DOI: 10.15439/2024F3931 ISSN 2300-5963 ACSIS, Vol. 39

Using a Textual DSL With Live Graphical Feedback to Improve the CPS' Design Workflow of Hardware Engineers

Twan Bolwerk
Philips in Best, The Netherlands & Radboud University in Nijmegen, The Netherlands

Marco Alonso
Philips in Best, The Netherlands

Mathijs Schuts
Philips in Best, The Netherlands & Radboud University in Nijmegen, The Netherlands

***Abstract*—Cyber-Physical Systems are designed and developed using multi-disciplinary teams. Handovers from one discipline to another often occur using text documents written in natural language, which can be imprecise, ambiguous, and lead to errors. To improve this situation, we created a textual Domain Specific Language with live graphical feedback to enhance the handover between mechanical and mechatronic engineers working on medical robots at Philips IGT. The Domain Specific Language formalizes the system description and provides immediate live graphical feedback to prevent mistakes from being made, such as editing the wrong physical parts and by visualizing the differences of two versions of a system. In addition, our approach leverages multiple industry standards and it enables bi-directional navigation between languages.**

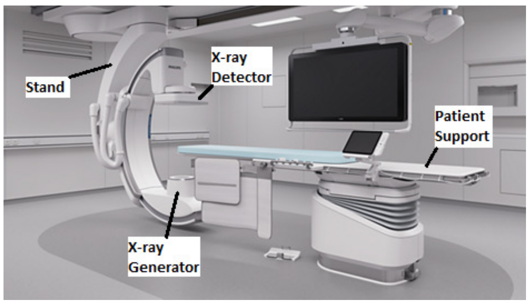


Fig. 1. Interventional X-ray system

I. Introduction

A CYBER-PHYSICAL System (CPS) [4] is a complex system composed of both hardware and software components. These systems are designed and developed with multi-disciplinary teams. Often, a CPS consists of moving parts such as in the case of robots, cars, airplanes, etc. For the hardware component development, mechanical and mechatronics engineers are involved. The mechanical engineer creates 3D models of the physical components using a Computer Aided Design (CAD) tool [27] and performs measurements, i.e., on weight and tolerances. The mechatronics engineer makes these physical systems move by creating control solutions using tools such as Matlab and Simulink [21]. Both disciplines use their own specialized software tools. Currently, the workflow involves manually written documents that are used to handover designs and measurement information from the mechanical engineer to the mechatronics engineer. Due to the informal nature of these documents, they can be imprecise, ambiguous, and prone to errors. Additionally, changes between document versions may go unnoticed.

At Philips IGT, we create interventional X-ray systems such as the Azurion system in Figure 1, which are used for minimally invasive procedures. These large medical robots feature motorized moving parts that can be operated using joysticks [24].

In this paper, we present an improved workflow for the development of these CPSs. After a mechanical component has been modelled using a CAD tool, the mechanical engineer can export the 3D model in the Unified Robot Description Format (URDF) [17], which is based on eXtensible Markup Language (XML) [7]. However, one downside of URDF is that weights and tolerances are not included, and cannot be added. Additionally, it lacks an import mechanism for reusing components across similar robots. These limitations can be addressed using the XML macro language (Xacro) [2], which requires manual editing to add weights and tolerances. Both formats are in XML, which is not a user-friendly way of editing. Furthermore, we place these files in a version controlled system, but merging XML-based files is challenging.

The first author of this paper created a textual Domain Specific Language (DSL) [11] called Geometry Specific Language (GSL) or GeometrySL. The language extends Xacro but is not based on XML. Instances of this language are placed in a version controlled system and handed over from mechanical engineers to mechatronics engineers. By using formal GSL files instead of informal documents, we reduce the likelihood of errors due to handovers. The GSL provides immediate live graphical feedback when editing the textual instance, showing which part is being edited within the robot's 3D model. It also has facilities for graphically comparing two versions of a robot, highlighting parts that are different in a 3D model. Additionally, it supports bi-directional navigation from a graphical part to the corresponding DSL fragments and vice versa. This allows seamless navigation between graphical views, URDF instances, Xacro instances, GSL instances, and back again using shortcuts.

To the best of our knowledge, the novelty of this research lies in the creation of the GSL, a DSL that defines how differences between robot representations are visualized. By leveraging multiple languages, including industry standards like Xacro and URDF, this approach enables bi-directional navigation and offers a unique method for visualizing differences in robot descriptions.

The paper is organized as follows. In Section II, we provide an overview of related work. We describe the current and proposed workflows in more detail in Section III. The GSL, Xacro and URDF languages are presented in Section IV. Section V describes the design of the tool. The resulting tool is shown in Section VI. Discussion is in Section VII. In this section, we also discuss how our work is related to the work of others. And we conclude our paper in Section VIII.

II. Related Work

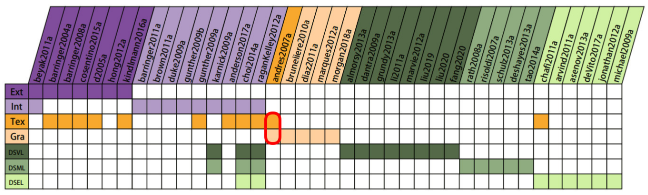


Fig. 2. DSL categories from [26]

Shen et al. [26] categorized recent studies on DSL based on three concerns: concrete syntax, abstract syntax and semantics. They analyzed the parsing and mapping strategies of these studies to classify them into categories such as external/internal, textual/graphical, modeling/visualizing/embedding. This study aims to address research gaps in DSL categorization. The vertical axis of Figure 2 lists literature references while horizontal axis list the following categories:

- **External (Ext):** Standalone languages with their own syntax and grammar, distinct from any host language, providing specific solutions within a particular domain.
- **Internal (Int):** Embedded within an existing general-purpose programming language, leveraging the host language's syntax and features to implement domain specific constructs.
- **Textual (Tex):** Use text-based syntax, similar to traditional programming languages, designed for domain experts familiar with coding or scripting.
- **Graphical (Gra):** Employ visual representations such as diagrams and flowcharts to define domain specific constructs, useful for users preferring visual over textual representation.
- **Domain Specific Visual Language (DSVL):** A subset of graphical DSLs using specialized visual notations to represent domain concepts, facilitating understanding and communication among stakeholders.
- **Domain Specific Modeling Language (DSML):** Focus on creating models specific to a domain using specialized syntax and semantics, providing tools for simulation, validation, and code generation.
- **Domain Specific Embedded Language (DSEL):** A type of internal DSL embedding domain specific constructs within a host language, integrating domain specific functionality directly into general-purpose language code.

The figure features a red box highlighting the scarcity of research focused on Gra and Tex DSL combinations. While enabling live graphical feedback reduces cognitive load, improves collaboration and communication in engineering projects. This study bridges computing science and behavioral research domains to improve DSL design with live graphical feedback for better communication and collaboration in complex engineering projects.

To visualize property changes in language instances, we need to consider their visibility. Munzner's book [19] recommends automatic highlighting with varied colors, shapes, or positions can emphasize distinctions between properties. The "pop-out" effect in Munzner's book helps users spot differences quickly without focused attention.

In [14], Joshua Horowitz et al. define programming qualities. The focus is on immediate feedback (liveness), domain specific editing (richness) and composability. Composability enables the inclusion of external libraries or components, separating responsibilities over multiple sources. Programming tasks often require using multiple composed tools, so effects should be visible with minimal distraction and effort. Liveness and richness often fail to retain their composability according to Horowitz et al. conclusion. They identified a trend where interfaces lacking composability are standalone applications that offer limited utility in practice. This research explores the intersection of liveness, richness and composability by adhering to these qualities using familiarity with 3D graphical tools for hardware-engineers' workflow improvement and tool intuitiveness enhancement.

Van Rozen et al. [34] recognized the need for program execution observation in traditional programming, which requires re-execution of updated source code from the beginning. This process is time-consuming and distracting when valuable states are lost or difficult to reproduce. To address these challenges, they propose a more fluid and live experience in programming using Textual Model Diff (TMDiff) [31]. Tmdiff uses two key techniques: origin tracking (tracing semantic model elements back to their defining source code) and text differencing (identifying corresponding model elements when aligned names have the same origin). The deltas found by TMDiff are converted into run-time edit operations, which can be applied atomically using rmpatch. Custom state migrations extend rmpatch to avoid information loss or invalid run-time states. Events like user interactions and changes in source code are recorded for undo functionality, persistent application state and back-in-time debugging. They evaluated existing methods (Xtext and EMFCompare) using Eclipse Modeling Framework (EMF) and found TMDiff's scope-handling ability more flexible. Their goal is to minimize distractions and preserve intermediate visual state for a smoother programming experience.

In [10], Cooper et al. present requirements and challenges to integrate graphical editors using Sirius within EMF. They note that while Sirius allows creation of custom modeling editors, it has a steep learning curve. To address this limitation, they propose five requirements for a hybrid textual-graphical workbench: syntax-aware editing, scoping and referencing, rename refactoring, error/warning marker display and accessibility to the textual model. These requirements aim to improve productivity, reduce errors and facilitate collaboration by providing seamless integration between graphical and textual models. Their proposed solution aligns with a case study on hybrid modelling workbenches.

A DSML for UML profiles was created by combining Papyrus and Xtext using EMF. One unique feature is shared storage base for both textual and graphical views, reducing synchronization efforts. The tool is tested their by four scenarios (Create1, Modify1, Create2, Modify2) with experienced developers in the UML language. Results showed that creating elements and setting properties were faster in textual notation while constructing state machines was quicker in the graphical view. Renaming operations were faster in textual mode due to regex search and replace efficiency. The hybrid solution was superior in efficiency, doubling the speed in mentioned scenarios. This demonstrates potential of combined graphical-textual approach for DSMLs [1].

In game development, rapid adjustments of rules are made using tools like Machinations[1]. Van Rozen et al. created a DSL called Micro-Machinations (MM) using the tool Rascal [30] to balance games. MM allows for direct visualization model editing, shortening feedback loops and reducing design iteration times by improving flexibility and adaptability. The Rascal Language WorkBench (LWB) with SPIN model checker [5] is used for analyzing MM, providing an IDE that reads textual MM and displays a visual model interactively. The MM Lib is embedded in the game itself to tackle interoperability, traceability and debugging challenges [29]. An immediate feedback loop greatly improves multi-disciplinary team collaboration in gaming domain similar to engineering domains.

Perez et al. [22] created DSVL using both textual and graphical views with AToM tool[2]. They followed meta-model centric approach where EBNF grammar was generated based on the meta-model, allowing decision to be made later whether to use graphical or textual syntax. Another issue is that produced Abstract Syntax Tree (AST) from parsing is not formally defined causing problems in integration with multi-view DSL proposed by them. They noticed that it is more natural to describe equations in a textual notation.

A more extensive version of this work can be found in [6].

[1]https://machinations.io/

[2]http://atom3.cs.mcgill.ca/

III. Workflow

In this section, we describe the current workflow of hardware engineers at Philips IGT and we propose a new improved workflow. The workflow involves two actors: mechanical engineers and mechatronics engineers.

A. Current workflow

The workflow process follows a waterfall approach, where mechanical engineers measure the system in the factory. These measurements are then communicated via various Office tools to mechatronics engineers. The tools used by these actors are depicted in Figure 3 and illustrate their interactions. The interactions between the tools can occur either automatically, with data being stored or transferred automatically between tools, or manually inputted by the user.

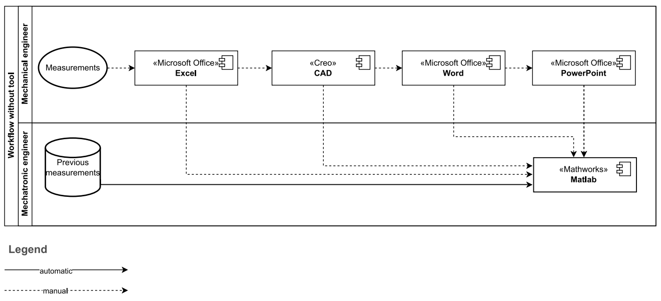


Fig. 3. Tools and actors.

We describe the actions per actor:

- Mechanical engineer. The mechanical engineer creates the hardware design of the system using the CREO[3] tool for opening and editing 3D CAD files. They also measure the real-world properties of the system, perform calculations in Excel and document any changes compared to the original CAD model using Word. These updates are presented via PowerPoint [25].
- Mechatronic engineer. The mechatronic engineer manually compares previous measurements saved as Matlab instances with the changed properties recorded in CAD and Excel to determine if the system still meets the specifications. For example, they ensure that the system adheres to the predefined tolerances for each link[4]. These assessments are crucial for maintaining the system's performance and reliability, and are calculated using complex calculations and simulations in Matlab and Simulink [36].

In sum, the handover from the mechanical engineer to the mechatronics engineer currently relies on informal document-based communication. This can result in misunderstandings and potential errors due to missed changes or ambiguities.

[3]https://www.ptc.com/en/products/creo/

[4]A link in robotics refers to a rigid component that forms part of a robot's structure, connecting to other links through joints.

B. Proposed workflow

In the proposed workflow, all Microsoft Office tools are replaced by Domain Specific Modelling (DSM) [12] using a single Geometry Specification Language (GSL) with live graphical feedback. It replaces the Microsoft Office tools by a unified language capable of presenting changes, serving as documentation and evaluating mathematical expressions that can be used to describe and calculate properties.

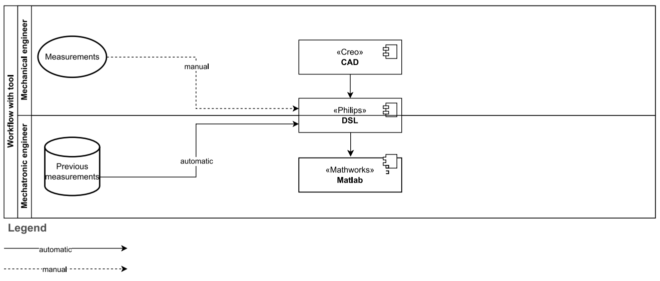


Fig. 4. Actors and tools workflow with DSL.

By eliminating the reliance on Microsoft Office tools and therefore multiple sources of truth, the complexity associated with using these tools is reduced. This can be observed in Figure 4 compared to Figure 3.

A CAD file can be converted into URDF which we are able to translate to our own DSL. The mechanical engineer can input measurements manually in an expressive manner similar to an Excel spreadsheet but now in the GSL. Meanwhile, the mechatronic engineer can compare previous measurements stored in the single source of truth DSL which is stored within Philips' version control system. Using a textual DSL makes, i.e., merging branches easy.

C. Use cases

In this section, we describe two main use cases. One involves presenting the textual DSL in a graphical manner and another focuses on visualizing differences between two versions of the DSL in a graphical representation.

1) Present: The first use case demonstrates effective communication of measurement changes between mechanical 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 physical link of the robot being hovered over. This visualization feature enhances communication by clearly highlighting and displaying the changes made to each physical link of the robot. It provides a seamless and intuitive way for stakeholders to understand and interpret updated measurements for every physical link in the robot.

The mechanical engineer can efficiently navigate, inspect, and present the robot's hardware properties using the tool's textual and graphical representations. This use case showcases how engineers can leverage the tool's features to communicate and demonstrate the robot's hardware-properties.

2) Highlight: The highlight use case offers an efficient solution for mechatronic engineers to automatically identify and highlight changed physical links between robot or component versions. This internal scenario eliminates the need for manual input of values into Matlab, as it leverages automatic highlighting and live graphical feedback. Engineers can quickly and accurately pinpoint variations between link versions using this feature, reducing reliance on manual comparisons and potential errors.

IV. Languages

In this section, we introduce the Language WorkBench (LWB) used to implement our tool as needed in order to understand this paper. We describe the URDF, Xacro and GSL languages. Finally, we provide example instances of these three languages.

A. Language workbench

A LWB provide concepts and mechanisms to define language syntax, semantics, and code generation for a language. They facilitate model-driven engineering by allowing 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, maintainable code, and closer collaboration between different stakeholders. LWBs bridge the gap between generalized programming languages and the unique requirements of specialized domains [8].

There are many language workbenches (LWBs) available. Both Eclipse Modeling Framework (EMF) and Rascal[5] are used at Philips; however, Rascal provides support for Visual Studio Code (VS Code). Given that VS Code[6] is the preferred tool at Philips and aligns with our previous experience using Rascal, we decided to use it for our tool.

Rascal [16] is a type-safe programming language featuring immutable data, built-in pattern matching, search, and relational calculus. It is a functional and procedural programming language with Java-like syntax. We introduce the language in this section as needed to understand the code fragments presented in this paper. The code snippets in Listing 1 are reused from [23].

```
loc l =
  |file:///Users/kees/.bashrc|(100,20,<2,0>,<2,20>)

data Boolean = true() | false() | and(Boolean lhs,
  Boolean rhs);
// extending Boolean with another constructor
data Boolean = or(Boolean lhs, Boolean rhs);
data Statement = \if(Expression c, Statement tt,
  Statement ff);

for (int i <- [1 .. 5]) println(i);

str w = "world"
println("Hello, <w>!");    // prints "Hello, world!"
```

Listing 1. Rascal code fragments

[5]https://www.rascal-mpl.org
[6]https://code.visualstudio.com

Rascal has numerical types such as **`int`** and booleans, represented by **`bool`**. It supports polymorphic **`list`**s, **`map`**s for collections and **`loc`** for location constants.

Rascal also provides the following built-in functions for working with **`map`**s:

1) Map := A list that uses any kind of data as index (called keys), to store a value.
2) rangeR := Expects a map and 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.

On Line 2 there is a constant **`loc`** that points to a file with the file scheme and selects the part on line 2 between the left margin and the 20^{th} column. Locations are used to refer to files, store information extracted from files and help in referring back to source locations.

In Rascal, Algebraic Data Types (ADTs) can be user-defined with their constructor functions. The fragment on Lines 4-9 shows a declaration of an ADT for the representation of Boolean expressions using three constructors. The next line extends the same ADT by adding an alternative to the existing declaration. Reserved keywords are not permitted as names of algebraic constructors; hence, **`if`** is escaped with `\if` when used as the name of an algebraic constructor.

Control structures such as **`for`** can be used to iterate over a value, while **`str`** literals in Rascal are not delimited by line endings.

We utilized the following two Rascal libraries: Salix and TypePal. Salix is a library that facilitates the development of web-based GUI programs; it runs user code on the server side instead of client-side execution. The library employs 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. TypePal is a typechecking and validator library for Rascal, designed to analyze and enforce type constraints, ensuring correct usage of types and detecting potential type-related errors in DSL instances. TypePal provides static type checking capabilities and can be used to improve the reliability and correctness of language instances [28].

B. URDF

Utilizing URDF offers numerous advantages, notably its capability to express a wide range of hardware properties, with the exception of tolerances. Moreover, URDF facilitates seamless conversion from formats like CAD files, streamlining the integration of engineering disciplines for our use case. Furthermore, the compatibility of URDF with visualization tools such as RViz[7] underscores its ability in conveying essential information using a 3D graphical representation.

The downside of URDF is its inability to scale and its cumbersome XML format, which is difficult to maintain and causes issues in the archiving system when merging changes. As the model becomes more complex, the lack of reusable components results in larger file sizes.

[7]http://wiki.ros.org/rviz

C. Xacro

To address the issue of large URDF files, especially for complex 3D robot models like interventional X-ray systems, a solution based on modularization and composition using the Xacro language can be employed. Xacro provides a way to create modular and reusable components, making robot descriptions more manageable and organized.

Although Xacro simplifies URDF composition and introduces expression evaluation, it still relies on an XML format that is neither user-friendly nor intuitive, making it difficult to version control and prone to merging issues.

D. GeometrySL

To overcome the previously described limitations, we introduce GeometrySL (GSL), which extends XacroSL. GeometrySL encapsulates Xacro and converts it to a more user-friendly syntax, making it easier to archive and merge changes. It enabled the creation of a more intuitive syntax that supports custom property definitions (including tolerances) and provides enhanced visualization options. The aim is to offer a more intuitive, flexible language for 3D robot modeling, addressing the drawbacks associated with traditional XML-based URDF descriptions.

Creating GeometrySL adds flexibility in defining and designing custom syntax, making it more intuitive to use and easier to extend with new semantics and introduce custom properties, enhancing its expressiveness and adaptability for different use cases.

One of these new use cases is the integration of visual semantics. This allows mechanical engineers to describe desired appearances of views using a language that engineers can understand. This approach enables better collaboration by providing live graphical feedback.

E. Example instances

Next, we will describe three features using language instances–custom features, modular includes, and highlighting differences between two robot versions. Due to confidentiality concerns, we use Franka's Panda robot[8] as an example instead of our interventional X-ray system.

```
1  robot {
2   link {
3     name="lbr_iiwa_link_0"
4     inertial {
5       origin := {
6         rpy="0 0 0"
7         xyz="-0.1 0 0.07"
8       }
9       mass := { value="0.2" }
10      tolerance := { value="200" }
11      inertia := {
12        ixx="0.05"
13        ixy="1"
14        ixz="0"
15        iyy="0.06"
16        iyz="0"
17        izz="0.03"
18      }
```

[8]https://support.franka.de/docs/franka_ros.html

```
    }
    visual {
      origin := {
        rpy="0 0 0"
        xyz="0.2 0.1 0"
      }
      geometry {
        mesh := { filename="meshes/link_0.stl" }
      }
    }
    collision {
      origin := {
        rpy="0 0 0"
        xyz="0 0 0"
      }
      geometry {
        mesh := { filename="meshes/link_0.stl" }
      }
    }
  }
}
```

Listing 2. GeometrySL instance example

Listing 2 shows an example of GeometrySL for the Panda robot, demonstrating how to define a link. A link has a name and various features such as inertial, visual, and collision properties. Inside the inertial feature, we added a custom property called "tolerance". This property can be exported to Xacro but not to URDF. The instance also references STereo Lithography (STL) files, which is a format used to describe 3D objects using triangles.

```
robot {
  name="lbr_iiwa"
  xmlns:xacro="http://www.ros.org/wiki/xacro"
  include "lbr_iiwa_link_0.gsl"
  ...
  include "lbr_iiwa_link_7.gsl"

  include "lbr_iiwa_joint_1.gsl"
  ...
  include "lbr_iiwa_joint_7.gsl"
}
```

Listing 3. GeometrySL instance example for modular includes

In Listing 3, an instance of GeometrySL is shown that describes the Panda robot. It includes other instances of GeometrySL to describe links and joints of the Panda robot. A link is represented as a mesh, while a joint defines the relation between exactly two joints.

```
<?xml version="1.0"?>
<robot name ="lbr_iiwa"
  xmlns:xacro="http://www.ros.org/wiki/xacro" >
<xacro:property name ="color" value ="Green"/>
<xacro:property name ="half" value ="0.1"/>
<xacro:include filename
  ="lbr_iiwa_link_0.gsl.xacro"/>
...
<xacro:include filename
  ="lbr_iiwa_link_7.gsl.xacro"/>

<xacro:include filename
  ="lbr_iiwa_joint_1.gsl.xacro"/>
...
<xacro:include filename
  ="lbr_iiwa_joint_7.gsl.xacro"/>
</robot>
```

Listing 4. Xacro instance example for modular includes

Listing 4 shows an example of how to represent the same information from Listing 3 in the Xacro language. It uses XML syntax instead.

```
<?xml version="1.0" ?>
<robot name="lbr_iiwa">
  <link name="lbr_iiwa_link_0">
    <inertial>
      <origin rpy="0 0 0" xyz="-0.1 0 0.07"/>
      <mass value="0.2"/>
      <inertia ixx="0.05" ixy="1" ixz="0"
        iyy="0.06" iyz="0" izz="0.03"/>
    </inertial>
    <visual>
      <origin rpy="0 0 0" xyz="0.2 0.1 0"/>
      <geometry>
        <mesh filename="meshes/link_0.stl"/>
      </geometry>
      <material name="Grey"/>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="meshes/link_0.stl"/>
      </geometry>
    </collision>
  </link>
  ...
  <link name="lbr_iiwa_link_7">
    ...
  </link>
  <joint name="lbr_iiwa_joint_1" type="revolute">
    <parent link="lbr_iiwa_link_0"/>
    <child link="lbr_iiwa_link_1"/>
    <origin rpy="0 0 0" xyz="0 0 0.1575"/>
    <axis xyz="1 0 1"/>
    <limit effort="300" lower="-2.96705972839"
      upper="2.96705972839" velocity="10"/>
    <dynamics damping="0.5"/>
  </joint>
  ...
  <joint name="lbr_iiwa_joint_7" type="revolute">
    ...
  </joint>
</robot>
```

Listing 5. URDF instance example for expanded Xacro instance

Listing 5 shows an example of how to represent the same information as Listing 4 using URDF syntax instead. It includes expanded information from Listing 2, but does not include the "tolerance" property that was present in GeometrySL and Xacro.

```
highlight
  robot "robot_v1.gsl"
difference
  robot "robot_v2.gsl"
```

Listing 6. GeometrySL instance example for highlighting differences for two versions of a robot

Listing 6 demonstrates a language instance of GeometrySL that visually shows the differences between two versions of the robot.

V. Design

In this section, we describe how we realized the use cases presented in Section III.

A. Conversion tool

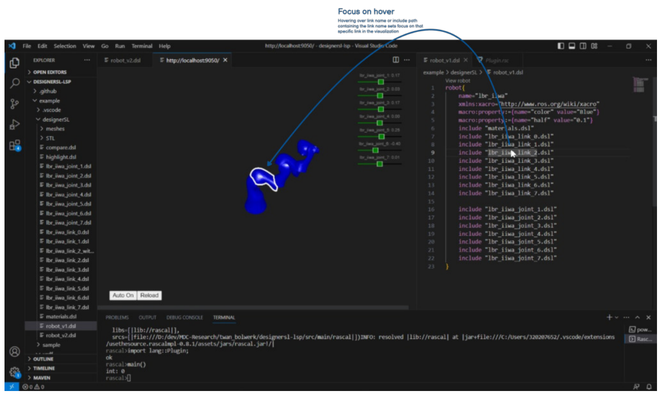
One of the benefits of using URDF, a widely adopted and standardized format is that other tools often have conversion features. For instance, the Blender tool[9] can be utilized as an intermediary that enables conversion from CAD export files into URDF. Blender is open-source, has free licensing, and 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.

B. VS Code URDF viewer

The VS Code URDF viewer[10] was utilized as the starting point, which incorporates BabylonJS[11], a JavaScript 3D graphics library. This library can load STL files and assemble them using URDF. However, it lacks certain features such as comparing changed properties, highlighting differences, bi-directional navigation and side-by-side views, and maintaining state after changes. Additionally, it requires the use of URDF, which has a non-user-friendly syntax and leads to large file sizes when designing systems like interventional X-ray systems. Nonetheless, despite these limitations, the VS Code URDF viewer 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 6. This visual representation establishes a direct connection between the textual and visual physical links of the robot, 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. Furthermore, the "reload" and "auto" buttons play a vital role in updating the view with the latest changes made in the GSL textual editor, whether through manual input or automatic updates.



Hovering over link name or include path containing the link name sets focus on that specific link in the visualization

```
robot{
    name="lbr_iiwa"
    xmlns:xacro="http://www.ros.org/wiki/xacro"
    macro:property:={name="color" value="Blue"}
    macro:property:={name="half" value="0.1"}
    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"
}
```

```
  libs=[|lib://rascal|],
  srcs=[|file:///D:/dev/MDC-Research/twan_bolwerk/designersl-lsp/src/main/rascal|])INFO: resolved |lib://rascal| at |jar+file:///C:/Users/320207652/.vscode/extensions
/usethesource.rascalmpl-0.8.1/assets/jars/rascal.jar!/|
rascal>import lang::Plugin;
ok
rascal>main()
int: 0
rascal>
```

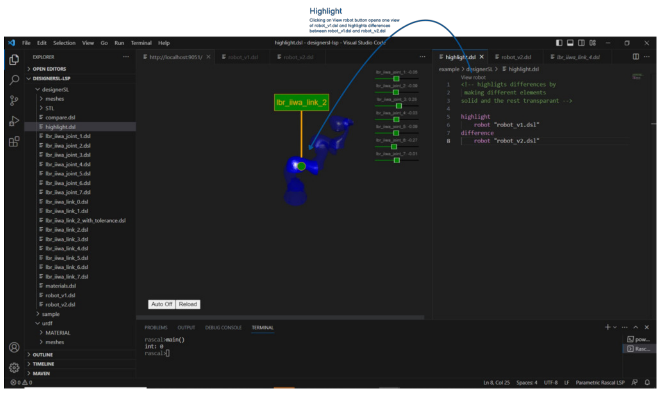
Fig. 5. Focus on hover link

[9]https://www.blender.org/
[10]https://github.com/javahacks/vscode-urdf-viewer
[11]https://www.babylonjs.com/

C. Show differences between robot versions

The next use case is showing differences between two robot versions. When the GSL instances are defined as the corresponding physical links of the robot are displayed as solid while the remaining links become transparent, offering a visual distinction.

Listing 6 provides a GSL instance to compare two robot versions. In Figure 6, we show the same GSL instance on the right and on the link we have the graphical view with differences.



Clicking on View robot button opens one view of robot_v1.dsl and highlights differences between robot_v1.dsl and robot_v2.dsl

```
<!-- highligts differences by
 making different elements
solid and the rest transparant -->

highlight
    robot "robot_v1.dsl"
difference
    robot "robot_v2.dsl"
```

```
rascal>main()
int: 0
rascal>
```

Fig. 6. Highlight robot

D. Present robot

The first use case we are going to describe is how we present the robot. The conversion tool has been executed and the VS Code URDF viewer is running. Salix is utilized to present the robot by 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 labeled "auto" is provided, as shown in Figure 5, which allows users to enable or disable the polling feature. This functionality offers a option to conserve system resources. With polling enabled the automatic updates are enabled, 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 webserver thread. Rather than refreshing the complete web-view, the web-server can update 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 then 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 then sends a HTTP message to our viewer. The viewer is equipped with a so-called listener (hooks) which update the visualization while preserving state.

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

TABLE I
RESPONSIBILITY PER COMPONENT

This significantly reduces distraction of reloading graphics and therefore improving usability.

The "polling" mechanism is used to focus on an element that is hovered over by the mouse. This improves the user's 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 as shown in Figure 5.

In this approach, a custom Xacro parser has been developed to convert Xacro code into GeometrySL. The shell "exec" function from Rascal is leveraged to call the Xacro compiler.

E. Tool's components diagram

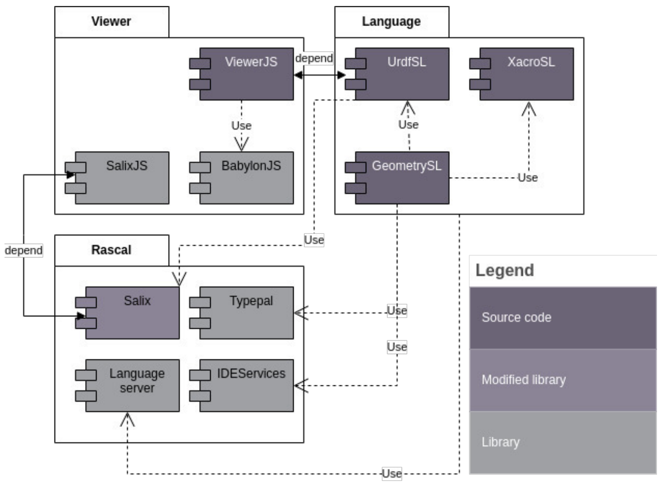


Fig. 7. Components

The component diagram in Figure 7 illustrates the software components and their relationships.

Each component shown in Figure 7 is mapped to its respective responsibility in Table I.

F. Link origin tracing

In our work, we trace link origin throughout the transformation process from GeometrySL to Xacro and finally to URDF, and vice versa. This approach draws inspiration from Inostroza et al. [15], where they track string origins during transformations. However, our adaptation involves storing origin locations from physical links of the robot instead of strings and preserving this information across three transformations instead of one. This feature enables seamless navigation between the URDF source, GSL instance, and graphical representation of the robot model. As demonstrated previously, hovering over textual elements triggers the graphical element to be highlighted with a white outline. When holding down the Ctrl-key and clicking on a visual element, the user is directed to the corresponding section in the URDF source code. Conversely, clicking on a visual element without holding down Ctrl-key directs the user to the corresponding section in the GSL instance. With this approach bi-directional navigation is created. 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 GeometrySL is reduced. The shared data structure, illustrated in Listing 7, encapsulates both Xacro and GeometrySL and is located in the Shared folder of the class diagram.

```
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 7. Shared XACRO datastructure

An important point to highlight in Listing 7 is that all objects and attributes are generic, allowing for easy extension of new robot property semantics such as tolerances. However, in order to support bi-directional navigation 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 8. GeometrySL::Semantics

By opting for this generic data structure, it accommodates 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 10. 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 the Appendix' Listing 13, are more specific and restrictive.

```
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;
}

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

Listing 9. Gather link location algorithm

In Listing 9 the link location map is a relation that can be 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.

G. Focus on hover

Different Integrated Development Environment (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 10. Cursor location algorithm

In the link translation from GeometrySL to URDF we keep track of the identifier, the name of the link and its location see Listing 10. This goes in two directions, hence bi-directional. The identifier is used to find the corresponding location in the link location mapping and in the other direction to find the identifier based on its location. 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.

H. Highlight differences

The functionality of highlighting differences compared to previous version(s). Link origin tracing must be ignored, in order to strictly check for value equality, this is different opposed to [31] where origin is actually added and used to ensure file equality. The links are hashed such that they can be compared and stored more efficiently, see *removeOriginFromLink* function Listing 11.

```
map[str,str] removeOriginFromLink(list[URDF] links){
  map[str,str] result = ();
  for (link <- links){
    str uniqueHash = md5Hash(link.attributes + link.elements);
    result += (getName(link).val: uniqueHash);
  }
  return result;
}
```

Listing 11. Remove origin from link implementation

The URDF data structure 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 store these in l1 and l2 respectively, such that we compare links only. Recall the explanation of the build-in `map` functions in Section IV-A. With rangeR we exclude the links in robot1 (r1) that do not exists in the robot2 (r2). Next the domain function is applied to the result, such that only the link identifiers are returned, since the keys are the link names, see Listing 11.

```
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 12. Compare algorithm

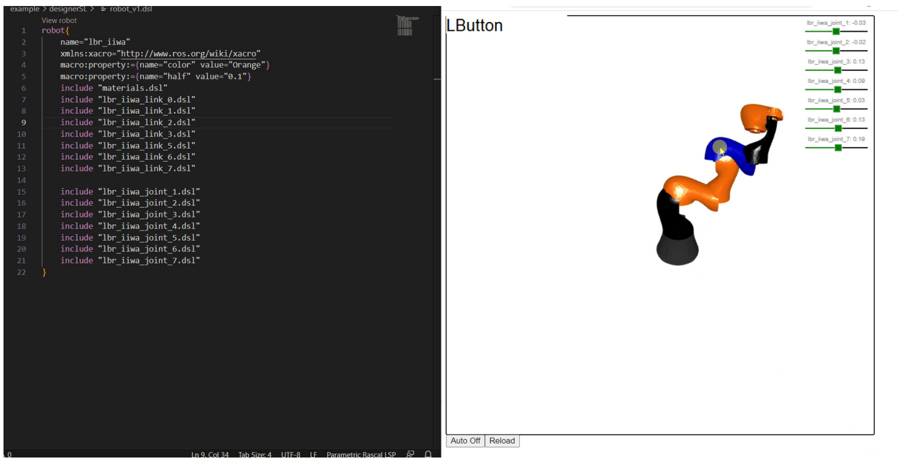
Note that we chose to apply the algorithm to the URDF data structure instead of GeometrySL or XacroSL due to the URDF's single-file nature, simplifying the process. We check for changes with the algorithm in Listing 12.

VI. Results

To see the final product in action, it is best to watch the demonstration videos. Due to confidentiality, we use the Panda robot instead of the Philips IGT interventional X-ray system. Figure 8 about bi-directional navigation[12] & figure 9 hover and highlight differences[13] are videos of our tool in use.

[12]https://www.youtube.com/watch?v=n71kg1OKVus&ab_channel=fedcsis3391

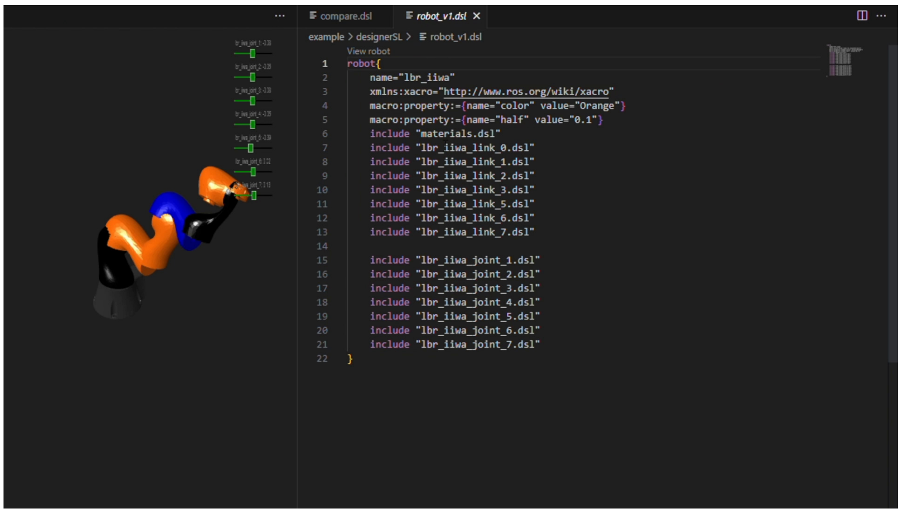
[13]https://www.youtube.com/watch?v=o_bJ8NsEODQ&ab_channel=fedcsis3391


example > designerSL > robot_v1.dsl
View robot
LButton
Auto Off
Reload
Ln 9, Col 34 Tab Size: 4 UTF-8 LF Parametric Rascal LSP

```
robot{
    name="lbr_iiwa"
    xmlns:xacro="http://www.ros.org/wiki/xacro"
    macro:property:={name="color" value="Orange"}
    macro:property:={name="half" value="0.1"}
    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_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"
}
```

Fig. 8. video of the GeometrySL tool bi-directional navigation



```
robot{
    name="lbr_iiwa"
    xmlns:xacro="http://www.ros.org/wiki/xacro"
    macro:property:={name="color" value="Orange"}
    macro:property:={name="half" value="0.1"}
    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_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"
}
```

Fig. 9. video of the GeometrySL tool hover and highlight

VII. Discussion

In contrast to the existing literature reviewed in Section II, 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.

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 GeometrySL 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.

A. *Bi-directional navigation*

The benefits of bi-directional interaction, as outlined by Witte et al. [35], is that it reduces cognitive load and improves the overall user experience. By integrating the view and text editor through a bi-directional approach, our tool 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 [33], 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 [14], 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.

B. Immediate feedback

Earlier research [9] and [13] have indicated that immediate feedback significantly improves debugging. More recent studies also successfully applied immediate feedback in their DSL [32], [20], [3], [18]. Additionally, incorporating immediate feedback ensures that the DSL adheres to the liveness quality, improving the programming experience.

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 the visualization serves a similar purpose as source code errors and warnings, and most languages perform these checks upon saving to achieve it. Additionally, this approach helps minimize the performance impact of running resource-intensive processes with each change.

Our language exhibits key qualities such as liveness, richness, and composability, as discussed in Horowitz et al.'s work on live programming [14].

Like mentioned in Munzner's book [19], we aim to create a solution that addresses the invisibility of property changes and enhances user experience by implementing these highlighting strategies while considering potential conflicts with user-defined materials, overlapping color use, or making components transparent for highlighting. We opted against animations due to their potential to cause change blindness, distracting users from subtle property changes. Instead, we focus on highlighting the changed links to clearly differentiate changes, enhancing user experience by ensuring modifications are easily perceived and understood within the graphical representation. The highlighting differences aim to make changes more distinguishable and improve the user experience.

TMDiff [31] utilizes an algorithm that relies on source location to detect changes, similar to existing tools like Linux's built-in diff and git diff. These tools typically compare lines of code, analyze differences on a line-by-line basis, and categorize changes as modified, deleted, or added.

In contrast, our solution specifically addresses the unique characteristics of URDF and GLS files, which represent robotic systems with distinct links and joints identified by their names. We introduce a novel approach by hashing the links and joints including their unique identifiers while excluding their source location into a hashmap. This allows us to efficiently determine whether links or joints already exist in the system, enabling effective management of modifications and comparisons.

Moreover, our visualization tool offers enhanced capabilities compared to TMDiff. While both systems can highlight changes per link and between linked entities (in the case of joints), our solution goes further by seamlessly accommodating rearrangements of links and joints. This means that even if the structure of the system is altered, our tool can accurately compare the old and modified versions, providing a more robust and flexible comparison mechanism for this use case.

In conclusion, our study has demonstrated that integrating liveness, richness, and composability into a DSL tailored for hardware engineers at Philips IGT, 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.

VIII. Concluding Remarks

We improved the hardware development workflow at Philips IGT by creating a textual Domain Specific Language (DSL) called GeometrySL or GSL. This DSL was used to formalize handovers from mechanical engineers to mechatronic engineers, preventing mistakes during the exchange process.

The novel approach of leveraging industry-standards URDF and Xacro, alongside techniques such as origin tracing and the LWB Rascal has resulted in a live graphical feedback on differences between Cyber-Physical System (CPS) versions. It enables bi-directional navigation among the graphical representation, URDF and the GSL itself, enhancing the efficiency and usability of the language.

The development of GeometrySL serves as a valuable case study that demonstrates the practical application of immediate, bi-directional visual feedback within an engineering context. The lessons learned from this project can inspire future research efforts and innovations in domain specific languages with live graphical feedback for CPS.

This research mainly focuses on graphically representing textual differences of 3D robot models. In our evaluation of the tool we had diverse feedback. The main challenge we faced was how to visualize changes made to invisible properties such as tolerances, mass inertia. A future work idea is to answer this question and improve our research. Explore innovative methods to visualize changes made to non-visual properties such as tolerances, mass, and inertia in robotic models.

While transparency has been utilized, investigate alternative approaches that effectively convey these modifications without significantly altering the overall view of the robot.

APPENDIX

```
data URDF =
    robot(map[str, URDFValue] attributes,
      map[str, list[URDF]] elements)
 ...
      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
  | plugin(map[str, URDFValue] attributes,
     map[str, list[URDF]] elements) // robot, link, or joint
  | counterbalance(map[str,URDFValue] attributes,
     map[str, list[URDF]] elements) // joint
  | tolerance(map[str, URDFValue] attributes,
     map[str, list[URDF]] elements) // robot custom property
  ;
```

Listing 13. Fragment of UrdfSL Abstract Data Structure

REFERENCES

[1] L. Addazi, F. Ciccozzi, P. Langer, and E. Posse, "Towards seamless hybrid graphical–textual modelling for UML and profiles," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Anjorin and H. Espinoza, Eds. Springer International Publishing, 2017, pp. 20–33.

[2] N. Albergo, V. Rathi, and J.-P. Ore, "Understanding xacro misunderstandings," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 6247–6252.

[3] D. Alique and M. Linares, "The importance of rapid and meaningful feedback on computer-aided graphic expression learning," vol. 27, pp. 54–60, 04 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1749772818300435

[4] R. Alur, *Principles of cyber-physical systems*. MIT press, 2015.

[5] M. Ben-Ari, *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[6] T. Bolwerk, "Improving the workflow for hardware engineers at philips with a domain-specific language and graphical feedback," 2023. [Online]. Available: https://www.cs.ru.nl/masters-theses/2023/T_Bolwerk___Improving_the_workflow_for_hardware_engineers_at_Philips_with_a_domain-specific_language_and_graphical_feedback.pdf

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0," 1998.

[8] W. Cazzola and L. Favalli, "Scrambled features for breakfast: Concepts of agile language development," *Communications of the ACM*, vol. 66, no. 11, pp. 50–60, 2023.

[9] C. Cook, M. Burnett, and D. Boom, "A bug's eye view of immediate visual feedback in direct-manipulation programming systems," in *Papers presented at the seventh workshop on Empirical studies of programmers*, ser. ESP '97. Association for Computing Machinery, 10 1997, pp. 20–41. [Online]. Available: https://doi.org/10.1145/266399.266403

[10] J. Cooper and D. Kolovos, "Engineering hybrid graphical-textual languages with sirius and xtext: Requirements and challenges," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 09 2019, pp. 322–325.

[11] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[12] J. Gray, S. Neema, J.-P. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-specific modeling." *Handbook of dynamic system modeling*, vol. 7, pp. 7–1, 2007.

[13] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," vol. 7, no. 2, pp. 131–174, 06 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X96900099

[14] J. Horowitz and J. Heer, *Live, Rich, and Composable: Qualities for Programming Beyond Static Text*, 03 2023.

[15] P. Inostroza, T. van Der Storm, and S. Erdweg, "Tracing program transformations with string origins," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2014, pp. 154–169.

[16] P. Klint, T. Van der Storm, and J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," ser. Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, 2009, pp. 168–177.

[17] L. Kunze, T. Roehm, and M. Beetz, "Towards semantic robot description languages," in *2011 IEEE International Conference on Robotics and Automation*, 05 2011, pp. 5589–5595, ISSN: 1050-4729.

[18] A. Lozano, K. Mens, and A. Kellens, "Usage contracts: Offering immediate feedback on violations of structural source-code regularities," vol. 105, pp. 73–91, 07 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016764231500012X

[19] T. Munzner, *Visualization Analysis and Design*. CRC Press, 12 2014, google-Books-ID: NfkYCwAAQBAJ.

[20] NDC Conferences, "Real-time prototyping using visual programming languages - rui martins," 10 2018. [Online]. Available: https://www.youtube.com/watch?v=cAiFJEcqwm4

[21] G. Onwubolu, *Mechatronics: principles and applications*. Elsevier, 2005.

[22] F. Pérez Andrés, J. de Lara, and E. Guerra, "Domain specific languages with graphical and textual views," in *Applications of Graph Transformations with Industrial Relevance*, ser. Lecture Notes in Computer Science, A. Schürr, M. Nagl, and A. Zündorf, Eds. Springer, 2008, pp. 82–97.

[23] M. Schuts, R. Aarssen, P. Tielemans, and J. Vinju, "Large-scale semi-automated migration of legacy c/c++ test code," *Software: Practice and Experience*, vol. 52, no. 7, pp. 1543–1580, 2022.

[24] M. Schuts, M. Alonso, and J. Hooman, "Industrial experiences with the evolution of a dsl," in *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, 2021, pp. 21–30.

[25] G. B. Shelly and M. E. Vermaat, *Microsoft Office 2010: Introductory*. Course Technology Press, 2012.

[26] L. Shen, X. Chen, R. Liu, H. Wang, and G. Ji, "Domain-specific language techniques for visual computing: A comprehensive study," vol. 28, no. 4, pp. 3113–3134, 06 2021. [Online]. Available: https://doi.org/10.1007/s11831-020-09492-4

[27] R. H. Shih, *Parametric Modeling with Creo Parametric 2.0*. Sdc Publications, 2013.

[28] T. van der Storm, "Semantics engineering with concrete syntax," in *Eelco Visser Commemorative Symposium (EVCS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[29] van Rozen, "Live game programming with micro-machinations and rascal," 11 2014. [Online]. Available: https://www.youtube.com/watch?v=YzsKaJEX4D4

[30] R. van Rozen and J. Dormans, "Adapting game mechanics with micro-machinations: International conference on the foundations of digital games," 03 2014, publisher: Society for the Advancement of the Science of Digital Games.

[31] R. van Rozen and T. van der Storm, "Origin tracking + text differencing = textual model differencing: International conference on model transformation," pp. 18–33, 07 2015, place: New York Publisher: Springer.

[32] R. van Rozen, "Cascade: A meta-language for change, cause and effect," 11 2022. [Online]. Available: https://ir.cwi.nl/pub/32568

[33] R. van Rozen, Y. Reijne, C. Julia, and G. Samaritaki, "First-person realtime collaborative metaprogramming adventures," 12 2021. [Online]. Available: https://ir.cwi.nl/pub/31301/

[34] R. van Rozen and T. van der Storm, "Toward live domain-specific languages: From text differencing to adapting models at run time," vol. 18, no. 1, pp. 195–212, 02 2019. [Online]. Available: http://link.springer.com/10.1007/s10270-017-0608-7

[35] T. Witte and M. Tichy, "A hybrid editor for fast robot mission prototyping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 11 2019, pp. 41–44, ISSN: 2151-0830.

[36] D. Xue and Y. Chen, *System simulation techniques with MATLAB and Simulink*. John Wiley & Sons, 2013.