PTm: State-aware Transactional Live Programming

Pablo Tesone
Unité de Recherche Informatique et Automatique
IMT Lille Douai, Univ. Lille
Lille, France
Inria Lille-Nord Europe
Lille, France
pablo-adrian.tesone@imt-lille-douai.fr

Guillermo Polito
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189
CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille
Lille, France
guillermo.polito@inria.fr

Luc Fabresse
Unité de Recherche Informatique et Automatique
IMT Lille Douai, Univ. Lille
Lille, France
luc.fabresse@imt-lille-douai.fr

Noury Bouraqadi
Unité de Recherche Informatique et Automatique
IMT Lille Douai, Univ. Lille
Lille, France
noury.bouraqadi@imt-lille-douai.fr

Stéphane Ducasse
Inria Lille-Nord Europe
Lille, France
stephane.ducasse@inria.fr

Abstract

Live programming environments, such as Pharo, allow developers to modify the code and application state while the application is running. This allows a faster development cycle compared with the *edit-compile-debug* process.

Pharo implements simple yet powerful mechanisms to migrate the application state after each change. It allows developers to modify both methods and classes. However, modifying classes with existing instances could lead to an inconsistent application state, because *e.g.,* new instance variables are left uninitialized or with obsolete state. As these modifications are applied while the live application is running, a naïve development session may break the application. This requires special care from the developer (*i.e.,* staging, sequencing, and doing intermediate changes) to keep the coherence of application state.

We propose a novel tool (*PTm*) that allows the developer to scope her changes isolating them from the running application. For this, *PTm* creates an alternative environment with all the classes, methods and instances that are modified. The developer uses this environment to execute her code isolated from the running application, to validate it without affecting the running environment. Finally, the developer decides to safely discard her changes or to apply them atomically in the running application.

CCS Concepts • **Software and its engineering → General programming languages**;

Keywords live programming, memory transactions, state migration

ACM Reference Format:
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. 2018. PTm: State-aware Transactional Live Programming. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'18).* ACM, New York, NY, USA, **??** pages. https://doi.org/

IWST'18, September 10–14th, 2018, Cagliari, Italy
2018.

1 Introduction

Live programming environments [?], such as Pharo [?], allow developers to modify the running application while it is executing. These environments include not only the code of the application but all the live instances representing the application state. Live Programming environments allow us to modify both, the application code and its live instances allowing a faster feed-back cycle if we compare it with the *edit-compile-debug* process. This faster feed-back cycle improves the overall speed of the whole development cycle. Any modification performed, including modifications to application code or library code, should preserve a coherent application state allowing it to continue running without problems [?].

Pharo is a Smalltalk [?] implementation that supports the live programming experience. It implements a simple yet powerful mechanism to migrate the application state after each change. This mechanism allows developers to modify both methods and classes. Pharo allows the developers to freely modify application code, core libraries and the language itself. It allows us to modify all the elements present in the environment. However, this mechanism does not cover all the possible changes leaving uninitialized variables and inconsistent global state. As a result, some modifications affect the stability of the system. Complex modifications require special care from the developer to maintain the stability of the running application (*i.e.,* staging, sequencing, doing intermediate changes). As this special care is not always possible [?], this requirement limits the flexibility and power of live programming environments (Section **??**). The described problem occurs when modifying both application and system code. Moreover, even a modification in application code could arise a stability problem.

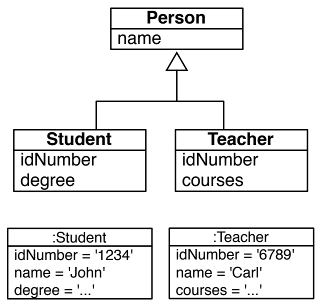
We propose to address this limitation scoping the modifications to a transactional environment. By doing so, the modifications to classes, methods and instances do not affect the running application. Scoping the changes is not enough, the developer should be able to perform new changes, test her changes and inspect and modify live instances. Also, she should be able to safely discard or apply these changes. As the application is running, the application of changes should be performed atomically. The changes to be applied are not only changes to methods or classes, also live instances should be migrated to maintain the application state coherence [?]. As not all the migrations could be calculated automatically [?], our solution provides a way of expressing the required migrations (Section ??).

We implemented our solution in Pharo. Our prototype tool (*PTm*) allows the developer to evaluate changes in a scoped environment. This environment includes the unmodified classes and instances from the application environment and the modified ones. This environment is created in an efficient way, only containing the referenced or modified classes and instances. This alternative environment is used to evaluate expressions, inspect and modify the instances and classes, allowing the developer to test her modifications. Once the modifications are ready, the developer applies or discard them. In case of needing migration strategies, the environment provides ways of detecting the need and provide ways of configuring the required migration strategies. Our solution handles the migration of live instances and globally accessible state (Section ??). We show that our solution handles the application of changes requiring migration of state and atomic changes (Section ??). The proposed solution enhance the ability to perform changes in the image side of the application, the changes requiring modifications in the VM or the runtime (*i.e.,* modifying the implementation of a VM primitive) are outside the scope of our solution. Also we discuss the design decisions, limitations and possible extensions to our implementation (Section ??).

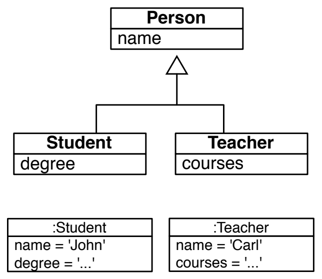
There are other solutions that provide a scoped transactional environment to isolate the changes [? ? ? ? ? ?]. However, these solutions do not take into account the migration of application state (Section ??). We finalize this paper presenting a conclusion of the impact of our work in Section ??.

2 Changes Corrupting Instances

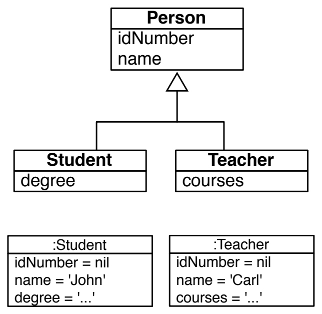
Needing Transactions. Pharo allows the user to modify the class structures and the methods of a running application. However, to maintain the coherence of the application state these changes have to be performed atomically or in a sequence of well-designed steps. Also, there are situations where such sequencing is not possible [?] requiring the atomic application of the changes.



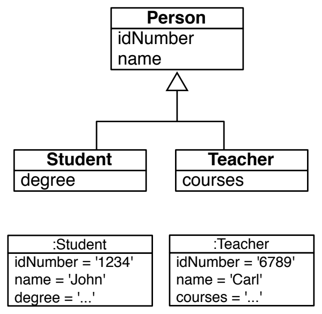
(a) Original State before the refactoring



(b) Step 1: Remove subclass instance variables



(c) Step 2: Add superclass instance variable



(d) Expected Result

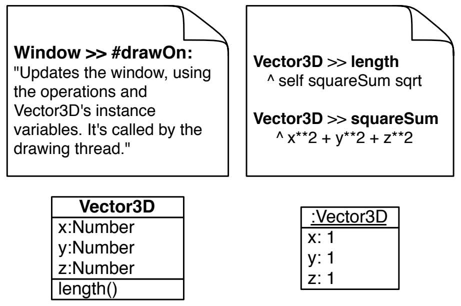
Figure 1. Step by Step of applying the Pull Up Instance Variable refactoring to the *idNumber* instance variable present in *Student* and *Teacher* classes.

An example of a situation requiring transactional changes is the *Pull up* refactoring (Figure ??). This refactoring performs the following steps:

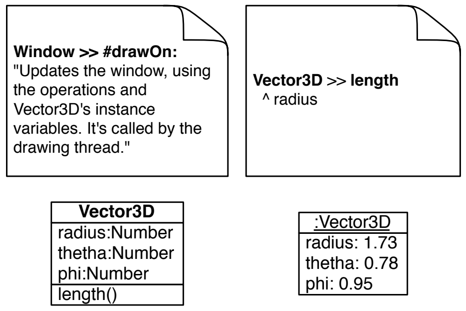
- Removes the instance variable idNumber from all the subclasses of Person.
- Adds the instance variable idNumber to the Person class.

This set of changes produced the expected result in the class structure. However, if these changes are performed naïvely the state of live instances is lost. In Pharo (and in many Smalltalks), when the first step is performed the instance variable is removed from all live instances of Student and Teacher. Once the instance variable is removed, their values are lost. When the instance variable is added back to the class Person, the instance variable values cannot be restored as they are already lost. This modification requires the atomic application of the changes and performing the instance migration after all the changes in the classes are performed.

Needing migrations. There are changes that require more than transactional changes, they require correct migration of instance state. If the modifications affect the structure of live instances or the instance variable value types, live instance should be migrated using custom logic [**?**].



(a) Original state



(b) After the desired change

Figure 2. Example of changes requiring migration of instances with custom logic.

In Figure **??**, we have a simple application that draws vectors in a window. The vectors are represented by instances of Vector3D. These instances have cartesian coordinates (x, y, z).

Figure **??** shows the initial implementation of the example and Figure **??** shows the desired changes to perform. The developer wants to change the representation of the vectors to use polar coordinates. To do so, the following changes are required:

- Update the methods Window » #drawOn: and Vector3D » #length.
- Remove the method Vector3D » #squareSum.
- Remove the instance variables x, y and z.
- Add the instance variables radius, thetha and phi.

After performing these changes, the instances of Vector3D are in an invalid state. When the old instance variables are removed, their values are discarded. Also, the new instance variables are initialized in nil. With this invalid state, the application using the instances crashes.

To avoid this, the instances of Vector3D should be regenerated (*e.g.,* from a persistent store), or they should be migrated. These set of changes requires a custom migration strategy [**?**]. Listing **??** shows a possible migration block. The block receives an old instance and a new instance.

```
[ :old :new |
   new radius: old length.
   new thetha: (old z / new radius) arcCos.
   new phi: (old y / old x) arcTan.
]
```

Listing 1. Migration Logic required for Vector3D

3 Transactional Changes

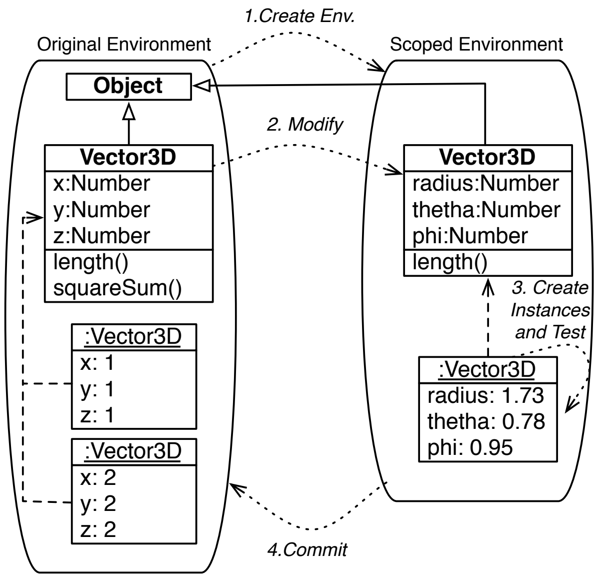


Figure 3. Overview of the Solution

We propose a technique to scope changes and instance state modifications in a transactional environment. To do so, we propose to use an alternative environment. This alternative environment is used to evaluate the code modifying the methods, classes and instances. In a nutshell, the developer is working in a copy of the environment and when the changes are complete and safe to apply, the old environment is replaced by the new objects. As all the instances and classes are living objects they are accessible through the environment.

```
"Step 1: Creation of the environment"
env := TMEnvironment new.

"Step 2: Applying the changes in the scoped environment"
env evaluate: [
   transaction createSubclassOf: #Object withNewName: #Vector3D
      slots: #(radius thetha phi)
      sharedVariables: ''
      package: 'Transactions-Tests'].

"Perform the changes in the methods"
env evaluate: [Vector3D removeSelector: #squareSum].
...
env evaluate: [Vector3D compile: 'phi: aVal. phi:=aVal'].

"Step 3: Create instances and test them"
aVector := env evaluate: [ Vector3DTest new ]
aVector radius: 1.73
...

"Step 4: Commit the transaction"
env evaluate: [transaction commit].
```

Listing 2. Using our solution

Figure **??** shows the overview of the solution and listing **??** presents an overview of how to do the different steps.

First, the scoped environment is created empty (Step 1). Next, the developer perform changes to the classes. When the classes are modified, copies of the classes are created in the scoped environment (Step 2). Only the required classes are created.

The developer is free to create instances and test her changes in the scoped environment (Step 3). The classes and instances in the original environment are not affected.

Finally, when the developer has tested her changes, she commits the operations in the original environment (Step 4).

4 Implementing PTm

Our solution presents a number of practical issues. We will address these issues in this section. First, we analyse how the new environment is created, which are the elements to include (*e.g.,* classes, objects and methods) in it and when the new environment is created and populated (Section **??**). Second, we provide a solution for state migration from and to the alternative environment (Section **??**). This migration has to handle the globally accessible state (Section **??**). Before applying the changes, it needs to detect and handle the conflicts in the state of the application (Section **??**). Finally, we provide the means to implement the application of changes (Section **??**).

4.1 Scoped Environment

An environment includes all the objects, classes and methods required to execute the application. In Pharo, the whole image is the environment of execution. All the modifications performed in our solution are performed in a copy of the environment. This copy is incrementally created. The classes are created in the new environment when they are referenced directly in the expressions evaluated in the environment or referenced in a method that is present in the environment. When a class is referenced in the new environment, the class is copied with the same definition that exists in the original environment. The superclasses of the copied classes are also copied. Also, the methods are compiled in the new environment and stored in the copied class.

To minimize the number of classes copied, we defined a set of classes that are only copied when they are modified (not when they are accessed). This set of classes includes core system Classes (*e.g.,* Object, Array, SmallInteger, Class, Metaclass). Not copying them on access improves the copy in most of the development scenarios, and allowing to copy them when modified allow us to handle transactional changes on them. This improvement is required to make the solution practical, without it the copy of the environment is not practical.

To transparently replace all the references to classes inside the alternative environment, the methods and expressions are compiled using the alternative environment to resolve the bindings. When a new binding is required, the environment creates a copy of it and creates the required class. This detection of the bindings and global variables is achieved by plugin on the compiler.

4.2 Global State

To be able to execute code in the scoped environment, our solution migrates the required globally accessible state of the application. In Smalltalk, the global state basically is of two types (1) global variables defined in the environment and (2) class-side variables. All the global state required by the scoped environment is copied in the new environment transparently.

The global variables are copied on access. Whenever an expression or method is compiled using a global variable this global variable is copied to the new scoped environment. To perform the copy during the compilation of methods the compiler has been extended. Extending the compiler allowed us to only copy the necessary global values. For example, literals, constants, and shared objects (*e.g.,* true, false, nil) are not copied.

The class-side variables are copied when the classes are included in the scoped environment.

In both cases, the value of the global state is copied. The instances pointed by the global state are copied in the scoped environment allowing us to be modified freely without affecting the running application.

4.3 State Conflicts Detection

After performing changes in the application code, the structure or use of the instance variables might be altered. If this is the case, these changes require migration of the instances from the old to the new version.

Our proposed solution first detects if the instance migration can be performed automatically. This is performed if one of the following conditions are met:

- There are no live instances (or subclasses instances) of the modified class in the old environment.
- The class structure is not changed in the number of slots or the names of slots.
- The types of the instances in the old environment and new environment are equivalent.
- The global state of the classes is the same (*i.e.,* all their class-side and shared variable values are the same).

If the migration cannot be performed automatically, the transactional environment cannot be committed in the original environment. Any attempt to commit it will raise an exception. To be able to apply the changes in the original environment, the developer should provide the missing migration strategies (Section **??**).

4.4 Applying Changes

Our solution applies the changes atomically on commit. To do so, we prepare all the elements that need to be replaced in the old environment.

These elements include:

- Classes and Metaclasses modified in the scoped environment.
- Existing instances in the old environment that requires migration.
- Global state requiring migration.

Once all the objects to replace are identified, the references to the old objects are modified to point to the new instances. This operation is performed using the Efficient Forwarders technique [**?**]. This novel implementation of *become* is implemented in the VM, also it assures the atomicity of the solution. We require this implementation to have a practical approach that does not require to traverse the whole memory to update the references.

To safely apply the changes, our prototype waits for a *Safe Update Point* [**?**]. We implemented a conservative strategy for detecting safe update points. We define a safe update point, as the moment that there is no thread with methods to be updated in its call stack. This conservative strategy might never find a safe update point, so the detection fails after a number of attempts. We prefer to find a safer update point than allowing to always commit.

The application of the changes is performed in a high priority thread. This thread is not interrupted by any other thread running in the image. It is interrupted by the events in the VM and the garbage collector, but these external interruptions do not affect the running update process.

4.5 State-Migration

The migration of state is performed in two stages by our solution. The first stage is the instance migration. In this stage live instances of the old environment are migrated to its new representation. This migration is performed following the guidelines in Tesone et al [**?**].

If the changes in the class structure do not add a new slot, the migration is performed automatically. Our solution is able to handle changes in the order of the slots, the position of the slot in the inheritance hierarchy (*i.e.,* moving a field to a super / sub class) or removing slots. In the automatic migration, the slots are copied from the old instances to the new instances by name

If the migration cannot be performed automatically, the developer should provide a migration strategy. A migration strategy encapsulates the logic to migrate from one version to the next one [**?**].

The second stage is the migration of global state. If the global state has been modified in the scoped environment, the developer should choose between three strategies: (1) using the state in the scoped environment (*using new global state*) or using the state in the original environment (*using old global state*). The selection of the strategy is performed in a class base. The developer selects for each of the classes with modified state which strategy to use.

4.6 Aborting the Transaction

As said before the changes are applied atomically by the become operation. This operation is atomic, it runs without interruption of the garbage collection process and the memory it requires is allocated before it is called (*e.g.,* the target objects are already allocated).

All the validations and possible errors occur before the execution of the become primitive. In case of an error during the impact of the changes the transaction is aborted. The abortion of the transaction is performed by discarding the new environment. As the original environment is not modified there is no impact of the aborted transaction.

5 Using PTm to safely apply changes

To validate our proposed solution, we implemented our tool in Pharo 7[1]. *PTm* provides an alternative environment to execute any modification in a scoped environment. The prototype is available as a Github project[2]. With this prototype, it is possible to perform the changes, execute code and tests, configure the migration strategies and commit or discard the changes.

We show in this section how to update the running application in the examples presented in Section **??**. As our solution is still a prototype, it is not integrated with the UI or the existing tools. However, it can be used from a workspace (GT

[1]https://pharo.org

[2]https://github.com/tesonep/transactions

Playground). So, in the examples, we will show the code to evaluate. The shown code only includes the required steps to implement the changes, although in the environment and in the development session many expressions and tests could be run.

```
"Creates a new environment"
env := TMEnvironment new.

"Redefines the Student class, removing idNumber"
env evaluate: [
  transaction createSubclassOf: #Person withNewName: #Student
    slots: #(degree)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Redefines the Teacher class, removing idNumber"
env evaluate: [
  transaction createSubclassOf: #Person withNewName: #Teacher
    slots: #(courses)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Redefines the Person class, adding idNumber"
env evaluate: [
  transaction createSubclassOf: #Object withNewName: #Person
    slots: #(name idNumber)
    sharedVariables: ''
    package: 'Transactions-Example'].

"Runs the tests to evaluate that the changes are ok"
result := env evaluate: [ StudentTest suite run ].
result defect isEmpty.
result := env evaluate: [ TeacherTest suite run ].
result defect isEmpty.

"Commits the transaction"
env evaluate: [ transaction commit ].
```

Listing 3. Atomic application of *Pull-up* refactoring using *PTm*

5.1 Transactional Changes

The example of the *Pull-up* refactoring requires transactional applications of the changes. Listing ?? shows the steps to perform this update using *PTm*. First, this example redefines both Teacher and Student classes removing the idNumber instance variable. The definition of the classes is performed through the transaction object. This object is available in the context of the new environment. This object is the main entry point of *PTm*.

Then, the Person class is redefined to have the new instance variable. Finally, the commit is performed. The changes are applied atomically. In this example, there is no need of custom instance migration, as the instance structure of Student or Teacher have not changed. The migration from the old to the new instances is performed automatically by name. This automatic migration is only possible as the changes are applied atomically.

5.2 Custom Migration

The second example in Section ??, the one migrating the structure of Vector3D class, requires custom migration of live instances. Listing ?? shows the step to safely perform this update using *PTm*.

```
env := TMEnvironment new.

"Redefines the class with the new structure"
env evaluate: [
    transaction createSubclassOf: #Object withNewName: #Vector3D
      slots: #(radius thetha phi)
      sharedVariables: ''
      package: 'Transactions-Tests'].

"Perform the changes in the methods"
env evaluate: [Vector3D removeSelector: #squareSum].
env evaluate: [Vector3D compile: 'length ^ radius'].

env evaluate: [Vector3D removeSelector: #x].
env evaluate: [Vector3D removeSelector: #y].
env evaluate: [Vector3D removeSelector: #z].
env evaluate: [Vector3D removeSelector: #x:].
env evaluate: [Vector3D removeSelector: #y:].
env evaluate: [Vector3D removeSelector: #z:].

env evaluate: [Vector3D compile: 'radius ^ radius'].
env evaluate: [Vector3D compile: 'thetha ^ thetha'].
env evaluate: [Vector3D compile: 'phi ^ phi'].

env evaluate: [Vector3D compile: 'radius: aVal. radius:=aVal'].
env evaluate: [Vector3D compile: 'thetha: aVal. thetha:=aVal'].
env evaluate: [Vector3D compile: 'phi: aVal. phi:=aVal'].

"Run the tests"
results := env evaluate: [ Vector3DTest suite run ]
"Check if the tests are ok"
results defects isEmpty.

"Try to commit the transaction.
It fails informing that a migration
is required for Vector3D"
env evaluate: [transaction commit].

"Provides a migration strategy for Vector3D"
env evaluate: [transaction migrate: Vector3D with: [:old :new |
    new radius: old length.
    new thetha: (old z / new radius) arcCos.
    new phi: (old y / old x) arcTan.
  ]].

"Commits the transaction"
env evaluate: [transaction commit].
```

Listing 4. Atomic update of Vector3D using *PTm*

If the developer tries to commit the transaction before setting a migration strategy and there are live instances of Vector3D, the operation will fail and it will produce an exception informing the situation to the developer. Then, the developer can decide to provide a migration strategy or discard the transaction without affecting the running application.

6 Discussion

One possible extension for our solution is the support for nesting environments. We have not implemented it in our current prototype, although one possible implementation may require to implement a polymorphic API between the image environment and the transactional environments.

Block Closures require particular considerations. In Pharo, a closure has a reference to its creating context to be able to access its state. This means that implementation-wise, a shallow copy of the closure will create a new closure sharing the creating context. Thus, any change in the context will affect both closures. In our implementation, the copy of the closures is shallow to minimize the impact in performance. Moreover this is a limited case because only applies to globally shared non-clean closures.

Our prototype does not implement any given support for concurrent transactions, the transactions are applied in the order they are committed. However, the detection of conflicts notifies the problems when trying to commit the second transaction. Having a proper locking of elements or versioning is a possible extension to this prototype.

We decided to use blocks to express the migration strategies. Using blocks allows the developer to easily provide a migration strategy. However, it lacks the ability to reuse this migration strategies. We consider to extend the support to using objects as migration strategies allowing a greater reuse. Regarding the migration strategies, we also consider the automatic generation of migration strategies using the information in the image environment and the alternative environment.

Other possible extension is the integration of our transactional environment with the existent tools in the Pharo Image providing a transparent experience to the user.

Finally, a possible extension is the use of Write Barriers and lazy binding of the associations to minimize the number of objects copied in the creation of the alternative environment.

7 Related Works

Mattis et al [?] implement transactional support for Squeak, they allow to modify the classes and methods scoping the changes to a set of threads. They also support to apply the changes back into the original environment. However, they do not propose any support to state migration or conflict detection.

Denker et al [?] propose a transactional solution *Changeboxes*. Changeboxes provide an object model representation of changes and scope the changes in a thread-local context. However, changeboxes are not intended to apply the changes back into the original environment, but to co-exist with the original environment. Also, they do not handle the migration of state or the detection of possible conflicts.

Lincke et al [?] propose a way of scoping the changes in a live programming environment. However, they do not allow to apply back the changes to the original environment and their solution does not arise all the problems of a class-based system, as it is based in a prototype system.

Wernli et al [?] propose a way of scoping changes inside contexts. A context scope the changes to the classes and methods, and the logic to migrate from one version to another and back. Their solution is designed to allow different versions of the same application to co-exists. They provide lazy migration from one version to the other. However, they do not implement detection of conflicts and requires more migration logic to interact with all the co-existing versions of the application. Our solution only keeps a single version of the running application, the alternative environments are only to edit and test code. There is no need to keep the different version running at the same time.

Casaccio et al [?] propose a technique for editing Pharo images without affecting the running environment. They propose to edit a client image from a server image. The modifications in the client image does not affect the server image. However, this solution requires to copy the whole image and does not include a way of applying back the changes in the running application.

8 Conclusion

In this paper, we quickly analyse the state migration requirements when applying changes to a live programming environment. These problems arise daily when modifying applications with live instances. Based on these problems, we propose a transactional modification solution that allows us to handle these daily problems.

We showed that our solution and prototype is able to handle these problems by applying the changes in a scoped environment and later applying them back to the original environment.

Our tool still requires additional work to be integrated in the existing Pharo tools. This is an important step to allow the developers to transparently benefit from our solution.

Acknowledgements

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.