects changes in the adjacent viewer. This synchronization can either be set to automatic via the "auto" button or can be manually updated using the "reload" button.
4.4.1 Document

This use case enables mechanical engineers to document measurements of a robot. This is achieved by converting the CAD format to URDF. This format URDF can be converted to DSL. By utilizing the DSL, engineers can describe tolerances and gain visual feedback on the specific link being edited.

![Figure 4.6: Wireframe: Bi-directional navigation](image)

Click on link

Clicking on a robot link opens up the link in the editor on the right.

```plaintext
robot
  link
    name="lr_1wo_link_0"
    inertial
      origin:
        rpy="0.0 0.0 -0.007"
      xyz="0.0 0.0 0.0"
    mass:
      value="0.2"
    inertia:
      lx="0.06"
      ly="0.06"
      lz="0.06"
    visual
      origin:
        rpy="0.0 0.0 0.0"
      xyz="0.0 0.0 0.0"
    geometry
      mesh:
        filename="meshes/link_0.stl"
```

Figure 4.6: Wireframe: Bi-directional navigation
Primary Actor: Mechanical engineer

Precondition: The mechanical engineer has an existing URDF file with the respective STL of the robot.

Postcondition: The mechanical engineer successfully documents the robot properties using the DSL.

Main Success Scenario

1. **Open URDF File:** The user initiates the tool and selects the option to open a URDF file.

2. **Load Model:** The user selects the URDF file to be loaded into the tool.

3. **Convert to DSL:** The tool processes the URDF file and converts it to the DSL integral part of the tool.

4. **View Robot:** The converted robot model is displayed in the viewer, providing a visual representation of the robot’s structure and links.

5. **Interact with Viewer:** The user interacts with the viewer, which provides camera controls and sliders for manipulating the robot links.

6. **Automatic Button Click:** The user clicks on a specific button within the viewer, triggering an automatic immediate update behavior associated with that button.

7. **Modify Property of Link:** The user double-clicks the desired link, *Link*, to modify it and adds a new property beneath its name.

8. **Property Addition:** The user adds the following property to *Link*:
   
   (a) **Limit:** This property specifies limitations for the joint connected to *Link*. It includes the following attributes:

   i. **Effort:** Set to "0"
   
   ii. **Lower:** Set to "-1.5707963267948966"
   
   iii. **Upper:** Set to "1.5707963267948966"
   
   iv. **Velocity:** Set to "1.3089969389957472"

9. **Visualization Update:** The tool updates the visualization of the robot model to reflect the modified property of *Link*, providing real-time feedback to the user.

This use case demonstrates how the language tool allows users to open, convert, visualize, interact with, and modify URDF files, providing a seamless workflow for working with robot models.
4.4.2 Present

This use case allows effective communication of measurement changes to colleagues and mechatronic engineers. It serves as a visual representation tool, enabling the presentation of changes through visualization. By hovering over the textual representation using the cursor as a pointer, the view automatically focuses on the specific link being hovered over. This visualization feature enhances the communication process by clearly highlighting and conveying the changes being made. It provides a seamless and intuitive way to communicate modifications, ensuring that all stakeholders can easily understand and interpret the updated measurements for each link.

Figure 4.7: Wireframe: Focus on hover link
Primary Actor: Mechanical engineer

Precondition: The mechanical engineer has successfully documented the robot properties using the tool. This resulted in the creation of a file, robot_v1.dsl, which includes all joint and link descriptions stored in separate files such as link1.dsl and joint1.dsl.

Postcondition: The mechanical engineer visualizes the changes made to the robot’s properties.

Main Success Scenario Description:
The mechanical engineer uses the tool to perform the following actions to inspect and modify robot specifications:

1. **Open robot_v1.dsl**: Within the tool’s interface, the mechanical engineer navigates to the file menu and selects the option to open the robot_v1.dsl file, which contains the specifications of the robot.

2. **View Robot**: After opening the robot_v1.dsl file, the mechanical engineer selects the "View Robot" option from within the tool or editor to visualize the robot in the main workspace.

3. **Click Button Auto**: In order to present and outline the active visual link.

4. **Hover Over the Text Editor**: To locate a specific link within the robot’s specifications, the mechanical engineer hovers the cursor over the text editor area. As the cursor moves over the content, the actively hovered link is outlined, aiding in identification.

5. **Right-Click on Link in view**: Once the correct link is identified, the mechanical engineer performs a right-click on the link within the viewer. This action brings up a context menu with the name of the related link, further clarifying the identity of the visual link.

6. **Double Left-Click on Link in view**: The mechanical engineer double left-click on the visual link. This action triggers the opening of an text editor at the associated detailed specification location of link. This allows the mechanical engineer to use both textual and graphical representation to further present changed properties.

Extensions

1. If the software fails to visualize the robot due to technical issues, an error message is logged in the log.txt file.

By following these steps, the mechanical engineer can efficiently navigate, inspect, and present the robot’s hardware properties using the tool’s textual and graphical representation. This use case showcases the engineer’s ability to leverage the tool’s features to present and communicate the robot’s hardware properties.
4.4.3 Calculate

The mechanical engineer wants to add a property to a specific link in the robot model. The user intends to view the robot and modify the limits by using a function that describes radians in terms of degrees so that it is easier to understand.

**Primary actor:** Mechanical engineer

**Precondition:** The mechanical engineer has a URDF containing the robot model and its geometric properties called.

**Postcondition:** The mechanical engineer successfully calculates the inertia properties of the robot.

**Main Success Scenario:**

1. **Load and Parse File:** The system loads and parses the specified URDF file and converts it to the DSL. This file contains the robot model's description and specifications in the language instance format.

2. **Request to View Robot Model:** After successfully parsing and converting the URDF file, the user requests to view the robot model within the system's interface. This action prompts the system to display the robot model, allowing the user to visualize it.

3. **Specify Link for Property Addition:** Within the interface, the user specifies the link to which the property should be added. In this use case, the user selects link for the property addition.

4. **Property Addition:** The user adds the following property to *Link*:

   (a) Limit: This property specifies limitations for the joint connected to *Link*. It includes the following attributes:

      i. Effort: Set to "0"
      
      ii. Lower: Set to "radians(-90)"
      
      iii. Upper: Set to "radians(90)"
      
      iv. Velocity: Set to "radians(75)"
      
      v. The system saves the modified language instance with the added property.

5. **Add Property:** Once the user provides the property details, the system adds the specified property under the name of the link in the language instance. This modification ensures that the updated property is associated with the correct link.

6. **Save Modification:** After adding the property, the system saves the modified language instance.

7. **Double-Click on Element with Ctrl Pressed:** After identifying the desired element, the mechanical engineer double-clicks on the element in the workspace. The updated file now includes the newly added property evaluated to the values:

   (a) Limit: This property specifies limitations for the joint connected to *Link*. It includes the following attributes:

      i. Effort: Set to "0"
      
      ii. Lower: Set to "-1.5707963267948966"
      
      iii. Upper: Set to "1.5707963267948966"
      
      iv. Velocity: Set to "1.3089969389957472"

By following these steps the mechanical engineer is able to express calculations similar to Excel.
4.4.4 Compare

Notably, the **Compare** use case of the mechatronic engineer is identified. This use case forms part of an external scenario, which involves the manual input of values into Matlab. The **Compare** use case allows the mechatronic engineer, to quickly spot differences between versions by leveraging live graphical feedback.

![Figure 4.8: Wireframe: Compare robot](image)

In the **Compare robot** use case, a side-by-side view provides users with a visual representation of changes that can be manually observed and compared.
Primary actor: Mechatronic engineer

Precondition: The mechatronic engineer has two different DSL versions of the same robot file with the respective STL or .obj files of the robot.

Postcondition: The Mechatronic Engineer successfully identifies the differences in properties

1. Open the Tool: The mechatronic engineer launches the tool on their computer.

2. Open compare.dsl: Within the tool's interface, the mechatronic engineer accesses the file menu and selects the option to open the compare.dsl file. This file contains the robot specifications for comparison purposes.

3. View Robot: After opening the compare.dsl file, the mechatronic engineer selects the "View Robot" option from the toolbar or menu. This action displays the robot model and its associated elements in the main workspace.

4. Right-Click on the Robot: To access a context menu that provide information about the robot link, the mechatronic engineer right-clicks on the robot in the workspace.

5. Compare with The mechatronic engineer types that it want to compare with two robot descriptions. The system loads the first robot descriptions into a left split view and the other one in the right split view the differences between the two versions can be observed by the mechatronic engineer.

6. Hover Over Element link2: To locate a specific robot element, the mechatronic engineer hovers the cursor over the element named link2. As the cursor moves over the workspace, the element is highlighted or outlined, facilitating identification in both views.

By following these steps, the mechanical engineer can effectively inspect robot elements, execute actions, and perform comparisons using the mechanical engineering tool. This use case demonstrates the engineer's ability to leverage the tool's features for detailed analysis and manipulation of robot specifications.
4.4.5 Highlight

Alternatively, the highlight use case offers an efficient solution for mechatronic engineers to automatically identify and highlight differences between links. This use case serves as an internal scenario, eliminating the need for manual input of values into Matlab. By leveraging automatic highlighting and live graphical feedback, engineers can quickly and accurately pinpoint variations between link versions. The highlight use case significantly reduces the reliance on manual comparisons, leading to a potential reduction in errors.

Figure 4.9: Wireframe: Highlight robot
Primary Actor: Mechatronic engineer

Precondition: The mechatronic engineer has two different DSL versions of the same robot file with the respective STL or .obj files of the robot.

Postcondition: The Mechatronic Engineer successfully identifies the differences in properties.

1. Open the Tool: The mechatronic engineer launches the tool on their computer.

2. Open **highlight.dsl**: Within the tool's interface, the mechatronic engineer accesses the file menu and selects the option to open the **highlight.dsl** file. This file contains the robot specifications for highlighting and comparison purposes.

3. View Robot: After successfully opening the **highlight.dsl** file, the mechatronic engineer selects the "View Robot" option from the toolbar or menu. This action triggers the display of the robot model and its associated elements in the main workspace.

4. Right-Click on the Robot: In order to access a context menu with options related to the robot and its elements, the mechatronic engineer performs a right-click on the robot within the workspace. This context menu provides information about the robot link.

5. Highlight Differences: The mechatronic engineer types that it want to highlight differences between two robot descriptions. The system loads the first robot descriptions into one view and highlight the differences between the two versions of the robot specifications. Elements that have changed between versions are displayed in solid color, while elements that have remained the same are displayed as transparent.

6. Inspect Highlighted Elements: The mechatronic engineer inspects the highlighted elements to identify the differences in their properties. By comparing the solid-colored elements to the transparent elements, the engineer can discern the changes made between the versions.

By following these steps, the mechatronic engineer can effectively utilize the tool to compare and analyze the differences in properties of robot elements between versions. The highlighting feature helps in visually identifying the changes, allowing for a detailed examination and comparison of the robot specifications. This facilitates informed decision-making and analysis for the engineer.
Chapter 5

Approach

The Approach chapter uses the Problem Statement and the Analysis chapter for the extraction of features that are essential for addressing the problem effectively. These features then form the basis for deriving both functional and non-functional requirements, providing a comprehensive understanding of the expected outcomes of the solution. To fulfill these requirements, the Analysis chapter explores various options and provides reasoning for each choice, ensuring that the selected solutions align with the identified features and requirements. Furthermore, prototyping is utilized to build confidence in the chosen technologies and validate their suitability for integration into the final design and implementation. By following this approach, the project gains a solid foundation for selecting the most appropriate solutions and ensuring their successful implementation.

5.1 Features

By combining problem identification and analysis of existing solutions, workflow and tools, a feature model is extracted, inspired by the study by Merino et al. [44]. This feature model is a direct mapping between the identified problems and the corresponding features. It involves categorizing the different features of a system to understand their relationships and relevance in addressing user requirements and problem-solving.
The features from Figure 5.1 address the following problems:

1. **Interoperability**: The feature of interoperability ensures integration into existing workflows by enabling compatibility between data-formats of different tools.

2. **Inconsistencies and Communication Gaps**: The feature of visualization ensures intuitive use among engineers familiar with graphical tools, reducing inconsistencies and facilitating effective communication through clear and visually represented information.

3. **Licensing Complexity**: The tool itself with all of its features such as evaluation, visualization and supportive features to documentation such as syntactic services address licensing complexity by allowing engineers to perform these actions within one tool itself, reducing dependence on multiple separate licenses.

4. **Absence of a Unified Archiving System**: Collaboration and interoperability features that fit the hardware engineers workflow and enable engineers at Philips to share a single source of truth using a single version control system.

In the Table 5.1, the terms are further explained.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interaction</td>
<td>Bi-directional navigation</td>
</tr>
<tr>
<td>Immediate</td>
<td>Live visual feedback</td>
</tr>
<tr>
<td>3D</td>
<td>Graphical view in three dimensions</td>
</tr>
<tr>
<td><strong>Visualization</strong></td>
<td>Graphical feedback</td>
</tr>
<tr>
<td>Include</td>
<td>Support for file inclusion to improve modularity</td>
</tr>
<tr>
<td>Calculation</td>
<td>Mathematical expression similar to Excel</td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td>Evaluating expressions</td>
</tr>
<tr>
<td>Highlighting</td>
<td>Syntax highlighting improves readability of DSL</td>
</tr>
<tr>
<td>Validation</td>
<td>Providing feedback that prevents errors</td>
</tr>
<tr>
<td><strong>Syntactic services</strong></td>
<td>Intuitive use emphasize user-friendly interface</td>
</tr>
<tr>
<td>Compare</td>
<td>Comparing robots for showcasing modifications</td>
</tr>
<tr>
<td>Archivable</td>
<td>Archiving within Philips’ version control system</td>
</tr>
<tr>
<td><strong>Collaboration</strong></td>
<td>Communication of changes for hardware engineers</td>
</tr>
<tr>
<td>URDF</td>
<td>Importing URDF into the tool</td>
</tr>
<tr>
<td>CAD</td>
<td>Importing CAD files into the tool</td>
</tr>
<tr>
<td><strong>Interoperability</strong></td>
<td>Compatibility in workflow of hardware engineers</td>
</tr>
</tbody>
</table>

**Figure 5.1**: Feature model

**Table 5.1**: Feature description
5.2 Stakeholders

The stakeholders defined in this section are the individuals that are directly influenced by this project with regard to their workflow.

1. Mechanic engineer
2. Mechatronic engineer

5.3 Functional Requirements

The functional requirements detail the tool’s operations and activities, prioritized via the MoSCoW method.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RQ-1]</td>
<td>The tool must accurately interpret and visually represent robots based on URDF file descriptions.</td>
</tr>
<tr>
<td>[RQ-2]</td>
<td>The tool should facilitate the input of mathematical expressions, mirroring Excel’s functionality.</td>
</tr>
<tr>
<td>[RQ-3]</td>
<td>The tool should permit the inclusion of separate files, allowing users to compartmentalize their code into distinct, reusable components.</td>
</tr>
<tr>
<td>[RQ-4]</td>
<td>The tool must serve as documentation and have presentation capabilities, thereby replacing office tools like Excel, Word, and PowerPoint.</td>
</tr>
<tr>
<td>[RQ-5]</td>
<td>The tool must allow modification of hardware properties, such as Tolerances, masses, Centers of Gravity (COG), and Motion of Inertia (MOI).</td>
</tr>
<tr>
<td>[RQ-6]</td>
<td>The tool could offer functionalities for visualizing properties.</td>
</tr>
<tr>
<td>[RQ-7]</td>
<td>The tool could provide functionalities to visualize selected elements in text.</td>
</tr>
<tr>
<td>[RQ-8]</td>
<td>The tool should compare two robot descriptions, presenting their differences and similarities.</td>
</tr>
<tr>
<td>[RQ-9]</td>
<td>The tool would be enhanced by a feature that allows understanding and processing CAD files as input, potentially by converting them to URDF files first.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must</td>
</tr>
<tr>
<td>Should</td>
</tr>
<tr>
<td>Should</td>
</tr>
<tr>
<td>Must</td>
</tr>
<tr>
<td>Could</td>
</tr>
<tr>
<td>Could</td>
</tr>
<tr>
<td>Should</td>
</tr>
<tr>
<td>Nice-to-have</td>
</tr>
</tbody>
</table>

Table 5.2: Functional Requirements of the Tool
5.4 Non-functional Requirements

The non-functional requirements provide criteria to evaluate the performance of the tool. Non-functional requirements lack prioritization as they encompass vital qualities essential for overall system effectiveness, including performance, maintainability and usability.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Quality attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NFRQ-1]</td>
<td>The tool should be user-friendly, enabling users to intuitively write, read, and understand code while receiving live graphical feedback.</td>
<td>Usability</td>
</tr>
<tr>
<td>[NFRQ-2]</td>
<td>The tool should efficiently parse and generate code, thereby ensuring high-performance visualization execution.</td>
<td>Efficiency</td>
</tr>
<tr>
<td>[NFRQ-3]</td>
<td>The tool should remain flexible and extensible, ready to accommodate future enhancements and requirements.</td>
<td>Extensibility</td>
</tr>
<tr>
<td>[NFRQ-4]</td>
<td>The tool should prevent crashes or incorrect results when issues arise.</td>
<td>Robustness</td>
</tr>
<tr>
<td>[NFRQ-5]</td>
<td>The tool should be equipped to visualize URDF.</td>
<td>Interoperability</td>
</tr>
<tr>
<td>[NFRQ-6]</td>
<td>Comprehensive and lucid documentation should be provided to guide both users and developers.</td>
<td>Documentation</td>
</tr>
<tr>
<td>[NFRQ-7]</td>
<td>The tool must function on Windows 10 operating systems without necessitating modifications.</td>
<td>Portability</td>
</tr>
<tr>
<td>[NFRQ-8]</td>
<td>The tool should be easy to maintain and upgrade, featuring clear and well-structured code.</td>
<td>Maintainability</td>
</tr>
</tbody>
</table>

Table 5.3: Non-Functional Requirements of the Tool

5.5 Solution choices

This section will give the reasoning behind choices made with regard to the case study and its requirements.

5.5.1 Language Workbench

A LWB provides tools and interfaces to define language syntax, semantics, and transformations, Language WorkBenches facilitate model-driven engineering. They enable developers and domain experts to work with high-level abstractions that closely resemble the specific problem domain. This can lead to more efficient development processes, more maintainable code, and closer collaboration between different stakeholders. Language WorkBenches are instrumental in bridging the gap between generalized programming languages and the unique requirements of specialized domains.

Within the range of Language WorkBenches available, Philips predominantly uses Xtext. However, the company has recently begun a transition from Xtext to Rascal, as it has been flagged as deprecated in the latest Xtext 2.31.0 Release Notes\(^1\).

Contrastingly, Rascal is under active development, with the creators committing to a substantial period of support, extending up to a decade. Given these circumstances, we opt for Rascal over Xtext, mainly due to its effective integration with VS Code.

The choice of Rascal aligns well with Philips' internal practices, as VS Code is widely used within the company. The preference within Philips remains with the original VS Code over the Eclipse IDE that Xtext defaults to. As a result, the shift towards Rascal is both strategic and practical, leveraging existing infrastructure and expertise within the organization.

\(^1\)https://www.eclipse.org/Xtext/releasenotes.html
5.5.2 Standardized format

The preference for using URDF over SRDL, as mentioned in Section 2.3, is based on several considerations. Our language focuses on communicating static robot properties, and therefore the need for semantic actions provided by SRDL is unnecessary for our purposes.

Using URDF offers the advantage of being able to express most hardware properties, with the exception of tolerances. Additionally, the availability of existing visualization tools like RViz validates the capability of the URDF format to provide the necessary information required to graphically represent robots.

Authors Daniella Tola and Peter Corke [5] present the findings from a survey conducted among robotic developers, which highlights that a large majority of participants have used URDF for simulating robots. The survey also identifies challenges and limitations associated with URDF, such as difficulties in modeling parallel linkages and closed-chain systems, limited documentation, and a restricted number of dynamic parameters for robot modeling. The future of URDF is explored, with opinions divided on whether it will become more prevalent or be replaced by other standards or tools. Overall, there is a consensus among participants regarding the need for improved tooling to ensure the continued use and development of URDF.

5.5.3 Xacro

To address the issue of large URDF files, particularly with complex robot models like the Azurion, a solution based on modularization and composition of URDF using the Xacro language can be employed. Xacro offers a way to create modular and reusable components, making robot descriptions more manageable and organized.

In this approach, a custom Xacro parser has been developed to convert Xacro code into DesignerSL, a custom language. The shell "exec" function from Rascal is leveraged to call the Xacro compiler. This "exec" function has also been utilized in the immediate feedback prototype Section 5.6.3.

Although Xacro simplifies URDF composition and also introduces expression evaluation, it still relies on XML format, which is not particularly user-friendly or intuitive to use.

5.5.4 DesignerSL

To overcome these limitations, the solution introduces DesignerSL, which extends XacroSL the language that encapsulated Xacro and is able to convert it to DesignerSL. DesignerSL enables the creation of a more user-friendly syntax, supports custom property definitions (including tolerances), and provides enhanced visualization options. The aim is to offer a more intuitive and flexible language for robot modeling, addressing the drawbacks associated with traditional XML-based URDF descriptions.

Additionally, creating DesignerSL adds the flexibility to define and design custom syntax, making the language more intuitive to use and easier to extend with new semantics and introduce custom properties, enhancing its expressiveness and adaptability to different use cases.

One of these new use cases is the integration of visual semantics. This allows mechanics to describe the desired appearance of the view in the language. This approach allows engineers to communicate not only via textual notation but also through graphical views, improving the collaboration and workflow by providing live graphical feedback.

As a result of these advancements, our language exhibits key qualities such as liveness, richness, and composability, as discussed in Horowitz et al.’s work on live programming [13].

5.5.5 Conversion tool

One of the benefits of using URDF, a widely adopted and standardized format is that other tools often have conversion tools available from a proprietary format to URDF. Blender\(^2\) has been selected as an intermediary tool that enables conversion from CAD export files into URDF. Blender is open-source, has free licensing, and it has widespread community support. This not only makes Blender cost-effective but also ensures that the tool is consistently updated and improved by a global community of developers. This solution can be traced back to requirement [RQ-9].

\(^2\)https://www.blender.org/
5.5.6 Visualize change of (invisible) properties

To address the challenge of visualizing changes in properties, we need to consider the visibility of these changes. For example, properties like mass may have different values but appear the same in the visualization. Based on relevant literature, we will implement and evaluate the most effective solution.

There are two main approaches to visualizing editing changes: comparing two instances and automatically highlighting differences. When comparing two versions, Munzner's book [45] recommends using a side-by-side view rather than animations. Change blindness, where users struggle to perceive alterations over time, supports this preference for immediate contrast and reduced cognitive load.

Alternatively, differences can be emphasized through automatic highlighting, such as using varied colors, shapes, or positions to accentuate distinctions between properties. Munzner's book introduces the concept of the "pop-out" effect, where certain visual features instantly stand out, allowing users to quickly spot differences without focused attention. By implementing both the side-by-side comparison and highlighting strategies, we aim to create a solution that addresses the invisibility of property changes and enhances the user experience.

The decision to use side-by-side comparison and highlighting is based on several factors. Firstly, the side-by-side view helps counter change blindness by allowing users to identify differences more easily. It reduces cognitive load by minimizing reliance on memory and context switches. Automatic highlighting of differences provides immediate visual cues to accentuate changes, even for invisible properties like mass. However, it is important to consider potential conflicts with user-defined materials and overlapping color use. Alternative methods, such as making components transparent to underscore changes, can also be used for highlighting. Animations were not chosen due to their potential to induce change blindness, which goes against our goal of making changes more obvious to users. The combination of side-by-side comparison and highlighting differences aims to make changes more distinguishable and improve the user experience.

5.6 Feasibility

This section outlines the prototypes that have been developed during our iterative development process mentioned in the Methodology chapter. The goal of our prototypes is to find solutions for issues and verify that a given technology or techniques solves it.

5.6.1 Visualization

An existing artifact[3] was utilized as the starting point, which incorporates BabylonJS[4], a JavaScript 3D graphical library. This library can load STL files and assemble them using the URDF. However, it lacks certain features such as comparing changed properties, highlighting differences, bi-directional interaction and side-by-side views, and maintaining state after changes. Additionally, it requires the use of the URDF format, which has a non-user-friendly syntax and leads to large file sizes when designing systems like Azurion. Nonetheless, despite these limitations, the artifact serves as a beneficial starting point.

The visualization is extended with a context menu. The view's context menu enhances user experience by enabling the identification of links through right-click actions, revealing a menu that displays the link's name, depicted in Figure 4.5. This visual representation establishes a direct connection between the textual and visual links, improving cohesiveness and comprehension of both representations.

The sliders facilitate joint movement, enhancing the view by providing an interactive experience. While primarily focused on static properties, this feature can greatly improve the visualization of specific links. Moreover, the reload and auto buttons play a vital role in updating the view with the latest changes made in the text editor, whether through manual input or automatic updates. These functionalities will be further elaborated upon in subsequent sections, delving into their mechanisms and benefits in greater detail.

5.6.2 Bi-directional navigation

The benefits of bi-directional interaction, as outlined by Thomas Witte and Matthias Tichy [34], is that it reduces cognitive load and improves the overall user experience. By integrating the view and text editor through a bi-directional approach, this approach aims to alleviate cognitive load and further enhance the user experience. It allow users to interact with visual links by clicking on them, instantly directing them to the specific location in the text editor where that element is specified.

Salix has been used in game development to improve collaboration among multiple disciplines [46], resulting in improved collaboration. By leveraging the Salix library, a successful prototype of bi-directional input visualization was achieved. This feature enables users to effortlessly click on visual elements, which then redirects them to the corresponding location within the text editor.

Building upon the insights from [13], our approach embraces the concept of richness by not only providing visual feedback but also integrating the textual and graphical representations. Although persistent changes are not supported via the visualization interface, this integration fosters a cohesive and intuitive user experience by establishing a strong connection between the two interfaces [NFRQ-1].

5.6.3 Immediate feedback

Earlier research [47] and [48] have indicated that immediate feedback significantly improves debugging. More recent studies also successfully applied immediate feedback in their DSLs [17], [49], [50] and [51]. Additionally, incorporating immediate feedback ensures that the DSL adheres to the liveness quality, improving the programming experience. To fulfill the non-functional requirement [NFRQ-1] of creating an intuitive language, integration of this immediate feedback with the chosen technologies, Rascal and VS Code, integration is necessary.

In VS Code, the "summarize event" of the Rascal Language server library is triggered each time the file is saved. We take advantage of this event to re-render the visualization using the latest valid state of the DSL, ensuring an up-to-date representation of the data.

This approach of fully refreshing the visualization, causes a brief moment of disappearance and reappearance of the robot, upon saving the file. This can be improved in terms of user experience. A more gradual change that maintains the robot in the exact same state and keeps it constantly in view would provide a smoother and more user-friendly way of communicating the modifications, minimizing context-switches and enhancing overall usability. By avoiding the robot’s disappearance, users can maintain continuous visual feedback and better understand the impact of their changes.

It makes sense to make this part of the summarize behavior as besides source code errors and warnings, the visualization serves the same purpose, and most languages perform this update upon saving to achieve it. Additionally, this approach helps minimize the performance impact of running resource-intensive processes with each change.

5.6.4 Automatically update and focus on hover

Salix provides a polling feature that performs an action over a specified interval as short as one second. Although this feature may impose certain performance overhead, it helps our language to fulfill the ‘liveness’ quality criteria by consistently checking for differences and updating the view accordingly. A toggle button "auto" is included, see Figure 4.5, this enables or disables the polling feature as it can consume some system resources in terms of performance. With polling enabled the automatic updates are enables, thus allowing for immediate feedback.

Upon detection of any discrepancy, a message with the updated model is send to our viewer which runs on a separate web-server thread. Rather than refreshing the complete web-view, the web-server can updates the visualization gradually. This ensures that the robot is always maintained in the view during these gradual updates, reducing distractions from changing visual context. This feature enhances the user experience by providing a seamless transition during updates and maintaining continuity in the visualization.

Each time the user changes the source code and than saves the source code, a Rascal "summarize" event is triggered. In this event we set a flag to true, the polling mechanism that runs on a separate web-server thread than sends a http message to our viewer. The viewer is equipped with so-called listener (hooks) which update the visualization while preserving state. This significantly reduces distraction of reloading graphics and therefore improving usability.
This same mechanism "polling" is used to focus on an element that is hovered over by mouse. This improves users understanding of where the user is in source code. The focus mechanism sets the camera focus on that specific element and marks the element with a specific outline color.

5.7 Visual semantics

This approach offers a configurable view utilizing a DSL with visual semantics. This has several advantages. Firstly, when comparing robots constructed with different materials, the distinctions that are apparent when viewed side by side may become less evident. However, by highlighting differences, these variations can be easily discerned. Secondly, when alterations are made to invisible properties, they may go unnoticed in a side-by-side comparison but become apparent through the highlight differences feature. By incorporating visual semantics instead of relying on buttons, the system can preserve this functionality in the archiving system and easily accommodate future extensions.

Let's consider a robot $R$ as the combination of links $L$ and joints $J$. Such that Robot $R := L \times J$. A link can be visually represented as a mesh. A joint describes the relation between exactly two links.

DesignerSL receives a model instance $M$ as an input, we also need to specify a conditional visualization, denoted as $C$. If we intend to visualize the model, we set $C$ to true. Additionally, we define $\mathcal{L}(\text{element})$ as a list, which represents a container that can hold zero or more elements.

Below the types of all syntactic functions are given:

$$\langle \text{DesignerSL} \rangle : M_{\text{DesignerSL}} \rightarrow C \rightarrow \mathcal{L}(R) \quad (5.1)$$

In Equation 5.2 we see that a single robot description, either by file path or full description will result in exactly one robot.

$$\langle \text{Robot} \rangle : M_{\text{Robot}} \rightarrow R \quad (5.2)$$

Each element composes multiple attributes containing keys. Links and joints contain elements. While defining operations, we choose to abstract away from these specific details, and therefore, they are omitted from the operation definitions. However, it is important to note that keys and attributes are integral parts of defining the links and joints as depicted in the semantics of elements Equation 5.3.

$$\langle \text{Element} \rangle : M_{\text{Element}} \rightarrow \mathcal{L}(L) \times \mathcal{L}(J) \quad (5.3)$$

Visualization occurs only upon user request. When an active viewer is present, the robot description, whether it is an actual description or a reference to one, will be evaluated and visualized. Without user interaction, there will be no evaluation or visualization.

A reference to a robot description or its links and joints can be referenced using file paths. The modularity requirement [RQ-3] led to the introduction of the `include` statement in DesignerSL. This statement allows for the inclusion of separate robot component files within the main robot file description. The purpose of the `include` statement is to modularize the robot description by breaking it down into smaller, reusable components.

In addition to the `include` statement for robot components, DesignerSL also provides the `robot` statement. The `robot` statement is used to include a complete robot description, allowing for visualization, highlighting, or comparison of two different robots.

By utilizing the `include` statement for robot components and the `robot` statement for complete robot descriptions, DesignerSL offers a flexible and modular approach to building and visualizing robot models. This enables users to create and manage complex robot configurations while maintaining consistency and reusability of components.

**Definition 5.1 (Load Operation).** Given a DSL file $F$, the Load operation is defined as:
Load(F) → R

Despite the robot and include statement both using a file path F as an input, both return different results, being it either a robot R or link L or joint J. Where R is the Robot description consisting of links L and joints J.

With Definition 5.1 the following semantics are mapping to Listing 6.7.

\[
\text{robot } "(F)" = F_{\text{Load}}^R
\]

With Definition 5.1 the following semantics are mapping to Listing 6.10.

\[
\text{include } "(F)" = F_{\text{Load}}^{L,J}
\]

5.7.1 Compare robot

Definition 5.2 (Display Operation). Given two sets of robot descriptions \( R_1 \) and \( R_2 \), the Display operation is defined as:

\[
\text{Display}(R_1, R_2) \rightarrow (R'_1, R'_2)
\]

where \( R'_1 \) and \( R'_2 \) are visually represented robots corresponding to their description \( R_1 \) and \( R_2 \), respectively, for side-by-side comparison.

Given a DSL model that defines compare (Robot) with (Robot) results in Definition 5.2 seen in Figure 4.9.

\[
\text{compare } \langle R_1 \rangle \text{ with } \langle R_2 \rangle \rightarrow \text{Display}(R_1, R_2)
\]

5.7.2 Highlight difference

Definition 5.3 (Diff Operation). Given two sets of robot descriptions \( R_1 \) and \( R_2 \), the Diff operation is defined as:

\[
\text{Diff}(R_1, R_2) \rightarrow L
\]

Here, \( L \) denotes the set of links that captures distinctions between the two robots, with transparent links denoting identical aspects and thereby emphasizing differences; the design of this visualization is illustrated in Figure 4.9.

Definition 5.4 (Highlight Operation). Given a set of robot descriptions \( R \) and a set of links \( L \) that differ from the initial version, the Highlight operation is defined as:

\[
\text{Highlight}(R, L) \rightarrow R'
\]

where \( R' \) is a Robot where the differences are visually highlighted.

Given a DSL model that defines highlight (Robot) difference (Robot) results in Definition 5.4 seen in Figure 4.9.

\[
\text{highlight } \langle R_1 \rangle \text{ difference } \langle R_2 \rangle \rightarrow \text{Highlight}(R_1, \text{Diff}(R_1, R_2))
\]
Chapter 6

Implementation

6.1 Software design

In this section, the design of the solution is presented using Unified Modeling Language (UML) diagrams. The software design is integral to our language as it extends the language with live graphical feedback. The interaction with the software system enables and disables certain visual semantics, as discussed in more detail in Section 6.2.

6.1.1 Component diagram

The component diagram in Figure 6.1 illustrates the software components and their relationships.

![Component diagram DSL](image)

Figure 6.1: Component diagram DSL.

Each component shown in Figure 6.1 is mapped to its respective responsibility in Table 6.1.
### Component Responsibility

<table>
<thead>
<tr>
<th>Component</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewerJS and UrdfSL</td>
<td>Visualization of robots and changes</td>
</tr>
<tr>
<td>Babylon JS</td>
<td>3D visualization</td>
</tr>
<tr>
<td>SalixJS and Salix</td>
<td>Bi-directional interaction</td>
</tr>
<tr>
<td>TypePal</td>
<td>Checking path existence and navigation</td>
</tr>
<tr>
<td>LanguageServer</td>
<td>Integration with IDE, using events (summarize, document, lenses)</td>
</tr>
<tr>
<td>IDEServices</td>
<td>Opening files in IDE or opening interactive content</td>
</tr>
<tr>
<td>XacroSL</td>
<td>Expression evaluation and resolving include path, using Xacro</td>
</tr>
<tr>
<td>DesignerSL</td>
<td>Providing visual semantics and URDF conversion</td>
</tr>
</tbody>
</table>

**Table 6.1:** Responsibility per component

### 6.1.2 Class diagram

The class diagram in Figure 6.2 showcases the components of the system, including UrdfSL, XacroSL, and DesignerSL.

![Class diagram of languages](image)

**Figure 6.2:** Class diagram of languages

The language can be divided among different sub-languages: UrdfSL for visualization, XacroSL for handling logic, evaluation, and composability, and DesignerSL for composing these functionalities together and extending them with visual semantics. The conversion capability is also present, with UrdfSL converting URDF to DesignerSL and XacroSL converting Xacro to DesignerSL.
Class | Responsibility
---|---
Translator | Translating syntax to an abstract data structure.
Syntax | Parsing raw text and mapping it to a syntactical tree.
View | Handling interaction and visualization.
Controller | Managing user input.
Model | Managing data between the controller and view.
Plugin | Serving as an entry point for the language(s).
Generator | Generating text based on the abstract data structure.
Compiler | Combining syntax, semantics, and generator, and performing writing to disk or calling executable functions.
Checker | Verifies the existence of file paths and collects these as definition using Type-Pal. This enables users to easily navigate using the Integrated Development Environments (IDEs) go-to definition functionality by holding CTRL and click on file path, opening the corresponding file.
Log | Handling risky operations and logging the results.
AbstractXacro | Containing the shared data structure Xacro.

Table 6.2: Class Responsibilities

In Table 6.2, each class is mapped to its corresponding responsibility. The interactions between these classes and methods are further explored in the sequence diagrams presented in Section 6.1.3, which align with the use cases outlined in Chapter 4. This detailed analysis provides insights into the intricacies of how these classes and methods interact with each other.

### 6.1.3 Sequence diagrams

In this section, we delve into the noteworthy interactions observed among various software components. Each sequence diagram start with a initial call that comes from different sources. A source can be the user that clicks on an element in the viewer.

In the context of VS Code, various events are triggered based on user interactions. For example, the summarizer event is invoked when a file is saved. When a user hovers over the active text editor, the documenter event is triggered, providing relevant documentation or information about the code being inspected. Additionally, interacting with code lenses (actionable contextual information interspersed)\(^1\), typically through clicking or selecting options from the context menu in the text editor, triggers the lenses functionality, which offers additional options and actions related to the code. The language server library in Rascal allows for interaction with VS Code events such as the summarizer, documenter, and lenses.

#### 6.1.3.1 Document

The Figure 6.3 implements [RQ-1], by converting URDF to our own DSL and opening the DSL file that was generated by calling edit on the location.

---

\(^1\)https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup
The summarizer event depicted in Figure 6.4 is utilized to initiate a reload of the visualization by updating it with the most recent saved language instance, taking advantage of the unique characteristic of the summarizer event to trigger only upon saving.

By pressing the Ctrl key while clicking on a robot link, the user can effortlessly open either a URDF or DesignerSL file, enabling them to examine the compiled output and verify complex calculations, such as a degrees to radians conversion [RQ-2].
This feature also serves as a seamless navigation mechanism, allowing users to effortlessly transition from the visualization to the corresponding code instance. This intuitive functionality greatly aids in locating and comprehending code segments, particularly when highlighted links indicate changes, facilitating quick navigation to the relevant code section.

Figure 6.5: Sequence diagram: Click on robot link

Figure 6.6: Result click
6.1.3.2 Present

This complex sequence diagram is separated into Figure 6.7 and Figure 6.8. These sequence diagrams implement [RQ-6] and [RQ-7].

Figure 6.7: Sequence diagram: Compile robot properties

The initial run(DesignerSL) seen in Figure 6.8 is a function that is called when the user clicks on a code lens in the text editor.

Figure 6.8: Sequence diagram: View robot properties

The property of the documenter event provides the exact location of the cursor at the text editor. With the gatherLinkLocation function, we created a mapping from link and locations. We lookup the name of the link at the cursor location. Send this link id to our ViewerJS that in turn put focus on this link. We put focus on the link by outlining the mesh with a white color and changing the camera angle to center around the mesh.
When a user clicks on the viewer, a ray is casted, and if it intersects with an element in the scene, the metadata containing a link name is sent to the Salix::App. This link name is then used to lookup the corresponding location in the link location map, which is subsequently utilized to open or edit the location within the VS Code editor.

The compileRobots function is executed upon the reload of the visualization, fulfilling the requirement of comparing robot descriptions, outlined in [RQ-8]. When the DSL is semantically defined as "highlightDifference," the function
setHighlightIds receives the link identifiers, which are then transmitted to Viewerjs. As a result, the corresponding links are displayed as solid while the remaining links become transparent, offering a visual distinction. When the DSL is semantically defined as "compareWith," the function collects link location maps from both robots. This enables focusing while hovering over link identifiers. The corresponding elements of both robots, if they share the same hovered link, will be focused, aiding in comparative analysis.

Figure 6.11: Result compare

To establish the file location for each link, a mapping mechanism is implemented, as illustrated in Figure 6.12. The gaterLinkLocation function is employed to extract the location information of individual links, subsequently stored within the linkLocationMap, as depicted in Figure 6.9. Notably, Robot A denotes the primary robot definition, while Robot B represents a subsequent one. Therefore, the introduction of linkLocationA and linkLocationB proves crucial in distinguishing these two locations. This distinction takes on particular significance when a user engages with Robot A, as the utilization of the linkLocationA mapping ensures accurate access to the corresponding file, even in scenarios where links share identical names between both Robot A and Robot B.
In the context of the `highlightDifference` case within Figure 6.12, the compare function operates by utilizing the output locations derived from the compilation of DesignerSL. This implies that a comparison is conducted between the representations of **Robot A** and **Robot B** in the URDF format. Opting for URDF comparison offers the advantage of consolidating all properties into a single file, streamlining the comparative process. This approach proves more straightforward compared to alternatives like Xacro or DesignerSL, where the incorporation of includes and other intricate evaluations may introduce complexities that differ upon evaluation or are not consistent.
6.1.4 Link origin tracing

In our work, we trace link origin throughout the transformation process from DesignerSL to Xacro and finally to URDF, as depicted in Figure 6.14. This approach draws inspiration from Hutchison et al.’s article on tracing string origin in [52], where they track string origins during transformations. However, our adaptation involves storing origin locations from links (robot components) instead of strings and preserving this information across three transformations instead of one. With this approach bi-directional navigation is created, as elaborated on in this section. Upon hovering over text, the camera dynamically adjusts to center the active link, enhancing visual focus. When clicking on the link, a text editor is triggered, displaying the corresponding element’s description for detailed examination. The tracing of robot link elements is chosen because links contain the visually represented graphical mesh, allowing for bi-directional navigation with the graphical representation.

By re-using the shared abstract data structure of Xacro, the compatibility with Xacro is maintained and the development time of DesignerSL is reduced. The shared data structure, illustrated in Listing 6.1, encapsulates both Xacro and DesignerSL and is located in the Shared folder of the class diagram depicted in Figure 6.2.
data Id_ = id (str id);

data XACRO_Attribute = attribute (Id_ \ type, str val)
| xacro_attribute (Id_ \ type1, Id_ \ type2, str val);

data XACRO_Object = ...
// irrelevant alternatives omitted
| link (Id_ name,
  list [XACRO_Attribute] attributes,
  list [XACRO_Object] elements,
  loc origin=|unknown://///|); // link origin tracing

data XACRO =
  robot__ (list [XACRO_Attribute] attributes
  , list [XACRO_Object] elements
  , loc origin=|unknown://///|); // link origin tracing

Listing 6.1: Shared XACRO datastructure

An important point to highlight in Listing 6.1 is that all objects and attributes are generic, allowing for easy extension of new robot properties semantics such as tolerances. However in order to support bi-directional interaction a more concrete data structure is needed. The link is specifically specified to incorporate storing origin locations.

tuple [XACRO_Object, map [str, XACRO]] translate (Object obj, map [str, XACRO] includes) {
  <xacro_obj, includes> = translate (obj.\type, obj, includes);
  if (xacro_obj.\type.id == "link") {
    name = get (l. attributes, "name");
    return <link (id(name),
      xacro_obj. attributes,
      xacro_obj. elements,
      origin=obj@\loc), // keep track of link location
      includes>;
  }
  return <xacro_obj, includes>;
}

Listing 6.2: DesignerSL::Semantics

By opting for this generic data structure, it accommodate all valid XML formats, enabling the parsing and semantic translation of custom properties. However, a trade-off of this approach is that specific tasks like determining link origins requires iterating through all, as illustrated in Listing 6.4. Furthermore, the semantics of the generic data structure do not verify the validity of elements, accepting all inputs, whereas the UrdfSL semantics, as depicted in Listing A.1, are more specific and restrictive. In future improvements, it would be beneficial to include the validation of properties against the URDF data structure using TypePal to inform that only valid properties are described.

map [str, loc] linkLocationMap = ();

map [str, loc] gatherLinkLocation (map [str, XACRO] xacros) {
  map [str, loc] result = ();
  for (key <- xacros) {
    result += gatherLinkLocation (xacros [key]);
  }
  return result;
}
CHAPTER 6. IMPLEMENTATION

map[str,loc] gatherLinkLocation(XACRO xacro) {
  map[str,loc] result = ();
  for (obj <= xacro.elements) {
    if (link(_, _, _) := obj) { // pattern match on link elements
      result += (obj.name.id:obj.origin); // map link name and location
    }
  }
  return result;
}

Listing 6.3: Gather link location algorithm

In Listing 6.3 the link location map is a mapping that can is used in both directions. We can search on the link name but also look for its location to get the name of the link, a functionality that proves notably convenient.

6.1.5 Focus on hover

Different IDE events are utilized, specifically employing the documenter event. This event has information about where the cursor of the user is located. This active cursor information is used to determine what link is being inspected. In our implementation we even were able to look up the active link through include statements.

bool isCursorLocationInLocation(loc cursor, loc linkLocation) {
  return cursor.begin >= linkLocation.begin && cursor.end <= linkLocation.end;
}

Listing 6.4: Cursor location algorithm

In the link translation from DesignerSL to URDF we keep track of the identifier, the name of the link and its location see Listing 6.4. This goes in two directions, hence bi-directional. The identifier is used to find the corresponding location in the link location mapping and the other direction to find the identifier based on its location, see Section 5.6.2. This location lookup checks if a certain location is in the same file and in between the row and column. If this is the case it will return its identifier Section 5.6.4.

6.1.6 Highlight differences

The functionality of highlighting differences compared to previous version(s). We check for changes via the following algorithm.

list[str] compare(URDF r1, URDF r2) {
  l1 = getAll(r1, "link");
  l2 = getAll(r2, "link");
  return toList(domain(rangeR(
    removeOriginFromLink(l1)
    ,range(removeOriginFromLink(l2))
  )))
}

Listing 6.5: Compare algorithm

In order to understand what is going on in Listing 6.5, we first introduce the powerful built-in Rascal functions for Map.

1. Map:= A list that uses any kind of data as index (called keys), to store a value.
2. rangeR:= Expects a map and expects values that exists in the map. Returns a map of key,value pairs that match the values.
3. range:= Returns a list of values.
4. **domain**:= Returns a list of keys.

Note that we chose to apply the algorithm to the URDF data structure instead of DesignerSL or XacroSL due to the URDF’s single-file nature, simplifying the process.

The URDF data structure, seen in Listing A.1, consists of concrete data types for each URDF property. Each property has elements and attributes, parsed from the URDF robots. In order to compare robot1 (r1) and robot2 (r2), we extract from the elements mapping the “link” and and store these in l1 and l2 respectively, such that we compare links only. With rangeR we exclude the links in robot1 (r1) that do not exists in the robot2 (r2). Next domain is applied to the result, such that only the link identifiers are returned, since the keys are the link names, seen in Listing 6.6.

```java
1 map[str, str] removeOriginFromLink(list [URDF] links) {
2     map[str, str] result = ();
3     for (link <- links) {
4         str uniqueHash = md5Hash(link.attributes + link.elements);
5         result += (getName(link).val: uniqueHash);
6     }
7     return result;
8 }
```

**Listing 6.6:** Remove origin from link implementation

Link origin tracing must be ignored Section 5.6.2, in order to strictly check for value equality, this is different opposed to [15] where origin is actually added and used to ensure file equality. The link is hashed such that it can be compared and stored more efficiently, seen in `removeOriginFromLink` function Listing 6.6.

### 6.2 Language design

DesignerSL, is an extended version of Xacro, which is already a superset of URDF. Making DesignerSL effectively a superset of Xacro and URDF.

![DesignerSL ⊃ Xacro ⊃ URDF](image)

DesignerSL inherits the functionality that Xacro offers. Xacro uses Python to evaluate expressions, allowing the use of commonly used Python functions available within the Python namespace. Furthermore, DesignerSL supports all URDF properties.

Given the well-established nature of Xacro and URDF, the focus will be on introducing visual semantics unique to DesignerSL.

#### 6.2.1 Context free grammar

This section describes the context free grammar of DesignerSL. The F denotes a file path, ( and ) mark the grammar rule that is applied in place. We made textual syntax bold to make a clear resemblance of the Section 6.2.2.

\[
\langle DesignerSL \rangle \rightarrow \\
| \text{compare} \langle Robot \rangle \text{ with } \langle Robot \rangle \\
| \text{highlight} \langle Robot \rangle \text{ difference } \langle Robot \rangle \\
| \langle Robot \rangle \\
\]

\[
\langle Robot \rangle \rightarrow \\
| \text{robot } \{ \langle Attribute \rangle \langle Element \rangle \} \\
| \text{robot } "(F)" \\
\]

(6.1)

(6.2)
Building upon Xacro and URDF, both of which utilize the XML format, DesignerSL emerges as their super set language that improves usability by eliminating the traditional begin and end tags characteristic of XML. Despite this, the underlying semantics remain unchanged. This continuation of XML-like semantics within DesignerSL can be seen in Equation 6.3 and Equation 6.4. By maintaining the inherent expressiveness of the original languages, it is possible to extend it with custom properties, such as tolerances, using the Equation 6.3.

DesignerSL presents a more user-friendly adaptation without compromising on the depth of expression provided by its predecessors.

### 6.2.2 Syntax

Listing 6.7: DesignerSL syntax load robot

Listing 6.8: Benefit of referencing a robot file locations. Describing robots, such as Listing 6.10, in their entirety can obscure the understanding of how they will be visualized.
CHAPTER 6. IMPLEMENTATION

List 6.9: DesignerSL comparing robots side by side view

```
compare
   robot "robot_v1.dsl"
with
   robot "robot_v2.dsl"
```

Listing 6.8: DesignerSL highlighting difference between robots

The also applies to the compare with Listing 6.9. A robot description itself such as Listing 6.10 can be further separated on components based. Each component is described in its own file, using include statements to compose them together.

```
robot {
   name="lbr_iiwa"
   xmlns:xacro="http://www.ros.org/wiki/xacro"
   macro:property:=
      {name="color" value="Blue"}
   include "materials.dsl"
   include "lbr_iiwa_link_0.dsl"
   include "lbr_iiwa_link_1.dsl"
   include "lbr_iiwa_link_2.dsl"
   include "lbr_iiwa_link_3.dsl"
   include "lbr_iiwa_link_4.dsl"
   include "lbr_iiwa_link_5.dsl"
   include "lbr_iiwa_link_6.dsl"
   include "lbr_iiwa_link_7.dsl"
   include "lbr_iiwa_joint_1.dsl"
   include "lbr_iiwa_joint_2.dsl"
   include "lbr_iiwa_joint_3.dsl"
   include "lbr_iiwa_joint_4.dsl"
   include "lbr_iiwa_joint_5.dsl"
   include "lbr_iiwa_joint_6.dsl"
   include "lbr_iiwa_joint_7.dsl"
}
```

Listing 6.10: DesignerSL including components

An example of such include file is given in Listing 6.11. It contains the actual robot properties, mesh file path etc. The same applies to the joint files that describes the relation between 2 links and its behavior. Note that ${}$ encapsulates an expression, which is inherited from the Xacro language. In the Listing 6.11 it is used to expand to the color value that is set in the Listing 6.10 as Blue, the actual color encoding lives in the material.dsl file.
robot{
  link{
    name="lbr_iwa_link_2"
    inertial{
      origin{
        rpy="0 0 0"
        xyz="0.0003 0.059 0.042"
      }
      mass{
        value="52"
      }
      tolerance{
        value="200"
      }
      inertia{
        ixx="0.05"
        ixy="0"
        ixz="0"
        iyy="0.018"
        iyz="0"
        izz="0.044"
      }
    }
    visual{
      origin{
        rpy="0 0 0"
        xyz="0 0 0"
      }
      geometry{
        mesh{
          filename="meshes/link_2.stl"
        }
      }
      material{
        name="${color}"n
      }
    }
    collision{
      origin{
        rpy="0 0 0"
        xyz="0 0 0"
      }
      geometry{
        mesh{
          filename="meshes/link_2.stl"
        }
      }
    }
  }
}

Listing 6.11: lbr_iwa_link_2.dsl file
Chapter 7

Evaluation

For the evaluation of DesignerSL, a preparation deployment is conducted followed by the evaluation of two phases: an alpha test phase and a beta test phase.

7.1 Deployment

In the preliminary evaluation, it was found that the installation process of the language(s) posed challenges for hardware engineers, as it involved installation of multiple programs. The Xacro functionality has been incorporated, and now only Python and the Rascal extension for VS Code are required.

7.2 Alpha

During the alpha testing phase, it was observed that the execution of import and command-line instructions was not intuitive for mechanical engineers with limited programming background. To address this challenge, the decision was made to deploy the language as a VS Code extension. This approach involves creating a ".vsx" executable, simplifying the setup process. By relying on the installation of VS Code, the barriers to tool usage are reduced. As an interim solution, a script has been implemented to execute the command-line instructions with a single click, providing a more user-friendly experience.

Furthermore, the alpha testing revealed that not only links but also joints were being edited. To enhance clarity, the aim is to visualize joints by highlighting both the parent and child links affected by the changes. This visual representation will provide users with a clearer understanding of the impact of their modifications.

In addition, the current tool converts a URDF to DesignerSL with a single click, but the links and joints are not separated into individual files. As a future improvement, the workflow will be enhanced by automatically separating the links and joints into separate files. This modification will simplify maintenance and further streamline the transition from URDF to DesignerSL.

7.3 Beta

Due to timing constraints and the incomplete CAD models, the evaluation of the new tool by mechanical engineers could not be conducted. Currently, the mechanical engineers are actively working on finalizing the CAD models using Creo to ensure the correct URDF model can be exported. This step is crucial in determining the usefulness of the tool. Consequently, the current limitations prevent the mechanical engineers from providing valuable feedback on the tool's usefulness at this time. This is something that can be addressed in future work.

The beta test group conducted tasks derived from the use cases outlined in Analysis chapter. The specific tasks can be found in Appendix B. To calculate the Points per Minute (PPM) metric, each task was accompanied by a question. The time taken for each task was recorded through the logs. Upon completion of the tasks, the participants were asked to fill out the questionnaire detailed in Methodology chapter. The participant group comprised 5 mechatronic engineers and 4 software engineers, with the latter excluded from the test results due to their non-inclusion as intended
users. The results revealed variations among mechatronic engineers in task completion within the 20-minute time frame, as seen in Table 7.1, which displays each task's completion time in minutes, RCS, and PPM results. The evaluation form and comments were also taken into account. The following key points provide an in-depth analysis of these outcomes, emphasizing tasks completed consistently and mechatronic users' perceptions of the tool.

It's worth emphasizing that mechatronic engineers didn't use the office stack that this new tool now supersedes. As a result, they do not utilize the document, presentation, and calculation use cases, and they do not make comparisons with these specific tasks (1, 2, 3, 4, 5). Their workflow involves notion of changes in measurements. Instead of manually comparing textual variations, they now benefit from visual feedback, which enables them to effortlessly observe changes, make comparisons, and identify and highlight differences, which are then used in Matlab to calculate and verify the performance of the system. In Table 7.1, we have listed the mechatronic engineers who took part in the beta evaluation. Each column corresponds to tasks 1 through 6. A task was awarded full points when the result was correct. If a task was only partially correct, it received 0.5 points. Tasks that were not completed or received no response are denoted by a '-' symbol.

1. **Points per Minute (PPM) metric**: To assess the efficiency of the users during task performance, the PPM metric was employed [32]. This calculation depends on the RCS [31] eq. (3.1) formula described in Chapter 3 eq. (3.2). The evaluation session had a total duration of 40 minutes, excluding the introduction and feedback round. The evaluation began with a set of six initial steps designed to familiarize participants with the tool. Subsequently, participants engaged in a task-based approach, which lasted for 20 minutes and encompassed six tasks. Since our target group consists of hardware engineers, we have excluded the performance of software engineers from these results.

<table>
<thead>
<tr>
<th>Role</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Minutes</th>
<th>RT in seconds</th>
<th>RCS</th>
<th>PPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechatronic Engineer</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>900</td>
<td>0.006666667</td>
<td>0.4</td>
</tr>
<tr>
<td>Mechatronic Engineer</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>1200</td>
<td>0.000833333</td>
<td>0.05</td>
</tr>
<tr>
<td>Mechatronic Engineer</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>1200</td>
<td>0.001666667</td>
<td>0.1</td>
</tr>
<tr>
<td>Mechatronic Engineer</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>17</td>
<td>1020</td>
<td>0.005882353</td>
<td>0.352941176</td>
</tr>
<tr>
<td>Mechatronic Engineer</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>1200</td>
<td>0.005</td>
<td>0.3</td>
</tr>
</tbody>
</table>

**Table 7.1: Evaluation Results**

2. **Environment Impact**: The beta test group conducted the tasks in a conference room using their laptops. However, the lack of additional monitors made it difficult for them to use the tool with split-screen view, especially on smaller screens.

3. **User Interface Observations**: The group also made some observations about the user interface of the software:
   - "It looks really cool and visuals are responsive"
   - "The software does not allow changing the robot position after focusing on hover."
   - "Users prefer automatic updates."
   - "Opening the text editor replaces the view in Visual Studio Code, but this issue was resolved by using a web browser."
   - "It is really cramped up to work with on a single screen."

4. **Language Consensus**: The users liked the ability to use mathematical expressions like PI and radians in the tool and the inclusion of files. The language was understandable especially when having a background with using URDF.

5. **Recommendations for Future Evaluations**: It would be more beneficial to conduct the next evaluation at each participant's workplace, as they would be more comfortable with their environment, would likely have access to dual monitors, and could use the web browser to view the robot instead of the integrated view in Visual Studio Code. This would prevent some of the issues participants encountered that hindered their ease of use.
The respondents had positive initial impressions and found various features of the language useful. More detailed responses can be found in Appendix D. They provided suggestions for improvements in live graphical feedback, such as highlighting, GUI controls and use of a secondary monitor. The language itself was commonly perceived as intuitive, although each participant held their unique perspective on what constitutes intuitiveness especially with regard to the visualization. One of the participants expressed a dislike for the transparency of unchanged properties as it hinders comprehensibility, while another participant preferred a more detailed visualization of the changed attributes. This highlights the complexity of designing graphical representations that strike a delicate balance between providing detailed enough information while avoiding clutter that can impede comprehensibility. The ability to split the robot components into multiple files and include them using an include statement along with their respective file paths was appreciated.

The multiple choice response ratings are depicted using a box plot in Figure 7.1, which illustrates the variability in the responses. To enhance readability, each question is correspondingly mapped in Table 7.2.

<table>
<thead>
<tr>
<th>Number</th>
<th>Question Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>The live graphical feedback helps me to understand what robot properties have changed.</td>
</tr>
<tr>
<td>7</td>
<td>The live graphical feedback helps me to understand on what robot properties I am working.</td>
</tr>
<tr>
<td>8</td>
<td>The graphical language helps me to communicate changed robot properties to my colleagues.</td>
</tr>
<tr>
<td>9</td>
<td>How intuitive did you find the graphical language?</td>
</tr>
<tr>
<td>10</td>
<td>How intuitive did you find the live graphical feedback?</td>
</tr>
<tr>
<td>11</td>
<td>Live graphical feedback would improve my ability to understand and work with robot properties?</td>
</tr>
<tr>
<td>12</td>
<td>Does the new workflow improve the communication between colleagues?</td>
</tr>
</tbody>
</table>

Table 7.2: Survey Multiple Choice Questions

Based on the given ratings, we can analyze the reception of the DesignerSL with live graphical feedback among mechatronic engineers for each question:

- **Question 6**: The live graphical feedback helps me to understand what robot properties have changed.
  - Mean rating: 3.0
The engineers’ ratings for this question vary, with a mean rating of 3.0. The standard deviation of 1.22 suggests some level of disagreement or inconsistency among the engineers. Some engineers may find the live graphical feedback helpful in understanding changes to robot properties, while others may not.

• **Question 7: The live graphical feedback helps me to understand on what robot properties I am working.**
  
  – Mean rating: 3.4
  – Standard deviation: 1.14

The engineers, on average, rated the live graphical feedback slightly higher for this question, with a mean rating of 3.4. However, the standard deviation of 1.14 indicates that there is still some variability in their opinions. Some engineers find the live graphical feedback helpful in understanding the robot properties they are working on, while others may not perceive it as useful.

• **Question 8: The graphical language helps me to communicate changed robot properties to my colleagues.**
  
  – Mean rating: 2.6
  – Standard deviation: 0.89

The engineers’ ratings for this question are relatively low, with a mean rating of 2.6. The low mean rating suggests that the graphical language may not be perceived as effective in communicating changed robot properties to colleagues. The lower standard deviation of 0.89 indicates that the engineers’ opinions are more consistent for this question compared to the previous ones.

• **Question 9: How intuitive did you find the graphical language?**
  
  – Mean rating: 3.4
  – Standard deviation: 0.55

The engineers, on average, found the graphical language relatively intuitive, with a mean rating of 3.4. The low standard deviation of 0.55 suggests that there is a higher level of agreement among the engineers regarding the intuitiveness of the graphical language.

• **Question 10: How intuitive did you find the live graphical feedback?**
  
  – Mean rating: 3.2
  – Standard deviation: 0.45

The engineers rated the live graphical feedback as moderately intuitive, with a mean rating of 3.2. The low standard deviation of 0.45 indicates that there is a relatively high level of agreement among the engineers regarding the intuitiveness of the live graphical feedback.

• **Question 11: live graphical feedback would improve my ability to understand and work with robot properties?**
  
  – Mean rating: 3.2
  – Standard deviation: 1.10

The engineers’ opinions are divided for this question, with a mean rating of 3.2 and a higher standard deviation of 1.10. Some engineers believe that live graphical feedback would improve their ability to understand and work with robot properties, while others may not share the same perception.

• **Question 12: Does the new workflow improve the communication between colleagues?**
  
  – Mean rating: 2.8
  – Standard deviation: 0.84
The engineers’ ratings indicate that the new workflow does not significantly improve communication between colleagues, with a mean rating of 2.8. The standard deviation of 0.84 suggests a moderate level of agreement among the engineers regarding the lack of improvement in communication.

Overall, based on the given ratings, it appears that the mechatronic engineers have mixed opinions about the DesignerSL with live graphical feedback. While they generally find the live graphical feedback and graphical language to be intuitive, their perception of the effectiveness in communicating changed robot properties and improving communication between colleagues is relatively low. There is also some variability in their opinions, as indicated by the standard deviations in the ratings.

The lower scores we observed in colleagues’ visual communication of changes (questions 3 and 7) could be attributed to the task sequence. Specifically, task 6, which involves comparing and highlighting use cases from Figure 4.4, was only completed by three participants. As not all participants reached this task, as evident in Table 7.1 where column six and row participant ‘-‘ indicate task 6 was not completed, their feedback on this aspect was based on tasks that did not specifically cover this use case. Consequently, these responses may not offer a complete understanding of how the use cases compare and highlight were received by the mechatronic engineers. The limited exposure to these use cases is also reflected in the open questions of the questionnaire, detailed in Appendix D.
Chapter 8

Results

This chapter addresses the research questions introduced in Section 1.5 and presents the results of the requirements outlined in Chapter 5. It provides a detailed description of how each requirement has been tackled and resolved, highlighting the implemented solutions.

8.1 Research answers

1) What domain-specific programming languages with graphical feedback do exist?

Some of the more recent DSL with graphical feedback are explored in Section 4.1. These languages cover various domains, including robotics, game development, behavior specification, education, and UML modeling. These languages provide specialized tools and approaches to address specific needs and challenges within their respective domains. They leverage visual representations, simulations, and interactive feedback to enhance usability, collaboration, reduce debugging effort, and improve comprehension and learning. The combination of graphical and textual views in some languages offers flexibility, allowing users to choose the most suitable representation for their tasks. Overall, these domain-specific graphical languages demonstrate the value of graphical feedback, bi-directional editing and seamless integration between graphical and textual representations in various application domains.

1.a) What are the industry examples of domain-specific languages that incorporate graphical feedback?

In Section 4.1 DSLs are explored that incorporate graphical feedback some are used in the industry. These serve as examples in how and what technologies could be used at Philips to achieve similar results. One of these visualization technologies is RViz. This is utilized for 3D visualization in robot programming, B-Motion Studio and its plugin for Rodin are used for visualizing DSL models at ASML, Rascal was used to create a DSL that serves as a framework for designing game rules in game development which involves graphical feedback, Sirius and Xtext are utilized for engineering hybrid graphical-textual languages for a FSM and AToM³ is used for creating DSLs with both textual and graphical views for graph transformations. These tools provide capabilities such as visual feedback, model visualization, model validation, and transformation, enabling the development and effective use of domain-specific graphical languages in their respective domains.

1.b) What technologies and tools are used at Philips IGT by the hardware engineers?

At Philips IGT the hardware engineers, utilize a variety of technologies and tools to perform their tasks, a more in depth analysis can be found in Chapter 4. Mechanical engineers employ tools such as CREO for working with 3D CAD files, Excel for recording and performing calculations on system properties, Word for documenting changes compared to the original CAD model, and PowerPoint for presentation purposes. Mechatronic engineers, on the other hand, rely on Matlab and Simulink for complex calculations and simulations. Specifically, mechatronic engineers compare measurements stored in a Matlab instance with changed properties recorded in CAD and Excel to ensure that
systems conform to specifications, and they conduct assessments critical for maintaining system performance and reliability. The proposed workflow suggests a transition to a unified language and tool, a DSL with live graphical feedback, replacing the existing office tool-stack. This CAD would integrate tasks like presenting changes, documentation, and mathematical evaluations to describe properties, streamlining the complexity associated with multiple tools and sources of truth while also enhancing version control and archivability.

2) How can graphical domain-specific language technologies be applied at Philips IGT given their existing technologies?

Considering Philips’ current technology stack, Rascal or Xtext would be suitable options for a LWB. A pragmatic approach would involve creating a dedicated DSL tool that seamlessly integrates with the widely used office tools like Excel, Word, and PowerPoint, which are used by hardware engineers. This envisioned DSL tool would combine documentation, visualization, evaluation, and modification of robot properties, streamlining workflows suitable with the existing Git archiving. Engineers would gain the ability to manipulate URDF files, interact with robot models, adjust properties, express mathematical concepts, compare specifications, and highlight differences. Enhanced with a user-friendly graphical interface and real-time feedback. By incorporating these capabilities within familiar IDEs like Eclipse IDE or VS Code, Philips would tap into graphical DSL technologies to boost engineers’ efficiency, accuracy, and overall performance in managing robot properties and specifications. This approach aligns with Philips’ existing technologies.

2.a) What technologies and tools are used at Philips IGT?

At Philips model-based engineering technologies are used to improve various software workflows, tools used include the LWB such as Xtext and recently Rascal. The workflow of hardware engineers at Philips IGT utilize various technologies and tools to support their work as described in Chapter 4. These include CREO a 3D CAD modelling tool for designing and documenting system properties, Excel for performing calculations based on those properties, Word for documenting changes made compared to the original CAD model, and PowerPoint for presenting and communicating the modifications. Other tools such as Matlab and Simulink enables more complex calculations and simulations to verify the systems integrity.

2.b) How to visualize editing changes with the given technologies?

The visualization of editing changes can be achieved through the use of CAD software exporting to URDF. This format consists of the necessary properties required for visualizing robots. When hardware engineers make changes in the new tool it provides a visual representation of the modified design in the viewer of this new tool, it does this by making the modified link solid and the unmodified components transparent, this design can be found in Chapter 5. This allows engineers to visually inspect and verify the changes they have made. Additionally, the tool can be utilized to interact with the edited properties of the robot model. This visualization capability helps engineers assess the impact of their changes and ensure that the system adheres to the desired specifications. The workflow including the new tool and its use cases are described in Chapter 4.

2.c) When does a live editing change propagate to the visual presentation?

Live editing changes propagate to the visual presentation immediate (each second with “polling” enabled) when the user saves the file using the summarize event. The immediate propagation of changes allows engineers to have immediate visual feedback and assess the impact of their edits on the robot model as soon as the user saves the file, enabling them to make informed decisions and ensure the desired behavior of the system, however since polling is required, which is a costly process with regard to system resources, it is an optional feature that must be enabled by the user, otherwise changes will only propagate when the user explicitly requests a the visualization to reload, a more detailed description is given in Chapter 5.
3) How can a domain-specific language be intuitive to use for mechanic and mechatronic engineers?

To make a DSL intuitive for mechanic and mechatronic engineers, it should leverage their existing knowledge and tools. By providing 3D graphical representation of the robot, the DSL adheres to the richness quality, allowing engineers to visualize their designs and better understand their implications. Interactive immediate feedback follows the liveness quality, reducing cognitive load and enabling engineers to make informed decisions directly while editing. Supporting modularity and reusability embodies the composability quality, allowing engineers to combine and reuse language features in a structured manner. Additionally, by offering a simplified syntax that aligns with their domain ensures that the DSL is easily understood and used by engineers. Seamless integration with existing workflows and tools further enhances the usability of the DSL, making it a natural replacement in their current workflow. By considering these factors, the DSL can be designed to meet the specific needs and requirements of hardware engineers, improving their workflow and collaboration.

3.a) What is the current workflow of the hardware engineers?

The current workflow, for more detail in Section 4.2, of hardware engineers at Philips involves utilizing various tools and formats for their work. The workflow begins with the engineers using tools such as CAD, Excel, Word, and PowerPoint for tasks such as measuring system properties, tracking changes, performing calculations, and documenting their work. They interact with CAD models and Excel spreadsheets to manage and analyze the properties of the system components. Any changes made to the CAD model are documented in Word and presented using PowerPoint. This workflow demonstrates the engineers' reliance on multiple tools and formats to perform their tasks, leading to potential complexity and a need for more streamlined and integrated solutions.

3.b) What is the desired workflow for the hardware engineers using a domain-specific language?

The desired workflow for hardware engineers, as outlined in Section 4.3, revolves around a unified tool that replaces the traditional office stack. This tool is designed to handle various tasks such as documentation, presentation, expression, and evaluation of mathematical formulas. Detailed features and requirements of the tool can be found in Chapter 5. To facilitate the integration of CAD files, Blender is used as an intermediate step to convert them into URDF format. The tool enables engineers to import URDF files and work with the domain-specific language called DesignerSL. Changes made in DesignerSL can be visualized using customized graphical representations, allowing users to define how they want to highlight differences between versions. The tool also ensures seamless integration with the existing archiving solution used at Philips, enabling easy storage and retrieval of project files. Bi-directional navigation between the graphical and textual representations further enhances user experience and understanding of changed robot properties. Overall, this desired workflow promotes effective communication among different disciplines by providing a unified language supported by live graphical feedback.

3.c) Does the domain-specific language fit the hardware engineers?

Based on the evaluation in Chapter 7, DesignerSL shows promise in meeting the needs of hardware engineers, particularly those familiar with working with URDF. However, the evaluation also identified several areas for improvement. One notable aspect is the graphical representation of changes, although not reviewed fully by each participant, it received mixed feedback. The diverse feedback underscores the subjectivity of user preferences. Despite these identified areas for improvement, the evaluation highlights the potential value of DesignerSL for hardware engineers and emphasizes the importance of considering needs and preferences when implementing graphical representations in domain-specific languages.

The transition from URDF to DesignerSL seems to be appreciated, given the feature to convert URDF to DesignerSL with a single click. The ability to use mathematical expressions in the language is also a plus, as it aligns with the mathematical orientation of engineers. During the evaluation, it became evident that participants really liked the automatic update and even would prefer it to be on by default. Another liked feature is the separation of links and joints into individual files, as this improves readability and maintenance significantly.
The use of VS Code as a platform for using the language is beneficial in reducing barriers for users with limited programming background as it provides useful syntactic services. However, the user interface appears to have some limitations, especially with screen real estate on smaller or single monitors.

The evaluation for mechatronic engineers was successfully conducted. However, due to incomplete CAD models and time constraints, the planned evaluation specifically targeting mechanical engineers could not be carried out as originally intended. The incomplete CAD models posed a limitation, preventing a comprehensive assessment of the tool’s suitability and effectiveness in meeting the specific needs of mechanical engineers. Therefore, it is recommended that this aspect of the evaluation be considered as future work to be undertaken, allowing for a more thorough examination of the tool’s application and relevance to the mechanical engineers.

### 8.2 Requirements Fulfillment

<table>
<thead>
<tr>
<th>ID</th>
<th>Solution</th>
<th>Link</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RQ-1]</td>
<td>The tool converts URDF to DesignerSL, that provides multiple views depending on specification. The tool leverages Xacro that is able to evaluate mathematical expressions and even use Python expressions as it internally uses Python.</td>
<td>Section 6.1.3.1</td>
<td>Yes</td>
</tr>
<tr>
<td>[RQ-2]</td>
<td>The tool leverages Xacro and extends DesignerSL with the include statement. DesignerSL highlights changed properties when edited and enables focus on hover over text editor, using the cursor as a pointer for presenting changed links.</td>
<td>Section 5.7</td>
<td>Yes</td>
</tr>
<tr>
<td>[RQ-3]</td>
<td>The tool can describe properties by extending URDF. Creating DesignerSL with added hardware properties, such as tolerances, masses, centers of gravity (COG), and moments of inertia (MOI).</td>
<td>Section 5.5.4</td>
<td>Yes</td>
</tr>
<tr>
<td>[RQ-4]</td>
<td>The tool can only shows that a link containing (invisible) properties has changed</td>
<td>Section 5.5.6</td>
<td>No</td>
</tr>
<tr>
<td>[RQ-5]</td>
<td>The tool focuses on textual hovering over text effectively visualizes selected elements</td>
<td>Section 6.1.5</td>
<td>Yes</td>
</tr>
<tr>
<td>[RQ-6]</td>
<td>The tool offers two views one side by side robot comparison and a highlighted view of changed links</td>
<td>Section 6.1.3.3</td>
<td>Yes</td>
</tr>
<tr>
<td>[RQ-7]</td>
<td>The tool is able to convert URDF to DesignerSL only, it cannot export from CAD directly</td>
<td>Section 5.5.5</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 8.1: Fulfillment of Functional Requirements
CHAPTER 8. RESULTS

<table>
<thead>
<tr>
<th>ID</th>
<th>Solution</th>
<th>Link</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NFRQ-1]</td>
<td>Bi-directional navigation has been implemented, enabling users to intuitively navigate around the graphical and textual link representation of a robot. The textual representation uses the URDF concepts but without the XML tags, further improving intuitiveness. The performance of the tool is dependent on system resources. The tool provides a button to enable or disable immediate feedback, which can be taxing in terms of performance. The user can also manually update the visualization. DesignerSL is built on top of XacroSL and UrdfSL, allowing for custom syntax and semantics, as demonstrated with visual semantics. The tool gives warnings when including files that do not exist using Typepal, preventing undesired behavior and mistakes. The viewer runs independently from the compiler, meaning that even when the compilation fails, the viewer shows the latest successful robot representation. The tool is able to convert URDF to DesignerSL and requires third-party software to convert CAD to URDF. Documentation is provided, including a step-by-step introduction and a showcase of use cases in the form of tasks. The tool is compatible with Windows 10 and above, MacOS, and Linux. It only depends on the host having Python. The tool uses design patterns such as Model-View-Controller (MVC) for the UI and minimizes code duplication using shared data structures. The code separates concerns using multiple files.</td>
<td>Section 5.6.2</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-2]</td>
<td></td>
<td>Section 5.6.3</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-3]</td>
<td></td>
<td>Section 5.7</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-4]</td>
<td></td>
<td>Section 6.1.1</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-5]</td>
<td></td>
<td>Section 5.5.5</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-6]</td>
<td></td>
<td>Appendix B</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-7]</td>
<td></td>
<td>Section 6.2</td>
<td>Yes</td>
</tr>
<tr>
<td>[NFRQ-8]</td>
<td></td>
<td>Section 6.1.2</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8.2: Fulfillment of Non-Functional Requirements

8.3 Discussion

In this section we reflect on the obtained results, their relation to the literature review and their significance in the context to related work. We aim to evaluate the meaning, importance, and relevance of our findings and assess their implications for the field of hardware engineers and DSL development.

The literature review and related work offer valuable insights and perspectives into the design of programming languages and tools, emphasizing live feedback, richness, composability, and usability considerations. While the literature review provides a comprehensive overview of these concepts across diverse domains, the related work delves into specific techniques and instances that have contributed to the design and implementation of DSLs featuring live graphical feedback. Collectively, these resources enrich our understanding of best practices in programming language and tool design. The incorporation of liveness, richness, and composability qualities were positively received by the mechatronic engineers.

In contrast to the existing literature reviewed in Chapter 4 and Chapter 2, which predominantly concentrates on visually representing properties of robots which can be visually represented such as positions of components and movements, our research places a strong emphasis on highlighting changes that cannot be visually represented, such as inertia and mass. During our evaluation, we observed varied responses from mechatronic engineers regarding the novel realization of our approach. Some find the use of white outlining and transparency of the robot components intuitive, while others do not share the same sentiment.
8.3.1 Implications

Our research has several implications for the development and use of DSLs and the consideration of programming qualities.

8.3.1.1 Improved Usability

The positive feedback from mechatronic users regarding liveness, richness, and composability demonstrates that incorporating these qualities into a DSL can significantly improve its usability and user satisfaction.

8.3.1.2 Pre-existing Knowledge

Leveraging the widely adopted URDF standard tapped into the existing knowledge base of end-users, simplifying their adoption and adaptation to the DSL. This underscores the notion that harnessing established standards can expedite the learning curve for new DSLs. Furthermore, through the incorporation of Xacro, we achieved composability within the language. Moreover, the development of DesignerSL expanded its capabilities with a visual language that empowers hardware engineers to precisely define how the robot is visually represented. This extension encompasses all known keywords and components while eliminating the cumbersome XML syntax, enhancing the user experience.

8.3.1.3 Conclusion

In conclusion, our study has demonstrated that integrating liveness, richness, and composability into a DSL tailored for hardware engineers at Philips, featuring enhancements like bi-directional navigation, live graphical feedback, and the inclusion of robot components, can significantly enhance usability and effectiveness. Furthermore, it has underscored the challenge of visualizing invisible hardware properties, which often hinges on personal preferences and perceptions. These findings make meaningful contributions to the ongoing development of DSLs within this domain, emphasizing the critical role of user feedback in designing intuitive graphical feedback languages.

8.3.2 Limitations

One limitation of this research is the limited test group, comprised exclusively of mechatronic engineers with some programming experience, while mechanical engineers with little to no programming background were not included in the evaluation. This omission affects our ability to assess the language's intuitiveness for this specific group of engineers.
Chapter 9

Conclusion

This thesis presents the development of a DSL with immediate graphical feedback. It introduces a technique called link-origin tracing, which equips the tool with bi-directional navigation capabilities. This enables users to navigate between text and graphics, offering instant visual feedback upon interacting with the text. The language is designed to support hardware engineers at Philips, offering a means to communicate and visualize changes in hardware properties. The primary goal is to enhance collaboration and understanding of modifications to robot hardware properties.

The developed DSL, DesignerSL, implemented using Rascal, has successfully addressed the needs of mechatronic engineers and has the potential to improve the workflow of hardware engineers at Philips IGT. By enabling the description of robot properties and providing live graphical feedback of changes, DesignerSL facilitates effective collaboration and understanding among users.

The evaluation of DesignerSL has yielded positive feedback regarding key features such as URDF to DesignerSL conversion, support for mathematical expressions, interactive visual feedback, and automatic updates. These features have proven valuable in enhancing the usability and efficiency of the tool. However, the evaluation also identified some limitations, including screen real estate issues and the need for more comprehensive visual representation of changes.

To further validate the effectiveness of DesignerSL for all use cases, future evaluations should involve mechanical engineers when complete CAD models are available. Conducting evaluations in users’ own workspaces would provide valuable insights into the practical application of the tool. Additionally, exploring alternative visualization options and expanding upon the visual semantics could allow for more personalized and flexible ways of visualizing changes.

Despite the incomplete evaluation for mechanical engineers, the overall findings underscore the potential value of DesignerSL and emphasize the importance of considering user preferences when implementing graphical representations in DSLs. The anticipated usage of the tool by 15 mechatronic and 30 mechanical engineers at Philips further validates its practical application and potential impact in real-world engineering scenarios.

The development of DesignerSL serves as a valuable case study, providing insights into the practical application of immediate, bi-directional visual feedback within an engineering context. The lessons learned from this project can inspire future research endeavors and innovations, contributing to the continued evolution of domain-specific languages with live graphical feedback in the CPS. The creation of initial prototypes within our research framework has further strengthened our confidence in the feasibility and potential of the employed technologies.

Overall, the development and evaluation of DesignerSL have demonstrated the effectiveness and potential value of integrating live graphical feedback in DSLs for hardware engineers. This work contributes to the advancement of engineering collaboration, usability, and documentation management within multi-disciplinary teams.
Bibliography


Appendix A

UrdfSL Abstract Data Structure

```python
data URDF =
  robot (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot
  material (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot
  color (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // material
  link (map [str, URDFValue] attributes, map [str, list [URDF]] elements, loc origin = [unknown ://]) // robot
  visual (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // link
  origin (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // visual or joint
  geometry (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // visual, collision
  cylinder (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // geometry
  sphere (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // geometry
  box (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // geometry
  mesh (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // geometry
  inertial (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // inertial
  collision (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // inertial
  joint (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // link
  dynamics (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot
  limit (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  parent (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  child (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  axis (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  transmission (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot
  actuator (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // transmission
  sensor (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot
  camera (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // sensor
  plugin (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // robot, link, joint, or sensor
  counterbalance (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  tolerance (map [str, URDFValue] attributes, map [str, list [URDF]] elements) // joint
  custom property
  xacroproperty (map [str, URDFValue] attributes, map [str, list [URDF]] elements)
```

Listing A.1: UrdfSL Abstract Data Structure
Appendix B

Evaluation tasks

This chapter contains the manual of the tool [NFRQ-6] and the test cases that form the basis of the task based approach.
DesignerSL evaluation

After the initial 6 steps we have 4 test cases that is either successful or failure based on the result. Please submit the results to this (email: twan.bolwerk@philips.com) mail to.

subject: Evaluation DesignerSL

body:

1)
2)
3)
4)
5)
6)

+ appendix 7

(optional) **install_vscode.bat** in case VScode is not installed
(optional) **install_rascal.bat** in case Rascal is not installed
double left click **open_designersl.bat** to start step 1 (press yes in dialogs) (might need to run twice in case a trust dialog appears)

Step 1
(1) left click on Run in Rascal Terminal

Step 2

file explorer in the left most side panel 1st icon open `example\urdf\model_free_base.urdf`

(2) left click on Convert to DSL

Step 3
Step 2 creates and opens a new file `example\urdf\model_free_base.urdf.dsl`

(3) Left click on the **split editor right icon** on the right upper corner

**Step 4**

Step 3 creates and opens a split screen view of `example\urdf\model_free_base.urdf.dsl`

(4) Left click on **View robot** to view this robot description

**Step 5**
Step 4 opens a viewer in the left screen **localhost:9050** you can also choose to open **localhost:9050** in a webbrowser as it may be more userfriendly to use in the upcoming testcases.

(5..11) left click on the **sliders** in the viewer to fold out the robot.

(12) left click on **auto** button in the viewer to start immediate update.

**Step 6**

Step 5 should unfold the robot and enabled automatic update of the viewer whenever the robot description is changed and saved (Ctrl-c).

(13) **Right click** on robot link shows the name of that link, double left click opens it in the text editor. While holding CTRL pressed it opens up URDF in order to inspect output.

(14) **Hover** over the link section in text editor will mark the link in the visualization with white outlining. (seen in the visualization)
Test cases

For each test case Step 6 has the correct setup, with regard to opened panels, file tree (left), viewer (center) and text editor (right).

Test case: Present

file explorer in the left most side panel 1st icon open example\DesignerSL\robot_v1.dsl
left click view robot
click button auto in view
hover over the text editor until the correct link lbr_iiwa_link_2 is outlined.
right click on link lbr_iiwa_link_2 in viewer to ensure this is link lbr_iiwa_link_2 double left click on link lbr_iiwa_link_2 in viewer without ctrl pressed (this opens up the text editor and a section automatically gets selected in text editor)

1. then ctrl-c ctrl-v in order to paste this section in email twan.bolwerk@philips.com

Test case: Document

file explorer in the left most side panel 1st icon open example\DesignerSL\robot_v1.dsl
hold-ctrl-and-press-left-mouse-button on "lbr_iiwa_link_2.dsl" in text editor to open file containing link
{ name = "lbr_iiwa_link_2" ...} add the following code snip-it below name of link 2

```
limit:={
    effort="0"
    lower="-1.5707963267948966"
    upper="1.5707963267948966"
    velocity="1.3089969389957472"
}
```

make sure viewer is visable and auto is on then, ctrl-s to save the file

2. What do you see in the viewer, did anything change? (append your own observations as a result in email twan.bolwerk@philips.com)

left click on link lbr_iiwa_link_2 in viewer with ctrl pressed (this opens up the text editor and a section automatically gets selected in text editor)

3. then ctrl-c ctrl-v in order to paste this section in email twan.bolwerk@philips.com

Test case: Calculate

file explorer in the left most side panel 1st icon open example\DesignerSL\robot_v1.dsl
hold-ctrl-and-press-left-mouse-button on "lbr_iiwa_link_2.dsl" in text editor to open file containing link
{ name = "lbr_iiwa_link_2" ...}
replace the following code snip-it with the previous limit from document below name of link 2
limit:=
    lower="${radians(-90)}"
    upper="${radians(90)}"
    effort="pi"
    velocity="${radians(75)}"

left click on link lbr_iwa_link_2 in viewer with ctrl pressed (this opens up the text editor and a section automatically gets selected in text editor)

4. then ctrl-c ctrl-v in order to paste this section in email twan.bolwerk@philips.com

Test case: Compare

Compare with

file explorer in the left most side panel 1st icon open example\DesignerSL\compare.dsl
left click view robot
right click in viewer this opens a text balloon with its name. Resize the viewer if the text is partially hidden (like Step 6).

5. What link or joint was different in compare.dsl? write result in email twan.bolwerk@philips.com

Highlight

In case you would hover in text over an transparent element / link in view, the focus of the camera will change however the element does not get the white outline.

file explorer in the left most side panel 1st icon open example\DesignerSL\highlight.dsl
left click view robot
right click in viewer this opens a text balloon with its name. Resize the viewer if the text is partially hidden (like Step 6).

6. What link or joint was different in highlight.dsl? write result in email twan.bolwerk@philips.com

DesignerSL evaluation form

https://forms.office.com/Pages/ResponsePage.aspx?id=LXpAGnV2F02GkrOsKFMG5K1vOpiQJRtDvLu64hjDiMNuOTJGqlg5MVdQ0xPVFZPMhMWUdFRzIGQy4u

Log file

7. email twan.bolwerk@philips.com log.txt as appendix file designer-lsp\target\classes\build\log.txt
### Appendix C

#### Questionnaire

<table>
<thead>
<tr>
<th>No.</th>
<th>Question</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What were your initial impressions?</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>What existing features of the language would be most useful to you?</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>What improvements or additions would you like to see in the graphical feedback?</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Was the language itself intuitive, what did you find less intuitive about the language?</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Role (mechanical engineer, mechatronic engineer)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>To what extent does the graphical feedback help you understand what robot properties have changed? (1 - Very Poor, 5 - Excellent)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>How effective is the graphical feedback in helping you understand what robot properties you are working? (1 - Very Ineffective, 5 - Very Effective)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>How much does the graphical language aid you in communicating changes in robot properties to your colleagues? (1 - Not at All, 5 - Significantly)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>How intuitive did you find the textual language? (1 - Not Intuitive at All, 5 - Extremely Intuitive)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>How intuitive did you find the graphical feedback? (1 - Not Intuitive at All, 5 - Extremely Intuitive)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>To what extent do you believe that graphical feedback would improve your ability to understand and work with robot properties? (1 - No Improvement, 5 - Significant Improvement)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Does the new workflow improve the communication between colleagues? (1 - Not at All, 5 - Significantly)</td>
<td></td>
</tr>
</tbody>
</table>

*Table C.1: User feedback and ratings questionnaire*
Appendix D

Questionnaire responses questions

Based on the responses provided, here is a categorization on role and the feedback for each question:

1. **Question: What were your initial impressions?**
   - Mechatronic engineer: Nice to have the visualization.
   - Mechatronic engineer: Confused by the many different files, a simple graphical overview would have helped.
   - Mechatronic engineer: It looks really cool and visuals are responsive, but it’s cramped up on a single screen.
   - Mechatronic engineer: Good improvement, very helpful visualizations.
   - Mechatronic engineer: Nice to have visual feedback on textual input.

2. **Question: What existing features of the language would be most useful to you?**
   - Mechatronic engineer: Including files, using PI.
   - Mechatronic engineer: The ability to directly visualize changes in the URDF; use of equations in Xacro.
   - Mechatronic engineer: It is very useful for understanding an existing URDF (by clicking on the different sections of text).
   - Mechatronic engineer: The correctness of the URDF; checking if the joints are correctly defined.
   - Mechatronic engineer: I may not have reached the point to experience where the "language" comes in and therefore I do not know how I can benefit from it - I've just experienced it as a visualization tool next for the URDF.

3. **Question: What improvements or additions would you like to see in the graphical feedback?**
   - Mechatronic engineer: Robot position should not reset when selecting another tab, auto-update should be on by default, improve transparency of links.
   - Mechatronic engineer: Graphical overview to explain the relationship between files and visualization.
   - Mechatronic engineer: Easily observe differences at a more detailed level of the link or joint visually.
   - Mechatronic engineer: The visualization was much better in a web browser than in Visual Code (the black links were very confusing with the black background).
   - Mechatronic engineer: I found the constant switching between tabs annoying and then having to reload the graphical view because it got back to its default positions.

4. **Question: Was the language itself intuitive, what did you find less intuitive about the language?**
   - Mechatronic engineer: No specific comment provided.
   - Mechatronic engineer: The language was quite understandable, but found the many different files and their relationship less intuitive.
• Mechatronic engineer: The language and its benefits were not experienced, so the intuition is unclear.
• Mechatronic engineer: Did not spend a lot of time looking into the language.
• Mechatronic engineer: Got the hang of the language quickly, so it was considered intuitive.