Alce: Predicting Software Migration

Santiago Bragagnolo, Stéphane Ducasse, Nicolas Anquetil, Abderrahmane Seriai, Mustapha Derras

Alce: Predicting Software Migration

1st Santiago Bragagnolo
Université de Lille, CNRS, Inria
Centrale Lille,UMR 9189 – CRIStAL
Berger-Levrault
Lille, France
santiago.bragagnolo@berger-levrault.com

2nd Stéphane Ducasse
Université de Lille, CNRS, Inria
Centrale Lille,UMR 9189 – CRIStAL
Lille, France
stephane.ducasse@inria.fr

3rd Nicolas Anquetil
Université de Lille, CNRS, Inria
Centrale Lille,UMR 9189 – CRIStAL
Lille, France
nicolas.anquetil@univ-lille.fr

4th Abderrahmane Seriai
Berger-Levrault
Perols, France
abderrahmane.seriai@berger-levrault.com

5th Mustapha Derras
Berger-Levrault
Paris, France
mustapha.derras@berger-levrault.com

***Abstract*—The constant apparition of new technologies challenging and disrupting the way to develop software pushes day-by-day software migration to become more and more common.**

Despite the "normality" of software migration, it is a problem that had ruined more than one company in the past. It is no wonder that different methods to migrate software have been the driver of many efforts and the centre of many discussions for years, resulting in multiple solutions and strategies to accomplish the desired migration.

However, there is a lack of efforts on how software reengineering can be used to assess the process of planning by measuring and predicting the cost of a migration.

In this article, we present Alce, a software migration assessment and prediction tool under development in the context of a collaboration with Berger-Levrault, for migrating Microsoft Access applications. We present a simple use case that represents most of the usages we had given to the tool during the analysis and reporting of two different applications to be migrated, to assess the extremely hard task of planning a software migration. We present as well a second use for task definition and prioritisation in the process of library migration. We discuss future features based on the interaction with one of the project managers, and finally, we discuss the lack of software reengineering tools usage in the context of software migration.

***Index Terms*—Software Migration, Software Analysis, Migration Planning Assessment, Reverse Engineering**

I. Introduction

In the context of a collaboration with Berger-Levrault, a major IT company, we are working on the migration of Microsoft Access monolithic applications to web front-end and microservices back-end. The constraint of the migration project are those often encountered in the industry: The software quality is uncertain, development must continue during the migration, resources are limited, the desired target architecture is sketchy, developers are yet to fully master the target technology, managers do not fully understand the magnitude of the enterprise. Migration is a daunting task that had sank entire companies in the past [7].

We started a research project two years ago to propose tools and processes to migrate more than 90 applications of different sizes to web technologies such as Java+SpringBoot and Typescript+Angular. To give an idea of the magnitude, the two projects we have analyzed have 700k/900k lines of code and 1000/2700 UI widgets.

Shortly after starting the research project, we had a first meeting with the key developers and managers of the teams working on those projects. In this first meeting, the first question that pops up was: "How much time is it going to take to migrate this application?". We cannot answer this question. However, we are able to provide enough data to allow team leaders and managers to estimate it based on arguments and understanding of the dimensions of the projects.

For achieving this task we read and discussed how to understand software modernization [3], We learned and proposed how to measure the complexity of a migration [4]. We learned how to do reverse engineering for Microsoft Access [2] in order to be able to produce a Famix model [1] for Microsoft Access (https://github.com/impetuosa/alce), what allow us to use and extend the Moose(https://modularmoose.org/) platform. With all this set, we extended and adapted Moose platform for developing a software migration assessment tool.

Section II presents briefly the challenges of our specific migration scenario. Section III presents Alce: a Moose-based analysis tool for predicting migration complexity. Section IV presents the supported metrics, their visualizations and what they reveal of the migration. Section V presents the architectural visualizations and how the architecture constraints the migration. Section VI presents the process used during the reporting and building of a project reporting for migration planning of the *ePaie* project. Section VII presents a study on assessing the definition of prioritisation of the tasks related to third-party artifact replacement in the migration of the *ePaie* project. Section VIII we discuss future work based on enabling comparative analysis as proposed by a manager. Section IX argues that reverse engineering is underrated and undervalued in the context of software migration and planning. Section X Resume and concludes the article.

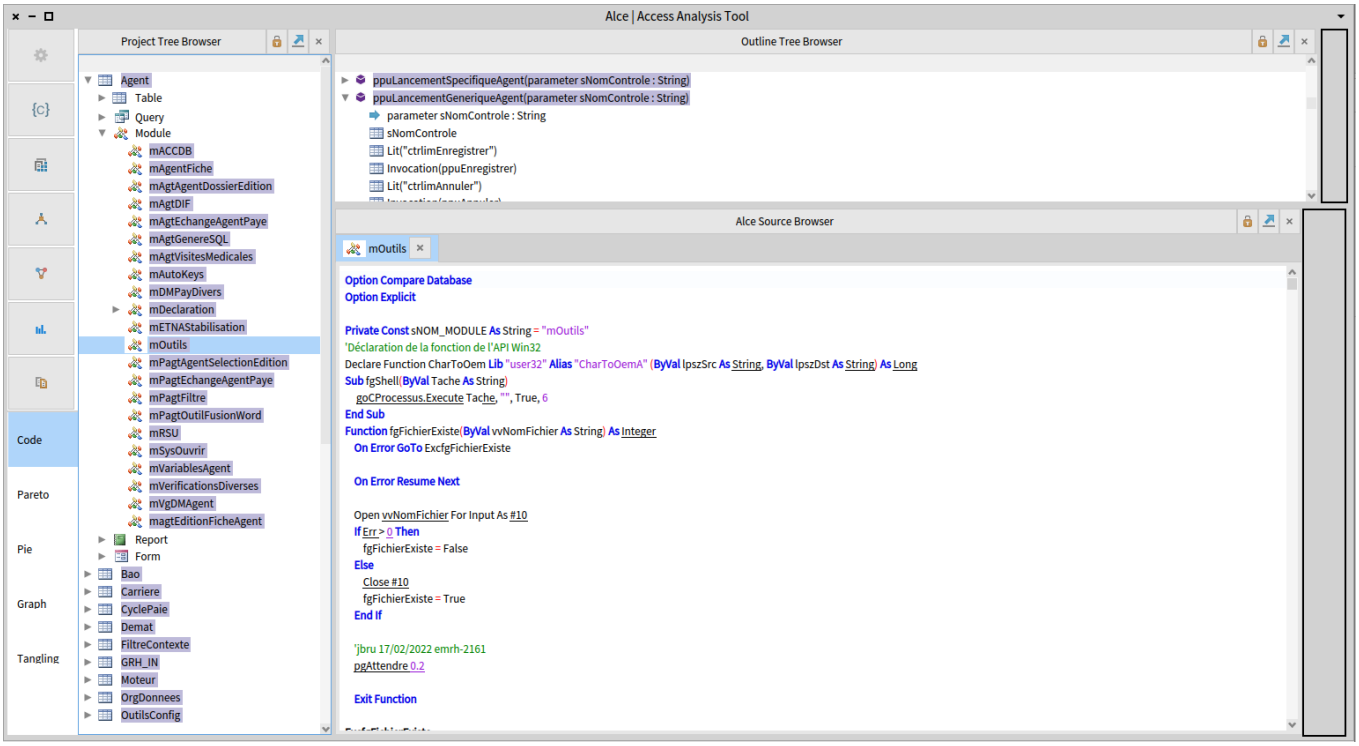

Fig. 1. Tool Screenshot: From the left to the right, Toolbar, Project Tree Browser (to navigate the project), Outline (To navigate the elements defined within the selected element), Source Browser (to read and navigate the source code of the selected element).

II. Challenges

The migration project from a monolithic Microsoft Access application to a front-end (Typescript + Angular) and back-end (Java + SpringBoot) application, entails the following challenges:

- Programming language migration: From Visual Basic for Applications (VBA) to multiple targets: HTML, TypeScript, and Java; This urges us to understand what are the differences between the languages and paradigms and to measure how much is our code affected by these differences.
- Library migration: E.g, The "long" type usages in Microsoft Access must be migrated as "BigInteger" in Java or "bigint" in Typescript. MsgBox function usages are transformed for using "alert" in Typecript or a Log4j "logger" in Java with SpringBoot; This urges us to understand and measure what libraries are used and how.
- Architectural Migrations: The migration must (i) split a stand-alone to front-end/back-end and (ii) split concerns from monolithic software to micro-services. This urges us to understand and measure how different architectural concerns interact in our code.

III. Alce Analysis Tool

A migration project like ours involves a large project and a large process. Therefore we argue that stakeholders require assistance to understand both the source project and the migration process. This process is tightly related to the source project, the target technology and the stakeholders expectations. We claim that analysing a project to understand the cost and complexities of migration requires more than just analysing the project as we are used for software maintenance [4]. For that, we implemented Alce Analysis Tool [1]. Figure 1 depicts the general view of our tool. You can check out a running version of our tool set up for the Microsoft Access Northwind traders project at https://github.com/impetuosa/alce-tool-northwind.

A. The ePaie project

ePaie is Microsoft Access project that consists of 18 sub-projects, summing up 1000 UI widgets and 700k lines of code and using 19 different libraries. The target technology of the migration is Java+Springboot for the back-end and Typescript+Angular for the front-end. *Agent* is one of the 18 different sub-project of *ePaie*. All the examples used in this article come from the same source: the *Agent* sub-project.

B. Tool presentation

We implemented a tool responding to this requirement to help Berger-Levrault in its decision-making process during migration. This tool leverages already existing Moose infrastructure and know-how to put together visualizations that helped us to understand the complexity of migrating projects.

On the left of Figure 1, we see a toolbar that lets the user choose a specific visualization. Once the visualization is open

[1]https://github.com/impetuosa/alcides

we can either observe it on the spot or deck it into the right part of our tool. On the left of Figure 1, after the toolbar, we find the Project Tree Browser. This widget allows the user to navigate and contextualize the opened visualizations. Selecting an item in this widget triggers the update of all the visualizations, except those that are locked to avoid updating.

IV. Metrics

In [4] we introduced different interesting aspects to measure to understand the complexity of a migration project. The idea behind the proposed indicators is to count the appearance of entities that may be a problem during migration. Following we describe three different indicators introduced by [4].

a) Grammatical complexity: Counts the apparition of grammatical constructions that are not available in the destination languages. It is used to estimate the possibility of finding problematic cases during a language translation process. Many VBA grammatical entities are used for error handling and control flow and for which we do not have an equivalent in Typescript / Java, we mention: Resume Label, Resume Empty, Error Resume Next, OnError GoTo,... The higher is the value of this indicator, the more complex.

b) Paradigmatic complexity: Counts the apparition of paradigmatic constructions that are not available on the destination languages. It is used to estimate the possibility of finding problematic cases during a language translation process. Among all the first-class citizens of VBA language, we find the followings that have no equivalent on our target platforms: Modules, Tables, Queries, Macros. The higher is the value of this indicator, the more complex.

c) Dependency complexity: Counts the usage of different dependencies in the code, accumulated by architectural concern or by artifact. It is used to estimate the number of places that are required to change during language translation since none of the libraries used in Microsoft Access is available in any of our targets. The higher is the value of this indicator, the more complex.

d) Implementation: The implementation (http://github.com/impetuosa/metro) is based on the analysis of a Famix model for Microsoft Access that visits and counts the apparition of specific entities and it accumulates the values according to the kind of metric. In our tool, we propose two ways to visualize these metrics: the Pareto chart and the pie chart.

A. Pareto chart

It was extensively used in [4] for presenting and analyzing data. As largely explained in books as [5], the purpose of the Pareto chart is to identify important individuals in a sample of data. The diagram follows the Pareto principle (also known as the 80-20 principle). The Pareto chart consists of two graphs: a histogram of frequencies on the measured variable where the individual values are represented in descending order by bars, and an accumulation line.

Figure 2 depicts the three proposed metrics presented using a Pareto chart, accumulating at the level of the Agent sub-project . Most of the grammatical complexity seems to be agglomerated in forms and the paradigmatic complexity in tables. Most of the dependencies, after language, are on the common libraries followed up closely by data access.

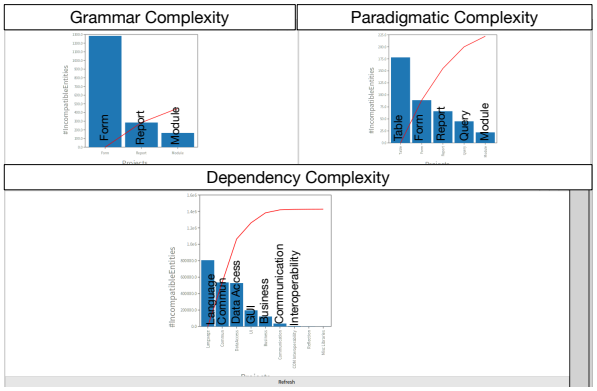

Fig. 2. Example of metrics presented with a Pareto Chart. The Grammatical and Paradigmatic charts presents how many incompatible entities are in the Agent sub-project by first-citizen. The dependency complexity shows how many usages we have of each library, clusterized by library tag.

B. Pie chart

Pie charts do not require an introduction. They are used to illustrate the numerical proportion of each represented category. For the case of Grammatical and Paradigmatic complexity, we use pie charts to contrast the incompatible entities with the compatible entities. For the case of Dependency complexity, we use pie charts to represent the different proportions of architectural concerns. Figure 3 depicts the three proposed metrics presented using a Pie chart, accumulating at the level of the Agent sub-project, contrasting proportions.

We can see that even when the grammatical complexity seems a lot in the Pareto chart, where the number of incompatibilities in forms reaches 1300 cases, is less than 25% of the total of grammatical constructions. On the Paradigmatic pie chart, we can see that Agent uses no classes (the only compatible first-class citizen). We also realise that none of the first-class-citizen in this module are compatible with Java or Typescript.

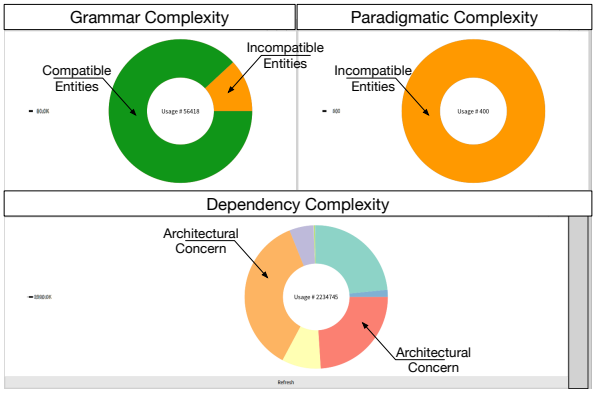

Fig. 3. Example of metrics presented with a Pie Chart. The Grammatical and Paradigmatic complexity charts show the proportion of compatible and incompatible artifacts. The dependency shows the proportion of usage of different libraries.

Finally, we the dependency complexity show us can find an arc-en-ciel of architectural concerns in this sub-project which responds to 6 different concerns.

V. Architectural Analysis

Forms, Reports, Modules and Classes in Microsoft Access can implement multiple concerns such as GUI, GUI navigation, data accessing, interoperability with other systems, and of course, business logic. Knowing that the migration project aims to split the applications into at least front-end and back-end applications, we present a graph that reveals the concerns of the selected artifact.

A. Tagging artifacts

One of the most complex problems in software migration and in our specific case is the unexpected mixture of architectural concerns. For the sake of analysis, we expect the different artifacts to be tagged with what represents their architectural concern. To tag artifacts we use a hierarchical tagging system: if we tag a library as UI, all the artifacts defined within this scope are considered to be UI, unless another tag has been explicitly set. Mixing this tagging with the dependency analysis allows users to recognize that a piece of code has UI dependencies because it's using one or another specific type or function. Different architectural tags have different colours to ease the reading.

B. Class architectural dependency graph

This chart clusters all the accesses between the different components for offering an overview of the dependencies in between all the artifacts within the selected artifact.

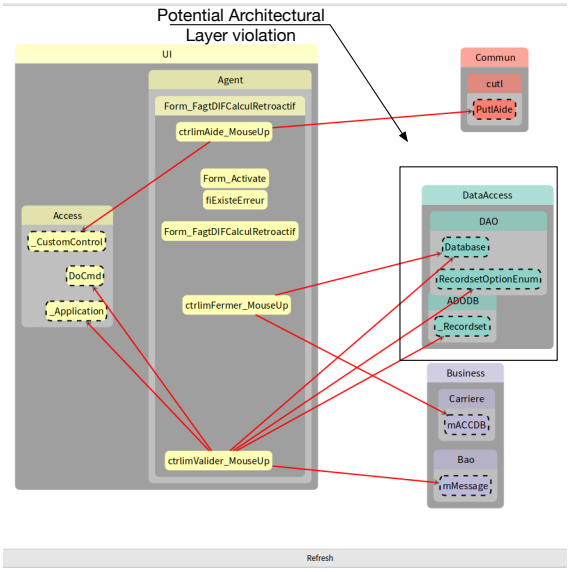

Fig. 4. Tool Screenshot: Example of Class architectural dependency graph. We observe that the analyzed form uses elements from Access library (tagged as UI), cutl (tagged as *Commun*), DAO/ADODB, (both tagged as Data Access) and Carriere/Bao (both user-defined projects, tagged as Business). In Microsoft Access is common to access data in a UI. However, our target (web technology) is considered an architectural violation.

Figure 4 shows the dependency graph of a simple form in the Agent sub-project. It is not a surprise that a Form enters into the GUI set in the graph, but, what is threatening is the fact that besides dealing with UI concerns, the code within the form deals with data access, *Commun* and business logic.

C. Architectural tangling highlight

This chart gives an architectural complexity overview over all a Microsoft Access sub-project, by kind (Form, Report ...) or by the entity. Figure 6 displays the schematics of the chart.

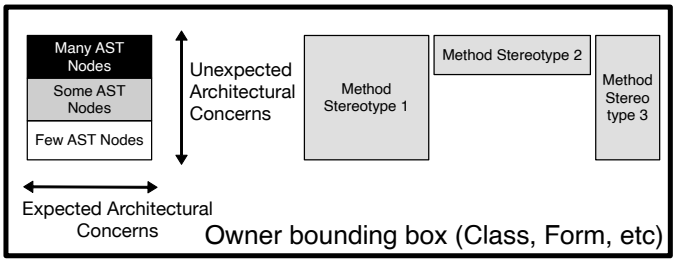

Fig. 6. Reading Architectural tangling highlight: Schematics and stereotypes for fast reading

Each coloured block represents one behavioural entity (method, function or sub-procedure). Blocks are grouped by ownership: all the blocks representing the behaviour of a single first-class-citizen (Forms, Modules, Reports or Classes) are together inside a bounding box; these boxes representing first-class-citizen, are grouped by kind.

This chart encodes three features of the architectural dependencies per behavioural entity into a block.

- A Block's *width* indicates the amount of code related to the architectural concern the element belongs to.
- A Block's *height* indicates the amount of code related to any architectural concern the element does not belong to.
- A Block's *opacity* indicates the amount of AST nodes extracted from the code of the element. The more elements the darker the block.

All the dimensions and opacity of the block are on a logarithmic scale. Finally, some blocks are of fixed size and red. These blocks are there to represent external functions and sub-procedures.

Figure 6 proposes 3 stereotypes to ease reading.

- Stereotype 1: It uses many elements that belong to both the same as different architectural concerns. It is related to probable architectural layer violation.
- Stereotype 2: It uses many elements that belong to the same architectures, and few that belong to other architectures. It is related to no architectural layer violation.
- Stereotype 3: It uses many elements that belong to different architectures, and few that belong to the same architectures. It is related to either probable architectural layer violation or misplacement (should this method belong to this class?).

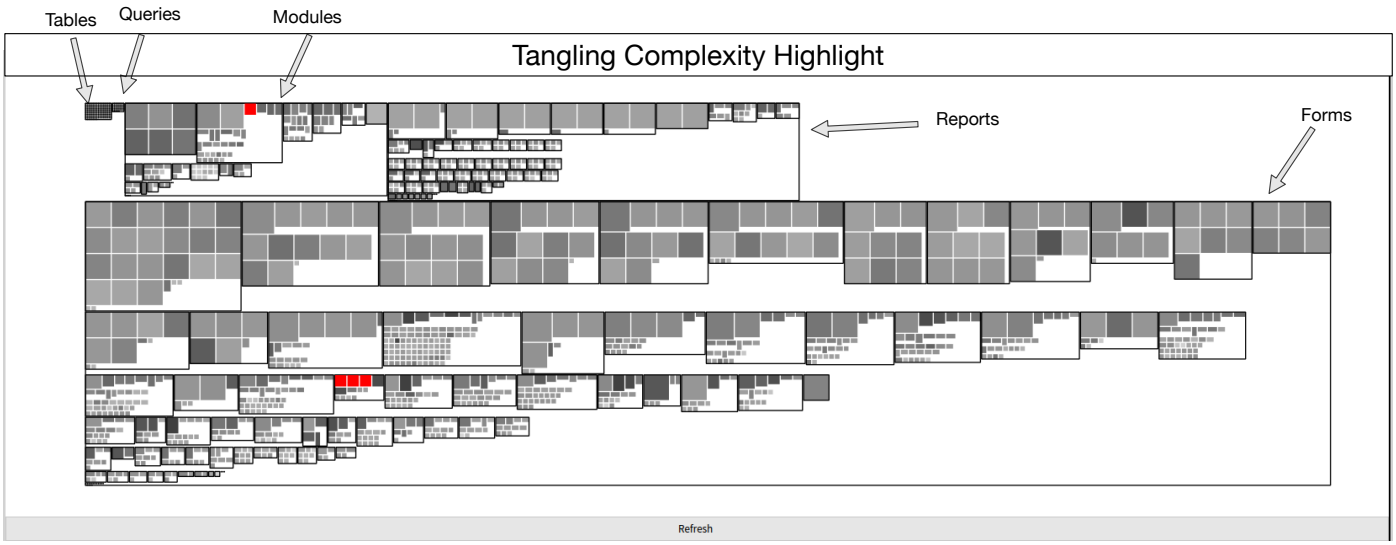

Fig. 5. Tool Screenshot: Example of Architectural Tangling Highlight. This chart represents all the different artifacts declared in the Agent sub-project.

Blocks are confined in boxes that represent the defining entity (class, module, form, etc). Blocks complexity is calculated by multiplying the three dimensions: width * height * opacity. Boxes (class, module, form, etc) complexity is calculated by summing up the complexity of all the contained blocks. Boxes and Blocks are sorted by complexity. Finally, all the boxes are organized by kind: Table, Query, Module, Report and Form.

D. Summary

Our tool allows software migration to reveal, (i) grammatical, paradigmatic and dependency complexities; (ii) The aggregation analysis of architectural "mess", with the tangling highlight, and (iii) the analysis centred on first-class-citizen (Class, Module, Form, Reports, . . .). All of these are calculated by using a Microsoft Access Famix model and extending the Moose platform with the pertinent algorithms for generating these visualizations.

VI. The tool in action: A process for migration assessment on the *ePaie* project

One of the real-life experiences we had in our company is to help key developers in the development of a migration blueprint for the *ePaie* project. The blueprint building consists of reporting the architectural and functional reality of the migrating project. This report is expected to respect the requirements of a consultant company on software migration for helping with the issuing of a migrating plan.

The tool has many aggregation visualizations, meaning that we could start with any of them according to the specificities and the kind of evidence that we are looking for.

In this article, we present the usage we did for getting information for help during the blueprint construction. The main concern of the blueprint according to the consultant company is the architecture and the features of the software. Following we expose how we use the tool for our case of study, focusing on Agent sub-project.

a) Architectural tangling highlight: The tangling highlight chart can give us a powerful overview of all that we have in a sub-project in terms of architectural complexity.

Observing the Figure 5 we decide to focus on the forms. Figure 7 zooms in into the forms section of the visualization. We can guess quickly that most of the forms are handling more than GUI and business, meaning that there are high chances to find architectural layer violations, and with it, highly complex pieces of code.

We pick the first form, which is the most complex in general (even when it does not hold the most complex method). We magnify the architectural tangling highlight in Figure 8. In here we observe that most of the methods are complex since most of them respond to the *stereotype one*. There is one apparition of *stereotype 3*, which means that the method may be even misplaced in this form.

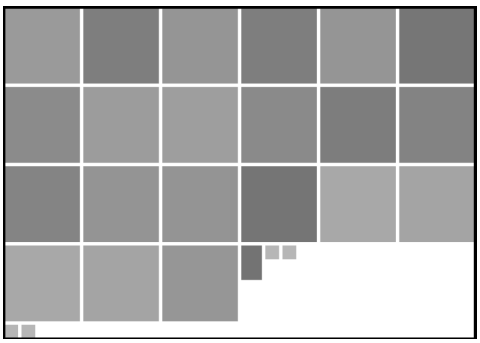

Fig. 8. Architectural Tangling Highlight: Magnification of the first and more complex Form in the Agent sub-project: Form_FagtVisiteMedicaleSelection.

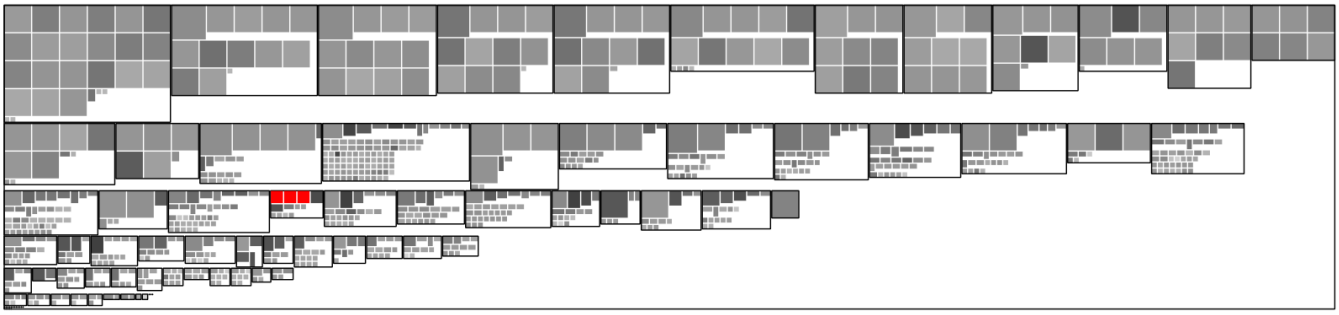

Fig. 7. Architectural tangling highlight. This chart represents all the Forms declared in the Agent sub-project

b) Dependency pie chart: In order to get an overview of what are the architectural concerns mixed up in our form, we can use the dependency metric on its pie chart version. Figure 9 shows all the different architectural concerns found in the selected form.

With this overview, we can know easily if there is something smelly or not on the implied architectural concerns. This case is special. There are too many concerns. We would like to look a bit deeper to understand what are the dependencies we are using.

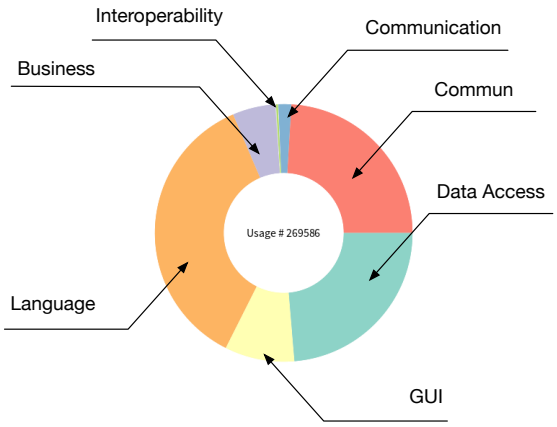

Fig. 9. Dependency Pie chart: All the architectural concerns found in Form: Form_FagtVisiteMedicaleSelection.

c) Class architectural dependency graph: In the previous step, we found evidence that this form requires reporting since it is concerned with many things at the same time. For understanding better the kind of responsibilities we use the dependency graph that will show us what functions and sub-procedures are used from what libraries, clusterized by architectural concern.

The Figure 10 shows the dependency graph of this Form, and with the proper navigation, it allows the user to discover which function in the form has what dependency.

Once we read the usage details, we find out that there is one kind of usage that is undesirable: the direct tangling with Data Access features. Even more, this form uses two different ways to access the database: DAO and ADODB libraries.

If we follow the arrows we can easily differentiate the functions and sub-procedures that do data access from those that do not.

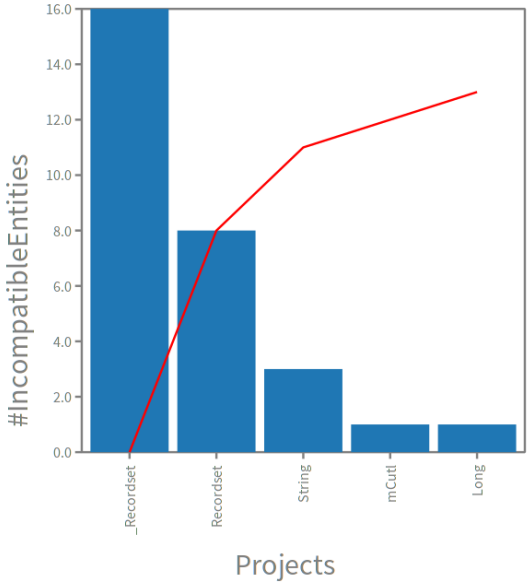

Fig. 11. Analysing fiListeCritere: Dependency complexity metric on Pareto chart.

d) Analysing fiListeCritere: We choose this method because it is one of the richest methods, using GUI, *Commun*, Business and DataAccess.

For having a first overview of the method we use the metrics of this specific function, visualizing them with the Pareto chart. At the level of the function, the chart of grammar complexity is not interesting, since it does not reveal much more than the existence of two incompatible entities. The dependency complexity of this function is more interesting and revealing, shown in Figure 11. We do not expose proprietary code, however, we will describe this finding. It is a function with two parameters. It is written along 48 lines of code, without counting comments or empty lines. It parses into 237 AST nodes. The code we found in this function is effectively interesting. It is a function that based on the parameters and information obtained from the form builds a query by concate-

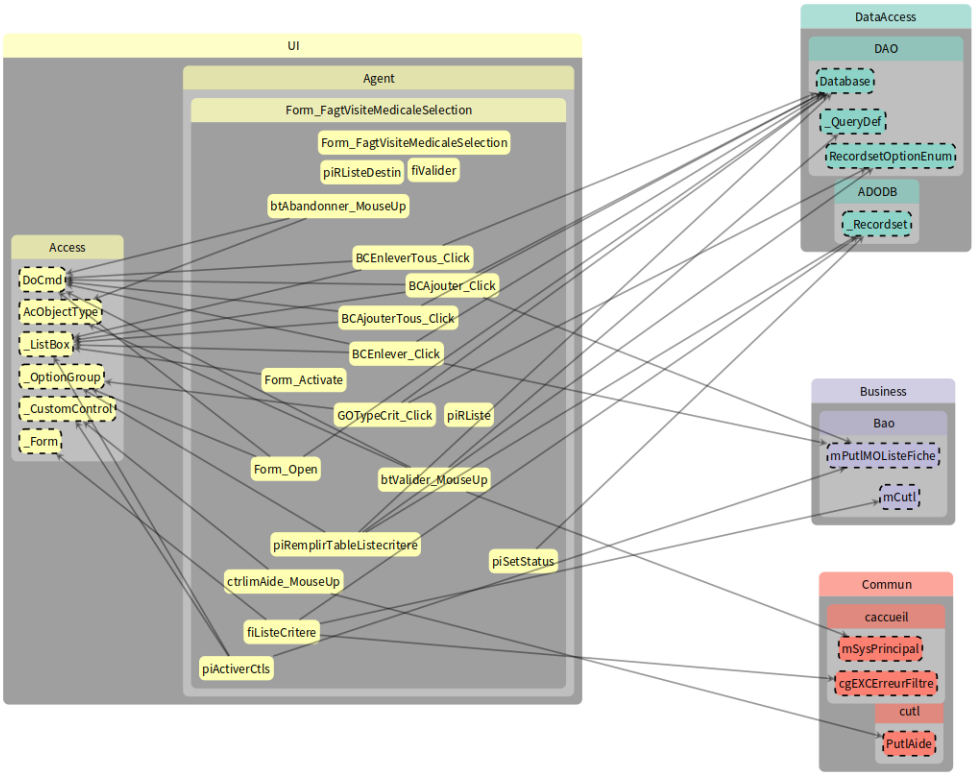

Fig. 10. Architectural dependency graph applied to Form_FagtVisiteMedicaleSelection. We can observe in here a strong dependency on Data Access.

nating strings and accesses directly the database, generates a string containing information split by a special character.

VII. Case of study: Selecting and prioritising the tasks involving library migration for the *ePaie* project

During the migration process, we need to plan how to replace different library elements of the source environment with those in the target. Especially for our case, since our migrating approach relies on code translation. None of the basic libraries offered by Microsoft Access is either available nor desirable to use in any of our target environment.

This means that all the code we translate must be adapted to the target environment libraries. For adapting the translated source code to use another library in a target environment we have to come up with different strategies.

Let's consider the translation to java in the following two examples:

- All the usages of *CStr(x)* (a function that receives any element and returns its string representation) must be replaced with *x.toString()*;
- All the usages of *Recordset* (class used for accessing the database) must be replaced by more than one artifact, or it may require the development of a special class giving the same service, since there is not simple equivalent in Java.

The project *ePaie* uses almost 690 different elements from different libraries. To have strategies for 690 different elements is a lot of work. This work requires prioritisation and some criteria to recognise how vital is it in translation. Instead of rushing on having strategies and implementations for all the 690 cases, we propose the usage of the dependency metrics for the whole project in a Pareto chart.

We argue that in this case, we have to distinguish types from functions and variables or constants since each of these elements accomplishes a different task in our code. This is why we refined the dependency metric to be able to measure only one kind of element at a time: (i) function, sub-procedure or method; (ii) member (access to some kind variable: global, class variable, etc) ; (iii) types.

The Pareto chart by itself already tells us which artifacts to take care of first. We propose the usage of this chart to evaluate what is the most representative set of artifacts, therefore, which artifacts we must focus on first. And to use the same order to prioritise the tasks.

For all the following charts, the distance between each tag on the X-axis is 10 elements. The first tag belongs to the first element. Figure 12 shows a Pareto chart on the usage of behavioural entities defined in different libraries. 80% of the total invocations are reached at the tenth element (at the function named Report). We prefer to focus on the first 90 out of 430 functions – 20%.

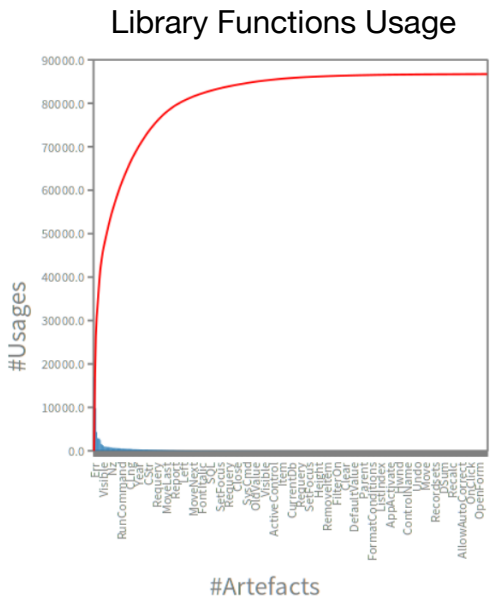

Fig. 12. Analysing *ePaie*: The most used functions. We have to ensure to have a java replacement of the first 90 functions.

Figure 13 shows a Pareto chart on the usage of member entities defined in different libraries. 80% of the total invocations are reached in between the third and fourth element of the member named Form. This means that we have ensured to have a java replacement for the first 25 out of 160 members – 15%. Figure 14 shows a Pareto chart on the usage of types defined in different libraries. 80% of the total invocations are reached at the fourth element the member named Form. This means that we have ensured to have a java replacement for the first 30 out of 200 types – 15%.

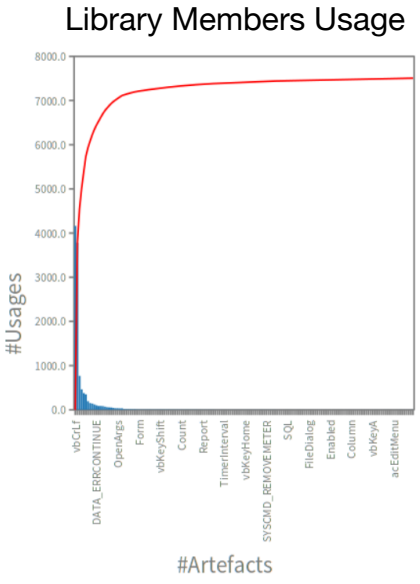

Fig. 13. Analysing *ePaie*: The most used members (variables, globals, constants). We have to ensure to have a java replacement of the first 25 members.

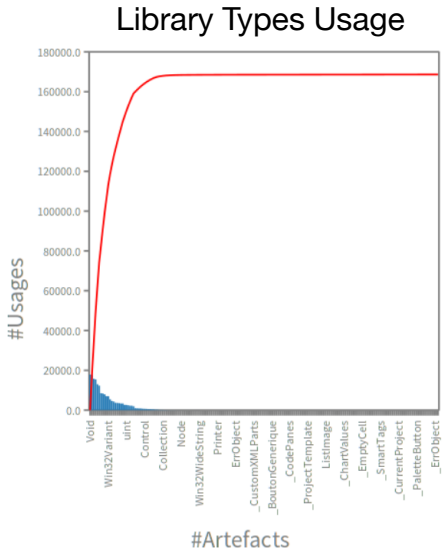

Fig. 14. Analysing *ePaie*: The most used types. We have to ensure to have a java replacement of the first 30 types.

By applying this method we are able to figure out what are the most worthy elements to work with, reducing our initial list of compulsory elements from 690 to 145 – 21% of the total. The rest of the elements can be treated on demand.

We propose also the usage of the order used by the Pareto chart as a first iteration of task prioritisation: the left-most the first.

VIII. Future work: Comparing two applications

During our analysis, the project manager of one of the projects to migrate expressed the intention to validate his planning by using the tool not only in his project but also by comparing the measures with those of another project, known to be more complex.

For us, this is a small success, because the people involved are using directly or indirectly using the tool, and also we find that a comparative analysis is a great idea for putting in dimension measures. So far we do not support any kind of metric or graphic that allows us to compare side by side two projects. We think it would be a good feature to add to our tool, but it may require some research beforehand, just to understand if it is a worthy feature. There are many reasons for thinking that allowing comparative analysis in the context of migration would be a reasonable next step: (i) Many of the projects developed with Microsoft Access use a suite of common libraries named *Commun*. Nevertheless, the usage and versions of the library deployed on different projects are, in many cases, different. To be able to understand differences could be important for yielding fine migration planning. (ii) Comparative analysis gives an idea of dimension based on previous experiences. It is doubtless an expertise tool since it requires the user to be familiar. But it helps to leverage this expertise to gain confidence in the process of measuring, predicting and planning. (iii) Once the first migration is

accomplished, we can propose an explicit comparison with this first migrated project. What will leverage actual expertise on the process of migration into the process of planning.

IX. Discussion

Sentences like *"we usually do not do reverse-engineering because we are consultancies and we cannot have tools for reverse-engineering of the different programming languages of all possible customers. We gain an understanding of the legacy systems with practices similar to those of requirements engineering such as workshops and interviews with the customers. In short, reverse engineering of the pre-existing system is not favourable considering the little Return On Investment (ROI)"* [6], make us think that the work done over the software analysis tools may not be mature enough.

Leveraging the Moose platform and the Famix meta-model [1] helped us to implement most of these tools in a short time.

Platforms and communities such as Moose have been there for more than 20 years ago. Platforms developed by open source communities that participate in both research and engineering, with the clear goal of easing the work of software engineering and coping with complexity.

Adding support to new languages is a complex task, but we argue that most of the time are a job that, when done intelligently, is done once, and reused later. It is also true that when this kind of task is leveraged by an open-source community, the costs and efforts are shared.

It is our duty as software engineers to leverage the multiple outcomes of reengineering and modelling studies for proposing solutions that help the stakeholders of the different projects (E.g. other software engineers, including ourselves) to be able to work efficiently. It is chief to understand that tasks such as software migration do not bring any new income, are always extremely expensive and become more and more likely to happen. Therefore, participating in open-source communities and community projects is the smartest approach.

X. Conclusion

Assessing and predicting outcomes of software evolution is quite a challenging domain. We recognize that a lot of work has been done on the development of useful tools for assessing our fellow software engineers and other human beings. We developed a tool that puts this technology in the quite specific perspective of software migration because is the only way we found to make it a tool that is meaningful for our project and our stakeholders.

In this article, we present our software migration analysis tool. A tool released with an MIT license just like Moose itself. We shared our experience on how to leverage the available visualizations for assessing complexity by presenting a process of assessment Section VI and the study case of task selection and prioritisation Section VII. We discussed our next steps Section VIII. We discussed the value of software reengineering in Section IX. And we open our doors to collaborate and share the heavy lift of software migration in particular and software evolution in general.

References

[1] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrôme Sudich, and Mustapha Derras. Modular moose: A new generation of software reengineering platform. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020.

[2] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Seriai Abderrahmane, and Mustapha Derras. Analysing microsoft access projects: Building a model in a partially observable domain. In *International Conference on Software and Systems Reuse (ICSR'20)*, number 12541 in LNCS, December 2020.

[3] Santiago Bragagnolo, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai, and Mustapha Derras. Software migration: A theoretical framework (a grounded theory approach on systematic literature review). Technical report, Berger-Levrault and Inria Lille Nord Europe, 2021.

[4] Santiago Bragagnolo, Abderrahmane Seriai, Stéphane Ducasse, and Mustapha Derras. Risk and complexity assessment on the context of language migration. In *International Conference on the Quality of Information and Communications Technology, QUATIC'2021*, September 2021.

[5] Stephen H. Kan. *Metrics and models in software quality engineering*. O'Reilly, 2006.

[6] Maryam Razavian and Patricia Lago. A lean and mean strategy for migration to services. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, page 61 to 68, New York, NY, USA, 2012. Association for Computing Machinery.

[7] A. A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November 2000.