Modular Moose: A new generation software reverse engineering environment

Nicolas Anquetil, Anne Etien, Mahugnon Houekpetodji, Benoit Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich, Moustapha Derras

Modular Moose: A new generation software reverse engineering environment

Accepted to ICSR2020

Nicolas Anquetil[1]✉[0000−0003−1486−8399], Anne Etien[1][0000−0003−3034−873X], Mahugnon H. Houekpetodji[1,3], Benoit Verhaeghe[1,4], Stéphane Ducasse[2][0000−0001−6070−6599], Clotilde Toullec[2], Fatiha Djareddir[3], Jerôme Sudich[3], and Moustapha Derras[4]

[1] Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France
nicolas.anquetil@univ-lille.fr
[2] Université de Lille, Inria, CNRS, Centrale Lille, UMR 9189 – CRIStAL, France
[3] CIM, Lille, France
[4] Berger-Levrault, Montpellier, France

Abstract. Advanced reverse engineering tools are required to cope with the complexity of software systems and the specific requirements of numerous different tasks (re-architecturing, migration, evolution). Consequently, reverse engineering tools should adapt to a wide range of situations. Yet, because they require a large infrastructure investment, being able to reuse these tools is key. Moose is a reverse engineering environment answering these requirements. While Moose started as a research project 20 years ago, it is also used in industrial projects, exposing itself to all these difficulties. In this paper we present ModMoose, the new version of Moose. ModMoose revolves around a new meta-model, modular and extensible; a new toolset of generic tools (query module, visualization engine, ...); and an open architecture supporting the synchronization and interaction of tools per task. With ModMoose, tool developers can develop specific meta-models by reusing existing elementary concepts, and dedicated reverse engineering tools that can interact with the existing ones.

1 Introduction

As software technologies evolve, old systems experience a growing need to follow this evolution. From the end-user point of view, they need to offer functionalities entirely unforeseen when they were first conceived. From the developer point of view, they need to adapt to the new technologies that would allow one to implement these functionalities [8].

Given the size of these systems and lack of resources in the industry, such evolution can only happen with the help of automated tooling [3, 15]. Concurrently, because such tooling requires a large infrastructure investment, it must be generic and reusable across technologies and reverse engineering tasks. This tooling needs to cope with the following problems:

– *Diversity of languages and analyses.* Many programming languages and versions of such languages are used in the industry. Meta-modeling was proposed to cope with that diversity, but it does not solve all problems. Software reverse engineering requires to represent source code with a high degree of details that are specific to each programming language and to the reverse engineering tasks themselves. How to model the different needs for details while remaining generic is an issue meta-models have not tackled yet.
– *Sheer amount of data.* Tools are particularly useful for large systems (millions of lines of code). The size of the analyzed systems stresses the modeling capabilities of the tools and the efficiency of the analyses.
– *Reverse engineering task diversity.* The evolution needs are numerous, from restructuring a system towards a micro component architecture [5], to migrating its graphical user interface [24], evolving its database [7], or cleaning the code [2]. This calls for various tools (query module, visualizations, metrics, navigation, analyses) that must integrate together to answer each need still acting as a coherent tool.
– *Specific tasks require specific tools.* Some tools can be useful in various situations, but specialized tools are also needed[5]. If a tool is too specific to a technology, its advantages are lost when working with others. It is important that they can be easily added and integrated into a reverse engineering environment.

We report here our experience with Moose, a software analysis platform answering to the basic needs of reverse engineering, reusable in different situations, extensible and adaptable to new needs [21]. Moose was initiated as a research project and still fulfills this role. But it is also used in a number of industrial projects [2, 7, 24].

Our redesign of Moose has the following goals: (1) the ability to develop specific meta-models by reusing existing elementary concepts, (2) the ability to develop dedicated reverse engineering tools by configuring and composing existing tools or developing new ones, (3) the ability to seamlessly integrate new tools in the existing reverse engineering environment. ModMoose, the new version of Moose described in this article, revolves around a modular extensible meta-model, a new toolset of generic tools (*e.g.,* query module, visualization engine) and an open architecture supporting the synchronization and interaction of tools.

The contributions of this article are (1) the description of ModMoose new generation reverse engineering environment, (2) FamixNG supporting the definition of new meta-models based on the composition of elementary concepts implemented using stateful traits, and (3) a bus architecture to support the communication between independent and reusable tools.

In the following sections, we discuss the difficulties of having a generic, multi-language, software meta-model (Section 2) and how we solved them with FamixNG (Section 3). Then, we briefly present the new generic tools and how

[5] According to Bruneliere [6], the plurality of reengineering projects requires adaptable/portable reverse engineering tools.

the new open architecture of ModMoose supports the collaboration of group of tools and their synchronization (Section 4). We conclude in Section 5.

2 Software Representation and Reverse Engineering Tool Challenges

To analyze code, tools need to represent it. Such a representation should be *generic* (programming languages are all build around similar concepts that should be reusable), support *multi-language*, and *detailed* (a proper analysis must pay attention to little, meaningful, details).

Each one of these challenges may be tackled by meta-models. For example, existing IDEs (Integrated Development Environments) like Eclipse, or software quality tools (*e.g.,* CAST, SonarQube) offer tools based on an internal representation of the source code. But in Eclipse, the model is not the same for version management and source code highlighting. In that way, they can represent with enough details the analyzed systems. The Lightweight Multilingual Software Analysis approach proposed by Lyons *et al.,* uses different parsers (one for each programming language analyzed) to populate their *different* models: "variable assignment locations", or "reverse control flow". The challenge is still how to support the development of multiple dedicated and specific meta-models for different languages in a tractable way.

2.1 Problem: Handling the Diversity of Languages

Many publications highlight the need, for modern software analysis tools, to deal with multi-language systems (*e.g.,* [9, 13, 18–20]). The generality and reusability of the analysis tools depend on their ability to work with all these languages.

Meta-models were supposed to solve this issue. Naively, one could hope that a single "object-oriented meta-model" would be able to represent programs in all OO programming languages and another "procedural meta-model" would represent all procedural programming languages.

Thus, for example, in the Dagstuhl Middle Meta-model [18] "the notions of routine, function and subroutine are all treated the same." But in practice, one soon realizes that "various programming languages have minor semantic differences regarding how they implement these concepts" ([18] again.) To be meaningful and allow precise analyses, the meta-models must represent these little differences and the tools must "understand" their specific semantics. As a consequence, Washizaki *et al.,* [25] report that "even if a meta-model is stated to be language independent, we often found that it actually only supports a very limited number of languages." A good meta-model must represent all details to allow meaningful analyses. Software is not linear, small details may have huge impacts. Reverse engineering is about abstracting from the source code, but also sticking tightly to it because one does not know in advance what "details" are making a difference for different tasks.

Visibility rules are a concrete example of the complexity to represent varying details for apparently universal concepts: Abstracting the API of a class requires understanding the visibility rules of classes, methods, and attributes. They are not exactly the same for Java (public, protected, private, default package, and now export) or C++ (public, protected, private, and friend). And the tools need to be aware of each different semantics to make the proper inferences.

Other problems are raised by the containment relationship (*i.e.,* ownership). Many meta-models assume a hierarchical containment tree: all entities (except the root entity) are owned by exactly one parent entity. However, in C#, class definitions may be scattered over several files (*partial* definitions), and in Objective-C or Smalltalk a package may add methods to a class defined in another package (extension methods). In these contexts, the notion of single owner is less consensual.

Our experience designing and using a programming language meta-model, Famix, for more than 15 years showed that we need dedicated meta-models for each language.

2.2 Single inheritance: Tyranny of the dominant decomposition

Moose was based on a single meta-model (Famix [10, 11, 21]) extensible with plugins. It did allow to model many different programming languages[6] (Ada, C, C++, FORTRAN, 4D[7], Java, JavaScript, Mantis[8], Pharo, PHP, PostgresSQL/PgPlSQL, PowerBuilder[9]), and even other sources of information (*e.g.,* SVN logs). But extending Famix was an awkward process since it relied on a single inheritance tree of concepts.

To address language variety, the choice was to have core entities representing "typical features" of programming languages (*e.g.,* function, methods, package, class) and extend this core for language specific entities (*e.g.,* the macros in C). The Dagstuhl Middle Meta-model [18] made the same choice.

However this was only possible through a careful definition of the meta-model and the hierarchy of modeled entities. In practice, Famix core was tailored towards Java and Smalltalk, its initial focus. Several extensions covered other OOP languages with varying degrees of success. For example, Enum and Class were both Types (see Figure 1, right). In Java, both accept an implements relationship to Interface (another type). Therefore this relationship is defined at the level of the Type concept. But then, a PrimitiveType (also a Type) inherits this same property which does not make sense in Java, and also an Interface is allowed to implements another Interface, which again is not correct in Java.

Other generic meta-models have the same issue. In the Dagstuhl Middle Meta-model [18] (see Figure 1, left), all relationships (*e.g.,*IsMethodOf, Includes, Invokes) may occur between ModelElements (*e.g.,*Type, Routine, Field). As a result

[6] Some languages are only partially supported

[7] `http://en.wikipedia.org/wiki/4th_Dimension_(software)`

[8] `http://www.lookupmainframesoftware.com/soft_detail/dispsoft/339`

[9] `http://en.wikipedia.org/wiki/PowerBuilder`

many entities (*e.g.,*Field) accept relationships (*e.g.,*IsMethodOf) that they don't use.

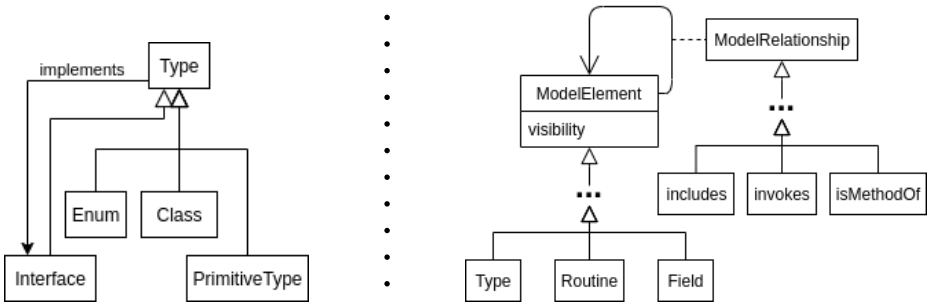

Fig. 1. Examples of the Dominant Decomposition problem in the Dagstuhl Middle Meta-model (left) and in Famix (right)

A generic single inheritance meta-model thus turns out to be too permissive with entities owning properties that they don't use. Incidentally, this has consequences on the size of the models since these properties occupy uselessly memory that multiplied by the number of instances (tens or hundreds of thousands) adds up to significant memory loss.

EMF (Eclipse Modeling Framework) attempts to solve the problems raised by single inheritance inheritance tree. EMF implementation uses inheritance and interfaces, thus mimicking multiple inheritance. But this still imposes to choose one dominant decomposition for inheritance and the other decompositions have to implement interfaces' API.

2.3 ModMoose's Goals

To design a new generation and modular reverse engineering environment we set goals at two different levels: meta-modeling and tooling.

Meta-modeling goals. We are looking for generic modeling that leaves open a wealth of analyses (metrics, dead-code, design patterns, anti-patterns, dependency analysis, etc). So we needed a unified representation that:

1. Supports the precise representation of various programming language features (identified in [22]);
2. Is compatible with many analysis tools (the *processing environment* identified in [25]);
3. Is wary of memory consumption (identified in [17]);
4. Is extensible (allow for easy addition of new languages).

Tooling goals. Tools are needed to [3, 15]:

- Automate tedious tasks to bring tremendous speed-up with less errors.[10]
- Handle large quantity of data and in our case all the details of multi-millions lines of code. If information size can become an issue even for automated tools, this is several orders of magnitude above what a human can handle.
- Summarize data to allow a human abstracting a big picture understanding.
- Quickly verify hypotheses so that there is little or no cost attached to wrong hypotheses, and several concurrent hypotheses can easily be compared.

3 A Composable Meta-model of Programming Languages

FamixNG is a redesign of Famix around the idea of composing entities from language properties represented as traits [23]. With FamixNG (See Section 3.1), one defines a new meta-model out of reusable elements describing elementary concepts (See Section 3.2).

3.1 FamixNG

To support the reuse of elementary language concept, FamixNG relies on *stateful traits* [23].

What is a trait? "A trait is a set of methods, divorced from any class hierarchy. Traits can be composed in arbitrary order. The composite entity (class or trait) has complete control over the composition and can resolve conflicts explicitly". In their original form, traits were stateless, *i.e.,* pure groups of methods without any attributes. Stateful traits extend stateless traits by introducing a single attribute access operator to give clients of the trait control over the visibility of attributes [23]. A class or another trait is composed from traits and it retains full control of the composition being able to ignore or rename some methods of the traits it uses.

In FamixNG, all properties that were previously defined in classes are now defined as independent traits: For example, the trait TNamedEntity only defines a name property (a string) and may belong to trait TNamespaceEntity. Therefore any entity using this trait will have a name and can be part of a namespace entity. Similarly, entities (presumably typed variables or functions) composed with the trait TTypedEntity have a declaredType pointing to an entity using the trait TType.

Four types of traits can be used to compose a new meta-model:

Associations represent usage of entities in the source code. This includes the four former associations of Famix: inheritance, invocation (of a function or

[10] A source code model computes in seconds a method call graph that takes weeks to recover by hand. We had the case in two different companies.

a method), access (to a variable), and reference (to a type). In FamixNG, we also found a need for three more specialized associations: DereferencedInvocation (call of a pointer to a function in C), FileInclude (also in C), and TraitUsage.

Technical traits do not model programming language entities but are used to implement Moose functionality. Currently, this includes several types of TSourceAnchors, associated to TSourcedEntity to allow recovering their source code (a typical TSourceAnchor contains a filename, and start and end positions in this file.) Other *Technical traits* implement software engineering metric computation, or are used to implement the generic query engine of Moose (see Section 4.2). There are 16 *Technical traits* currently in FamixNG.

Core traits model composable properties that source code entities may possess. This includes TNamedEntity and TTypedEntity (see above), or a number of entities modeling ownership: TWithGlobalVariables (entities that can own TGlobalVariables), TWithFunctions (entities that can own TFunctions),... There are 46 *Core traits* currently in FamixNG including 38 traits modeling ownership of various possible kind of entities.

Terminal traits model entities that can be found in the source code such as Functions, Classes, Exceptions, ... These entities are often defined as a composition of some of the *Core traits*. For example, TClass is composed of: TInvocationsReceiver (class can be receiver of static messages), TPackageable, TType (classes can be used to type other entities), TWithAttributes, TWithComments, TWithInheritances, TWithMethods. The name *Terminal trait* refers to the fact that they can be used directly to create a programming language concept (a class, a package), whereas *Core traits* are typically composed with other traits to make a meaningful programming language concept. There are 38 such *Terminal traits* currently in FamixNG.

Contrary to the old Famix, the *Terminal traits* are rather minimal definitions (intersection of languages) than maximal ones (union of languages). They are intended to have only the properties found in any programming language. For example TClass does not use the TWithModifiers trait (for attaching visibility) since not all languages explicitly define visibility of classes. For any given meta-model, additional properties may be added to these "*Terminal traits*", for example JavaClass uses the traits TClass and TWithModifiers since Java classes can be declared public, private, ... More specialized languages (*e.g.,* SQL) can always compose new entities from the set of *Core traits* offered by FamixNG.

We validated this new approach by defining different meta-models (*e.g.,* GUI meta-model [24], Java, OpenStack cloud infrastructure, Pharo, PowerBuilder, SQL/PLPGSQL).

3.2 Creating a new Meta-model with FamixNG

To create a new meta-model, the developer may extend an existing meta-model or start from scratch. This later case can be relevant if the language to model is quite different from what ModMoose currently handles, such as a specific

domain or SQL. For example, while being different from procedural languages, stored procedures in SQL bear resemblance to functions and can be composed from the same FamixNG traits.

Figure 2 shows the definition of a toy meta-model (left) with the meta-model itself (right). There are four entities: the root Entity, and File, Commit and Author. In grey, the TNamedEntity trait, part of FamixNG. The embedded DSL needs first the entity creation, then definition of inheritance relationships, properties of each entity, and finally the relationships between entities. To improve readability, entity descriptions are stored in variables having the same name (*e.g.,* author variable for Author entity description). One can see an example of using a predefined trait, TNamedEntity that adds a "name : String" property to the classes using it. The same syntax, "--|>", is used for class inheritance and traits usage. From this, the builder generates all the corresponding classes and methods.

```
entity := builder newClassNamed: 'Entity'.
file := builder newClassNamed: 'File'.
commit := builder newClassNamed: 'Commit'.
author := builder newClassNamed: 'Author'.

"inheritance"
file --|> entity.
file --|> TNamedEntity.
commit  --|> entity.
author  --|> entity.
author --|> TNamedEntity.

"properties"
commit property: 'revision' type: #Number.
commit property: 'date' type: #Object.
commit property: 'message' type: #String.

"associations"
file *-* commit.
commit *- author.
```

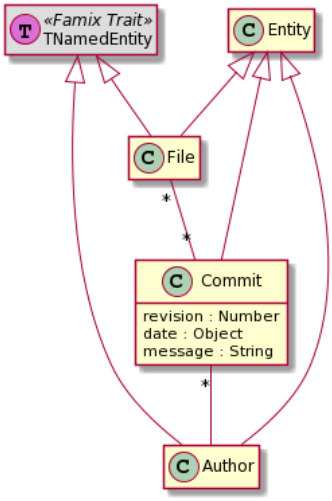

Fig. 2. A simple File/Commit/Author meta-model (left: building script; right: UML view; gray: predefined FamixNG trait)

Meta-model reuse and extension. The extension of existing meta-model is done by specifying which meta-model to extend (similarly to import instruction in Java), then the existing traits can be reused at will. It is also possible to manage multi-language systems by composing several meta-models. We cannot illustrate these two points for lack of space.

4 ModMoose: A Reverse Engineering Environment

Program comprehension is still primarily a manual task. It requires knowledge on computer science, the application domain, the system, the organization using it, etc [1]. Yet the sheer size of the current software systems (millions of lines of code, hundred of thousands of entities modeled) precludes any software development team to fully understand these systems. Any significant program comprehension activity imposes the use of specialized tools.

ModMoose lets software engineers design specialized reverse engineering tools by (1) using infrastructure tools (see Section 4.2); (2) taking advantage of a bus architecture supporting smart interactions between tools (see Section 4.3); and, (3) reuse a set of generic and configurable tools (see Section 4.4).

4.1 ModMoose Architecture

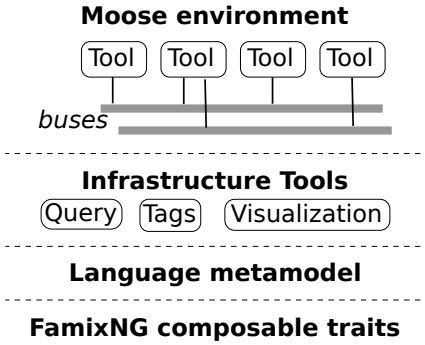

Fig. 3. ModMoose architecture.

ModMoose is architectured on the following principles:

- Tools are part of the ModMoose environment which acts as a master and centralizes data;
- Tools communicate through buses, they "read" model entities on their bus(es) and "write" entities back on their bus(es) (see below);
- Tools are focusing on a single task: *e.g.,* the Query Browser works on a set of model entities and produces another set of entities.

These simple principles ensure that tools can be easily added to ModMoose and collaborate between themselves in a flexible manner.

4.2 Infrastructure Tools

We identified three important requirement to analyze software systems: (i) query and navigate a model to find entities of interest; (ii) visualise the software to

abstract information; and (iii) annotate entities to represent meaning and reach a higher level of abstraction.

Moose Query is an API to programmatically navigate FamixNG models. For any FamixNG meta-model, Moose Query[11] computes the answer to generic queries:

- containment: parent or children entities of a given type from a current entity.
- dependencies: following incoming or outgoing associations from or to a given entity. Dependencies are also deduced from parent/children relationships, *e.g.,* dependencies between methods can be uplifted to dependencies between their parent classes or parent packages.
- description: all properties of a given entity.

In a FamixNG meta-model, relationships denoting a containment are declared as such in the association part of the meta-model construction (see Figure 2), *e.g.,* `method *-<> class` or `class <>-* method`. Containment queries may go up and down the containment tree based on the expected entity type: *e.g.,* from a Method one can ask to go up to any entity(ies) of type Class owning it.

Navigating dependencies is based on containment and association relationships. It is possible to navigate through a specific association (*e.g.,* "all invoked methods") or all types of association (*e.g.,* "all dependencies") to other elements. Direction of navigation (incoming, outgoing) must be specified. Dependencies between children of entities can be automatically abstracted at the level of their parents.

Package level communication example. A frequent query is to find how packages interact with each other via method calls. This can be done by iterating over all the methods of all the classes of a package, collecting what other methods they invoke. From this one finds the parent classes and parent packages of the invoked methods.

With Moose Query, such a query is simply expressed:
`(aPackage queryOutgoing: FamixInvocation) atScope: FamixPackage` (*i.e.,* find all invocations stemming from `aPackage`, and raise the result at the level of the receiving package).

Stored procedures referencing a column example. In SQL, one may want to know all the stored procedures[12] accessing a given column of a database table. These stored procedures can directly reference columns, in the case of triggers, or contain SQL queries that reference the columns. SQL queries contain clauses that themselves can contain other queries. So, the answer can be computed from a given column of a table, by collecting the references targeting this column, and recursively analyze them to identify the ones that are inside a stored procedure. Due to the possible nesting of SQL queries, this has to be a recursive process.

[11] `https://moosequery.ferlicot.fr/`
[12] functions directly defined inside the database management system

With Moose Query, this query is simply expressed as follows:
`aColumn queryIncomingDependencies atScope: FamixSQLStoredProcedure` (*i.e.,* find all incoming dependencies to `aColumn` and raise the result at the level of the stored procedures. Incoming dependencies not stemming from a stored procedures are simply dropped here).

Analysis. These examples show that MooseQuery API is independent from the meta-model used. Obviously, each query depends on the meta-model of the model it is applied (one cannot ask for methods in a SQL model). In the first example (Java), there are different kind of dependencies in the model (invocation, inheritance, access, reference), therefore the kind one is interested in must be specified (`queryOutgoing: FamixInvocation`). In the second example there exist only one kind of dependencies to a column. Consequently, the query can be simpler: `queryIncomingDependencies`.

In the second example, Moose Query implicitly filters out all queries that do not stem from a stored procedure. In the first example, no such filtering occurs since all methods belong to a package in Java.

Visualisation Engines: ModMoose uses Roassal [4], a generic visualization engine, to script interactive graphs or charts. Roassal is primarily used to display software entities in different forms or colors and their relationships. Possible examples are to display the classes of a package as a UML class diagram, or as a Dependency Structure Matrix[13]. Roassal visualization are interactive.

ModMoose also uses the Telescope, more abstract, visualization engine that eases building new visualizations in terms of entities, their relationships, their forms and colors, positioning, etc [16]. Telescope offers abstractions such as predefined visualizations and interaction, and relies on Roassal to do the actual drawing. For example, Telescope offers a predefined "Butterfly" visualization, centered on an entity and showing to the left all its incoming dependencies and to the right all its outgoing dependencies.

Tags are labels attached by the user to any entities either interactively or as result of queries [14]. Tags enrich models with extra information. They have many different uses:

- to decorate entities with a virtual property *e.g.,* tagging all methods directly accessing a database.
- to represent entities that are not directly expressed in the programming language constructs: *e.g.,* representing architectural elements or tagging subsets of a god class' members as belonging to different virtual classes [2].
- to mark *to do* and *done* entities in a repetitive process.

An important property of ModMoose tags is that they are first class entities. Tags are not only an easy way to search entities, they can also be manipulated as any other model entity: a query may collect all dependencies from or

[13] `https://en.wikipedia.org/wiki/Design_structure_matrix`

to a tag, the visualization engines may display tags as container entities with actual entities within them, one can compute software engineering metrics (*e.g.,* cohesion/coupling) on tags considered as virtual packages.

4.3 Smart Tool Interactions through Buses

The power of MODMOOSE comes from the generic interaction of specialized tools. In particular, it is key that: (i) different tools display the *same entities* from different points of view, or (ii) different instances of the same tool may be used to compare different entities from the same point of view. MODMOOSE supports such scenarios using an open architecture based on buses and tool behavior controls.

Communication Buses. Tools communicate through buses. They read entities produced by other tools on the buses and write entities on the same buses for other tools to consume. Several buses can be created, and groups of tools can be set on different buses. This allows one to explore different parts of the model or different courses of actions: *e.g.,* imagine two buses each one attached to a Query browser and a Dependency Graph browser (described in Section 4.4). Each Query browser selects different entities from the model, the Dependency Graph browsers display them, and the two buses ensure interaction within each pair of browsers.

Tools can also be detached from all available buses to keep their current result and presumably allow one to compare it with other results from other instances of the same tools.

Since tools can be attached to more than one bus, a tool can be set as a bridge between all buses: It listens to all buses and forwards activity on them. A natural candidate for this is the Logger (described in Section 4.4). By selecting an interesting set of entities in the Logger, one can propagate these entities to all buses, thus synchronizing them.

Tool Behavior Controls. On top of the bus architecture, three behavior controls (*frozen*, *highlighting*, *following*) fine-tune the tool reaction to bus events:

- *Following* is the default state where a tool reads entities that pass on its bus(es) and writes entities to the same bus(es). In such mode, a tool reacts to read entities.
- *Frozen* is a state where the tool does not listen to the bus(es) for incoming entities and therefore keeps its state and display independent of entities written on the bus(es). The tool is not detached from bus(es) and can still write entities on them. A reverse engineer can interact with a frozen tool and other tools will be informed of this.
- *Highlighting* is a sort of intermediary state, where the current entities on which the tool works remain the same (similar to *Frozen*), but when new entities pass on the bus(es), the tool looks for them in its "frozen" display and highlight them if they are present (they are ignored otherwise). This is

useful to highlight a new result in the context of a preceding result that was already displayed in the tool.

4.4 Specialized Reusable Tools

On top of the modular meta-model (Section 3) and infrastructure tools (Section 4.2), MODMOOSE offers some specialized tools that answer different recurring needs in software analysis.

There is a number of reusable tools already implemented, among which:

Model Browser: Imports or creates new models of given software systems, selects a model, and browses the model entities. It outputs the selected model. This is the entry point for working with the environment.

Entity Inspector: Lists all properties of a selected entity and their values (such as metrics, tags and others). If the input is a group of entities, it lists all their common properties. It allows navigating the model when the value of a property is another entity (*e.g.,* the methods property of a Class). It outputs any entity selected for navigation.

Query Browser: Offers a graphical interface to the Query module (Section 4.2). It works on the entities found on the bus(es) and can filter and/or navigate from these entities. It outputs the result of the queries.

Dependency Graph Browser: Shows as a graph all direct incoming and outgoing dependencies of a group of entities (see Figure 5, right). Entities are selectable and are written on the bus for other tools to read them allowing interactive navigation.

Duplication Browser: Computes and displays duplication in the source code of a group of entities. In Figure 4, left, the big boxes are software entities (typically methods), and the small squares are the duplication fragments found in them. Each duplication fragment has a color to identify it in all its occurrences (in the various software entities). Entities are selectable and can be output on the bus(es).

Source Code: Displays the source code of an input entity. If there are several entities in input or an entity with no source code (a model for example), does not do anything. Currently produces no output.

Logger: Records all entities (individually or in a group) that pass on a bus. It lets the user come back to a previous stage by selecting the entities that were produced at that stage. It can export entities in files (txt, csv). It outputs any selected entity or group of entities.

4.5 Creating a new Tool

New tools can be added and are planned, for example, a metric dashboard with standard metrics on the set of entities given in input, or new software maps such a Distribution Map visualisation [12].

Since tools are specialized, they are easy to develop, particularly with the help of the visualization engines and the query module. The open architecture with the buses makes the integration of new tools smooth and easy.

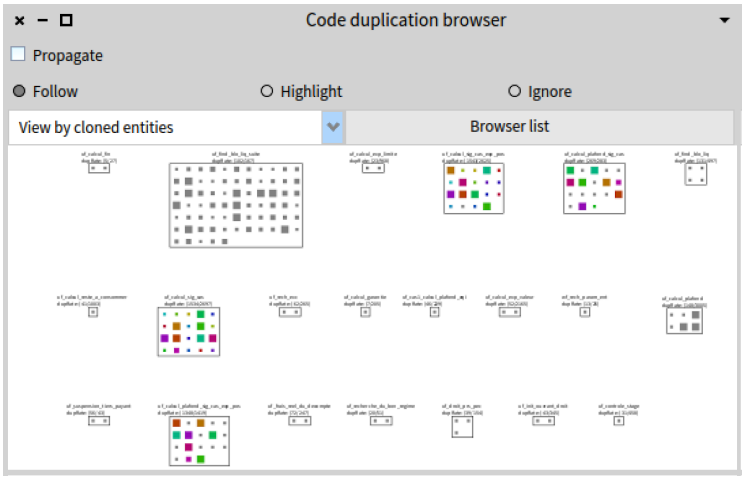

Fig. 4. MODMOOSE example of specialized tools: Duplication browser.

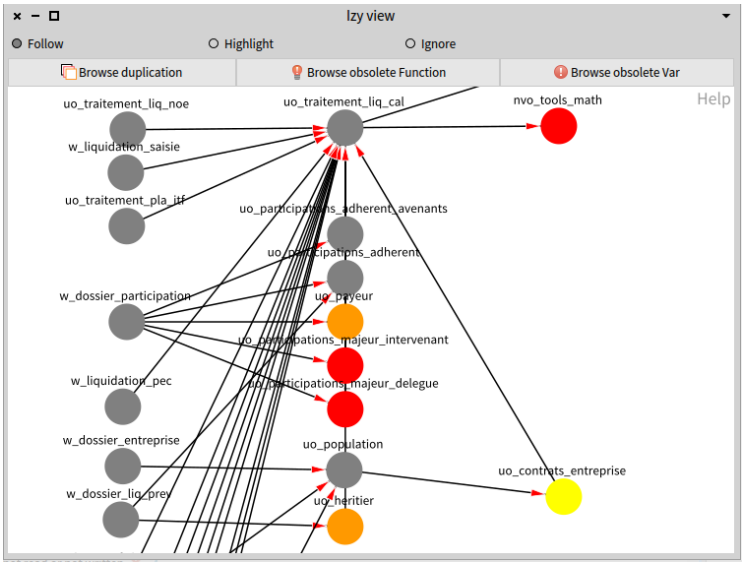

Fig. 5. MODMOOSE example of specialized tools: Call graph browser.

5 Conclusion

From literature and our industrial experience, we identified key aspects that a program comprehension environment must fulfill: a meta-model easily extensible and adaptable to represent new programming languages or sources of information; interoperating tools that can be adapted to the comprehension task at hand.

In this paper, we presented the new architecture of MODMOOSE, our reverse engineering environment. It is first based on FAMIXNG a new way to express meta-models by composing new entities from a set of traits, each describing individual properties that are generally encountered in programming languages. We developed *infrastructure* tools to manipulate models and an architecture with specialized end-user tools that interact through information buses.

This is not the end of the road. We will complete our tool suite to respond to other aspects of reverse engineering and re-engineering. Two research directions are drawing our attention: (1) how to more easily develop importers for new programming languages; (2) how to generate new code from the models in any programming language for which we have a meta-model.

References

1. N. Anquetil, K. M. de Oliveira, K. D. de Sousa, and M. G. Batista Dias. Software maintenance seen as a knowledge management issue. *Information Software Technology*, 49(5):515–529, 2007.
2. N. Anquetil, A. Etien, G. Andreo, and S. Ducasse. Decomposing God Classes at Siemens. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2019.
3. B. Bellay and H. Gall. An evaluation of reverse engineering tools. *Journal of Software Maintenance: Research and Practice*, 1998.
4. A. Bergel. *Agile Visualization*. LULU Press, 2016.
5. S. Bragagnolo, N. Anquetil, S. Ducasse, S. Abderrahmane, and M. Derras. Analysing microsoft access projects: Building a model in a partially observable domain. In *International Conference on Software and Systems Reuse, ICSR2020*, dec 2020. in submission.
6. H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014.
7. J. Delplanque, A. Etien, N. Anquetil, and S. Ducasse. Recommendations for evolving relational databases. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 12127 of *LNCS*, pages 498–514. Springer, 2020.
8. S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
9. S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings of the International Conference on The Unified Modeling Language (UML'99)*, volume 1723 of *LNCS*, pages 630–644, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

10. S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In *International Conference on The Unified Modeling Language (UML'99)*, volume 1723, pages 630–644. Springer, 1999.
11. S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.
12. S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, ICSM'06, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.
13. A. Egyed and N. Medvidovic. A formal approach to heterogeneous software modeling. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 178–192. Springer, Berlin, Heidelberg, Mar. 2000.
14. B. Govin, N. Anquetil, A. Etien, S. Ducasse, and A. Monegier Du Sorbier. Managing an Industrial Software Rearchitecting Project With Source Code Labelling. In *Complex Systems Design & Management conference (CSD&M)*, Paris, France, Dec. 2017.
15. H. M. Kienle and H. A. Müller. The tools perspective on software reverse engineering: Requirements, construction, and evaluation. In *Advanced in Computers*, volume 79, pages 189–290. Elsevier, 2010.
16. G. Larcheveque, U. Bhatti, N. Anquetil, and S. Ducasse. Telescope: A high-level model to build dynamic visualizations. In *International Workshop on Smalltalk Technologies (IWST'15)*, 2015.
17. J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming (SCP)*, 76(12):1177–1193, May 2011.
18. T. Lethbridge, S. Tichelaar, and E. Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.
19. D. M. Lyons, A. M. Bogar, and D. Baird. Lightweight Multilingual Software Analysis. *Challenges and Opportunities in ICT Research Projects*, 2018.
20. P. Mayer. A taxonomy of cross-language linking mechanisms in open source frameworks. *Computing*, 99(7):701–724, July 2017.
21. O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In M. Wermelinger and H. Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. Invited paper.
22. A. Shatnawi, H. Mili, M. Abdellatif, Y.-G. Guéhéneuc, N. Moha, G. Hecht, G. E. Boussaidi, and J. Privat. Static Code Analysis of Multilanguage Software Systems, June 2019. arXiv: 1906.00815.
23. P. Tesone, S. Ducasse, G. Polito, L. Fabresse, and N. Bouraqadi. A new modular implementation for stateful traits. *Science of Computer Programming*, 2020.
24. B. Verhaeghe, A. Etien, N. Anquetil, A. Seriai, L. Deruelle, S. Ducasse, and M. Derras. GUI migration using MDE from GWT to Angular 6: An industrial case. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.
25. H. Washizaki, Y.-G. Gueheneuc, and F. Khomh. A Taxonomy for Program Metamodels in Program Reverse Engineering. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 44–55, Raleigh, NC, USA, Oct. 2016. IEEE.