Towards easy program migration using language virtualization

Théo Rogliano, Guillermo Polito, Pablo Tesone

To cite this version:

Théo Rogliano, Guillermo Polito, Pablo Tesone. Towards easy program migration using language virtualization. IWST19 - International Workshop on Smalltalk Technologies, Aug 2019, Cologne, Germany. hal-02297756

HAL Id: hal-02297756

https://hal.archives-ouvertes.fr/hal-02297756

Submitted on 26 Sep 2019

Towards easy program migration using language virtualization

Théo Rogliano
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
Lille, France

Guillermo Polito
CNRS - UMR 9189 - CRIStAL, Univ. Lille, Centrale Lille, Inria
France
guillermo.polito@univ-lille.fr

Pablo Tesone
Pharo Consortium
INRIA Nord Lille Europe
Lille, France

Abstract

Migrating programs between language versions is a daunting task. A developer writes a program in a particular version of a language and cannot foresee future language changes. In this article, we explore a solution to gradual program migration based on virtualization at the programming language level. Our language virtualization approach adds a backwards-compatibility layer on top of a recent language version, allowing developers to load and run old programs on the more recent infrastructure. Developers are then able to migrate the program to the new language version or are able to run it as it is. Our virtualization technique is based on a dynamic module implementation and code intercession techniques. Migrated and non-migrated parts co-exist in the meantime allowing an incremental migration procedure. We validate it by migrating legacy Pharo programs, MuTalk and Fuel.

Keywords Migration, modularity, virtualization.

1 Introduction

Migrating programs is a complex task. A developer writes her programs targeting a version of a programming language. As languages, libraries and frameworks evolve, programs using them become obsolete. The developer tries to keep her programs as much as possible compatible but it is not always possible or straightforward [12, 19]. Changes in the language may break her programs. For example, in Pharo 2 there was a trait named TBehavior whereas now it does not exist anymore. Programs using that trait are now broken. Moreover, this problem is not exclusive to program forward-porting (Pharo2 -> Pharo7) but also to back-porting (Pharo7 -> Pharo2) and to porting between language dialects (Squeak -> Pharo).

When a language evolves to a certain point, the differences are so important that it becomes impossible to execute pieces of code written in an old version of the language. We identified the following evolutions provoking incompatible code (Section 2):

Syntax changes. Changes in the syntax make programs that were valid for one version invalid in another.

Standard Library changes. Changes in the classes of the standard library invalidate the code that uses it *e.g.*, methods or classes removed, or methods with modified semantics.

Meta model changes. Reflective languages expose their meta model to the programs [11]. If the meta model changes from one version to another, programs using it are not able to run correctly anymore.

We explore the problems of program migration using a virtualization approach. Virtualization is well-known in Operating System (OS) and the problem it solves is similar to ours. For example, Games running in Windows XP do not run natively on Windows 10 due to different kernel infrastructure, file system and so on. Windows uses virtualization in the form of a compatibility layer to solve this issue. The technique consists of interposing a layer between a program and the OS, called *hypervisor*. The role of the hypervisor is to be an adapter: if the behavior is present it transmits calls to the OS else it adapts them to make them compatible with the new API.

We want to achieve the same results with a programming language instead of an OS. This means interposing a virtualization layer between a program and a language in order to run old programs in a newer version of the language.

Toward this goal, we present a set of techniques required to be able to load and run programs intended for older versions of the host language (Section 3). We implemented the virtualization layer through the use of dynamic modules and metalinks for code rewriting. Once the program is loaded, developers are able to update the program to the new language version or to run it as it is using the virtualization layer. We validate the chosen infrastructure with different migrating scenarios of Pharo programs (Section 5).

Contribution. The contributions of this paper are:

1. the design of a dynamic module system that allows us to load partially broken code;
2. the identification of a set of migration problems and a corresponding set of intercession techniques to make them compatible to recent systems transparently;
3. an experience report on migrating two real-world libraries written from 7-9 years.

IWST'19, July, 2019, Cologne, Germany
2019.

2 Motivation: Incompatible programs

When a developer tries to execute programs developed in an older version into a newer version of the language, 3 main situations occur:

Successfully loaded and running. This is the state we want the program to be in.

Successfully loaded but not running. The program was loaded but some unexpected bugs appear. They make the program unusable. Since the program is loaded the developer is able to debug it. She changes the source code of the program thus making it incompatible again with the version it was conceived for.

Unloadable. The program cannot be loaded. In order to correctly load the program, the developer should debug the loading process. Once the incompatibilities are discovered, the program source code is modified to comply with the newer language version. Finally, the program is loaded and it arrives to one of the previous state.

The 2 last cases are the problematic ones and are due to language changes. We illustrate these changes through examples:

2.1 Syntax changes

Syntax changes break the loading of a program. For example, in old Pharo versions, the parser accepted two different syntaxes for assignments, with an underscore a _ 2 or with a colon followed by an equal $a := 2$. The former gradually disappeared and now is not supported anymore. A program using underscore as assignment is then not understood anymore by the compiler therefore it does not compile.

2.2 Standard library changes.

Standard library changes break both loading and execution.

We can split them in three categories:

Renaming of methods or classes. For example, the method String » includesSubString: changed into String »includesSubstring:. Both the loader (to get the source code) and the program relied on the former one resulting in a doesNotUnderstand during the loading process then during the execution.

Removal of methods or classes. The class PackageInfo does not exist anymore breaking program relying on it.

Behavioral modifications. A method classNamed: which was returning nil if the class is absent, now returns an Exception. The code expecting a nil in this case is not valid anymore and must catch the exception instead. [1]

[1]We consider refactorings as a composition of the previous changes. We will not detail them.

2.3 Meta-model changes.

Changes in the meta model also occur and break the loading process. As an example, the Trait class has its own hierarchy similar to the one of Class in Pharo 2. Both inherit from Behavior and ClassDescription and use a trait TPureBehavior. In newer versions of Pharo, this hierarchy has been updated and Trait is now a subclass of Class. The meta side of traits has been reworked making their creation different and any code relying on reflection on traits is obsolete.

3 Modules for Virtualization

To illustrate with an image we will reuse, an object oriented program can be viewed as a patchwork or a puzzle consisting of different entities, namely packages, classes, methods and objects (Figure 1). They are linked by mainly three relationships: subclassing, reference and message-send.

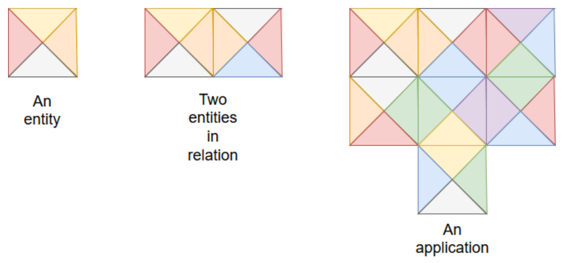

Figure 1. A program seen as many related entities forming a puzzle.

Changes are associated with loosing some pieces of the puzzles or the way they are linked (Figure 2). Our goal is to make new pieces looking alike the old ones and manage to link them. This is the role of the hypervisor.

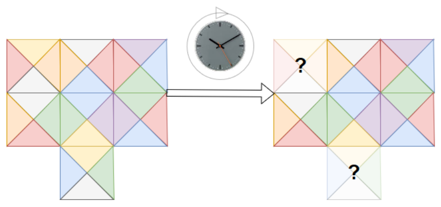

Figure 2. Missing parts as time goes (*e.g.*, removing classes from standard library)

We propose the introduction of language hypervisor *i.e.*, a compatibility layer at the level of the programming language. This component intercepts all the interactions of the loaded code and the language environment (Figure 3). In this section, we will detail the prerequisites for creating a hypervisor as we intend. The following features are mandatory.

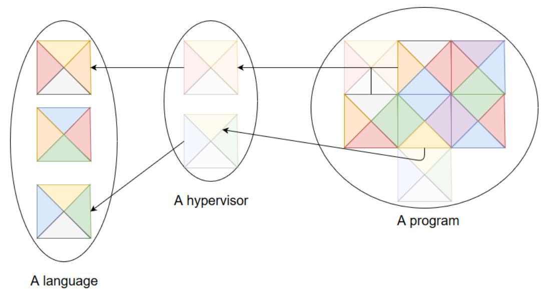

Figure 3. All the calls to missing parts from the program are intercepted by the hypervisor and redirect to an equivalent in the host environment.

3.1 Encapsulating changes

An unexpected side effect of loading an old application is the corruption of the host language environment. For example, the program depends on a different version of the class String. The loading process manages to install version of String required by the program, replacing its own String. The host environment ends up with this version of String which is not compatible with other current components. This is the worst scenario as it breaks all the new libraries and cannot be debugged. We require that the loading process does not affect the host environment. We propose to solve this problem using encapsulation techniques, here in the form of a module system (Figure 4) .

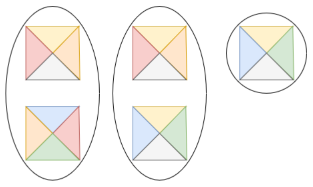

Figure 4. Encapsulated entities in different modules. They are only aware of what is inside the module (oval).

3.2 The module system in a nutshell

Our modules are composed of sets of name bindings:

Defined. A set containing the entities defined inside the module.

Import. A set containing a binding to the entities we used that come from the outside.

Undefined. A set containing existing references in the module to entities not defined nor imported.

A module has the same API as the host environment API so tools in the host environment are able to interact with the modules transparently. Also the modules introduce the API needed for its own management such as class creation for itself, import, undefined references and so on. A same name is allowed to appear in different modules, as each of these bindings are independent.

Now, with the earlier example, the loading process loads the program in a module. The class String expected by the program will be loaded in the module and will be different than the one of the host environment. Doing so, the incompatibility is isolated inside the loaded module. It only affects its execution. Moreover, the developer is able to debug this incompatibility and fix it without affecting the stability of the host environment.

In our proposed solution, we have 3 modules: the hypervisor, the incompatible program under test and the host language environment.

3.3 Late class creation

Now that we have the possibility to safely load the program to virtualize, we can take a look on the loading difficulties. The first cause of loading failure are missing classes that prevents us from loading all the hierarchy below them (Figure 5). New classes in Pharo are created by sending the message #subclass: to the superclass as shown in Listing 1.

```
Point subclass: #ColorPoint
   instanceVariableNames: 'color'
   classVariableNames: ''
   category:''
```

Listing 1. Example of class creation.

The superclass is then in charge of class creation by configuring a class builder with the arguments of the message. In case of a missing superclass, which is a common scenario, we cannot create all the subclasses, the compiler will not manage to resolve the receiver of the message subclass:. In Listing 1, if the class Point is not available, we cannot send the subclass message to it, hence we cannot create the class ColorPoint. A similar problem appears with missing traits, the users of the missing trait will not be configured properly and will certainly be broken. From now on, we will use the term *behavior* to refer to classes and traits.

To handle this problem, instead of executing a class definition directly; we transform the class definition message-send into an Abstract Syntax Tree. It allows us to detect missing references and to react to them. The class builder has been enhanced to be configurable with such ASTs. For example, a possible solution is to change the superclass of ColorPoint to another class (*e.g.*, Object) in the model and ask the builder to make the class.

This solution works when there is a direct replacement of the missing class. (*e.g.*, if the class Point has just been renamed.) However, if there is no direct replacement we change the intended original definition and it will continue to break the program. So, we require to have a better abstraction for missing entities.

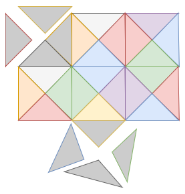

Figure 5. Behaviors are missing and we cannot load the fractured pieces.

3.4 Representation of missing entities

We need to have a representation of missing behaviors for two reasons. As stated above, for resolving, at least in a temporary way, the missing references to those behaviors. The second reason is we want to keep track of the dependencies of the loaded program.

We want to load the program under test without modifying it. In term of pieces of a puzzle, we want to keep the expected shape of the pieces (Figure 6).

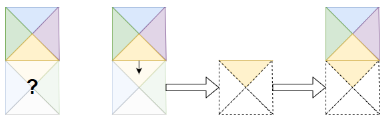

Figure 6. We know a piece is missing. We know it was expecting a yellow part on the top. We create a stub piece (dashed lines) with a yellow part on top and link it, waiting for a better replacement

For that purpose, we used stubs classes or traits that will contain the information obtained through static dependency analysis. We call those stubs *undefined classes* for classes and *undefined traits* for traits.

We encounter three scenarios:

Undefined classes. We create an undefined class by subclassing the class UndefinedClass. We followed the proposal solution of Polito et al [10]. The latter redefine basicNew and doesNotUnderstand to prevent instantiation.

Undefined traits. UndefinedTrait is a subclass of Trait. Only the method isDefined is redefined to return false. The creation of undefined trait is the same as for regular trait. A trait composition with an undefined trait becomes undefined. An undefined trait is created when a behavior refer to a missing trait.

Undefined traited classes. An undefined traited class is a class that uses an undefined trait. As for undefined classes we prevent the instantiation, here by applying UndefinedTraitedClass on the class side of the target class which redefines basicNew.

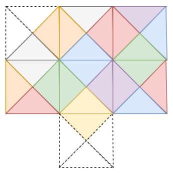

Figure 7. A program loaded with two undefined pieces. The undefined pieces will be replaced later with real pieces.

At first, we load our program with stub classes (Figure 7). Once we found the corresponding classes, we are able to throw away those stubs easily and replace them.

3.5 Dependency management

In some cases, the missing entities are still present in the host environment and are compatible, so we want to use them. In the last example Point is missing in the module containing the program but is present in the host environment. We would like to create a stub for Point and make it inherit from the Point from the host environment. We need then to import the point from the host. This creates a name conflict, we have two times "Point" inside the same module. Point from the host which will stay in the host but is imported and the subclass Point we want to create. For doing so, we suggest aliasing as a solution for this problem. We import Point by giving it another name, an alias. With this mechanism, you can solve the problem in different ways. As show in Figure 8), you can use alias in the program's module or in the hypervisor.

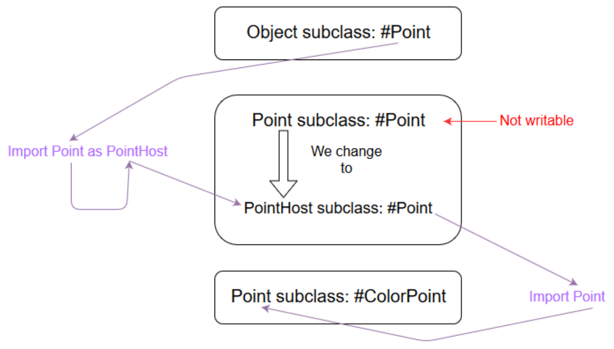

Figure 8. We cannot ask for the first subclass message in the hypervisor. The compiler will resolve it as the same and we will build a circular hierarchy. Instead we rename the Point from the host environment in the import.

With late class creation, undefined entities, modules and aliases we are able to handle all the hindrance for creating a hypervisor.

4 Virtualizer

The hypervisor is a module containing all that it is necessary for the program to work and be compatible with the newer version of the language. It contains all modifications and adaptations required to run the program in the host environment without modifying the program nor the language environment.

For example, it contains the undefined entities of the program's module, the program's module imports them from the hypervisor. Having the undefined entities in a module allows us to resolve them without modifying the host environment nor the loaded program (Figure 9).

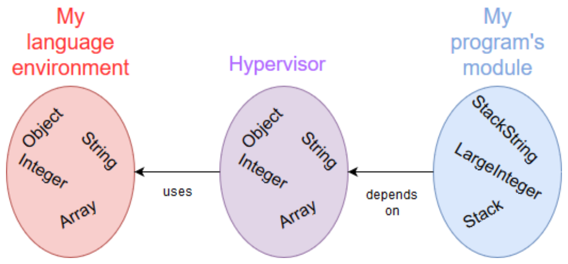

Figure 9. The undefined entities of the program are in a module called hypervisor. They are isolated and modifying them impact only the hypervisor.

4.1 Intercession techniques for virtualization

We have 3 strategies for resolving incompatibilities:

Kernel indirection. We still have an entity with the same name in the host environment. We assume some methods are similar and we want to reuse them. We then discard the undefined behavior and declare a new subclass with the same format than the one from the host environment. This way the methods are found with a regular look up. If a method is still missing we create it in the newly created behavior in the virtualization layer.

Catching messages. This strategy is for missing classes without similarities with the ones of the host environment. Our only information is their names, we lost their methods or their layout. In some cases, we still have the opportunity to consult old versions of the language to retrieve the original classes. Under other circumstances, we have to implement them from the start. We define those by subclassing Object. We then run the tests and catch the doesNotUnderstand messages which contains the names of the expected methods. We then implement them by assuming the behavior by testing (similar to Test Driven Developpement [1]).

Metalinks. In less common cases, a method still exists but has been renamed. The lookup raises a doesNotUnderstand since the method was not found. We solved the problem with metalinks [13]. Metalinks are used to annotate AST nodes. An annotated AST is expanded, compiled and executed on the fly thanks to the ReflectiveMethod/CompiledMethod twin. In our case, using metalinks allows us to replace the execution of the original method with the new one. Even more, we are able to translate different message-sends (*e.g.*, the parameters have different orders between the versions). The source code stays untouched while we still execute the expected method.

4.2 Using the virtualizer: an example

In order to use our virtualization solution the developer should perform the following steps.

1. She installs the program that works in a different version of our language inside a module. During the loading of the program, the module system detects missing dependencies and generates undefined entities for them.
2. The virtualizer component transfers the undefined entities to another module that is going to become the hypervisor. We decomposed this in 2 stages, creating the missing dependencies in the hypervisor and then importing them.
3. She runs the program's tests to find out potential problems to solve.
4. She analyzes and determines the best course of options (if there is one) to simulate the missing behaviors or the semantic mismatchs applying one of the stated strategies.
5. When all the tests are processed and passed, it means that the program is operational.

5 Validation

In order to validate our infrastructure, we port 2 old Pharo programs to a newer version of Pharo.

Methodology. We consider a migration is successful if all the tests are passed. If all the tests are passed, we assume the program is working. Moreover we assume that the tests are passed in the original version of Pharo. We only choose programs with tests for this purpose.

For each of the programs to run we executed the steps stated in Section 4.

Selected Scenarios. We chose 3 scenarios. The first one is an exchange of libraries and serves the purpose of testing the module system. The other two are program migrations bringing each different challenges. The migration of MuTalk was chosen for compiler challenges and the migration of Fuel for file and objects creation challenges.

Threat to validity. In the case the test coverage is low, we can only show that the tested parts are in the same state as in earlier version. Non tested code is still supposed broken.

5.1 System/container collection

This scenario is our calibration test for modules. We use the system tests (the tests of the current implementation of the language) with an alternate implementations of some collections and, the opposite, use another tests implementation on the system collections. It allows us to assert that we can run tests on independent implementations.

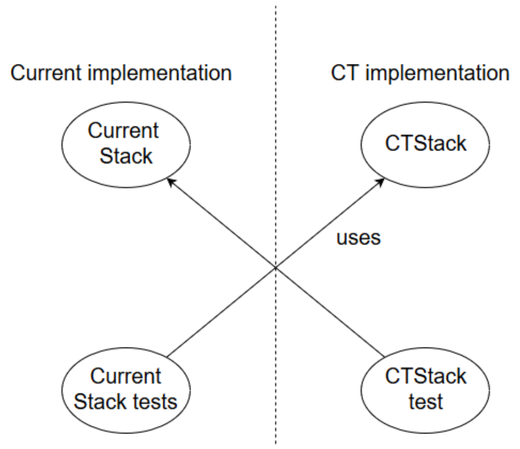

Figure 10. Exchanging tests between implementation.

We chose the Stack and LinkedList collections. In the system, the implementation of Stack is based on LinkedList whereas we introduce another implementation, called CTStack, using an array (Figure 10). In the modules containing the tests, we decide which implementation to use by redefining the Stack import. Both implementation pass the tests of the other one. For LinkedList the general principle is the same, except two redefinitions of import are done, LinkedList and Link. We noticed that it's possible to mix the system-LinkedList with the CT-Link and vice versa.

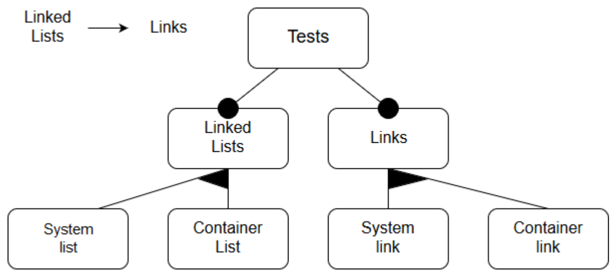

Figure 11. Feature diagram of linked list tests

As for the results, all tests pass with a green light. This scenario showed us that the module system produces isolated modules that are freely linkable between each other.

5.2 MuTalk

In this scenario, we produce a hypervisor for MuTalk[2]. MuTalk a mutation testing framework which last commit dates from 2013. We know the framework was working in old versions of the language (*i.e.,* all tests were green). We chose it because mutation testing implies modification of methods at runtime bringing interesting challenges since there is a need of interfacing with the compiler.

Indeed thanks to this experiment we found a problem when the program is modifying CompiledMethod objects (object representing methods). In order to generate mutants the program asks methods for their source code with the getSource message and modifies it. In this scenario the getter for source code has changed to sourceCode and we are stuck with a doesNotUnderstand message (Figure 12). The hypervisor is not involved in the process since the compiler uses the CompiledMethod from the system. This problem is a metamodel problem and allows us to develop our metalinks strategy.

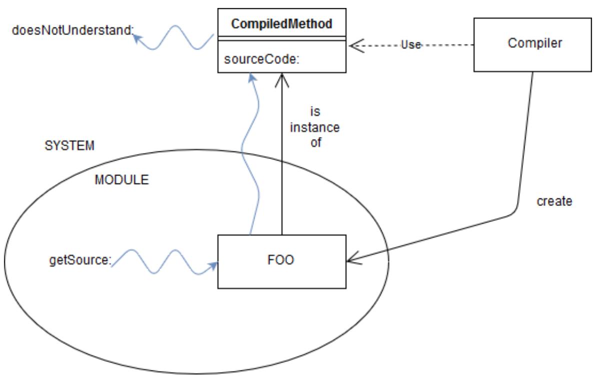

Figure 12. Scenario with a doesNotUnderstand

In this validation scenario, we identified 45 references to missing entities in the framework. Out of the 45, 30 are classes with the same name that a classes in the system. We used the kernel indirection strategy in those cases. For the others classes we used catching messages strategy. We retrieved methods from Pharo 2 and implemented them in only 3 classes. With our different strategies and thanks to the use of metalinks, we managed to pass 355 out of 362. The 4 failures and 3 errors remaining means some behaviors are still missing or that we did not manage to retrieve the correct semantic for some methods.

Note: We only talked about fixing 33 references out of 45. In the 12 remaining, 4 are global objects referenced with the same name that we still have in the system. We applied something similar to the kernel indirection strategy. The missing 8 references could be the cause of the failing tests or just present in the program but not used at all (dead code).

5.3 Fuel

This scenario is about the virtualization of Fuel [3] version 1.6, last comitted in 2012. Fuel is a serialization/materialization program still maintened to this day. Serialization implies I/O management, file management and assumptions on objects.

[2]https://github.com/pavel-krivanek/mutalk

[3]https://rmod.inria.fr/web/software/Fuel

This scenario presents an extra challenge. A newer version of Fuel exists in Pharo and conflicts with our.

This time, we identified 79 missing references. 66 have been resolved using the kernel indirection strategy. We quickly got results in applying the catching message strategy since the former bring us to our first hindrance.

This version of Fuel uses the class FileDirectory which does not exist anymore. We retrieved an implementation of FileDirectory from Pharo 1. In addition, a problem arises because Fuel relies on an old implementation of streams. To get around this matter we added in the hypervisor a stream compatibility for files using the new implementation of streams and FileReference. Thanks to this workaround we managed to use an erzatz of FileDirectory and to get back the file management.

Fuel uses extension methods on system classes but we do not have any support for them yet. The question is how to handle extension methods without breaking the properties of our modules. For example, an extension method foo expects to be installed in the class String. We do not want it to be installed in the class String of the host environment. Conflicts arise in case another method foo has been implemented in the meantime. Moreover, by doing so we break our property to keep the host environment intact. Creating a String class in the hypervisor specifically and installing the method do not work. Moreover, similarly to the CompiledMethod problem in MuTalk, the compiler use the String of the host environment for creating strings. Those strings will then not understand the method foo. We plan on handling extension methods in the future.

Finally, Fuel assumes that it works in a single environment, the Smalltalk global environment. It serializes and materializes objects in this environment. Since we installed it in an isolated environment, it does not have free access to the host environment (nowadays Smalltalk) creating incompatibilities. For example, asking Fuel to materialize a LargeInteger fails. Fuel asks the class using reflection to create the object. The class LargeInteger was not an import of Fuel and is not in its environment because it accesses it dynamically. It cannot ask LargeInteger to create the large integer requested. Moreover, Fuel also uses Smalltalk for reflective operations on itself and does not find itself in it provoking incompatibilities too. Reflective libraries require more work in case of their introduction into a module system, for example an installing environment for objects. We also plan to overcome this problem in future works.

The solution is incomplete with failure of a lot of tests (19 passes/239 tests) but this scenario taught us a lot on unforeseen difficulties and on a new kind of incompatibilities.

5.4 Lessons learned

Thanks to the previous validation scenarios, we learned that our solution encounters its limits when the program tries to do reflective operations. For example, in a library exists a specialization of String called MyString with a method lookForAnother that look for a specific method only in the superclass (String). Since in our solution, we insert a class between String and MyString, the method lookForAnother fails since it will look in the superclass which is not String anymore. Those cases need to be analyzed one by one and are resolved with metalinks. We have a similar problem with the resolution of symbols. They are, in Pharo, resolved in a global scope. With the introduction of a module system symbols are now resolved inside the module scope instead of the global one. Though, in some cases like in Fuel, we would like to not only resolve symbols in a module scope but in something more "global" like a set of modules.

6 Related Work

6.1 OS virtualization

A similar solution is used for porting applications in different versions of operating systems. Windows's application compatibility layers (Application Compatibility Toolkit[4]) attempts to run poorly written applications or those written for earlier versions of the platform. It exposes a layer that emulates the old versions of the API.

FreeBSD includes a Linux compatibility layer (https://www.freebsd.org/doc/handbook/linuxemu.html) that enables binaries built specifically for Linux to run on FreeBSD. FreeBSD also provides other Unix-like system emulations, including NDIS, NetBSD, PECoff, SVR4, and different CPU versions of FreeBSD. NetBSD includes similar Unix-like system emulation.

All these virtualization layers are provided by the platform without any support to modify them. Moreover, it is impossible to create extensions to support applications that expect old versions of third party libraries (*i.e.,* libraries that are not part of the operating system).

6.2 Application migration

In the literature, solutions exist to help developers in migrating applications. For example, they analyze and extract the required refactorings from one version of the host environment to another [5, 6, 8, 19]. These solutions provide different level of automatic generation of refactorings. For example, Wu et al [19] propose a solution to automatically extract the required refactorings from one version to another of a library comparing the source code and the call graph. Henkel et al [8] propose to generate the set of refactorings from capturing the changes performed by the developer. However, they all modify the original program making it incompatible with the older version of the language.

[4]https://docs.microsoft.com/en-us/windows/desktop/win7appqual/application-compatibility-toolkit--act-

6.3 Encapsulation techniques

Solutions exist to encapsulate changes. Allan Wills propose Capsules. Capsules in Fresco [17] chose to modify the class definition in order to have the same name several times. In their solution, a class owns meta-data such as the author name, the date of creation and the prerequisites of the classes. The idea is to encapsulate a class like data in a network. Modular Smalltalk [18], The Java Module System [16] Newspeak [4] and Matriona [15] offer encpsulation solutions based on name scoping. Some solutions based , this time, on methods scoping [3] also exist. Bergel et al. analyze module diversity and explain many of them [2, 14]. Finally some solutions [7, 9] allow the developer to modify the program under migration without affecting the host environment. However, these solutions do not provide a way of performing the changes without affecting the original source code.

7 Conclusion

In this article, we presented a set of language changes causing problems in loading or running programs. We proposed a solution based on virtualization to migrate programs despite those problems. We designed an infrastructure based on dynamic modules and metalinks in this goal and chose to validate it with real migration scenarios. Although our solution does not yet cover the entire selected scenarios, the selected programs load and are expected to run. The infrastructure needs to take into account extension methods and have a better environment handling.

As a future work, we plan to extend our solution to include the missing features. Moreover, we plan to continue using this solution to migrate old application to newer versions of Pharo. Other possible path of evolution is the automatic hypervisor generation using refactoring detection techniques[8, 19].

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. The work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.

References

[1] Kent Beck. 2002. *Test Driven Development: By Example.* Addison-Wesley Longman.

[2] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. 2005. Analyzing Module Diversity. *Journal of Universal Computer Science* 11, 10 (Nov. 2005), 1613–1644. http://rmod.inria.fr/archives/papers/Berg05cModuleDiversity.pdf

[3] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. 2003. Classboxes: A Minimal Module Model Supporting Local Rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03) (LNCS)*, Vol. 2789. Springer-Verlag, 122–131. https://doi.org/10.1007/b12023 Best Paper Award.

[4] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10)*, Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 405–428. http://dl.acm.org/citation.cfm?id=1883978.1884007

[5] Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 507–511.

[6] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes.. In *icsm*, Vol. 96. 359.

[7] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. 2007. Encapsulating and Exploiting Change with Changeboxes. In *Proceedings of International Conference on Dynamic Languages (ICDL 2007)*. ACM Digital Library, 25–49. https://doi.org/10.1145/1352678.1352681

[8] Johannes Henkel and Amer Diwan. 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings International Conference on Software Engineering (ICSE 2005)*. 274–283.

[9] Jens Lincke and Robert Hirschfeld. 2012. Scoping changes in self-supporting development environments using context-oriented programming. In *Proceedings of the International Workshop on Context-Oriented Programming*. ACM, 2.

[10] Guillermo Polito, Stéphane Ducasse, and Luc Fabresse. 2017. First-Class Undefined Classes for Pharo. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies (IWST '17)*. ACM, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/3139903.3139914

[11] Fred Rivard. 1996. Smalltalk: a Reflective Language. In *Proceedings of REFLECTION '96*. 21–38.

[12] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 56, 11 pages. https://doi.org/10.1145/2393596.2393662

[13] David Röthlisberger, Marcus Denker, and Éric Tanter. 2008. Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime. *Journal of Computer Languages, Systems and Structures* 34, 2-3 (July 2008), 46–65. https://doi.org/10.1016/j.cl.2007.05.001

[14] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. 2004. Composable Encapsulation Policies. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'04) (LNCS)*, Vol. 3086. Springer Verlag, 26–50. https://doi.org/10.1007/b98195

[15] M. Springer, H. Masuhara, and R. Hirschfeld. 2016. Hierarchical Layer-Based Class Extensions in Squeak/Smalltalk. In *Modularity'2016*.

[16] Rok Strniša, Peter Sewell, and Matthew Parkinson. 2007. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*. ACM, New York, NY, USA, 499–514. https://doi.org/10.1145/1297027.1297064

[17] Alan Wills. 1991. Capsules and Types in Fresco. In *Proceedings ECOOP '91 (LNCS)*, P. America (Ed.), Vol. 512. Springer-Verlag, Geneva, Switzerland, 59–76.

[18] Allen Wirfs-Brock and Brian Wilkerson. 1988. An Overview of Modular Smalltalk. In *Proceedings OOPSLA '88*. 123–134.

[19] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 325–334.