FASTTypeScript Metamodel Generation Using FAST Traits and Tree-Sitter Project

Aless Hosry[1], Benoit Verhaeghe[1]

[1]Berger-Levrault, 69760 Limonest, France

Abstract

The TypeScript programming language is a popular language when building web applications all around the world. However, analyzing applications developed in TypeScript is challenging because of the lack of modern analysis platforms dealing with this programming language. Moose is an existing platform developed in Pharo that aims to ease the analysis of programming language by using pre-existing composable traits leading to usable generic tools. However, building a new metamodel for a new programming language is still challenging. When considering the support of the TypeScript language, one needs (1) a parser to parse the TypeScript source code, along with (2) a metamodel capable of efficiently representing the Abstract Syntax Tree (AST) of the language. However, building these tools from scratch is a particularly costly task in terms of effort, time, and required knowledge. To ease the task of supporting a new programming language, we introduce a new approach leveraging the existing well-maintained parser Tree-Sitter. In this paper, we detail how we use it to build a first version of FASTTypeScript, a metamodel for TypeScript AST working in Moose. The project was completed within weeks, and initial results demonstrate that our method is effective and promising for application to other programming languages.

Keywords

Abstract Syntax Tree, TypeScript, Moose, Parsing, Metamodel

1. Introduction

TypeScript is becoming increasingly popular [1], especially with frameworks like Angular and ReactJS, which use it as a primary programming language. In parallel, analyzing software that uses TypeScript is becoming a high demand, especially for industry to apply, for example, code modifications for quality enhancement or security reinforcement. However, it is still complicated to find analysis platforms that support newer languages quickly.

Moose is a software analysis platform built in Pharo that offers tools for importing models, parsing data, querying, and visualizing [2]. It can already analyze Java and Fortran programs by representing their Abstract Syntax Tree (AST) using FAST-Java[1] and FAST-Fortran[2] metamodels. However, Moose cannot analyze TypeScript AST yet. Additionally, to proceed with such analysis, this requires not just metamodels but also the ability to parse the source code and to import data into Moose correctly to represent it using these metamodels. This process can take months or years to complete [3].

We aim to benefit from the capabilities of Moose to analyze TypeScript programs. However, we face challenges such as limited expertise in TypeScript and parsing techniques. Moreover, we need to work quickly due to industry demands for software analysis, especially since it uses Angular for web application development.

Section 2 details our methodology for designing the parser and the metamodel. Section 3 details an industrial use case we use to test our parser and metamodel. Section 4 presents our findings. Section 5 presents future work.

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland

aless.hosry@berger-levrault.com (A. Hosry); benoit.verhaeghe@berger-levrault.com (B. Verhaeghe)

0009-0007-1106-2492 (A. Hosry); 0000-0002-4588-2698 (B. Verhaeghe)

[1]FAST Java (https://github.com/moosetechnology/FAST-JAVA)

[2]FAST Fortran (https://github.com/moosetechnology/FAST-Fortran/)

2. Methodology

In this section, we explain our methodology for building a new metamodel for the TypeScript AST that is compatible with Moose. We followed four main steps: (1) Integrating an external tool capable of efficiently parsing TypeScript source code and generating the AST in Pharo. (2) Generating an initial version of a metamodel that can represent the TypeScript AST based on the external tool but also compatible with Moose. (3) Upgrading the metamodel and enrich it for better support to TypeScript software analysis. (4) Creating an importer to link instances from both metamodels: the original one generated using the external tool, and the upgraded one compatible with Moose.

2.1. Integrating Tree-Sitter in Pharo

In the first stage, we decided to create a project that works in Pharo and is able to parse TypeScript by connecting to an external parser. Using an existing parser will save us time from building a native parser in Pharo. We opted for using Tree-Sitter [3], which is an existing parser developed in C. We decided to use it for two key reasons:

- Its ability to parse multiple languages, including TypeScript, provides flexibility for future extensions if our strategy proves successful.
- Its support for incremental parsing [4], based on the algorithms of Wagner et al. [5]. Incremental parsing is needed for handling large source code files by avoiding full-file parsing after each modification. For example, with Tree-Sitter, one can parse a method, apply changes to it, and reparse it. The second parse generates a new model by updating the initial one and does not generate a full model again. Traditional Pharo parsers, such as SmaCC [6] and PetitParser2 [7], do not support this feature and are typically used to parse languages like Java, Python, and Fortran.

To apply the integration, we developed "Pharo-Tree-Sitter" [8], a new project that integrates Tree-Sitter with Pharo via FFI protocols. The new project contains multiple classes and APIs. One class, `TSLibrary`, is responsible for calling the original APIs of Tree-Sitter using the FFI protocol. It contains a set of APIs such as methods to create a new parser, set a specific language, and access the tree nodes. We also created classes to encapsulate the results of the parsing, such as `TSTree` and `TSNode`. Each one of them encapsulates a set of methods to handle the tree and access the nodes. Moreover, we created a package called "TreeSitter-Libraries" to handle the generation of Tree-Sitter libraries and move them to the correct location for different operating systems automatically in order to grant a smooth installation and usage of the new project in Pharo. After setting up the project, we began parsing TypeScript code within Pharo and developed a set of tests to ensure the reliability of our integration.

Listing 1: Pharo-Tree-Sitter example parsing for TypeScript

```
1  parser := TSParser new.
2  parser language: TSLanguage typescript.
3
4  string := 'class Person {
5      name: string;
6  }'.
7
8  tree := parser parseString: string.
9  rootNode := tree rootNode.
```

Listing 1 is an example of how we can use "Pharo-Tree-Sitter" in Pharo to parse TypeScript code:

[3] Tree-Sitter (https://github.com/tree-sitter/tree-sitter)

- In line 1, a new Tree-Sitter parser in Pharo is created using the `TSParser` class and encapsulated in `parser`.
- In line 2, `parser` invokes the method `#language:` to specify which language should be parsed, which is TypeScript in our case.
- In lines 3 to 6, we define a TypeScript code snippet and assign it to the `string` variable.
- In line 8, `#parseString:` is invoked by `parser` and takes the `string` as a parameter. Here, the parsing is done by calling the generated libraries of TreeSitter original project through FFI protocol. The result, of type `TSTree` is stored in `tree`.
- Finally, in line 9, we use `rootNode` to access the root of the generated tree.

At this stage, "Pharo-Tree-Sitter" works with Pharo and is able to generate a model that represents TypeScript AST. However, the generated model is not compatible with Moose. And we are still at this stage, unable to benefit from Moose features to work with the model. To address this limitation, the next step is to adapt "Pharo-Tree-Sitter" with Moose.

2.2. Generating a first version of a new metamodel compatible with Moose

In the second stage we worked on generating a new metamodel for TypeScript AST that is compatible with Moose and based on Tree-Sitter.

Building metamodels in Moose is automated through a metamodel generator. Each generator is unique and has specific properties that allow specifying the metamodel characteristics. A generator creates a new package with classes where each one represents a node of the language's AST. To do that, we rely on the grammar of Tree-Sitter that is capable of representing all the AST nodes of many languages.

We created the "TSFASTBuilder" class in the "Pharo-Tree-Sitter" project, responsible for generating a first version of any metamodel generator based on Tree-Sitter and by specifying the requested language.

This is a generic method for a first version metamodel generation using "Pharo-Tree-Sitter," on the condition that the requested language is supported by this project.

Listing 2: Generation of metamodel generator using "Pharo-Tree-Sitter" project

```
1  builder := TSFASTBuilder new.
2  builder tsLanguage: TSLanguage typescript.
3  builder build.
```

Listing 2 represents the statements used to generate a first version of our new metamodel generator for TypeScript AST. After the execution of this listing, we got a first version of the metamodel generator. For the naming convention, all packages and classes are preceded by prefix FAST which is used for all the metamodels that represent the AST in Pharo. We will elaborate more on FAST in the upcoming subsection. Below are the details of the automated generation:

- A new package "FAST-TypeScript-Model-Generator" encapsulating the class "FASTTypeScript-MetamodelGenerator" that inherits from "FamixMetamodelGenerator" was created. The class is responsible for the automatic generation of the "FASTTypeScript" metamodel compatible with Moose using `FASTTypeScriptMetamodelGenerator new generate`.
- `packageName` on the class side: precises the package name in which the new metamodel is encapsulated. In our case it is "FAST-TypeScript-Model".
- `prefix` on the class side: specifies the prefix of the generated classes of the metamodel, which is "FASTTypeScript" in our case.
- `slots`: this is responsible in Pharo for defining the slots to be managed in the methods of the class. In our case, each defined slot represents an AST node of TypeScript such as `#boolean` and `#forInStatement`. `slots` was filled by calling an API from the "Pharo-Tree-Sitter" project that returns a list of all the grammar used by Tree-Sitter to represent the AST of TypeScript.

- `#defineClasses`: this is a method where all the classes of the metamodel are defined. Thus, each modification, suppression or addition must be handled in this method. The method was configured to be filled automatically with statements responsible for the creation of the classes of the metamodel. For instance: `boolean := builder ensureClassNamed: #Boolean` leads to the creation of "FASTTypeScriptBoolean" class in "FAST-TypeScript-Model" package once the metamodel is generated.
- `#defineHierarchy`: this method is responsible for establishing the inheritance relationships between the classes. For example, "FASTTypeScriptBoolean" inherits from "FASTTypeScriptLiteral".
- `#defineRelations`: this method defines the selectors of each class, which will make the metamodel rich with the info that it provides through the properties of each class. For example: "FASTTypeScriptDeclaration" has `#body` and `#name` as properties.
- `#defineTraits`: this method defines Traits (which creation is supported in Pharo). Traits are collections of methods that can be reused by multiple classes without inheritance. Using traits allows to share code between different classes without duplicating that code. This method defines customized traits responsible to handle this metamodel. For example: "FASTTypeScriptTOptionalField" is a trait responsible for handling the optional properties of the classes like `#return_type` for "FASTTypeScriptMethodDefinition".

We executed the generator and created a first version of the metamodel of TypeScript AST, which we named "FASTTypeScript". The first version of "FASTTypeScript" contains empty classes that correspond to the AST nodes of TypeScript imported from Tree-Sitter. For instance, the "FASTTypeScript" metamodel can now represent a method defined in TypeScript but does not provide the developers with its name and its body because this kind of info is not yet provided to the metamodel. In the next stage we explain how we upgraded this metamodel to enrich it with more properties useful for TypeScript analysis.

2.3. Upgrading the metamodel

In the third stage, we refined "FASTTypeScript" to add more properties for each class in two ways: using FAST traits and by importing some properties from the original Tree-Sitter project.

In Pharo, FAST [4] is a list of traits that is is already predefined. It provides a set of useful methods that can be used by each class that uses such traits. For instance "FASTTypeScriptString" is a class that represents String nodes of any generated FASTTypeScript model. To refine it, we used the "FASTTStringLiteral" trait from the FAST package because it provides a set of methods for handling string literals. To fulfill this upgrade, we updated the method `#defineHierarchy` and this statement `string -|> #TStringLiteral.` before generating again the metamodel.

Moreover, to enrich our metamodel, we reviewed each class and parsed examples to generate models with nodes of these types. During this process, we identified properties covered by Tree-Sitter but not covered by existing FAST traits and still essential for these nodes. We also implemented them for each node. For instance, we added 3 properties for `FASTTypeScriptTryStatement`: `#body` that returns the body inside `try`, `#handler` that returns everything inside `catch()` and `#finalizer` that returns everything inside `finally` .

2.4. Creating an importer

In the previous steps, we were able to create a project that allows parsing TypeScript from Pharo using Tree-Sitter. However, the generated AST from "Pharo-Tree-Sitter" was not compatible with Moose, so we created a new metamodel in the second stage and upgraded it in the third one. In this stage, we want to ensure a transition from the "Pharo-Tree-Sitter" model to the "FASTTypeScript" model. This is why we implemented an importer that traverses the former and generates a model of the latter.

[4] FAST (https://github.com/moosetechnology/FAST)

The transition is straightforward because all the node types are copied from the original Tree-Sitter grammar. Essentially, the importer copies the nodes, their corresponding source code positions, and the full source code for the root node. Thanks to FAST traits, the source code of each node becomes accessible, allowing the corresponding methods to be executed.

Listing 3: FASTTypeScript example

```
1 parser := FASTTypeScriptParser new.
2 string := 'with (person) {
3   console.log(address.city);
4 }'.
5 model := parser parse: string.
```

Listing 3 provides an example of how a string can be parsed and a new model of Type "FASTTypeScript" can be generated in Pharo:

- In line 1, a new `parser` is created using the `FASTTypeScriptParser` class.
- In lines 2 and 4, a TypeScript code snippet is defined and encapsulated in `string`.
- Finally, in line 5, the parser receives the `#parse` message to parse the TypeScript string, and the resulting object of type `FASTTypeScriptModel` is stored in `model`.

Finally, figure 1 is a summary of the 4 previous stages for parsing TypeScript in Pharo. It illustrates an example of an interface **Person** being parsed. Initially, the "Pharo-Tree-Sitter" project calls the external library (step 1), *i.e.,* the dll file on Windows operating system, generated from the original Tree-Sitter project through the FFI protocol to parse the given source code. The result of the parsing in Pharo is a model in step 2. Then this model is translated to a new model of type "FASTTypeScript" in step 3. As shown in the figure, the Abstract Syntax Tree (AST) can be visualized using Moose through the FAST tab. This tab has been developed by the original developers of FAST and FAST-Java for visualizing Java AST and is made compatible with our "FASTTypeScript" model thanks to the use of generic FAST Traits.

3. Use cases

We started to use "FASTTypeScript" for an internal project. The goal of this project is to enhance the quality of software being developed by the developers of the company by providing comments after pushing to source code management platforms (like GitLab) and suggesting modifications. This project takes as input a Moose model and a set of "rules" to be applied using another project.

Rules are encoded in a separate project implemented in Pharo. This rules engine project will be called by separate internal projects that are in the process of development and by the software quality enhancement project listed previously. Each rule is specific to a language and a violation. A rule contains a "pattern", which is responsible for searching for specific source code that violates a condition following specific criteria, and a "fix", which is responsible for launching a specific action over the source code caught previously to fix the violation.

To search for the source code, we used MoTion [9], a declarative object matching language. MoTion is already working by matching Java, Smalltalk and XML models imported in Pharo. It can also be used with "FASTTypeScript" to do pattern matching. It would have been more challenging if we did not have "FASTTypeScript" to import TypeScript models in Moose. MoTion can match any object by expressing patterns in a declarative way that describe the type of the object using the class names like `FASTTypeScriptIfStatement` and the properties using the method names like `condition` with their values.

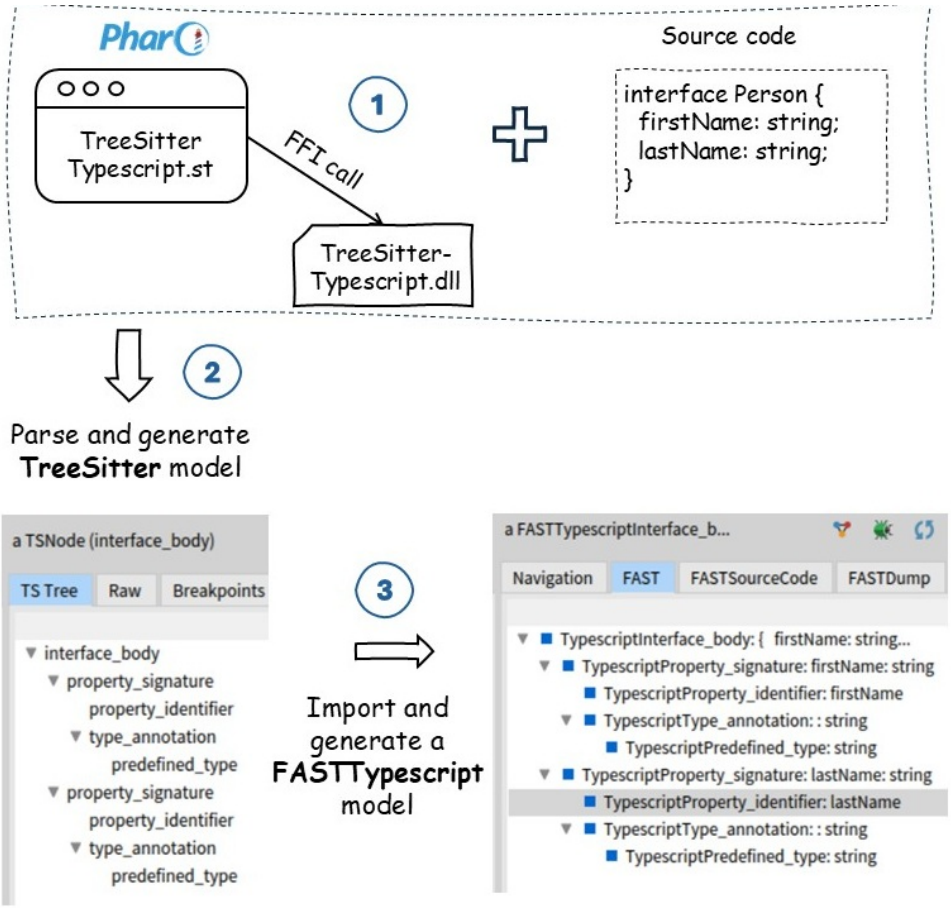

Figure 1: Steps of parsing a TypeScript interface

Listing 4: MoTion with FASTTypeScript example

```
1  pattern := FASTTypeScriptModel % {
2    #entities <=>  FASTTypeScriptMethodDefinition % {
3        #name <=>  'get\s+.*'.
4        #children <=> FASTTypeScriptStatementBlock % {
5            #children <=> { #'@child1'. #'*otherChilds'. }
6        }.
7      }.
8    }.
9
10 pattern match: aTypeScriptModel.
```

Listing 4 shows an example of how MoTion can be used to describe a pattern that matches a FASTTypeScript model. In this example we are looking for all the getters that contain more than one statement inside:

- The pattern is defined between lines 1 and 8.
- The pattern begins by specifying that the object to match is of type `FASTTypeScriptModel`, using the syntax `FASTTypeScriptModel % { ... }`.
- To narrow the scope, the `#entities` property is used to access all entities within the model. We further restrict this to match only method definitions by using `#entities <=> FASTTypeScriptMethodDefinition % { ... }`.
- In line 3, a regular expression is used to match the name of each method that begins with the keyword `get`, potentially followed by other characters.

- Line 4 allows us to access the body of the method using `FASTTypeScriptStatementBlock`, which leads to the access to the statements of the method inside the body.
- Line 5 employs collection matching to verify that the method contains more than one statement. Specifically, it looks for one mandatory child (`#'@child1'`) and any number of additional children (`#'*otherChilds'`).
- Finally, line 10 shows how we can match the defined `pattern` with a `aTypeScriptModel` generated using "FASTTypeScript".

4. Findings

In this section, we present our findings, following the creation of "FASTTypeScript" and its usage.

To verify the ability of our methodology for parsing the TypeScript files and representing their AST in Moose, we executed the "FASTTypeScript" parser over some projects of the company. We started with 2,696 TypeScript files that we needed to initially analyze. The length of these files can reach around 700 lines of code per file, providing a more realistic case for our test. Our findings show that 99% of the collected files could be parsed using "Pharo-Tree-Sitter". Less than 1% of the parsed files generated an AST that contained ERROR nodes that we are still investigating to figure out the issues. Normally the code is compiled and pushed to GitLab without errors, but we still need to investigate more to discover the main reason for the existence of errors after parsing. This indicates that our approach, which was implemented within 5 months, proved to be promising to continue evolving with "FASTTypeScript".

In terms of correctness of the generated ASTs, we conducted a comparison of the ASTs generated from small source code samples. This comparison was made between the model generated from the original project using "Pharo-Tree-Sitter" and the model represented using the "FASTTypeScript" metamodel. We did this comparison because we trust Tree-Sitter being tested by the community and used by thousands of developers, so we trust its ability to parse the source code and represent all the corresponding nodes of TypeScript source code. We found that we got the same hierarchical structure of nodes for both models. This indicates that the import of a model generated using an external tool for the generation of "FASTTypeScript" models led to a successful transition.

Moreover, following the multiple usages of "FASTTypeScript", we found that it was practical to traverse the generated AST using FAST methods such as `children` and `allParents`. Additionally, FAST methods were practical for detecting the source code of each node using `sourceCode`. As for the methods transferred from the original Tree-Sitter project, we found that they were useful, especially when accessing a node's details. By enriching the metamodel with these methods, we found that they were very helpful for developers to add as much description as possible while searching for specific nodes in "FASTTypeScript".

Our findings extend beyond technical implementation and also show a methodological impact. We realized that other developers started to follow our methodology to cover additional programming languages such as C [5] and Perl [6]. But their implementations are still at the beginning.

5. Future work

Our implementation is not finished yet.

First, we still need to continue looping over all the node types of the Tree-Sitter grammar, generate tests and add all the missing properties to "FASTTypeScript" metamodel. Despite that, the findings are encouraging to keep upgrading the metamodel and using FAST traits to ensure more stability of "FASTTypeScript" with Moose.

Second, this implementation helps us to expand our work in several areas. We can create additional FAST metamodels using this methodology, extending Moose to support more languages like what is being done with C and Perl by other developers.

[5] Tree Sitter and C in Moose (https://github.com/moosetechnology/TreeSitterCLanguage)
[6] Tree Sitter and Perl in Moose (https://github.com/moosetechnology/Famix-Perl)

Third, we also plan to further develop the "FASTTypeScript" metamodel and create a bridge with Famix TypeScript[7]. While FAST focuses on ASTs, Famix provides a higher-level, abstract representation of the source code of a software. By combining these approaches, we analyze TypeScript projects more efficiently.

Fourth, beyond the internal project focused on defining rules for TypeScript applications, we aim to expand our software analysis efforts to cover not just monolingual projects but also multilingual systems. It is common to encounter projects that combine TypeScript with other languages. This new expansion requires the use of another tool, Adonis [10], which is specifically designed to help developers identify external dependencies that exist between entities across different languages or even different machines.

6. Conclusion

Analyzing TypeScript source code is in high demand, yet there are few tools available to support this language. To address this problem, we aim to extend Moose, an existing tool in Pharo to analyze software, to support TypeScript analysis. To achieve this, we developed a new metamodel to represent TypeScript ASTs.

In this paper, we present our methodology for creating "FASTTypeScript" using Tree-Sitter project, which already supports TypeScript parsing. We also used FAST traits which contain sets of methods to access ASTs and allow accessing Moose features.

Although the implementation is new, multiple use cases have proven its effectiveness. We tested it using various TypeScript code snippets and real-world examples from GitHub. The models were generated successfully, and using FAST traits, we were able to benefit from existing methods to access model nodes and retrieve requested info such as the source code of the node.

This approach allows us to support more languages by creating metamodels using Tree-Sitter, avoiding the need to develop complex custom parsers for each language.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] J. D. Scarsbrook, M. Utting, R. K. Ko, Typescript's evolution: An analysis of feature adoption over time, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, 2023, pp. 109–114.

[2] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, M. Derras, Modular moose: a new generation of software reverse engineering platform, in: Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19, Springer, 2020, pp. 119–134.

[3] A. Afroozeh, A. Izmaylova, One parser to rule them all, in: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), 2015, pp. 151–170.

[4] C. Ghezzi, D. Mandrioli, Incremental parsing, ACM Transactions on Programming Languages and Systems (TOPLAS) 1 (1979) 58–70.

[5] T. A. Wagner, S. L. Graham, Efficient and flexible incremental parsing, ACM Transactions on Programming Languages and Systems (TOPLAS) 20 (1998) 980–1013.

[6] J. Brant, J. Lecerf, T. Goubier, S. Ducasse, A. Black, Smacc: a compiler-compiler, 2017.

[7] Famix TypeScript (https://fuhrmanator.github.io/posts/typescript-in-moose/index.html)

[7] J. Kurs, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, J. Laval, Petitparser: Building modular parsers (2013).
[8] B. Verhaeghe, A. Hosry, N. Hlad, Evref-bl/pharo-tree-sitter: v1.0.2, 2025. URL: https://doi.org/10.5281/zenodo.15089054. doi:10.5281/zenodo.15089054.
[9] A. Hosry, V. Aranega, N. Anquetil, Motion: A new declarative object matching approach in pharo, Journal of Computer Languages 81 (2024) 101290.
[10] A. Hosry, N. Anquetil, External dependencies in software development, in: International Conference on the Quality of Information and Communications Technology, Springer, 2023, pp. 215–232.