# Analyzing Dart Language with Pharo: Report and early results

Nicolas Hlad, Benoît Verhaeghe, Mustapha Derras

# Analyzing Dart Language with Pharo: Report and early results

Nicolas Hlad[1,*,†], Benoit Verhaeghe[1,†] and Mustapha Derras[1]

[1]*Berger-Levrault, Toulouse, France*

Abstract

Dart is a programming language introduced by Google in 2011. Today, it is mainly used in Flutter, a cross-platform SDK to build native mobile applications, desktop applications, and web-based applications. Since its introduction, Flutter has gained in popularity and so has Dart. Thus it has become crucial for the industry and the academic community to dispose of the proper tools to analyze, maintain and evolve projects in Dart. However, our research community has yet to propose tools for the static analysis of Dart. In this paper, we share our experience regarding the usage of Pharo 10 to support the analysis of Dart2.10 and we present our early work on a set of open-source tools. Our tools cover the parsing of Dart with SmaCC, the visualization of its AST with Roassal, and its meta-model analysis with Moose and Famix.

**Keywords**

Code Analysis, Parser, Model Driven Engineering, Visualizer

## 1. Introduction

Introduced in 2014 by Google, Flutter is initially an open-source framework to build cross-platform mobile apps, focused on iOS, Android, and Windows Phones [1]. From a single code base, the Flutter engine can natively compile for each platform, without relying on Javascript iFrame code (unlike Ionic and Cordova). Thus, Flutter was a direct alternative to Microsoft's Xamarin [2]. With passing years, this framework has grown to add more platforms: web-based applications in 2019 and desktop applications in 2021.

Google uses Dart as Flutter's main programming language. Initially launched in 2011, Dart[1] is a general-purpose language that supports object-oriented programming with a C-style syntax [3]. It also that supports modern language features such as just-in-time and ahead-of-time compilation, dynamic and static typing, garbage collection, asynchronous activities, etc.

According to *spectrum.ieee.org*, in 2022, Dart was the 15th trending language [2]. Therefore, we believe that there is a growing interest in analyzing Dart to support the analysis of Flutter

*IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France*

*Corresponding author.

†These authors contributed equally.

nicolas.hlad@berger-levrault.com (N. Hlad); benoit.verhaeghe@berger-levrault.com (B. Verhaeghe); mustapha.derras@berger-levrault.com (M. Derras)

https://badetitou.fr/ (B. Verhaeghe); https://www.research-bl.com (M. Derras)

0000-0003-4989-2508 (N. Hlad); 0000-0002-4588-2698 (B. Verhaeghe)



[1]https://dart.dev/

[2]see report : (https://spectrum.ieee.org/top-programming-languages-2022)

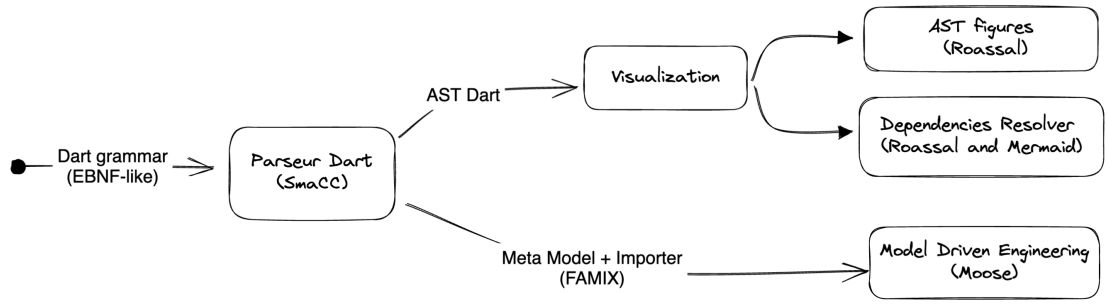


**Figure 1:** Analysis process set up for Dart in Pharo

applications. However, despite its increasing popularity, the Dart community faces an absence of academic and industrial tools to analyze this language. Furthermore, among the different ways to improve software quality, model-driven engineering (MDE) and re-engineering has outstanding impacts, as seen in [4, 5]. Yet, we observe that common software analysis tools and platforms, such as EMF/ECORE [6], have yet to cover new languages like Dart, and rather focus on legacy languages like Java.

To tackle the Dart programming language software quality tooling, we built a process for analyzing Dart source code and Flutter application. This process comes with the usage of tools to perform the first level of analysis: code parsing and model analysis, alongside with Flutter application visualizations. In this paper, we share our experience on how to leverage different tools to perform such first analysis of a new language such as Dart.

Our paper is structured as follows: section 2 presents an overview of the steps we follow to implement our analysis of Dart; section 3 details the parsing step of Dart and our use of SmaCC; section 4 presents two visualizations we introduce for Dart, one for the Dart AST with Roassal and another for the file dependencies in a Dart project with Roassal and Mermaid; section 5 introduces the early implementations of an importer for the Famix meta-model with Moose.

## 2. Overview

In our work, we want to exploit and reuse the tools' suit of Pharo, which has shown to be a reliable static analyzer of legacy languages like Java, Delphi, and C [7]. Furthermore, past works have proven the maturity of Pharo as an industrially-ready tool to analyze large applications with MooseIDE [8]. Thus, in this overview illustrated in Figure 1, we present the overall step-by-step process we follow to analyze Dart using Pharo's tools kit.

**Parsing Dart code.** The first step of our analysis is to be able to parse and perform symbol resolution of Dart source code. The goal of this step is to output an Abstract Syntax Tree (AST) for Dart. To achieve our goal, we need to formally define a grammar using Extended Backus-Naur Form (EBNF) for Dart and to implement its corresponding parser. The implementation of this parser required us to implement all the Dart entities inside their corresponding Pharo classes. So that when parsing a Dart declaration of a class, our parser creates a DartClass entity

(and so on for method, attribute, etc.). Inside the existing Pharo tool set, we found two options: PetitParser2[3] and SmaCC [9]. We use SmaCC because it generates Pharo's classes for each of the language's entities define in an EBNF grammar (more on this in section 3).

**Visualization tools.** With a given ENBF grammar, SmaCC generates a parser that we call *SmaCCDart*. A visitor is also generated, which allows us to walk through the Dart AST. Given the Dart AST, we implement visualizations to perform a first analysis of the source code. This is done using another tool in Pharo called Roassal[4]. Roassal is a graphical engine that can be linked to a data model, allowing users to interact with the model through a graphical user interface. Here, we build two visualizations: an AST visualizer and a file dependencies visualizer. These visualizations are presented in section 4.

**Model analysis.** Exploiting an AST with a visualization is practical when analyzing one or two source code files. However, it becomes more tedious when having to visualize more than a hundred files in one project. This is where Moose becomes handy [10]. Moose is a powerful and flexible framework for software analysis and visualization, which can help developers gain insights about their code and improve its quality and maintainability. It comes with a set of tools in Pharo to perform Model Driven Engineering for any given programming language, using the Famix model. Setting up the model analysis requires 3 steps: 1) we use Famix to build a meta-model of the Dart language; 2) we develop a *Famix Importer* called *Chartreuse-D*, to initialize a Famix-Dart meta-model when visiting the Dart AST; 3) finally, we execute a symbol resolution on the model instance to resolve the association between the model's entities. We detail this part in section 5.

In the following section, we detail each step of our analysis process of Dart in order. All the tools that we present are work in progress (with many improvements required) and their source code are available in their respective GitHub depot.

## 3. Parsing with SmaCC

The first step to perform advanced analysis of a programming language is to build a parser. Based on the parser output, one will then be able to create visualisations or perform reengineering.

However, unlike EBNF legacy programming languages (or long-live ones such as Java and C), we observe difficulties to find tooling to parse Dart code. Perhaps it is due to Dart being a relatively new language from 2011 and that it is still in evolution. Or, because Google has not introduced an official grammar for Dart, despite providing an official specification from Google [5], the 5th edition since 2014[6]. Tools to extract an AST are only available as Flutter packages, making them useful only in the context of Flutter development [7].

[3]PetitParser2: http://kursjan.github.io/petitparser2/
[4]Roassal3: https://github.com/ObjectProfile/Roassal3
[5]https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf
[6]https://www.ecma-international.org/publications-and-standards/standards/ecma-408/
[7]https://pub.dev/documentation/analyzer/latest/dart_ast_ast/dart_ast_ast-library.html

SmaCC (*Smalltalk Compiler-Compiler*) is a generator of scanners and parsers for language that comes with an EBNF grammar. SmaCC inputs an EBNF grammar and outputs a parser that match the grammar. SmaCC compilation process follows a two-step process: first, the scanning which converts a stream of characters into a stream of tokens; then, a parsing step which converts the stream of tokens into objects that are defined by the SmaCC user.

Listing 1: Declaring a token and a production rules with SmaCC

```
helloUser :
    "hello" <name> 'userName' {{HelloUser}} ;

<name> :
    [A-Za-z] \w* ;
```

Let's go over the basic of SmaCC with a grammar example in Listing 1. Using the EBNF grammar, we declare the *tokens* and the *production rules* used by SmaCC to output our *objects*. A token is declared with a unique token name associated with a specific regular expression. In our example, *<name>* is a token that matches any name starting with a capital letter. Production rules are non-terminal symbols associated with a set of possible production. Unlike tokens, each production rule is a none terminal symbol (the production name, like *helloUser*) and consists of a list of tokens or keywords, ended optionally by a Smalltalk statement. Keywords are declared using double quotes (like *"hello"*). A SmallTalk statement is enclosed inside curly brackets. Double curly brackets are used to declare an AST node object associated with a specific production, here *HelloUser* will be generated as a Pharo class. A token can be caught as an instance variable when associated with a symbol using a single quote inside the production rule. In our example, *'userName'* is an instance variable of the class *HelloUser* and that catches the value of the token *<name>*

To kick start our work on our parser, we've started from a community open-source grammar of Dart available in ANTLR4 GitHub project[8]. Although ANTLR grammar is in EBNF, SmaCC requires adding specific statements inside the EBNF file (like in Listing 1). Thus, manual editing is required to adapt an ANTLR4 grammar to a SmaCC grammar. Hopefully, SmaCC comes with a tool that eases this transformation[9]. In our work, we add a statement to declare a SmaCC node beside each production rule that is interesting to consider in our AST. The grammar we adapt and our SmaccDart parser are available on GitHub[10].

Here are some notes to conclude this section. First, the grammar we use from ANTLR reports that the official Dart specification is not up-to-date with the actual compiler code [11]. However, they are unclear about what exactly differs between their grammar and the official Dart specification.

Second, the authors of Dart specification are unclear about what is the proper starting symbol of Dart grammar. They hint that they may be at least two equivalent starting rules: ⟨partDirective⟩ or ⟨libraryDefinition⟩ (see the comment of section 19, page 204 of the specification).

[8] ANTLR Dart2 grammar : https://github.com/antlr/grammars-v4/tree/master/dart2
[9] ANTLR to SmaCC convertor : https://github.com/j-brant/SmaCC/tree/master/rewrites/antlr
[10] SmaCCDart : https://github.com/Evref-BL/SmaccDart
[11] see comment line 14 : https://github.com/antlr/grammars-v4/blob/c0ad64169f54727b7262b4eb1f7edd5cae14a5ce/dart2/Dart2Lexer.g4#L14

Thankfully, SmaCC let us define multiple starting symbols, thus resolving this ambiguity with a SmaCC directive like so: *%start libraryDefinition partDirective*.

Finally, in Dart, class constructor invocations are heavily used inside return expressions to compose a Flutter widget with multiple instances of other widgets. However, only a capital letter in the identifier distinguishes a constructor invocation from a function expression. This only difference is further exaggerated by the fact that the *new* keyword is optional in Dart2. Thus, our parser is forced to explore both the entire branch of function expression production rule and the constructor invocation, before choosing (from the longest branches of the two) which one is the proper AST node. This exploration has a significant performance impact on our parser. To solve it, we introduce a Dart refactoring step before the parsing that adds the *new* keyword to constructor invocations. This resolves the ambiguity between the *functionExpression* and *constructorInvocation* production rules, significantly increasing the parsing speed. For instance, using the implementation of the class *_MyHomePageState*[12], the parsing is over 5 times faster with the refactoring (1476ms without refactoring against 271ms with refactoring). Therefore, refactoring the code to remove ambiguities has a significant performance impact on the parser speed, but comes with the limitation of changing the initial code source.

## 4. Visualization with Roassal

SmaccDart give us the means to analyze Dart code and get a Dart AST. However, SmaCC itself does not provide a visualization tool to help developers visualize an AST (in the latest version of Pharo). To solve this, we use Roassal3 to implement an AST visualization. Roassal3 (or simply *Roassal*) is an agile visualization engine for Pharo. It provides a versatile framework for developing interactive and dynamic data visualizations in a range of forms, such as 2D images, animations, and interactive widgets. Its interactivity is one of its main benefits.

### 4.1. Roassal Dart AST visualizer

The conception of a Roassal visualizer can be summarized as: a) declaring shapes (*i.e.,* Roassal UI entities) in a canvas; b) associating the shapes to a model; and c) defining a global layout. Shapes are declared programmatically by visiting the produced AST of the previous step. During this visit, we declare a new *RSBox* shape for each node visited, and a link between these boxes for each *node to children* relation in the AST. Each shape can be set to a model entity using the *Shape»model: Object* method. It allows us to display a text value in each box that corresponds to the AST node type. Furthermore, it allows us to interact with the model, as we explain after. Finally, Roassal comes with a set of pre-loaded layouts that relies on links between shapes. For rendering the tree structure of the AST, we use the *RSTreeLayout*. The result is shown in Figure 2, from an AST based on a basic *hello world* Dart class.

The interaction with a Roassal visualization allows us to display more information about the model when interacting with the windows of Roassal. In Figure 2, dragging the mouse cursor over a shape displays an overlay panel containing an overview of the source code declared by

[12]Code extrated from Google's tutorial of Flutter https://github.com/flutter/codelabs/blob/b53da2544a1df2b0456d54cd83f37bc7d68805f0/namer/step_08/lib/main.dart#L53

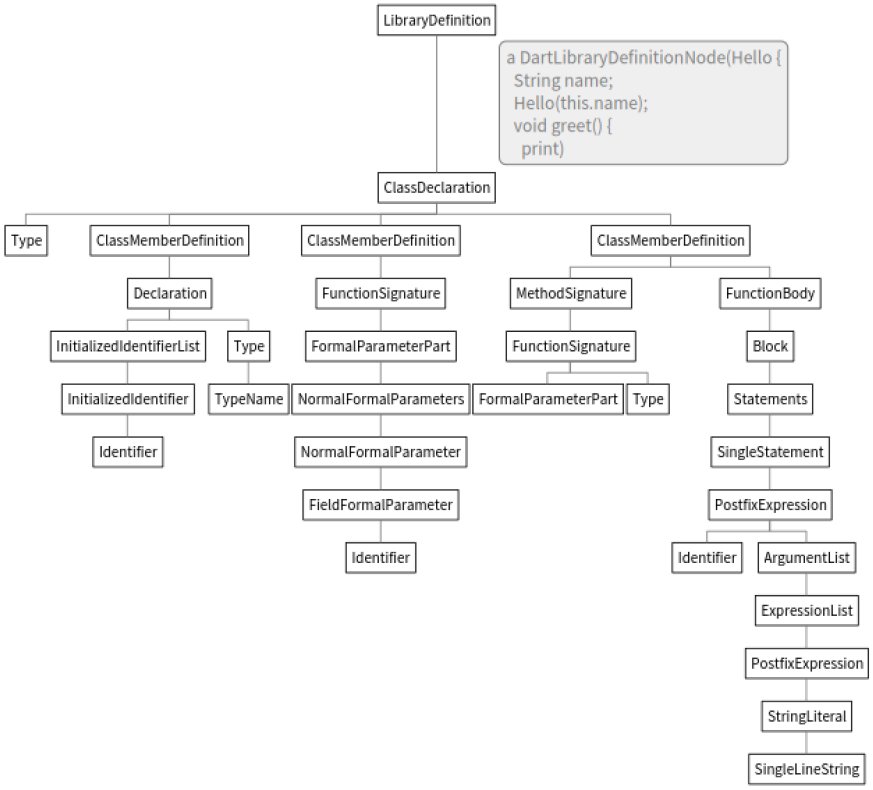


**Figure 2:** Visualization of a Dart AST built with Roassal

this node. By clicking a node, we open an *Inspector* window to the SmaCCDart object of the Roassal model, allowing further inspection of the AST. Our AST visualization is included with the Baseline dependencies of SmaCCDart.

## 4.2. Flutter Dependencies Resolver

We implement a second visualization that takes advantage of SmaCCDart to give a more abstract understanding of the Flutter code. We develop a Flutter dependencies resolver that looks at the imports/exports dependencies declared in each Dart file of a Flutter application. Our tool explores the content of each Dart file declared in a Flutter project with SmaCCDart. Then it maps the file import dependencies inside a graph, with each outbound edged representing a import declaration to a specific file.

Dependencies are visualized as a graph, where nodes are files and directed edges represent their specific imports. We color files that are declared in the Flutter project in orange, to distinguish them from the external dependencies (in gray). As an example, we analyze a GitHub open-source Dart project called *Animsearch*[13]. We produce the import dependencies graph of this Flutter application using a mermaid flowgraph in Figure 3. The result can be exported to a

[13]AnimSearch : https://github.com/ArizArmeidi/AnimSearch

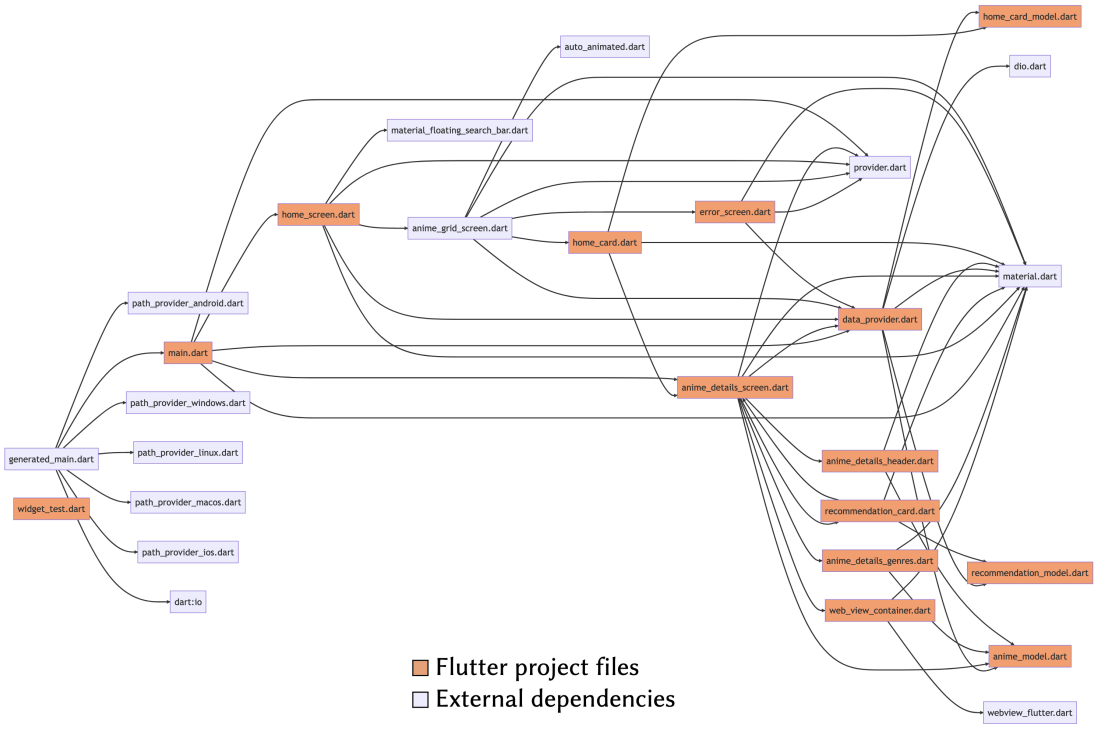


**Figure 3:** Dependency graph of Dart files in the Animsearch flutter app

mermaid[14] drawing that can be included in markdown text. Thus, developers can share this representation with non-Pharo experts.

The two proposed visualizations are convenient for small projects, but shows their limits at scale. For instance, the AST visualization is efficient for one file at a time, but becomes tedious to visualize the entire AST of one project because of the large amount of nodes. Similarly, a project with a large number of files and dependencies is difficult to represent inside our Mermaid graph. Therefore, to extend the analysis of Dart, we rely on model engineering and build a meta-model for Dart in section 5.

## 5. Model Driven Engineering with Famix and Moose

Based on the AST produced by SmaCC, one can build great visualization using Roassal. However, when it comes to analyzing industrial level software systems, these visualizations find their limits due to the large number of entities. Also, the algorithms to create them become increasingly difficult to write and less and less generalizable. It is in this context that one needs meta-modelization framework. Out of the existing one, Moose allows one to create easily new

[14]Mermaid : https://mermaid.js.org/

meta-models to support new programming languages with out-of-the-box compatible tools. In this section, we describe our early work with Moose for Dart project analysis.

### 5.1. Dart Meta-model definition

To construct the meta-model, we use the Famix framework [11] to create our model entities as Pharo's class. These class entities use *traits*, which are declared in the Famix framework. In the Java world, we would compare Traits to Java interfaces with default method implementation. In Famix, Traits can be combined to define complex entities. As of today, Famix has over 114 traits declared.

Declaring new entities is semi-automated with Famix since these classes can be generated. As an example, let's say that we want to add Dart classes in our Dart meta-model. First, we manually declare a Pharo class called *FamixDartGenerator*, a subclass of *FamixMetamodelGenerator*. The generator is responsible for defining the entity's classes, with the entity's traits and hierarchy. Thus, we add to *FamixDartGenerator* two methods: *defineClasses* and *defineHierarchy*, shown in Listing 2. Note that a Dart class has attributes and methods. In our meta-model, this relationship is translated using the traits *TWithAttributes* and *TWithMethods* (last two lines of *defineHierarchy* in Listing 3. Finally, we execute the method *FamixDartGenerator generate* and obtain a new package *Famix-Dart-Entities*, with our *FamixDartClass* class generated by Famix.

Listing 2: defineClasses method's code

```
defineClasses
    super defineClasses.
    class := builder newClassNamed: #Class
```

Listing 3: defineHierarchy method's code

```
defineHierarchy
    super defineHierarchy.
    class --|> #TClass.
    class --|> #TType.
    class --|> #THasVisibility.
    class --|> #TCanImplement.
    class --|> #TWithMethods.
    class --|> #TWithAttributes.
```

### 5.2. Importer and symbol resolver

Once a Famix entity is declared in our meta-model, it is ready to be created. To create a model, a good approach is to develop an importer, a program capable of visiting an object and returning instances of its corresponding model entities. In our case, the visiting object is the Dart AST, and its corresponding entities, the one we declare in our Famix-Dart meta-model.

Our Famix Importer is a work-in-progress project called *Chartreuse-D*[15]. As of today, it is only able to import classes, attributes, and methods. The methods to visit a Dart AST are provided

[15]Chartreuse-D : https://github.com/Evref-BL/Chartreuse-D

by SmaCC, using a trait it generates when we build our parser for Dart. Thus, we gain access to methods that are invoked when a specific AST node is visited, *e.g. visitClassDeclaration*. Inside, we set our importer to instantiate a new Famix Dart Entity corresponding to this currently visited class. The methods and attributes of this current class will be instantiated upon visiting their nodes.

Once the AST is fully visited, we complete our model with the resolution of the symbols. Symbol resolution associate the model entities between them. Thus, we look for dependencies such as *method invocations*, *variable and attribute accesses*, *type references*, and *type inheritances*. In our importer, this step takes place directly on the model instance itself.

### 5.3. Challenges ahead

In our current work on *Chartreuse-D*, we plan to implement the symbol resolution step after enough Dart entities are included in our meta-model. Our Roadmap plans a release for November 2023 of this importer, as an open-source project available on GitHub. However, this implementation is a specific *Famix Importer*, which comes with challenges. First, the behavior of the importer is tied to the visitor of the AST, thus our SmaCCDart parser. This means that maintenance of our parser will require maintenance of our importer. Second, Dart is a multiple paradigm programming language, which allows one to do Functional and Object-oriented programming. Therefore, functions can be declared and used inside objects' methods without restriction. This can prove challenging during the symbol resolution, as we will need to distinguish between inner method invocation and external function invocation. Finally, we need to study if the current Famix traits are sufficient for our Dart meta-model.

## 6. Conclusion

In this paper, we explore the potential of diverse tools built in Pharo to handle a static analysis of the Dart language. Our attempts cover the multiple steps of the static analysis process. First, we introduce SmaCCDart, a Pharo parser for Dart2 built with SmaCC. Second, we explore the potential of visualization conception made with Roassal3 and propose an interactive AST visualization. Plus, we propose a Flutter dependencies resolver that gives a quick view of the relation between the internal and external files in one Flutter project. Finally, we give an overview of how to conduct MDE with Moose and Famix for a Dart code. In this work in progress, we present the idea of a meta-model implementation for Dart based on the Famix model. We also present our plan to implement an importer that will instantiate our Famix Dart meta-model from visiting the AST produced by SmaCCDart.

Although this paper very much describes works in progress, it encourages the idea that new languages, such as Dart, can still be analyzed by existing tools already in Pharo. We plan on releasing regular updates of our presented tools and on advancing the development of the importer, set to be released by the end of 2023. Future works will be dedicated to the evaluation of our tools and process, using open source projects on online repository to analyze Flutter project written in Dart[16]. We believe that this work can open the way to new industrial tools

[16] https://github.com/tortuvshin/open-source-flutter-apps

promoting better-analyzing software systems, thus increasing the maintainability and evolution of new software built in Dart.

## References


[1] Flutter, Flutter releases, 2018. URL: https://docs.flutter.dev/release/archive.

[2] M. De Icaza, Announcing xamarin, 2011. URL: https://web.archive.org/web/20230505135708/https://tirania.org/blog/archive/2011/May-16.html.

[3] G. Bracha, The dart programming language, 2015. URL: https://books.google.fr/books?id=UHAlCwAAQBAJ. doi:10.1109/ICSM.2010.5609731.

[4] S. Silva, A. Tuyishime, T. Santilli, P. Pelliccione, L. Iovino, Quality metrics in software architecture, in: 2023 IEEE 20th International Conference on Software Architecture (ICSA), IEEE, 2023, pp. 58–69.

[5] B. T. Niang, G. Kahn, N. Amokrane, Y. Ouzrout, M. Derras, J. Laval, Using moose platform for the implementation of a software product line according to model-based delta-oriented programming, in: IWST22—International Workshop on Smalltalk Technologies, 2022.

[6] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008.

[7] J. Brant, D. Roberts, B. Plendl, J. Prince, Extreme maintenance: Transforming delphi into c#, in: R. Marinescu, M. Lanza, A. Marcus (Eds.), 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania, IEEE Computer Society, 2010, pp. 1–8. URL: https://doi.org/10.1109/ICSM.2010.5609731. doi:10.1109/ICSM.2010.5609731.

[8] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, M. Derras, Modular moose: A new generation software reverse engineering environment, arXiv preprint arXiv:2011.10975 (2020).

[9] H. Rocha, S. Ducasse, M. Denker, J. Lecerf, Solidity parsing using smacc: Challenges and irregularities, in: Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies, IWST '17, Association for Computing Machinery, New York, NY, USA, 2017. URL: https://doi.org/10.1145/3139903.3139906. doi:10.1145/3139903.3139906.

[10] 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: S. B. Sassi, S. Ducasse, H. Mili (Eds.), Reuse in Emerging Software Engineering Practices - 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2-4, 2020, Proceedings, volume 12541 of *Lecture Notes in Computer Science*, Springer, 2020, p. 119–134. URL: https://doi.org/10.1007/978-3-030-64694-3_8. doi:10.1007/978-3-030-64694-3_8.

[11] S. Tichelaar, S. Ducasse, S. Demeyer, Famix and xmi, in: Proceedings Seventh Working Conference on Reverse Engineering, 2000, p. 296–298. doi:10.1109/WCRE.2000.891485.