Reporting Context Aware Partial Translation engine based on immediate and delayed Rule application

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*—Language migration has been the driver of many efforts resulting in multiple solutions and strategies.**

One of the most popular approaches for dealing with it is source code translation: it proposes to translate the source code of an application to a target language. For doing so it leverages a set of translation rules based on the grammatical constructions provided by the source and target languages.

However, we notice that even when most of the literature acknowledges translating implies migrating the runtime, libraries and the Software Development Kit (SDK), none of them proposes a systematic way to solve this problem. Along with this, we notice that there are many proposals to shift the paradigm from procedural to object-oriented programming based on how to propose classes automatically. But we found nothing on how to translate the expressions that use functions into expressions that use methods. In the context of migration from Microsoft Access (MS Access) to web technologies, these two lacks threaten seriously any attempt to produce even a compilable version of the code on the target technology.

This article proposes a translation engine that split the translation process into two phases. A phase of language translation, and a phase of adaptation to the target environment. The first phase is in charge of producing declarations, and the second one is in charge of adapting the usage of this declaration to fit the translated version of our artefacts.

We argue that enabling to adapt the code to fit the translated version of our artefacts allows the definition of simple *adapting* rules able to deal with a large share of both problems: (i) runtime, libraries and the Software Development Kit (SDK), and (ii) simple paradigm shift.

This article presents some basic *adapting* rules and validates our approach by translating a battery of simple tests that feature the usage of a carefully chosen set of features.

***Index Terms*—Migration, Iterative, Incremental, Context-Aware, Microservice, Monolith, Standalone, Software Evolution, Refactor**

I. Introduction

During language migration, we are required to change the language of a program: to express the source program semantics in a new language.

Language migration is a deceiving task. It seems to be as simple as translating language constructs (something that can be already complex by itself). Even considering languages that have similar grammatical constructions, languages often run on different environments with different SDK and libraries available and, never the less, different best practices, which leads to substantially different ways to define APIs, and compose and reuse code.

After translating a project from one language to another with the purpose of migration, we expect to have the same behaviour and to use the target environment: runtime, SDK, libraries: all that makes this environment a desirable target.

We find out that even when syntax-based translation is implemented with **AST rewriting is a powerful tool**, it is hard to define rules that apply to all usages of a specific artefact, to replace one library with another or even one piece of our code with some library usage from the target environment.

In this article, we explore the possibility of a language migration based on language constructs and language semantics.

This article continues as follows: Section II specify in detail what has been done, and why is not enough for our

case of study. Section III presents the research questions that will guide our research. Section IV explains the importance of reifying the relation between declarations and usages and proposes a variation of the Abstract Semantic Graph. Section V proposes a translation engine able to apply *translating* and *adapting* rules. Section VI analyzes the potential of the adapting rules for migrating runtime, paradigm and libraries. Section VII proposes and conducts a validation that involves the translation of programs from MS Access to Java and Pharo Section VIII discusses the threats to the validity of the conducted validation. Section IX concludes and proposes future works on the subject.

II. Translating source code

In the context of a collaboration with Berger-Levrault, a major IT company, we are working on the migration of MS Access monolithic applications to web Typescript/Angular front-end and Java/Springboot microservices back-end. The minimum requirement of our software migration is to translate MS Access 's flavour of Visual Basic for Applications (VBA) to Java and Typescript, ensuring that the code uses their respective runtimes, SDK, libraries, frameworks and paradigm.

A. Source code Translation

Language translation has been addressed as a compiling problem [1–3]. Software migration is not compiling but many techniques used in compiling can be leveraged in our favour. This has been done in the past for conducting successfully language migrations [5, 8].

However, translating is more than changing the language. To change runtime and libraries is imposed (except in rare occasions [9]). Terekhov, Verhoef [10] points explicitly out how this problem has been neglected. Despite all these notices and recognition, it has not been dealt with systematically.

B. The problem

Lets consider the code in Listing 1. A Visual Basic Application module used for testing named Testing, with a function testing the function Len using HelloString global variable as a parameter. This global is defined in another module named ProjectGlobals.

```
Public Function testLen() : Void
    Call Assert(Len(HelloString), 5, "Should be 5")
End Function
```

Listing 1: Testing the function Len

If we use a translation approach with Java and JUnit 3 as a target in mind, we could quickly propose some rules of translation:

1. Translate the module as a class subclass of TestCase.
2. Translate the function as a method.
3. Translate the return type Void as void.
4. Translate the call to Assert(x,y,z) as ”this.assertEquals(z,x,y)”
5. Translate the access to the global HelloString as [ParentName].[GlobalName].
6. Translate the call to Len(x) as x.length().

If we define such rules in a translation engine such as the one used by Brant et al. [5] [1] we will be able to translate this method into Java. However, there are three main restrictions in this approach: By having rules that express how to write expressions such as [ParentName].[GlobalName], we make an assumption that impose strict constraints over **what**, **how** and in what **order** something is going to be translated.

a) The importance of what, how and when to translate something: **What to translate** is never a clear decision. Much of our code may have no sense of the target or being replaced by a target library, a service, etc. Are we sure that we want to translate ProjectGlobals? Are we sure we want to translate the global HelloString? If we decide to not translate the module ProjectGlobals we will be forced to review or create new rules to contemplate this exception. **How to translate** something is not an easy decision either. Is HelloString going to be translated as a static public variable? Without any accessor? If we decide finally to translate HelloString as an accessor getHelloString() we will be forced to review or create new rules to contemplate this exception. The **Order of translation** is even more complex. We could propose to translate things in order of dependency, which would solve many technical problems, but are we going to translate the full module ProjectGlobals to be able to translate our test? To our client's needs, this order may be a waste of time.

b) The importance of being agile: We argue that a translation is too complex and too risky to not be agile. We want to be able to do partial translations in order to do the minimum for being able to make one thing work at a time. We argue that changing our minds should be relatively easy and affordable.

1) Library migration: It is an area of study on its own. Even updating a library, or migrating from one version to a newer one, can be already challenging. Cossette, Walker [6] explains very carefully the problem of library migration, and it proposes to group API transformations, according to the degree to which they believe that they can automatically be enacted. Here we recover the part that applies to our study case.

- Fully automatable transformations involve API changes that do not alter the semantics of the API significantly. *i.e.,* function rename, remove a parameter, reorganize parameters.
- Hard-to-automate transformations are cases that represent significant alterations to the semantics of entities between libraries; a tool is unlikely to handle such a transformation without assistance from a developer. *i.e.,* Add parameters, Feature deletion (the feature does not exist on the target), API replacement.

This work is mainly inspired by [7]

2) The paradigm migration: Paradigm migration or Paradigm shift is also a subject that had a lot of attention. It is noted that we choose to take the easy way out of it, by using static “class methods” on all our destinations. This reduces drastically the complexity of our problem, but we still must

[1]https://github.com/j-brant/SmaCC

deal with the problem of changing the way we use this code to be adapted to the object-oriented paradigm. For example, all the function calls must be transformed into method invocation, which implies inferring which object or class the receiver has to resolve this method.

3) Sumary: Language translation rules must assume some knowledge of the language. Naive approaches to assumptions can lead to wrong translations. Both facts threaten the reusability and validity of rules. Library and paradigm migration have one thing in common: both affect what is available to use, and how to use it.

III. Research questions

We consider that most of programming languages respond to three different kinds of constructs: Declarations, Statements and Expressions. Declarations are the definition of structural and identifiable artefacts of some specific nature: types, functions, etc. Statements are syntactic entities defining actions to be carried out. Expressions are also actions to be carried out, but an expression also yields a value.

Declarations are to some degree, always defined in terms of other declarations. Either declared in the language or provided by the runtime like primitives. *e.g.,* Classes are declared referring to their superclasses. Behaviour is also defined in terms of other declarations. *e.g.,* Functions behaviour is based on applying other functions by referring them through function invocations. We call those elements used to relate an entity with a declaration *Reference*, or *referential code*. There is an implicit dependency between the nature of a declared artefact and the kind of reference able to interpellate it. If we use a function invocation (kind of reference) to execute a behaviour, therefore, the declared artefact is a function. If an artefact is declared as a function with one parameter, to be used for execution, it must be through a function invocation with one argument.

In this article we support the following claims: (i) given the reference of the source program, and the declaration that it is expected to refer in the target program, we can automatically propose a pertinent translated reference: including adapting different references to use pre-existing artefacts (SDK, Libraries, Types, etc) in the target system, and to adapt to the target system paradigm. *i.e.,* if we know that the reference is a function invocation and that the target is a static method, we can automatically write a static method invocation as translation. By doing so, we reduce the impact of decisions like **what** and **how** to translate an artefact. (ii) to delay the translation of a reference to the moment when the engine finds evidence of what the expected declaration to be referred to reduce the impact of the translation's **order**.

From these claims stem the following questions:

- RQ#1 Can we delay the translation of a reference to the moment when the referee is defined?
- RQ#2 Can a delayed translation rule transform a reference to use the target environment's pre-existing artefacts (SDK, Libraries, Types, etc)?
- RQ#3 Can a delayed translation rule transform a reference to use the target environment's paradigm?

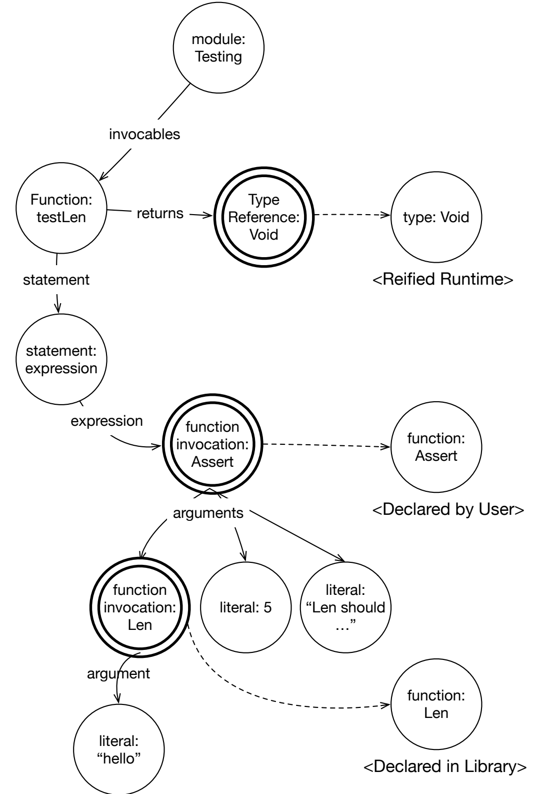

Fig. 1: ASG Object Model. Reference objects are denoted with double strokes.

IV. Representing the reference - declaration binding with an Abstract Semantic Graph

a) Abstract Semantic Graph: Our proposal is based on a common program representation model. For doing so we use a representation popular in the domain of program transformation named Abstract Semantic Graph (ASG). An ASG is a graph that extends the Abstract Syntax Tree (AST) structure by linking artefact uses to its definition through symbol resolution. In our version, the link between any entity and a Declaration is reified as a Reference object. A function declaration uses a TypeReference object for referring to it's returning type. A function or method invocation is considered to be a reference as well.

Figure 1 is a graphical representation of a ASG instance which represents the source code listed in Listing 1. Reference objects are denoted with double strokes. The dotted line represents the relation between a kind of reference object and a declaration. Like this, the function declaration node has an arrow pointing to a type-reference object, which has a dotted line arrow pointing to the type Void. We note that our ASG reifies and uses polymorphically runtime, library and user-defined artefacts. From our object model point of view, the difference between them is their tag. It is to note that a library function keeps its parameters but has no body.

b) Heterogeneous Abstract Semantic Graph: We call our ASG heterogeneous because we use it to represent many languages. Different concepts are modelled by different classes. When representing a new language the rationale is the follow-

ing: if we need to represent a concept that we already used and from the point of view of migration is equivalent (an expertise task that we are not going to discuss in this article), we reuse the same class. This is for example the case of the classes representing the If, While and For statements. If we do not find an equivalent, we add a new class. We always try to reuse what is possible but to avoid compromising its unique semantic value. This is why we have *ClassWithNamespace* to represent classes that like in Java can declare inner classes, *DecoratableClass* to represent classes that like in TypeScript can be decorated.

V. RQ#1: Can we delay the translation of a reference to the moment when the referee is defined?

To respond to this question we have to start with a quick proposal for a translation engine. The translation engine receives a source unit of code to be translated and the target where to aim to translate. By translation, we mean the same kind of syntactic tree transformation proposed by [5, 8]

A. General Approach

1) Load source project: The engine loads an ASG for the source project. Source entities are the units to be translated. MS Access offers many kinds, in this article, we focus exclusively on Modules and Classes. This ASG includes all the Modules and Classes defined by the users and those specified in runtime, libraries and SDK.

2) Load target project: We argue that no translation project starts from thin air. There is always an idea of architecture and what libraries we aim to use. The target model allows us to: (i) know what are the available libraries on the target *e.g.,* we may want to use specific third-party libraries. (ii) to aim where to translate something *e.g.,* a previously existing package, or a previously existing class.

The engine loads an ASG for the target project. This ASG includes all the artefacts defined by the users and those specified in runtime, libraries and SDK.

3) Configuring rules: Each translation requires different rules that are suitable for translating different languages. We model rules as objects to be able to easily add them to the engine according to the desired behaviour. Rules are discussed below in Section V-B

4) Mapping: Once we have entities from source and target projects we can establish semantic equivalences. We have two kinds of mappings: (i) Simple mappings that allow telling that a source entity is equivalent to a target entity. We use this mapping to relate types *e.g.,* the typed string from MS Access is mapped to the String class defined in the java.lang library. (ii) Nested mappings that allow telling that a source entity is equivalent to a target entity and that the direct children must be mapped as well *e.g.,* a source function may be mapped to a method, where the first argument is mapped to be the receiver of the method invocation and the rest of the arguments are kept in the same order. The user is expected to map all the possible entities before continuing to the next step.

5) Translation: The engine is given a set of source entities to translate and a target which to translate. *e.g.,* some MS Access modules as source entities, a Java package as a target. The engine applies different rules over the different source entities to produce the translated code within the given target. *e.g.,* to translate the MS Access modules as Java classes in the given package.

B. Rules

Rules are conditional operations, that applied over an accepted entity and return a new entity. We consider a rule to be an instance of a class that responds to two methods, a conditioning method and an operation method. *Condition* consists of a predicate that allows the user to define specific requirements for the operation to be applied. *Operation* consists of any systematic modification over the target. A rule returns a single entity.

Each time the engine creates a target entity with a rule, it traces down the relation: the target entity T was produced by applying the rule R to the source entity S. Each time the engine creates a **target declaration** out of **source declaration** it produces a simple mapping between these artefacts to establish a semantic equivalence, as explained in Section V-A4.

C. Translating phase

[2] The engine traverses hierarchically the source entity avoiding the links between reference and declaration objects. For each source entity, it applies a “translating rule” to produce a target entity. Each produced target declaration based on a source declaration is mapped as a simple mapping. We call this first part *Translating phase*.

1) Translating rules: Rules to be applied during *Translating phase*. They represent the production of a *target* entity based on a *source* entity.

a) Copy translation rule: Listing 2 lists the copy rule.

The first method represents the condition. It is parametrized with the source entity, the target context entity (where the target entity is to be written) and the writer to be used. In this case, it yields always true. The second method represents the operation. It is parametrized with the source entity, a targeted writer and the engine (for it may need to delegate to other rules). In this case, it attempts the copy the source entity into the target entity.

```
CopyRule>>matches: aSourceEntity context:
  aTargetContextEntity with: aTargetWriter
  ^ true
CopyRule>>applyTo: aSourceEntity destinationWriter:
  aTargetWriter engine: anEngine
  ^ CopyHelper copy: aSourceEntity into: aTargetWriter
```

Listing 2: The copy translation rule is an attempt of copying the source entity into the target

This rule is permissive, and it could end up producing function invocations in a java method, the thing that we know is wrong, but the copy can be **rewriten later** by an adapting rule.

[2]To traverse the ASG as an AST and use a visitor pattern.

b) Syntactic mapping example: Listing 3 lists and example which transforms a function or sub-procedure into a static method.

```
FunctionToMethod>>
matches: srcEntity context: tgtContextEntity with: tgtWriter
   ˆ srcEntity isFunctionOrSub and: [ tgtContextEntity isClass ].

FunctionToMethod>>
   applyTo: srcEntity destinationWriter: tgtWriter engine: anEngine
      ˆ tgtWriter writeMethod: [ :method |
         method selector: aSourceEntity selector.
            srcEntity isSubProcedure ifTrue: [
               method returnTypeRefence: tgtWriter refToTypeVoid.
            ].
            method setAsStatic.
            srcEntity allChildren do: [: child |
               anEngine migrate: child into: method
            ]
      ]
```

Listing 3: Syntactic mapping transformation example. Transforming a function or sub-procedure into a static method

The condition ensures that the source entity is a function or a sub-procedure. It also ensures that we are going to be writing inside a class. The operation writes a method: (i) it copies the selector, (ii) if it is a sub-procedure sets void as return type, (iii) sets the method as static (iv) delegates to the engine the migration of all the children.

Other declaration-related rules have similar implementations such as ModuleToClass or VariableToProperty.

c) Translating example: Let's consider the translation of the ASG proposed in Listing 1. From MS Access to Java. We will leverage the following rules: ModuleToClass, FunctionToMethod and Copy.

The outcome is shown in Figure 2.

We note that the invocations to Len and Assert functions in our result are wrong, but we leave it there to be adapted when there is an equivalent entity in the target.

2) Installing linked stubs: Up to this point, we can translate code eagerly by applying Translating rules. The translating phase finishes by marking all the references that have been created to be taken into account in the adapting phase. To mark these references we are going to set as referee a "linked stub".

This linked stub will hold a reference to the source reference's referee that produced the target reference *i.e.,* the buggy function call to Assert in our target will have as a referee a stub that links to the Assert function defined in the source model. This stub has two responsibilities (i) mark a reference to be processed (ii) trace fine-grained relations.

In order to build this stub we need to have an engine that keeps the traces between source and target entities. The thing that our engine does as explained before in Section V-B.

D. Adapting phase

The engine follows up with the *Adapting phase*. The engine iterates all the reference entities that were marked with linked stubs, as explained in Section V-C2. For each entity, it applies an "adapting rule" to ensure that the reference object is fit and able to refer to a target declaration.

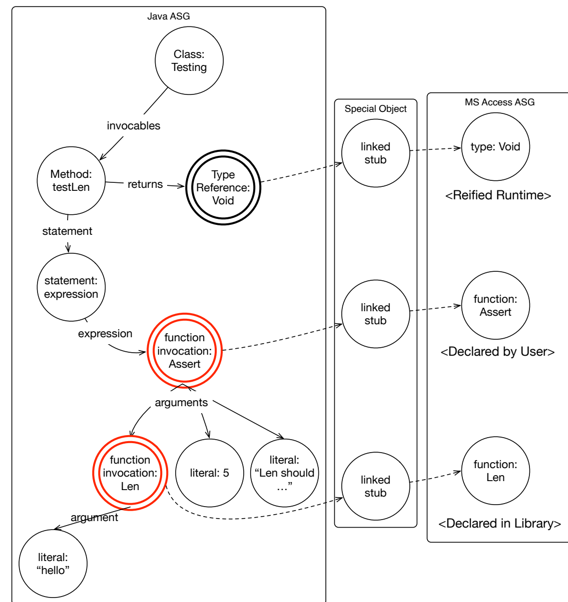

Fig. 2: Translation

1) Adapting rules: Rules to be applied during *Adapting phase*. These rules represent the modification or replacement of a *target reference entity* based on the context of usage, the available mapping information and the declaration artefact that the *target reference entity* should be able to interpellate.

a) Simple rename adapting rule: Listing 4 lists an adapting rule. The first method represents the condition. Parametrized with the target reference, the declaration held by the linked stub, and the available mappings so far. It yields true if: (i) there is an equivalent declaration in the target model. (ii) it can be referred by tgtReference (if it is a type-reference and the mapped declaration is a type this is true). (iii) no arguments are given to the reference and no parameters are required for the declaration. The condition ensures that the only real thing that could change is the name. Nothing else would matter. And that is what the operation does: it sets the referee to the mapped declaration and it sets the name, to ensure that the reference complies with the use of the same name as the declaration.

```
SimpleRenameRule>>
matches: tgtReference toAdaptAs: srcDeclaration mappings: mappings
   selectedMapping := mappings detect: [: map |
      (map source = srcDeclaration) and: [
    (tgtReference refersToObjectsLike: map target) and: [
      tgtReference hasNoArguments and: [
      map target hasNoParameters ]
    ]]
 ] ifNone: [ nil ] .
 ˆ selectedMapping isNotNil

SimpleRenameRule>>
adapt: tgtReference toFitTheEquivalentTo: srcDeclaration writer:
    aWriter using: aFylgja
 tgtReference name: selectedMapping name.
 tgtReference referee: selectedMapping target
 ˆ tgtReference
```

Listing 4: Rename adapting rule

Adapting rules examples are introduced and carefully anal-

ysed in Section VI.

E. Summary: Can we delay the translation of a reference to the moment when the referee is defined?

In this section, we proposed a method able to delay the application of rules to the moment when a mapping is possible. We content ourselves by knowing that there is at least one way, and it responds to the requirements.

VI. RQ#2 & RQ#3: Can a delayed translation rule transform a reference to use the target environment's artefacts & paradigm?

We remember that the engine gathers all the mapping information, either produced by the user during engine configuration as explained in Section V-A4, or produced by the translating phase as explained in Section V-C.

In this section, we respond to these questions by proposing five rules that will be verified (i) Simple rename, (ii) Rename Map arguments with static receiver, (iii) Rename Map arguments with this receiver, (iv) Rename Map arguments with argument receiver, (v) Autowrap missing library.

A. Simple and Nested mappings

Before stepping further into the definition of the adapting rules, we want to be crystal clear on what our mappings are.

a) Simple mapping: A simple mapping is an object that relates a source declaration with a target declaration. It works as a simple association and it implies that it implies that the source declaration is equivalent to the target declaration in the target ASG. This mapping is enough to map two entities without parameters, or two target entity that was produced based on the source entity (assuming that if there are parameters, they did not change order). *e.g.,* (Void => void); (String => String); etc.

b) Nested mapping: As a simple mapping, it associates a source declaration with a target declaration. It does as well map all the parameters between source and target declarations. This mapping allows also to specify if a parameter on the source declaration becomes a receiver in the target. *e.g.,* let's consider the mapping between function F(x,y,z) and method M(a,b,c). Three mappings examples could be:

- (F => M (a => z; b => y; c => x)): all the target parameters are mapped to all source parameters. The order changes.
- (F => M (a => x; b => y; c => x)): parameters a and c are mapped to x; parameter b is mapped to a. Parameter z is dismissed.
- (F => M (a => x; b => y; c => x; z => R)): parameters a and c are mapped to x; parameter b is mapped to a. Parameter z is proposed as receiver.

c) Mappings are only evidence: The proposed mappings do not offer any behaviour. There are there to log of "evidence" of the relation between two artefacts. A mapping can be the outcome of the user manually configuring the engine as explained in Section V-A4. It can also be the outcome of the engine establishing a relationship between two entities as explained in Section V-C. The mappings are maintained in a list to be queried by the adapting rules when being applied.

B. Simple rename: Renaming types and simple invocations automatically

Condition

1) There is a target declaration mapped **AND**
2) The target reference can refer to the target declaration (*e.g.,* A function invocation cannot refer to a method) **AND**
   a) Mapped source and target declarations have no parameters **OR**
   b) All the source parameters are optional **AND** The target reference has no arguments.

Operation

1) Ensure that the target reference uses as identifier the same identifier as the target declaration.
2) Ensure that the target reference refers to the target declaration.

This rule describes a version a bit more complicated than the one presented before by Listing 4. This version ensures that there are no parameters on the declaration. It contemplates the possibility of the source declaration being addressed with no arguments.

For example, the declaration of the variable name uses a type reference that refers to the primitive type string. We map the MS Access type string with the Java class String. The engine would execute this rule: (i) The type reference target is allowed to point to the String class. (ii) The type reference has no parameters, therefore all the conditions are met. The operation will check that the literal hold by the target reference is *string* instead of *String* (mind the uppercase), and therefore modify the type reference to fit in.

C. Rename & Map arguments with static receiver

Condition

1) There is a target declaration mapped **AND**
2) The target reference **cannot** refer to the mapped target declaration **AND**
3) The mapped target declaration is a method or an attribute **AND**
4) The mapped target declaration is *static*.

Operation

1) If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using the parent of the attribute as a receiver.
2) If the mapped target declaration is a method, write a method invocation expression (which is a reference) using the parent of the method as a receiver.
3) Set the arguments in the new target reference according to the mapping information.
4) Set the new target reference to refer to the mapped target declaration.

For example, invocation of the function Date (which yields the date of today). We map the function Date with the static method LocalDate.now. After copying the usage of this function as function invocation in java, the engine would

execute this rule: (i) there is a target declaration mapped: LocalDate.now. (ii) The function invocation cannot refer to it: is a method. (iii) LocalDate.now is static. With all the conditions met: the operation would yield a new expression that will replace the old one. This expression would be a method invocation "LocalDate.now()"

D. Rename & Map arguments with self *receiver*

Condition

1) There is a target declaration mapped **AND**
2) The target reference **cannot** refer to the mapped target declaration **AND**
3) The mapped target declaration is a method or an attribute **AND**
4) The target reference is used in the context of a method **AND**
5) The mapped target declaration is reachable from the calling context by using self or super.

Operation

1) If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using super or self as receiver.
2) If the mapped target declaration is a method, write a method invocation expression (which is a reference) using super or self as receiver.
3) Set the arguments in the new target reference according to the mapping information.
4) Set the new target reference to refer to the mapped target declaration.

For example, invocation of the function *Assert* as used in the example. We map the function Assert with the method Asserter.assertEquals. After copying the usage of this function as function invocation in java, inside a method of a subclass of TestCase (JUnit3). the engine would execute this rule: (i) there is target declaration mapped: Asserter.assertEquals. (ii) The function invocation cannot refer to it: is a method. (iii) Asserter.assertEquals is reachable by using self. With all the conditions met: the operation would yield a new expression that will replace the old one. This expression would be a method invocation "this.assertEquals(...)"

E. Rename & Map arguments with argument *receiver*

Condition

1) There is a target declaration mapped **AND**
2) The target reference **cannot** refer to the mapped target declaration **AND**
3) The mapped target declaration is a method or an attribute **AND**
4) The mapping offers a parameter as a receiver. **AND**
5) The type of receiver can reach the target declaration mapped.

Operation

1) If the mapped target declaration is an attribute, write an attribute access expression (which is a reference) using the argument that responds to the mapping as a receiver.
2) If the mapped target declaration is a method, write a method invocation expression (which is a reference) using the argument that responds to the mapping as a receiver.
3) Set the arguments in the new target reference according to the mapping information.
4) Set the new target reference to refer to the mapped target declaration.

For example, invocation of the function Len as used in the example. We map the function *Len* with the method from *String.length*, to be resolved by the argument.

After copying the usage of this function as function invocation in java, with a string as an argument. the engine would execute this rule: (i) there is target declaration mapped: *String.length*. (ii) The function invocation cannot refer to it: is a method. (iii) *String.length* is reachable by using the argument. With all the conditions met: the operation would yield a new expression that will replace the old one. This expression would be a method invocation "argument.length()"

F. Autowrap

[3] Defines a new declaration that wraps a call to the target declaration respecting the source declaration order of parameters.

Condition

1) There is **no** target declaration mapped **AND**
2) The source declaration was defined in a library **AND** It is **not** a type.

Operation

1) Automatically translates the library component into a pre-established destination.

If a piece of code uses a library artifact that we cannot map, we have two options: (i) dismiss the code,(ii) migrate the library manually. If we dismiss the code we would not be translating this piece of code. We consider then that the only alternative is to migrate manually the library. For easing the job, we propose to generate automatically an equivalent structure to let the user to define the behaviour.

G. Summary: Can a delayed translation rule transform a reference to use the target environment's artefacts & paradigm?

a) Simple rename: renames what ever element that is paradigmatically equivalent, and semantically chosen as equivalent by the mappings of the user.

b) Rename & Map ... with ... receiver: All of them address the problem of transforming a function or variable access into a method or attribute. All of them apply a rename if required, and all of them are able to shuffle parameters order based on the mapping configuration. All of them promote an object as receiver. The *static* rule promotes the defining class. The *self* rule promotes self or super. The *argument* rule promotes one of the arguments.

[3]its implementation may vary from language to language

c) Autowrap: produces a skeleton that responds to the used artefact, leaving the developer to fill-up the blanks.

VII. Validating the rules

As we said before our project stems from a collaboration with Berger-Levrault, a major IT company. Therefore our validation is concerning one of the running migration projects. For validating our approach we did as follow

1) Created a library testing project based on the analysis of an industrial project.
2) We mapped our engine for Java and Pharo languages.
3) We translated the tests on both destinations.
4) We measured by the following criteria
   - Is the resulting test Parsable?
   - Is the resulting test Compilable?
   - Is the resulting test Red, Yellow (or Blue) or Green?

A. Crafting a representative source project

There are many reasons to not use an industrial project as a source for validation. Firstly, the language translation is a big part of our industrial migration, but it is not all. The details required to grasp the overwhelming complexity of the problem cannot fit into half a page in a validation. Second, it will be an undisclosable source of proprietary code, which would make the example less than interesting.

We propose then to make up our source project. But to make it an interesting case of study we require two things: (i) it must respond to the library and runtime usage of a real project. (ii) it must be testable.

1) Choose the library elements: None of the projects we have to migrate uses everything from every library. To validate our approach we address one of our migration projects.

a) Demographics: The chosen project is Microsoft Access project that consists of **18 subprojects**, summing up **1000 UI widgets** and **700k lines of code** and using **19 different libraries**. This project uses almost **690 different elements** from different libraries.

b) First filtering: We use the analysis tool proposed by Bragagnolo et al. [4] to select the most meaning full by applying the Pareto principle (also known as the 80%-20% rule). That yielded a list of 160 elements that represent 80% of library usage in the project. This list is available in validation-results.xls.

c) Second filtering: From the yielded list, we filtered out all the elements regarding MS Access infrastructure, UI, file system or those which cannot be unit-tested.

MS Access infrastructure provides many helping elements unconceivable on the target. For example the“CurrentDB” global. Our language translation project does not consider UI (which will be migrated differently). Our target is either a back-end or a front-end program. While in MS Access makes sense to access the user's files, it is not the same for either back-end or front-end applications. Therefore all the code using those functions must be rewritten. Finally, the ADODB and DAO objects (Recordset, Database and Connection classes), cannot be unit-tested. We originally did the test and try to add it to the validation, but it cannot be unified, since a single testing method over the database requires to have some setup and cleanup. This list is available in validation-results.xls.

2) Writing tests: We implemented an extremely humble testing library in MS Access . Each test is a function. We have no setUp. We have a single implementation of Assert which receives two entities to compare for equality and a text to show if the assertion fails.

We created four different modules according to the kind of tested entity: types, functions/subprocedures, constants/globals and methods.

Testing types: The test asserts the default value of a declared variable. Example given in Listing 5.

```
Public Function test_ErrObject()
   Dim var As ErrObject
   Call Assert(var Is None, True, "None expected")
End Function
```

Listing 5: Default value is None.

Testing Constants: The test that asserts the default value. Example given in Listing 6.

```
Public Function testVbNullString()
   Call Assert(vbNullString, "", "Constant Expected")
End Function
```

Listing 6: Null string is an empty string.

Testing Functions: The test asserts examples of usage. Example given in Listing 7.

```
Public Function testUCase()
   Call Assert(UCase("hellow"), "HELLOW", "Upper case
     expected")
End Function
```

Listing 7: Upper case must be upper.

3) The source project: The source application is available PaperTest.accdb.

The source project we propose for the validation consists of **55 tests** split in four different modules. 14 tests of constants and globals, 18 tests of types, 21 tests of functions.

B. Getting to know our targets

The target applications are available in the target folder.

Our validation translates 53 tests, which were detailed in Section VII-A, to two different targets. Here we briefly describe what we find in these two target projects.

1) Common decisions: This validation is not about migrating tests to be used by a Test framework, since we have found none in MS Access and there is no usage of something similar in our industrial projects. Therefore, to simplify and avoid polluting the engine configuration, we expect the target projects to have four different testing classes to receive the translation of the tests of each of our four MS Access testing modules.

2) The Java target: Java is a popular statically typed object-oriented language. The target is a Maven project configured to compile Java 1.8 and with a single dependency: JUnit3. This dependency is required to support testing. The rest of the available dependencies are the many definitions available in the JDK. As previously existing code, to simplify the

experiment, we defined four empty test case classes using the Eclipse default template. These classes are a subclass of the junit.framework.TestCase class. We find also a class named Example. This class exists because of a technological restriction of our Java ASG loading module: it only loads used pieces of the libraries. This class is not used at any moment.

3) The Pharo target: Pharo is a popular open-source dynamically typed pure object-oriented language, inspired in Smalltalk. The target is a Pharo package that includes the four expected TestCase classes developed in Pharo 10. Pharo ships many dependencies within the Pharo image, including from the definition of Object and SmallInteger to the SUnit testing framework. During our Pharo ASG loading process we cut down the available dependencies to ensure that we do not load useless stuff. For this experiment we cut down the dependencies to the following packages: 'Kernel-BasicObjects', 'Kernel-Objects', 'Kernel-Numbers', 'Kernel-Chronology', 'Kernel-Exceptions', 'Collections-Abstract', 'Collections-Sequenceable', 'Collections-Unordered', 'Collections-Strings', 'SUnit-Core'.

C. Rules and Mappings

1) Installed Rules: For both engines, we used the same kind of translation and adapting rules. We use the following five translating rules: ModuleToClass, FunctionToMethod, Copy, (three of them discussed in Section V-C1) and GlobalToAttribute, which transforms all global variables or constants into a static attribute. We need an extra rule to transform three different binary operations according to the target language. CopyReplaceBinaryOperator, which copies a BinaryOperator structure replacing the operator. There are two ways to express equality in MS Access : A Is B and A = B. Both are translated as A == B in Java and as A = B in Pharo. There is also the concatenation in MS Access : A & B. Translated as A + B in Java, and A , B in Pharo. We want to note that this set of rules is not enough to translate a full MS Access program since control flow statements are not taken into account in our experiment, since they are out of scope. For the adapting rules, we use the five rules discussed in Section VI.

2) Mappings: From the mappings point of view, we configure the engine according to the target ASG. Table I show some elements that we mapped in one language and not in the other. The full list can be found in validation-results.xls.

TABLE I: Mapping examples:

VBA	Java	Pharo
Information.TypeName	None	Object.className
Interaction.IIf	None	Boolean.ifTrue:ifFalse:
Recordset.AddNew	ResultSet.moveToInsertRow	None
VBALang.Void	void	None

D. Results

The translating application ready to reproduce the experiment is available in Application.zip.

Table II presents an overview of the testing modules Constants, Functions and Type tests. The full detailed list of the results is available in validation-results.xls.

TABLE II: Translation Results.

Test case	Total	Mapped	Parsed	Compiled	Test Result		
					S	F	E
Constants	14	1	14	1	1	0	0
Functions	21	11	21	16	8	3	5
Type	18	13	18	9	4	4	1
Java Total	53	25	53	26	13	7	6
Constants	14	4	14	14	3	0	11
Functions	21	14	21	21	10	3	8
Type	18	13	18	18	7	10	1
Pharo Total	53	31	53	53	20	13	20

Test Legends: (S)uccess, (F)ailure, (E)rror.

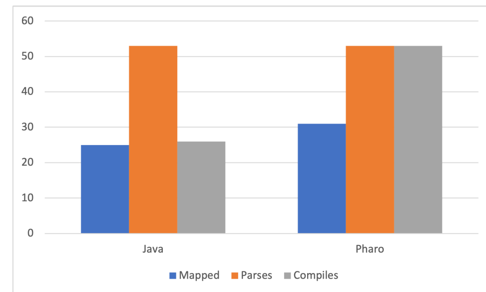

Fig. 3: Bar chart comparing the number of mappings and the impact on parsing and compiling.

We want to insist on the fact that the engine was configured with the same set of translation rules (with a slightly different configuration), and with the same set of adapting rules to produce two different OOP targets, with different types of typing systems and libraries. The main difference between the configurations was the source and target ASG mappings.

a) Mapping Parsing and compiling: Figure 3 shows how many entities were successfully mapped, how many translated tests were successfully parsed, and how many of them were correctly compiled. For Java, we mapped 47% of entities. 58% of them in Pharo. For both targets, the produced source code is verified by the SmaCC Java [4] parser and the Pharo parser. 47% of the translated testing methods are compiled by the Java compiler. 100% are compiled by the Pharo compiler. The compiling rate seems to be tightly related to the mapping in the case of Java since the compiling errors come from wrong typing. We argue that most of them stem from the MS Access type named Variant, which we mapped to Object in Java. MS Access Variant accepts any value but keeps the type of the element (object or not). In Java, the type Object accepts only objects (no primitive values allowed) but if a variable is typed as Object it loses all singularity.

b) Tests: Figure 4 shows in a stacked bar plot the amount of executed tests split by the result: success, failure and error, and at the same time, it split the errors and failures by kind. All the successes make sense, so we do not classify them. We have three kinds of failures: Only in Java do we find failures because of the wrong literal translation. *i.e.,* testing if a double variable value is 0, it must be used 0d. Only in Pharo do we find failures related to untyped variables. In Pharo, all the variables start with nil because all possible value is an objects.

[4]https://github.com/j-brant/SmaCC

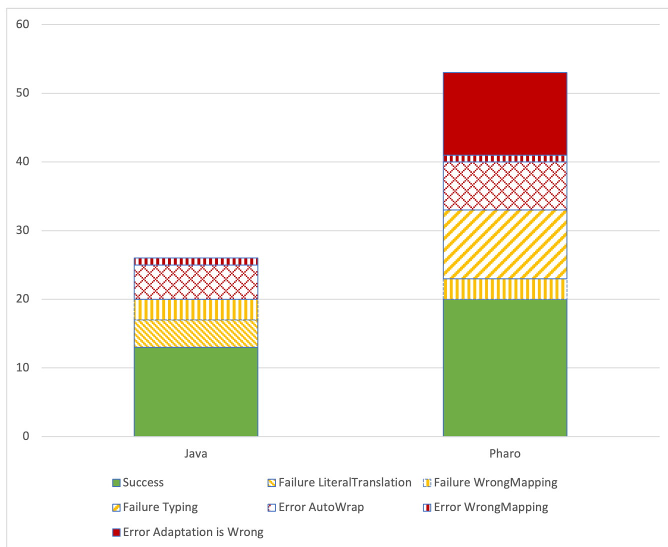

Fig. 4: Stacked bar chart comparing tests results per language

In both targets, we find failures due to bad mapping. Some functions seem to do the same this is not exactly true. Str(3) is expected to return " 3" note the whitespace. We found no method doing the same in Java or Pharo. We have three kinds of errors: On both targets, we find errors due to the usage of the Autowrap rule. These errors a fully expected, since the rule generates methods that raise exceptions asking the developer to implement the behaviour. On both targets, we find errors due to wrong mapping. *i.e.,* MS Access allows using strings indicating time to create a Date. Finally, only in Pharo do we find errors because the job of the Autowrap rule was not enough: In Pharo, all the attributes of an object or class are only accessible by the class itself or by subclasses. Meaning that when the Autrowrap rule creates equivalent globals (as class side "static" attributes) we must also implement the accessors. And all the accesses to attributes must be rewritten as accessor invocation.

VIII. Threats to Validity

There are two main threats to the validation proposed.

a) Tests cases coverage: The written tests are based on simple usage of functions. We did not investigate in depth to understand the limit of validity of the MS Access functions in order to test not only a simple case but also the limit cases.

b) Tests simplicity: The second threat is that the validation is done over a set of simple tests. We focused strictly on *Reference* objects, what took out of the equation another kind of construction such as control flow, and the analysis of how control flow translation rules work together with adaption rules.

IX. Future Work and Conclusion

The main hypothesis of this article is that we can adapt code based on the nature of the artefact. ....

References

[1] *Aho Alfred V., Sethi Ravi, Ullman Jeffrey D.* Compilers: Principles, Techniques and Tools. Reading, Mass.: Addison Wesley, 1986.

[2] *Aho Alfred V., Ullman Jeffrey D.* The Theory of Parsing, Translation and Compiling Volume I: Parsing. 1972.

[3] *Appel Andrew W.* Modern compiler implementation in Java. New York, NY, USA: Cambridge University Press, 2002. Second. with Jens Palsberg.

[4] *Bragagnolo Santiago, Ducasse Stéphane, Anquetil Nicolas, Seriai Abderrahmane, Derras Mustapha.* Alce: Predicting Software Migration. 2022. working paper or preprint.

[5] *Brant John, Roberts Don, Plendl Bill, Prince Jeff.* Extreme Maintenance: Transforming Delphi into C# // ICSM'10. 2010.

[6] *Cossette Bradley E., Walker Robert J.* Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries // Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. New York, NY, USA: ACM, 2012. 55:1–55:11. (FSE '12).

[7] *Ducasse Stéphane, Polito Guillermo, Zaitsev Oleksandr, Denker Marcus, Tesone Pablo.* Deprewriter: On the fly rewriting method deprecations // JOT. 2022.

[8] *Kontogiannis K., Martin J., Wong K., Gregory R., Müller H., Mylopoulos J.* Code Migration through Transformations: An Experience Report // Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research. 1998. 13. (CASCON '98).

[9] *Mateus Bruno Góis, Martinez Matias, Kolski Christophe.* Learning migration models for supporting incremental language migrations of software applications // Information and Software Technology. 2023. 153. 107082.

[10] *Terekhov A. A., Verhoef C.* The realities of language conversions // IEEE Software. XI 2000. 17, 6. 111–124.