Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Dynamic Software Update from Development to Production

Pablo Tesone[a,b] Guillermo Polito[c] Noury Bouraqadi[b]

Stéphane Ducasse[a] Luc Fabresse[b]

a. Inria Lille-Nord Europe, F- 59000 Lille, France

b. IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 59000 Lille, France

c. Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

Abstract

Dynamic Software Update (DSU) solutions update applications while they are executing. These solutions are typically used in production to minimize application downtime, or in integrated development environments to provide live programming support. Each of these scenarios presents different challenges, forcing existing solutions to be designed with only one of these use cases in mind. For example, DSUs for live programming typically do not implement safe point detection or instance migration, while production DSUs require manual generation of patches and lack IDE integration. Also, these solutions have limited ability to update themselves or the language core libraries, and some of them present execution penalties outside the update window. We propose a DSU (*gDSU*) that works for both live programming and production environments. Our solution implements safe update point detection using call stack manipulation and a reusable instance migration mechanism to minimize manual intervention in patch generation. Moreover, it also offers updates of core language libraries and the update mechanism itself. This is achieved by the incremental copy of the modified objects and an atomic commit operation. We show that our solution does not affect the global performance of the application and it presents only a run-time penalty during the update window. Our solution is able to apply an update impacting 100,000 instances in 1 second. In this 1 second, only during 250 milliseconds the application is not responsive. The rest of the time the application runs normally while *gDSU* is looking for the safe update point. The update only requires to copy the elements that are modified.

Pablo Tesone, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse. *Dynamic Software Update from Development to Production*. Licensed under Attribution 4.0 International (CC BY 4.0).
In Journal of Object Technology, vol. 17, no. 1, 2018, pages 2:1–36. doi:10.5381/jot.2018.17.1.a2

Keywords Atomic, dynamic software update, object-oriented, live environments.

1 Introduction

A Dynamic Software Update (DSU) [HN05, PH13] engine is a tool that manages the migration of a piece of software from version 1 to version 2 while it is running. Its basic idea is to turn the *stop, install, restart* cycle into a simple *update* action [PDF+15]. DSU engines perform such migrations minimizing downtime and guaranteeing that the software will continue working as expected. They are important because software must constantly evolve, otherwise it becomes obsolete [LB85, DDN02]. Examples of such evolution are adding new features, improving performance or fixing bugs and security failures.

DSU solutions (from now on DSUs) are typically used in two scenarios: a production DSU is designed to update long running applications *e.g.,* Web application servers; a development DSU is integrated within a development environment to provide live programming support. Each of these scenarios presents different requirements, making existing solutions to be specialized for only one of them. For example, a production DSU requires safer guarantees while a development DSU requires incremental updates and IDE integration.

Both kinds of DSUs share the challenges listed below and revisited in Section 2. Table 1 summarises how the different kinds of DSUs choose to balance their features to cope with these challenges:

- **State Migration.** Migrating the state of an application between versions is not a trivial activity. On the one hand, it requires a technique to replace old values by new values (*e.g.,* pointer swapping, lazy proxies). On the other hand, it requires a way to express value transformations which are usually application dependent and cannot be produced automatically.

- **Change Identification.** Determining the set of changes to apply (*e.g.,* classes to create, methods to modify, instances to migrate) is error prone if done manually. Moreover, doing it automatically lacks precision: the process may miss business dependent value transformations required for state migration.

- **Core Libraries and Self Update.** Updating the core parts of the run-time environment (*e.g.,* core language libraries) and the DSU itself is difficult [PDF+15]. Such updates introduce circular dependencies that may break the update and require special mechanisms to ensure atomicity.

- **Safe Point Detection.** Detecting and deciding the best moment to execute an update (*Quiescence Point*, *Safe Update Point* or *Alterability Point*) [NH09] presents a challenge for those applications that were not designed for it. Looking for a safe update point should be fast enough to minimize the suspension time and smart enough to detect as soon as possible when such update point will never happen.

- **Execution Penalty.** Implementing all the above requires smart strategies to avoid performance penalties outside the update itself. For example, the usage of lazy proxies for state migration introduces an additional level of indirection affecting the overall application performance.

Challenge	Development DSUs	Production DSUs
State Migration	Limited	Yes
Change Identification	Automatic	Manual
Core lib & Self Update	Limited	No
Safe Point Detection	No	Manual
Execution Penalties	Most	Most
Examples	Jrebel, Javeleon, Smalltalk	Rubah, DUSC, Pymoult

Table 1 – How different DSU solutions face the challenges.

Table 1 presents a summary of how the existing engines balance these requirements. As shown, the different types of DSU engines are designed to maximize some of the challenges (See Related Work in Section 7).

On the one hand, development DSUs such as Jrebel [Zer12], Javeleon [GJK+12], and the one found in Smalltalk [GR83] work as follows: they first identify the changes from the IDE usage, then they apply the changes immediately without performing a safe point detection. These solutions provide limited state migration because they do not support value transformations. On the other hand, production DSUs such as Rubah [PH13], DUSC [ORH02] and Pymoult [MDB15] receive a manually generated patch with all the changes to apply including the value tranformations, they detect and wait for a safe update point, they pause the application threads and apply the changes. Both kinds of DSUs have limitations: they only have limited or no support for core libraries or self update. In addition they often present performance penalties outside of the update window.

Contribution. We propose (*gDSU*), a DSU that is practical for both live programming and production environments (Sections 3 and 4). The entry point of *gDSU* is a patch containing the changes and migration rules to apply. *gDSU* uses a conservative safe update point detection strategy to decide when it is best to apply such update, based on call stack manipulation. To allow atomic core library and self updates and avoid meta-circularity problems, *gDSU* performs a copy of all objects affected by the update. It then leverages *forwarders*, a modern memory management technique existing in the Pharo VM, to apply the commit operation with minimal performance overhead [MB15].

To make *gDSU* practical for day-to-day usage, we implemented a state migration mechanism via semi-automatically generated patches. We identify the changes that should be included into a patch by inspecting the source code repository, or by listening to IDE events. Such integration with existing tools minimises manual intervention needed when identifying changes.

Even though our proposed solution is not the first one that is usable in both scenarios, it is the first one to address in a practical way the problems of both update scenarios. The other existing solutions are intended to be applied in one of the scenarios, this design decision minimizes the applicability in the other scenario. We demonstrate that our solution is applicable in both scenarios (Sections 5 and 7).

We implemented a working prototype of *gDSU* in Pharo [BDN+09]. We validated

our solution by updating applications both in development and production environments. We also show how our solution is able to update Pharo core libraries and the update engine (*gDSU*) itself. Moreover, we present a set of benchmarks showing that our solution is practical in both execution time and memory (Section 5). Our solution is able to apply an update impacting 100,000 instances in 1 second. In this 1 second, only during 250 milliseconds the application is not responsive. The rest of the time the application runs normally while *gDSU* is looking for the safe update point. The update only requires to copy the elements that are modified.

2 Challenges of Dynamic Software Update

In this section we present the challenges of applying a code change on a running application through an example. We then stress the issues that a real DSU engine should overcome to become practical in usage, such as performance impact and ease of usage. Finally, we draw a requirement list for a complete DSU engine.

2.1 Change Challenges Illustrated

Let us consider an application running a 3D visualization on a continuous stream of data. This application has a Window class that implements a drawOn: method responsible for rendering a single frame of our visualization in a 3D canvas. This application has a render thread with a loop invoking the drawOn: method. So, Window continuously draws in its canvas instances of Vector3D class. Figure 1 shows a screenshot of such application and Figure 2 illustrates its code.

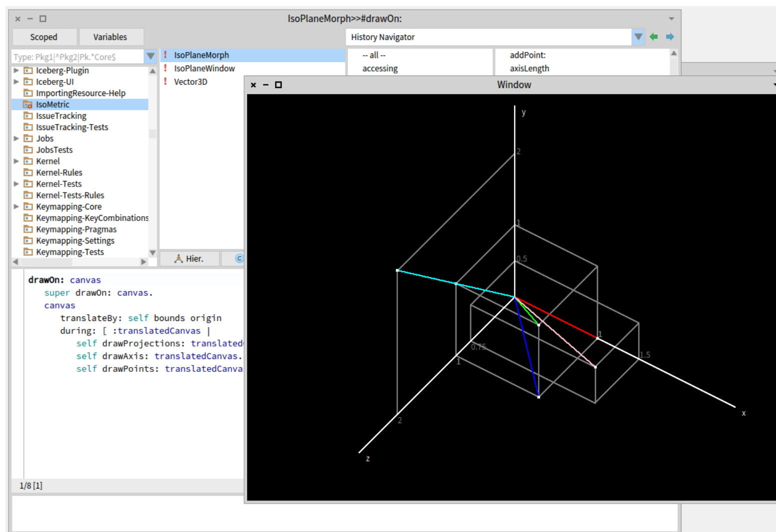


Figure 1 – Window of the rendering application in action.

Let us consider now that a developer wants to change the coordinate system from cartesian coordinates to spherical coordinates for performance reasons. Figure 3 presents the desired modification. This modification requires a number of different changes:

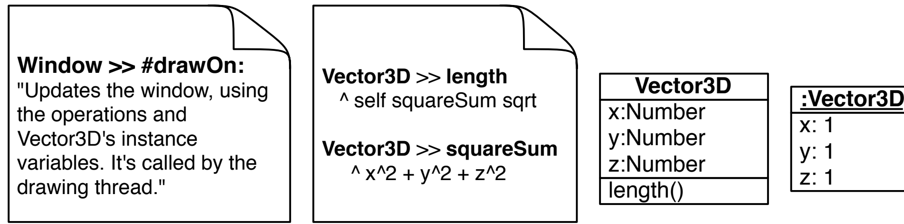


Figure 2 – Original version using cartesian coordinates

- The Window»drawOn: method will be updated to use the new coordinates of Vector3D.
- The Vector3D»length method will be replaced to use the *radius* instance variable.
- The Vector3D»squareSum method will be removed, as it is not used anymore.
- The structure of the class Vector3D will be changed replacing the existing instance variables with the new ones.

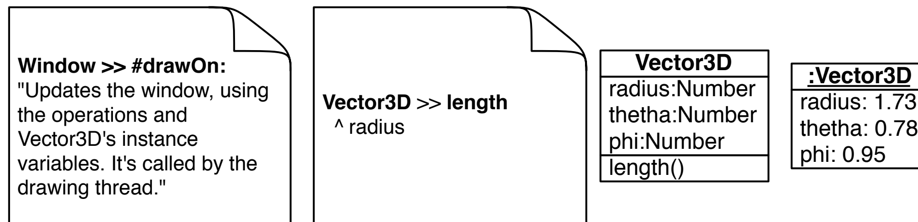


Figure 3 – New version using spherical coordinates

Applying these changes while the application is running is not a trivial task. This example clearly shows the three problems that occur when updating a running application:

State Inconsistency. Updating an application involves changing its internal state representation. However, naively initializing such state may produce information loss or even failures in the application. Most of the times, a migration is business dependent. In our example, the Vector3D instances change: initializing the new instance variables to null produces a null pointer exception, while initializing them to 0 will produce a loss of previous application state. Instances should be migrated from their cartesian coordinates to their corresponding spherical coordinates. The logic of this migration is clearly business dependent and cannot be generally inferred just from the changes in the classes and methods.

Change Interdependency. When updating an application, the modifications are usually interdependent because they relate to the same entities (*i.e.,* methods, instance variables, classes). In the example, the new Vector3D»length implementation requires the previous introduction of the new radius instance variable. Modifying this method before adding the instance variable is wrong, and it causes application failures. These interdependencies appear also between methods. For example, removing Vector3D»squareSum before updating Vector3D»length produces a missing method error during the execution of Vector3D»length.

Concurrency and Execution Inconsistency. While the application is running, some of the methods to be updated are present in the execution stacks of running threads. The update process should not lose or corrupt the application execution. Such a corruption occurs when the update alters local variables or control-flow of a method in any execution stack [HN12]. In our example, if the render thread enters the Vector3D»length method and the update is applied before the Vector3D»squareSum method is called, continuing the execution of the length method will fail as the Vector3D » squareSum method does not exist anymore.

These issues make applying code changes a challenging and interesting task, to which existing DSUs propose several techniques to solve. However, implementing a solution in a way that it is applicable in a real scenario imposes new challenges. The following section explores those challenges concerned with these more practical issues.

2.2 DSU Practical Concerns

DSUs usage presents a set of concerns that should be addressed to have a practical solution.

Performance. Making a program updateable should impact its performance as little as possible [HN05]. A DSU performance impact is divided in two stages: (1) during normal execution (outside update-window) and (2) during an update (inside update-window). A DSU should minimize the impact in both stages. Examples of impact are memory consumption, execution overhead and downtime during the update.

Ease of Use. The DSU engine should be easy to use by regular application developers. The less complicated the updating process is, the less error-prone it will tend to be [HN05]. A DSU solution should be integrated with the development tools used in the language. Also it should minimize manual interactions and simplify them when they are unavoidable. Finally, it should be present in the whole life-cycle of the application providing solutions during the development as well as during the evolution of systems in production.

Versatility. A DSU should be able to update any part of the running application [HN05]. The running application is not the only part that may require modifications. Core language libraries including the DSU engine itself also require updates (*e.g.,* adding new features, improving performance or fixing bugs and security failures).

2.3 Requirements for a General DSU

We consider a general DSU as a tool capable of updating applications in development and production scenarios. Taking in consideration the problems and concerns a practical general DSU should overcome, we have enumerated a set of requirements. Table 2 presents how the requirements cover the problems and concerns.

Atomicity. A DSU solution should perform atomically all changes in a single update. All the changes should be applied at once, or at least, the execution and state of the old and new versions should not be mixed.

Problem	Concerns
State Inconsistency	State Migration.
Change Interdependency	Atomicity.
Concurrence and Execution Inconsistency	Automatic Safe Point Detection.
Performance	Small Run-time Penalty. Minimal Application Downtime.
Versatility	Self and Core Lib Update.
Ease of Use	Patch Generation. Patch Reuse. Broad Applicability.

Table 2 – Requirement mapping to the problems and concerns of a DSU.

State Migration. A DSU solution should provide the means to migrate the state of the application from one version to the other. The needed migration logic might be produced automatically or provided by the developer. The migration logic for value transformations that depends on business logic cannot be generated automatically thus, a DSU solution should minimize required manual interventions.

Automatic Safe Point Detection. A DSU solution should detect the safe points to perform an update. As the application is running the program under update, the update should be performed while it does not have any impact on the running application or the solution should handle the impact. The detection should be done minimizing developer intervention.

Patch Generation. A DSU should automatically calculate the set of changes needed to pass from one version to the next one. This set of changes is a *patch* [SAM13]. A DSU should provide a clean integration with the development process providing *programmer transparency* [MME12]. This integration minimizes the need of manual intervention in patch creation.

Patch Reuse. If the DSU solution requires the participation of a developer, it should allow the developer to reuse and compose these elements. The reusing and composing of migration logic and patches ease the developers' manual work.

Self Update and Core Lib Update. As the bugs and improvements not only occurs in the application under modification, it is required that the DSU solution allows the developer to modify the core libraries of the language and the DSU itself.

Small Run-time Penalty. A DSU solution should minimize the performance impact it has on the application during its normal execution *i.e.,* outside the update window. Ideally, the application should run as if there is no present DSU solution. As a counter-example, techniques such as bytecode manipulation or lazy proxies introduce an additional level of indirection affecting the normal performance of the application.

Minimal Application Downtime. A DSU solution should minimize the downtime of the application during the update window.

Broad Applicability. A DSU should be applicable during the whole life-cycle of the application. It should be applicable in development and updating an application in production.

3 Our Solution in a Nutshell

We propose a new DSU solution called General Dynamic Software Update (*gDSU*). We call *gDSU* general because it is applicable to both productive and development scenarios fulfilling all their requirements in a practical way. *gDSU* allows developers to apply changes to an application, core language libraries and even on the DSU engine itself.

3.1 Update Life-cycle in *gDSU*

The entry point of the *gDSU* engine is a patch. A patch is a collection of changes that describe the update to perform. It includes the changes to apply in the code, such as method and class modifications, and the corresponding state migration logic. *gDSU* generates this patch semi-automatically by applying a VCS version diff or/and by recording code changes made in the IDE during a development session. In both approaches, business-related state migration policies should be provided by the developer (Sections 4.1 and 4.2).

If *gDSU* is used in production, the developer generates a patch in his development environment. Afterwards, she applies the patch using the *gDSU* engine deployed in the production environment. In the case of a development scenario, the patch is generated and applied at the same moment in the development environment.

gDSU takes the patch as input and performs the following steps to safely apply the update. These steps are the same regardless the update scenario. Figure 4 illustrates such steps.

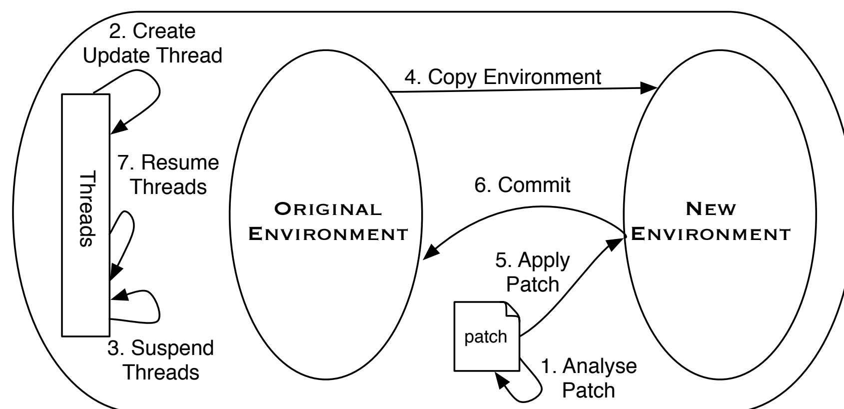


Figure 4 – Steps to apply an atomic update

1. Analyse Patch. *gDSU* analyses the patch and calculates the changes to perform

in the environment. This analysis is used to detect what live instances it should migrate and the conditions to reach a safe update point (Sections 4.3 and 4.4).

2. **Create Update Thread.** *gDSU* spawns the update thread. This thread is responsible of monitoring the other running threads looking for a safe update point and performing the update (Section 4.4).

3. **Suspend Threads.** When the safe point is reached, the update thread suspends all the other threads and the update process can begin (Section 4.4).

4. **Copy Environment.** *gDSU* copies the instances and classes that are impacted by the update inside an isolated environment (Section 4.3).

5. **Apply Patch.** *gDSU* performs all the changes in correct order on the new environment. It also migrates the state of all affected live instances (Section 4.5).

6. **Commit.** *gDSU* replaces all the instances in the original environment affected by the update with their corresponding instances in the new environment (Section 4.7). This step is only performed if the resulting environment is valid. If the validations are not correct, the new environment is just discarded and the update is not applied (Section 4.6).

7. **Resume Threads.** The application threads are finally resumed (Section 4.4).

3.2 Requirement Assessment

This section analyses how *gDSU* satisfies each of the stated requirements:

Atomicity. We satisfy the atomicity requirement by applying the changes in an isolated environment, and committing all the changes in a bulk replacement operation (Section 4.7).

State Migration. We satisfy state migration requirement through the use of migration policies (Section 4.5). The developer provides the required migration logic implementing one or more migration policies. The migration policies are reused in different updates and also the implementation provides generic migration policies (*e.g.,* the migration logic required by an automatic refactoring is already provided).

Automatic Safe Point Detection. We satisfy safe point detection requirement by the implementation of an automatic detection algorithm based on stack manipulation. The developer does not need to provide extra information to detect safe points (Section 4.4).

Patch Generation. We satisfy the patch generation requirement with a semi automatic patch generation (Section 4.2). Our proposed patch generation uses two strategies: (1) getting the information from the version control system and (2) storing the information while the developer changes the application. The first strategy is used when the patch is generated to migrate one version of the application to another. The second strategy is used when the application is modified using live programming. In both strategies, the patch is not totally generated. The developer should provide the business related migrations that are impossible to calculate.

Patch Reuse. Our proposed solution requires the implementation of custom migration and validation logic. However, the solution provides different mechanisms to reuse and combine this custom logic (Sections 4.5 and 4.6).

Self Update and Core Lib Update. Our solution satisfies the self update and core libraries update requirement using an atomic commit operation. Our solution performs all the changes in a copied environment. This level of isolation allows us to modify elements that are currently used by the update process. The real elements are replaced in a single operation using a bulk instance replacement mechanism (Section 4.7). This mechanism is implemented in the virtual machine and it is part of the garbage collection operation.

Small Run-time Penalty. We minimize the run-time penalty through the usage of eager migration of instances. Using eager migration does not require the use of lazy proxies or bytecode instrumentation. Our solution does not add any impact in the execution outside the update window. Moreover, no part of the *gDSU* engine runs outside the update process. We validate this in Section 5.6.

Minimal Application Downtime. Our solution minimizes the downtime of the application by minimizing the copy of objects. Only the modified parts of the application are copied and replaced in an update. The efficient copy of updated elements apply to live instances, classes and methods (Section 4.3). We validate this in Section 5.6.

Broad Applicability. Our proposed solution is compatible with both development and production update scenarios. The update life-cycle is exactly the same regardless the environment of application. The differences that may arise in these scenarios are attenuated by the *gDSU* support for automatic patch generation and safe point detection stated above. We validate this in Section 5.3.

4 Practical General Dynamic Software Update

The *gDSU* architecture presented in the previous section provides the means to perform safe atomic updates. However, a number of design issues have to be addressed to implement it efficiently. In this section, we analyse these issues and propose a solution to them: how is a patch composed (Section 4.1), how this patch is generated (Section 4.2), how to efficiently copy the environment during the update (Section 4.3), how to detect the safe update point (Section 4.4), how to make an efficient bulk replacement of instances during the commit (Section 4.7), and how to create reusable migration policies (Section 4.5) and validations (Section 4.6).

4.1 Design of Patch Content

gDSU requires a patch to contain all the information necessary to perform the update in a single operation. The patch's content consists of:

Structural Changes. A set of all changes to methods and class structures corresponding to the new version. This includes new classes and methods, their modifications and removals.

Instance Migration Policies. A set of migration policies. A migration policy describes how to migrate live instances of one type from the current version to the new version. Migration policies are further explained in Section 4.5.

Update Validations. A set of validations. These validations are meant to guarantee that the application state and behaviour are consistent after the update is applied but before the commit.

A patch describes the update in a declarative way, containing details of what to do. The *gDSU* engine is responsible of determining when and how to perform the update.

4.2 Patch Generation

To help users using *gDSU*, *gDSU* provides several tools to help generate patches semi-automatically:

VCS version Difference. *gDSU* calculates a patch from two versions of the application in the version control system (VCS). This approach is useful when the update target system is an application in production.

IDE Change Sets. *gDSU* records IDE events and stores code changes. This approach is most useful in a development environment, for example when using automatic refactorings.

These tools automatically calculate the structural changes, system level validations and basic migration policies. As an example of such a basic policy, if a class' structure changes the order of its instance variables, the *gDSU* engine proposes a migration policy copying the instance variable values by name, as Figure 5 shows.

```
ByInstVarNameMigration >>
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
    new class instanceVariables do: [ :newIV |
      old class instanceVariableNamed: newIV name
        ifFound: [ :oldIV | newIV write: (oldIV read: old) to: new ] ].
```

Figure 5 – Reusable Migration Policy: it migrates all the instance variables' values by name. It is used when the instance variable order changes.

The developer can then extend the patch: modify the structural changes to apply, add business related migration policies (Section 4.5) and validations (Section 4.6). This patch can moreover be reused in posterior updates.

4.3 Efficient Partial Copy of the Original Environment

gDSU makes a partial copy of the original environment into a new environment to guarantee that it can safely perform self-updates, core library updates and cancel failing updates. First, making a copy transforms the meta-circular update into a normal update because *gDSU* is not modifying itself but a copy of itself. Second, scoping the changes into a copy allows one to avoid affecting the updated application when problems appear during the update. Such problems can raise because of several causes *e.g.,* errors or bugs in the *gDSU* engine, the migration policies, or unsatisfied

validations. While handy, the copy of the original environment is a time consuming operation, so limiting the number of copied objects reduces the overall execution time of the update.

gDSU implements a partial copy of the environment that only includes objects (classes and instances) affected by the update. *gDSU* calculates which objects are affected using the structural changes and the migration policies in the patch. A class is considered affected if there is a structural change on it or on one of its superclasses. An instance is considered affected if its class is affected or if there exists a registered migration policy for its class or any of its class' superclasses. Informally speaking, a structural change in a class will affect all the instances of that class and it will recursively affect its subclasses. As methods are not down-copied in subclasses, a change in a method only affects the class containing the method. The subclasses use the updated method through the usual method lookup.

This copy process leaves the original objects intact and thus requires to replace afterwards all references to the old objects with references to the new objects. *gDSU* does such replacement during the bulk update in the commit operation, further explained in Section 4.7.

In the example presented in Section 2, the only objects to copy are the Window class (because its drawOn: method was modified), the Vector3D class (because its structure was modified) and all instances of Vector3D (because its class structure has changed).

4.4 Automatic Safe Update Point Detection

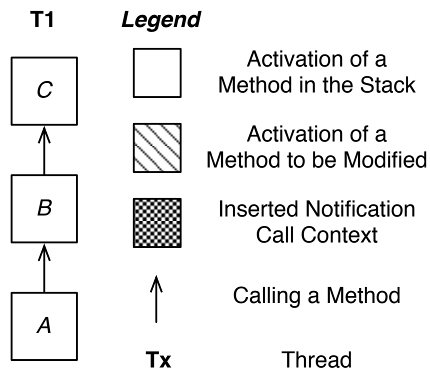
gDSU guarantees that the updated application is not running during the update by suspending all its related threads. Process suspension contributes to the atomicity of the update. To provide better guarantees and avoid creating execution inconsistencies, the application should not be suspended at any moment but at a point considered safe. During the update process the only running thread is the update engine thread.

An application is at a safe point if all its threads are at a safe point. A thread is at a safe point if its call stack does not contain methods affected by the update *i.e.,* the corresponding patch contains changes for that method.

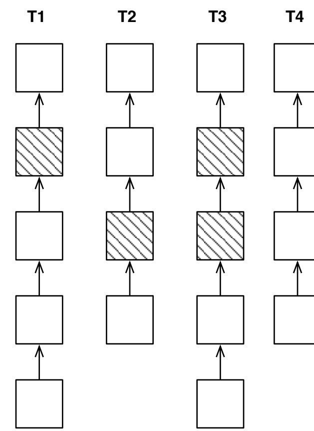
When an update is required, the *gDSU* engine spawns a new thread. This thread is responsible of performing the update. Then the *gDSU* engine waits until all other threads are at a safe point before performing the update. The update thread monitors all other threads using events rather than busy-waiting. When a thread returns from the execution of an affected method, an event is produced. Upon an event, the update thread checks if a safe point is reached, in which case starts applying the update. If the application is not at a safe point, the update thread yields the processor and waits for the next event to recheck.

gDSU implements such detection strategy with call-stack manipulation, as illustrated in Figure 6. It inserts in each thread call-stacks one notification call context just before the context of oldest affected method in the stack. When the notification call context is executed because the thread returns from the affected method, it suspends all the running threads and re-checks for the occurrence of a safe update point.

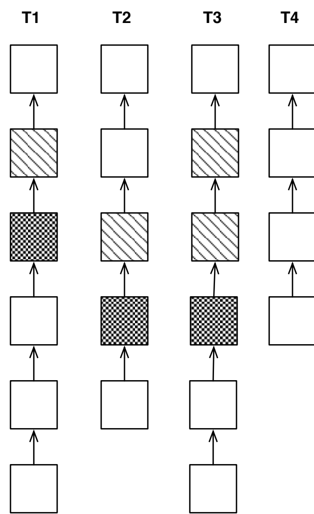
While checking for safe update point condition *gDSU* will install new notification call contexts in all threads that do not have one. This may happen because existing threads may have returned from the notification call stacks or new threads include affected methods in their call-stack. This conservative strategy may not converge if the application never reaches a quiescence point [NH09]. *gDSU* will timeout an update



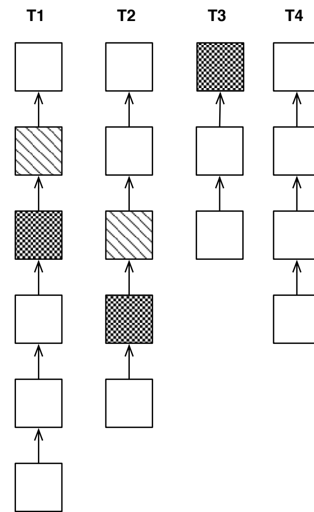
(a) Method A called B, and B called C.



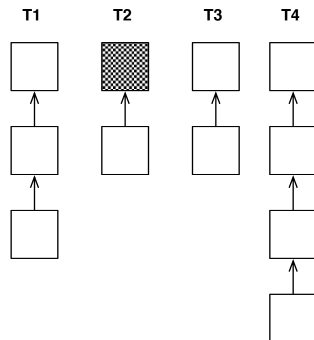
(b) Threads using methods to be modified, in this state the update cannot be applied.



(c) Context inserted in the call stacks just before the methods that should be updated.



(d) The *gDSU* checks, but the system is not at a Safe Update point because T1 and T2 are still executing methods to be modified.



(e) The *gDSU* checks, the system is at a Safe Update Point. The update is performed.

Figure 6 – Modification of call stack for the detection of Safe Update Points.

after several of retries, aborting the update. The reasoning of this choice is discussed in Section 6.

In the example presented in Section 2, the problematic thread is the drawing thread which is calling the drawOn: method. If during the safe point detection the drawing thread is executing the drawOn: method, the *gDSU* engine adds a notification call context just before this method. When the drawOn: method ends, the thread executes the notification call context and notifies the *gDSU* engine that is possible to suspend all threads and execute the update.

4.5 Reusable Instance State Migrations

To preserve the application state after an update, *gDSU* migrates the state from the old version to the new version with *Migration Policies*. A *migration policy* is a meta-object in charge of the state conversion. Figure 7 shows the meta-object protocol of such a migration policy.

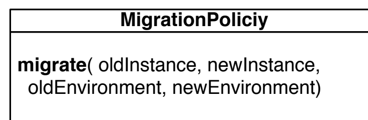


Figure 7 – Migration Policy interface

gDSU provides generic migration policies for common cases such as *e.g.,* refactorings. These generic migration policies are reused between different updates and even between different applications. Figure 8 shows a pull-up instance variable refactoring and the corresponding migration policy that is applicable every time the same refactoring is applied.

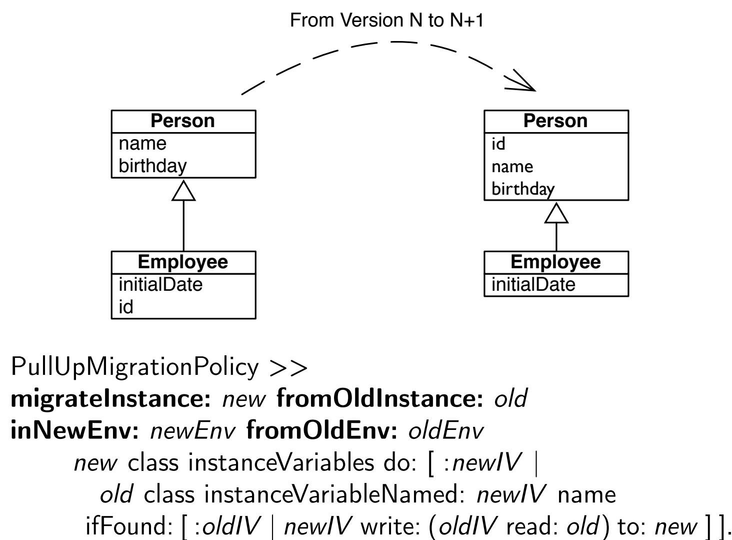
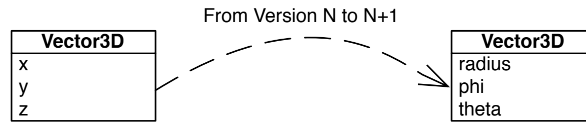


Figure 8 – Migrating instance variables per name: an example of application independent change

Moreover, the developer may extend this protocol to define business dependent migrations. For example, Figure 9 illustrates a policy to migrate cartesian to spherical coordinates.




```
CartesianToSphericalMigrationPolicy >>
migrateInstance: new fromOldInstance: old
inNewEnv: newEnv fromOldEnv: oldEnv
    new radius: old length.
    new thetha: (old z / new radius) arcCos.
    new phi: (old y / old x) arcTan.
```

Figure 9 – Migrating Vector3D: an example of application dependent change.

4.6 Reusable Validations

To guarantee that an application state and behaviour are consistent after an update, *gDSU* performs several *Validations* before committing the update. A *Validation* is a predicate function that validates the copied environment. If all validations are successful *gDSU* proceeds with the commit, otherwise the update is discarded. Although the validations are not needed by the update engine, their presence improves the stability of the updates avoiding invalid updates. We identify three different categories of validations that are easily reused in different updates.

System Level Validation. It checks the consistency of the running platform. They are independent of the update and the application, and they are executed in all the updates for a given platform. For example, one validation checks if the application meta-objects (classes and methods) and system structures have been correctly migrated.

Application Validation. It checks application invariants that should be consistent during all the life-time of the application. They are applied in all the updates of this application and are useful to guarantee business rules before committing an update. It is the responsibility of the developers to produce them. For example, an application validation may check that all Employee instances have a name.

Single Update Validation. It checks a condition that should hold when the update is applied. For example, if the structure of a core object is changed, it is useful to check if the state migration was correct for every object. Once this update is committed, this validation is not useful anymore and can be discarded. In the running example presented in Section 2, the developer can include a validation that asserts the correctness of all the points to prematurely detect and avoid problems in the drawing thread. Figure 10 shows an example of such a validation.

4.7 Bulk Instance Replacement

gDSU implements the commit operation as a bulk swap of object references. In other words, during the commit operation *gDSU* replaces all references to old affected objects by references to their corresponding copies. Bulk replacement is done atomically, making the update a true atomic process. The bulk replacement operation is

```
CartesianToSphericalValidation >>
validateFrom: oldEnvironment to: newEnvironment
  ^ (newEnvironment allInstancesPairFor: #Vector3D)
    allSatisfy:[:aPair | | old new |
      old := aPair first.
      new := aPair second.
      (old length = new length) & (new phi = (old y / old x) arcTan).
    ].
```

Figure 10 – This validation is used to guarantee the correct migration of the Vector3D from one coordinate system to the other.

crucial in the implementation of most DSUs. They are used in related works such as Rubah [PH13] and are present in most Smalltalk implementations. However, a naive implementation would not be satisfying from a performance point of view as it would require to full scan the memory to perform pointer replacement [MB15].

Bulk replacement is implemented in our prototype as the primitive Virtual Machine operation become:. become: takes two equal-sized arrays as arguments and performs a pointer-swap between each object in the first array and the object that occupies the same position in the second array. That is, a pointer swap between a and b makes all objects in memory referencing a change and point to b and vice-versa. For the atomic commit, *gDSU* uses a variant of become: called becomeForward:. becomeForward:, also called one-way-become, only performs the pointer replacement from the first to the second set of references. To efficiently implement become we leverage a novel memory management technique called forwarders that is available in the Pharo Virtual Machine. Forwarders allow one to perform lazy pointer swapping with the use of a partial read-barrier thus avoiding the need of a memory full scan [MB15].

5 Validation

To validate our model we implemented *gDSU* in Pharo [BDN+09]. Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback[1]. We selected Pharo because (1) it provides powerful meta-level operations, (2) most of its runtime is implemented in itself, (3) it represents basic concepts of the language and the environment as first-class objects. These characteristics allow the implementation of *gDSU* as a library. Moreover, Pharo covers all the platform requirements to implement our solution (Section 5.1).

This prototype is available in a Git repository[2]. It is loadable in the latest version (Pharo 6) of the platform and it is intended to be included in future Pharo versions.

gDSU is applicable not only for updating a running application, but also in daily development. We validated our solution in the following three scenarios. Each scenario has been validated as a long running application and using interactive live programming:

- Update of Application Code with live instances (Section 5.3).

[1] http://pharo.org/
[2] https://github.com/tesonep/pharo-AtomicClassInstaller

- Update of the DSU engine itself (Section 5.4).
- Update of the core libraries of the language (Section 5.5).

We also validated that our solution is practical using a set of benchmarks. These benchmarks show that the solution is viable in terms of execution time and used memory (Section 5.6). Before presenting the validation, we give an analysis of the requirements needed by our solution to be implemented in other platforms.

5.1 *gDSU* Platform Requirements

Implementing *gDSU* as a library requires that the underlying platform provides a number of requisites:

Class manipulation. *gDSU* needs to be able to query, add and remove classes; and change the superclass. It also needs to be able to add, remove and change instance variables. Finally it needs to be able to add, remove and modify methods.

Instance manipulation. *gDSU* needs to be able to create new instances from a given class, read and write all the instance variables in a given instance. And also, it needs the ability to list all the instances of a given class.

Thread manipulation. *gDSU* needs the ability to inspect the call stack of threads, stop and resume them. It needs to be able to modify the call stack inserting new method activations.

Environment manipulation. *gDSU* needs the ability to read and modify the elements in the global environment. It needs to make a copy of the environment and replace the environment with this copy.

Bulk object swap. *gDSU* needs the ability to perform a bulk replacement of objects as described in the Section 4.7.

IDE Integration. To be able to minimize the manual building of patches, *gDSU* needs the ability to integrate with the language IDE. *gDSU* needs a way of getting the changes performed by the user and the details of the program modified.

5.2 Validation Methodology

The following sections present three different scenarios we used to validate and benchmark our solution. These scenarios include the modification and refactoring of a stateful chat application. The modifications are performed in the application, the DSU and in the core system libraries. For each of these scenarios we applied the following methodology for the development and production environment:

Development Environment. We set-up a development environment with our application code. We run a simulation of requests to generate application objects. This simulation produces around 30 000 live instances. After the simulation, we have an environment with live instances that is useful to perform live programming. This environment replicates a common development environment where the developer has not only the code but also a set of data to try her changes.

Then, we perform modifications to the application. These modifications are performed programmatically, but the result is the same if they are produced using the existing IDE.

Production Environment. Using the same application a HTTP server is launched. This HTTP server replicates a production server. This is designed to be deployed as a productive application, as it uses the production ready frameworks and technologies used in Pharo. We generated 10 concurrent requests to our server during 2 minutes, generating an average of 700 requests per second. This simulation generates the load expected in a production server. Then, we apply the update at minute 1.

Finally, we apply measurements after the update is finished.

The scenario application, the set of test scripts and detailed instructions are available in GitHub[3]. Also Appendix A describes detailed instructions to replicate the validation experiments.

5.3 Validation 1: Application Update

Research Question. Is *gDSU* able to safely update a running stateful application in a development or production scenario?

Scenario. Our scenario application is a chat application. This application stores all the messages sent by all the users generating live instances in each request.

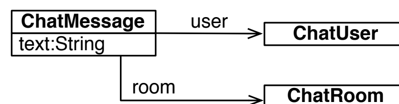


Figure 11 – Original Scenario. In this scenario all the messages are of a single class ChatMessage. This implementation has conditional code to handle the differences in messages from the system and from users.

Figure 11 shows the model of the application in Version 1. For presentation purposes, we show only the part that is relevant for the update. The application handles two types of messages. The first type is the one sent by a user in a room, the second type is the one produced by the room (*e.g.,* when a user enters or leave). In this version all the messages are instances of ChatMessage, when the message is an info message (that is not produced by a user) the user field is left as null.

Figure 12 shows the model of the application in Version 2. In this version the ChatMessage is refactored to include two subclasses. One for the user messages and the other for the info messages. Going from a version to the other requires the migration of live instances.

Results. *gDSU* updates correctly both the development and production applications. Instances are correctly migrated in an atomic fashion. No inconsistencies are introduced.

5.4 Validation 2: Update of the DSU

Research Question. Is *gDSU* able to update itself?

[3] https://github.com/tesonep/chatServer.git

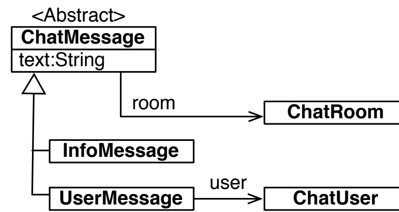


Figure 12 – The application is refactored to extract the different behavior in the messages in two subclasses (InfoMessage and UserMessage) to represent the messages sent by the system and by a user.

Scenarios. We have performed three updates on *gDSU* code:

1. Update the safe point detection algorithm.
2. Update the application of structural changes.
3. Update the internal representation of the update (*i.e.,* modifying a *gDSU* stateful class with live instances during the update).

Results. All updates were successful. All of them show that *gDSU* can update code and migrate instances that are related to and used by itself. Moreover, the first experiment shows that the safe update detection works even if the affected method is in the DSU thread.

5.5 Validation 3: Update of Language Core Libraries

Research Question. Is *gDSU* able to update core language libraries?

Scenarios. To validate the ability to update the language core libraries we experimented with the following two scenarios:

1. Update the OrderedCollection class, adding a new instance variable holding the size of the collection and modifying all related methods. The OrderedCollection class is a key part of Pharo's collections framework. It is the main collection used in the whole environment. In any given Pharo environment there around 46 000 live instances of this class. Also this class is extensively used by *gDSU.*
2. Update the class builder modifying the methods *gDSU* uses to create classes. All the operations modifying a class in Pharo are performed through the class builder. This component is a crucial part of the live programming capabilities of Pharo. Also, it is used as a key part of *gDSU.*

As said before, both elements are used during the execution of the update, introducing circularity issues similar to the changing *gDSU* itself.

Results. All updates were successful. These experiments demonstrate that the running core libraries are indeed isolated from the update. Otherwise, modifying the core libraries while performing the update risks to compromise the whole application stability.

5.6 Validation 4: Benchmarks

To evaluate the performance of our solution we have performed two series of benchmarks. The first one analyses how the number of live instances to migrate affects the update time. The second benchmark analyses the impact of the update process on the response time of a running application. The benchmark has been executed using Pharo 6.1 32-bits, in a machine running OS X 10.12.6 having a 2,6 GHz Intel Core i7 and 8 Gb of 1600 MHz RAM memory.

Number of Migrated Instances. This benchmark shows the behaviour in terms of memory and time with a varying number of instances to migrate. We performed 172 updates varying zero to ten million instances and analysed the time and memory required to perform the migration. The results, in Figure 13, show that (1) the memory consumption is linear and it grows with the number of instances to migrate and (2) the time to execute the update is almost constant (around 1 second) below ten thousand instances and then it grows linearly.

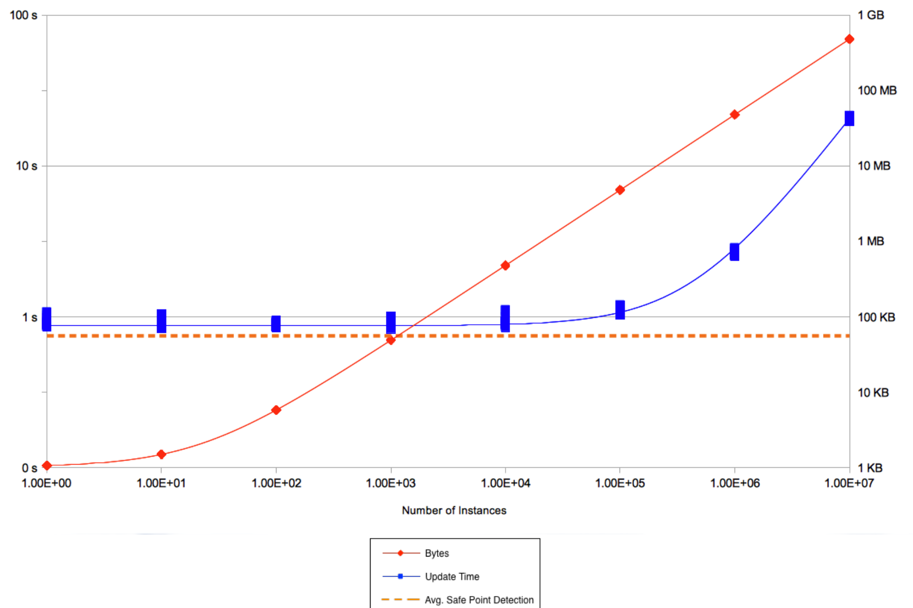


Figure 13 – Impact in memory space and execution time depending the number of instances to migrate.

The first result is due to the copy of the affected instances. The DSU process copies the modified instances and classes to perform the changes. The number of affected instances has a baseline of 13 instances when there are not live instances of the modified class to migrate. This set of instances are the core objects modified in the system *i.e.,* package manager, package, classes, global environment. So, any change will include at least these 13 instance to migrate. These instances are the minimum affected by an update. Starting from this baseline the update process only copies the instances to migrate.

The second result is due to two causes. The first one is that the detection of a safe update point takes an average of 750 milliseconds. During this time the application is running normally and the update process is just waiting. The second cause is that the bulk replacement takes a constant time of 250 milliseconds in average to create the

forwarders. If the number of forwarders does not fit in the available free memory, the bulk update runs a garbage collection and performs a traditional pointer swapping with a full scan [MB15]. Our benchmark shows that bulk replacement is able to handle about 25 thousand instances per second.

Server Response Time. This benchmark shows time measurements on the server application described in Section 5.2. The results, illustrated in Figure 14, shows the application response time. The response time during the update windows is of 1400 ms, with a number of instances to migrate around 28500. Contrastingly, the response time outside the update window ranges from 5 to 200 ms, with an average of 22 ms. This benchmark shows that the impact in response time is inside the parameters of the first benchmark.

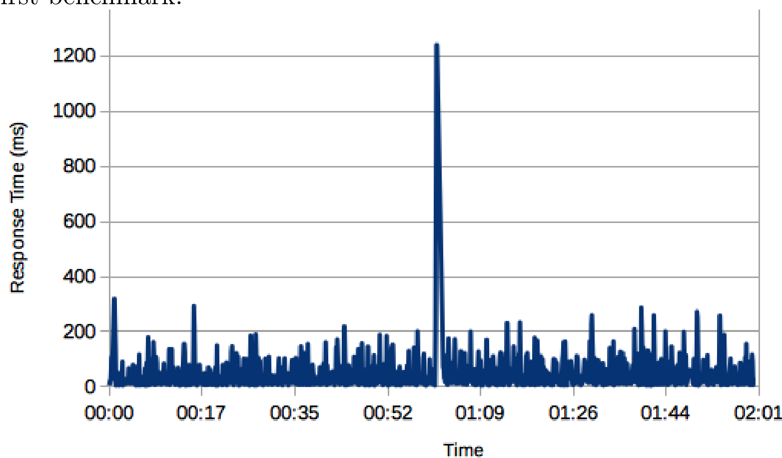


Figure 14 – The response time is only affected briefly during a small update window.

6 Discussion

Conservative Safe Update Points. We decided to look for safe update points instead of reconstructing the threads' call stack after the update. This decision simplifies the solution and permits the execution of any change in the instance structure or in the methods. Our solution does not stop the execution of the application while it is waiting for a safe update point, the application is running freely until this safe point is reached.

Rewriting the call stack allows the execution of the update in any moment, without needing to wait for reaching a safe update point. However, stack reconstruction techniques are limited in the number of method changes it can handle without developers' manual intervention, specially with loops and recursive methods. They also require full instrumentation of the code and enough history of previous runs.

Our solution provides a way of automatically detecting safe update points in a pretty conservative manner. We are aware that the proposed safe point detection does not always arrive to a safe point, since some programming patterns produce programs that never reach a safe point in our definition. For example, patterns such as the one in Figure 15 produces a method that never returns so it never reaches a safe update

point. We left the analysis of these programming patterns and how they could be detected, handled or rewritten outside the scope of this paper, but it is a good topic for future research.

```
execution
    [ true ] whileTrue: [
        self doAction1.
        ...
        self doAction2.
    ]
```

Figure 15 – Example of a Programming Pattern that does not allow the program to reach a safe update points if this method is updated in the patch

We have decided that aborting the update is better than performing the update without the guarantees needed to continue normal execution. Less restrictive conditions are possible, but these conditions are more complex to detect and they do not provide enough guarantees to execute all the possible changes. Other solutions [PH13, MME12] require explicit update points to handle these situations.

Eager vs. Lazy Migration. We have decided to use eager instance migration instead of lazy migration because lazy instance migration requires the use of proxies. By using eager migration, the penalty during the normal execution of the application is zero. During the normal execution of the application there is no need to execute code of the DSU solution.

Manual Development. In our solution the developer has to implement part of the patch *i.e.,* migration policies and validations. We have decided to do that because most of these components are heavily coupled with the application under update. However, the *gDSU* tool is shipped with already implemented versions of such elements. The shipped implementations cover the common scenarios, and provide a way of extending them. The shipped system validations assert the correct state of the Pharo environment after an update. We consider the validations and the migration policies as elements that are equivalent to unitary tests. The required elements are part of the application code base and they are created during the whole life cycle of the application.

7 Related Work

In this section we compare existing DSU approaches to our proposed solution. To compare these solutions we use the requirements stated in Section 2.3. Table 3 presents the results of the comparison using the proposed requirements.

We have classified existing solutions so far in two categories: DSUs designed to be used in production environments (Section 7.2) and DSUs designed to be used in development environments (Section 7.3).

It is true that many of these solutions are usable in both scenarios. However, as they are designed to satisfy the requirements of one of these scenarios, the other scenario's requirements are not fully satisfied. For example, DSUs designed for production environments do not provide automatic generation of patches limiting its usability as a development tool, but they provide an extensive API for state migration that is not present in DSUs for development environments.

We have been considering classical Live Programming environments such as Lisp or Smalltalk (Section 7.1) within the development oriented DSUs. However, in this section we study them separately. Although they provide programming language support for DSU scenarios, they are not fully engineered for this task and considering them amongst development DSUs would be unfair.

Category	**Requirements**									**Examples**
	Atomicity	*State Migration*	*Automatic Safe Point Detection*	*Small Run-time Penalty*	*Minimal Downtime*	*Broad Applicability*	*Patch Generation*	*Patch Reuse*	*Self and Core Update*	
Classical Live Programming	○	◐	○	●	●	○	A	○	◐	Lisp, Clos, Smalltalk
Development DSUs	●	○	○	○	◐	○	A	○	○	Jrebel, Javeleon, Jvolve
Production DSUs	●	●	◐	○	●	○	M	●	◐	Rubah, DuSTM, Pymoult
Our Solution	●	●	●	●	●	●	S	●	●	

●: Yes ○: No ◐: Limited
A: Automatic M: Manual S: Semi-Automatic

Table 3 – Comparison Table of Related Work and Our solution

7.1 Classical Live Programming Environments

Live programming environments [San78], such as Lisp [Ste90], CLOS [KR90], Smalltalk [GR83] and Javascript environments allow developers to modify the code while the program is running. These tools allow hot update of running code. When the class structure changes, the structure of live instances is updated. Often these languages offer a Meta-Object Protocol (MOP) [KdRB91] to support version migration of instances and code modification [Riv96].

However, the migration support is limited. For example, when new fields are added, these new fields are left uninitialized. These solutions do not apply the changes atomically. They apply the modifications one at the time. As they do not implement atomicity, the sequencing of changes is mandatory. As sequencing is not always possible [PDF+15] they have limited ability to update core libraries. Also they do not handle the concurrency or the detection of safe update points. Contrastingly, *gDSU* performs changes atomically and it is able to update core libraries. Also, it performs the changes only when the application is in a safe update point.

These solutions are designed to be used during development. The required support is implemented in the language infrastructure and they do not produce additional

run-time impact. Patch generation is done automatically because the IDE is integrated with the update tools. *gDSU* is designed to be used during development as it is also fully integrated with the IDE. Also, *gDSU* leverages language infrastructure and existing VM techniques. It does not produce additional run-time impact.

Finally, live programming tools for Javascript (*e.g.,* Nodemon[4], Firebug[5], Chrome Dev Tools[6]) do not handle the migration of live instances. *gDSU* migrates the state of live instances.

7.2 Production DSUs

DSUs designed for production provide the means for applying an update atomically and provide state migration mechanisms. They are designed to minimize the downtime during the execution of the update. While they require the manual generation of the patch, they provide reusable elements to compose the patches. Also the patches allows the expression of limited instance migration logic. They provide limited self and core update. *gDSU* is also designed to minimize the application downtime. However, it automatises the task of patch generation and provides a full support for self and core update.

Production DSUs present run-time impact. Pymoult [MDB15] requires modifications in the VM. DuSTM [PC11] and DUSC [ORH02] implement a Java DSU that requires bytecode transformations, and use lazy proxys to handle migration of data. Thus, they generate an impact in the size of the program and a penalty in the execution. Rubah [PH13] is a Java DSU implementation that uses also bytecode rewriting techniques and lazy proxy for instance migration. This solution presents an execution penalty during the normal execution of the application, affecting the application outside the update window. *gDSU* leverages existing VM techniques and it does not introduce run-time penalty outside the update window.

Regarding patch generation, Rubah [PH13] and Pymoult [MDB15] implement an API for the update of a program. The developer should express the changes implementing the update logic. This logic is executed and it is the responsible of performing all changes. The update objects can extend already implemented solutions. Contrastingly, *gDSU* is integrated with the language IDE and reduces the need of manual patch writing.

7.3 Development DSUs

Development DSUs such as Javeleon [GJK$^+$12], Jvolve [SHM09], DCEVM [WWS10] and JRebel [Zer12] are intended to be used during the development of the application. They are integrated with the language IDE and generates the patches automatically. However, they do not allow custom state migrations. *gDSU* is not only integrated with the IDE but it also allows expressing business state migration logic.

As they are designed to be used during development, these solutions were not designed with performance impact in mind. DCEVM requires a modification in the Java VM to run. JRebel does not require VM modifications but the amount of changes it can handle is limited (*e.g.,* it does not allow hierarchy changes). Jvolve implements a virtual machine level DSU, so it needs a special version of the Java Virtual Machine to run. JavAdaptor [PKC$^+$13] uses proxies and delegation, having an impact in

[4] https://github.com/remy/nodemon
[5] https://addons.mozilla.org/en-US/firefox/addon/firebug/
[6] https://developer.chrome.com/devtools

the execution of the application. JavAdaptor includes some of the characteristics of Production DSUs, although it lacks support to express complex instance structure migration (*i.e.,* renaming variables, changing the type of variables, changing from a native value to an object). Lacking this support is not a problem in development environments, but it limits its usability for production environments.

Javeleon requires bytecode instrumentation and class modification during the bootstrap of the application.

In all these solutions, the DSU related code is running all the time impacting the global performance of the application even outside the update window. Contrastingly, *gDSU* is designed to minimize the run-time impact.

Development DSUs are not able to apply changes to the DSU engine or the language core libraries, on the contrary *gDSU* allows core libraries update.

7.4 Related Techniques

The literature describes many other techniques that are not a full DSU implementation but could be applied to one. In this section we compare several such solutions with the design choices we included in *gDSU*.

Regarding safe point detection techniques, Cazzola et al. [CJ16] propose an automatic way to determine unsafe update points. Their technique uses static analysis of the changes to detect unsafe update points. Then the update process uses this information to avoid performing an update in those points. This solution could not handle the case when the problems arise from third party libraries, where the source code is not available.

Makris et al. [MB09] propose a way of updating software, without waiting for a safe update point, but in their solution the application source code should be instrumented before execution. Contrastingly, *gDSU* automatically detects update points in a conservative way. It does not require to anticipate them with code instrumentation or explicit code changes. This makes our solution simpler to use and has a smaller performance penalty at run time.

Regarding atomicity, Denker et al. [DGL+07] and Mattis et al. [MRH17] propose to isolate the applied changes during the execution of the application using a copy of the environment. Mattis et al. duplicate the modified methods and introduces a method lookup strategy that selects the correct version of the method to execute depending the running context. Also they modify the classes to store the values in a thread-scoped structure. As in our solution, they also limit the copy of instances to the affected ones. As our solution they look for safe update points. Denker et al. copy not only the modified methods but also the modified objects and classes. However, they do not address the problem of committing the changes or the migration of instances from one environment to the other, or the handling of globally shared state.

Penney et al. [PS87] propose a way of handling modifications, but they require modifications in the virtual machine. Wernli et al. [WLN13] propose to manage isolation using a copy and keeping alive both the old and new environment. The access to the modified objects is done through a lazy proxy, adding an execution penalty to the application after the update. This also produces problems with the identity of objects because the updated instances are duplicated. Our solution solves isolation with a copy of the environment. This copy is only used during the update window and replaces entirely the old objects on a commit operations. *gDSU* has a small run-time penalty because it copies only modified objects and it leverages existing support from the VM to do the atomic commit.

Regarding the safety guarantees provided by the DSU, Tedsuto [PH16] is a tool that provides a general framework for testing updates and detecting if they are successful. However, it requires manual intervention to create the validations and the invalid environment should be discarded so it cannot be applied to a productive environment. *gDSU* performs all validations in an isolated environment. This allows one to easily validate and cancel the update without affecting the running application.

There are also architectural alternatives to DSU [KM85, OMT98, PBJ98]. However, these solutions should manually take care of replication and persistence of application state. They require more complex and manual update schemes and special handling of running instances and processes. Also, they are only designed to handle anticipated update situations. Because of this, we left these solutions out of the scope of this paper.

8 Conclusion

In this paper, we described the existing problems in applying an update in a running application. We showed that safely applying an update requires an atomic dynamic software update engine. However, existing solutions so far are only applicable to either development or production scenarios but not both of them.

We proposed *gDSU*, a general DSU engine that proves useful to safely update applications in both kind of scenarios. With this purpose, *gDSU* implements safe update point detection, a partial copy of the environment and an efficient atomic commit that leverages existing VM technology. Moreover, *gDSU* provides several facilities to ease its usage: it provides automatic patch generation, it can update itself and the core language libraries.

We have validated our solution with a prototype in Pharo. We show that our solution is applicable to both use cases in several scenarios, and introduces only an execution penalty during the update window. Our solution is able to apply an update impacting 100,000 instances in 1 second. In this 1 second, only during 250 milliseconds the application is not responsive. The rest of the time the application is still running normally while *gDSU* is looking for the safe update point. The update only requires to copy the elements that are modified.

In the future, we will work on the automatic generation of patches and the integration with software development tools as automatic refactorings, providing more support to autogenerated migration policies and validations. We will be also working on a better implementation and on performance improvements. Our project's objective is not only to have an atomic DSU solution, but to provide a useful tool for developers. Another possible evolution path is to work on improved detection of safe update points, and the impact of different strategies to find them. Finally, other interesting approach could be an empirical study to analyse the manual work required in producing a patch and detect the required automations to improve its use in daily development.

A Instructions to Reproduce Validation Experiments

To validate the proposed solution we need to use a stateful bench application. We implemented a stateful chat server. This simple application presents all the problems and requirements described in this work. This application allows us to replicate and evaluate the design decisions in our proposed solution.

The bench application is used to validate the following scenarios:

- Updating application code while migrating live instances.
- Updating kernel libraries of the environment.
- Updating the DSU tool itself.
- Benchmarking the DSU implementation.

A.1 Installation

The bench application, tests scripts and more documentation are available in a GitHub repository. This repository is located in https://github.com/tesonep/chatServer.

The first step to install the bench application is to clone the given repository in a local machine. For the results included in this paper the validations have been executed in a machine running OS X 10.12.6 having a 2,6 GHz Intel Core i7 and 8 Gb of 1600 MHz RAM memory. However, the same validations will run in Linux or Windows machines.

The bench application runs in Pharo 6.1[7]. A basic knowledge of Pharo is required to execute the validations. The documentation of Pharo and beginner instructions are also available in the Pharo web site.

A 32-bits image and VM are needed to run the validations. They are available at the download site.

On this fresh image the bench application and the DSU tool should be installed. To do so, Listing 1 should be executed in the image.

```
EpMonitor current disable.
Deprecation showWarning: false.
Deprecation raiseWarning: false.

Metacello new
  baseline: 'ChatServer';
  repository: 'filetree://pathToClonedRepository';
  load.

EpMonitor current enable.
```

Listing 1 – Installing Bench Application

This script will install the bench application and all the required libraries. Including the DSU tool. For the DSU tool it uses a release called *JOTPaperVersion*.

To simplify the installation in the repository we provide a script that download the required elements and install them inside the *build* subdirectory. The file install.sh is the simplified installation script.

A.2 Executing Validations

All the validations have been executed using the same image. For running the image in development mode there is a script in the root of the repository called run.sh. This script runs the image in interactive mode.

Also this script opens a Pharo Playground to execute different statements. The Playground is a REPL to execute Smalltalk code. It is similar to the Scala Worksheet

[7] http://pharo.org

or the old Workspace in other Smalltalk dialects. These are the pieces of code to replicate the experiments.

In the Pharo Playground to evaluate a line, the line should be selected and with the context menu execute *Do it*.

A.2.1 Preparation

To execute the different validations, live instances are needed. Generation of instances is performed by executing listing 2. This script generates the users and messages instances to execute the validations. The number of instances can be modified to reflect different scenarios.

```
"Generate Instances"
ChatUpdate new generateInstances: 10000.
```

Listing 2 – Generating Bench Instances

The generated instances, and the code of the application is accessible to browse. Listing 3 presents the code needed to open the instance inspector and the source code browser. Checking the instances is needed to see if the instance migration is correctly performed.

```
"Inspect User Instances"
ChUser allInstances inspect.
"Inspect Message Instances"
ChMessage allSubInstances inspect.

"Browse Model"
'ChatServer-Model' asPackage browse.
```

Listing 3 – Browsing Code and Instances

A.2.2 Running Validations

For all the validations we have to execute the preparation steps.

For Validation 1, listing 4 shows the code to apply and revert the application update. Once the application is updated (or reverted) the changes are seen in the instance inspectors and in the source code browser.

```
"Update Model"
ChatUpdate new updateV1ToV2.

"Revert Update Model"
ChatUpdate new updateV2ToV1.
```

Listing 4 – Updating the Bench Application

Even more, there are tests in the code base of the bench application to test it.

For Validation 2, listing 5 shows the code to apply and revert the update on the DSU tool. After doing the update we can execute any of the other validations to see that the DSU tool is operative.

```
"Update DSU"
ChatUpdate new updateAtomicProcess.

"Revert DSU"
ChatUpdate new revertUpdateAtomicProcess.
```

Listing 5 – Updating the DSU itself

For Validation 3, listing 6 shows the code to apply and revert the update in the OrderedCollection class, and in the class builder. This classes are used by the DSU process and they are part of the kernel of Pharo language.

```
"Update Kernel libraries"
ChatUpdate new updateKernel.

"Revert Kernel Libraries"
ChatUpdate new revertUpdateKernel.
```

Listing 6 – Updating Core Libraries

To achieve repeatability we have scripted the changes in the methods that are used for each of the updates. These methods can be easily browsed to see the executed changes. Moreover, these methods can be modified to perform other updates. For example, the 3rd validation updates is implemented in the method updateKernel of the ChatUpdate class.

A.3 Executing Benchmarks

We implemented two different benchmarks, the first one measures the memory consumption of the DSU process and the second one the downtime during an update.

A.3.1 Memory Consumption

To measure memory consumption, we have implemented a benchmark that executes the application update 8 times. Using 1 to 10,000,000 live instances (using 10^n where n is in [0,8]). Listing 7 shows the code executing during this benchmark. This process outputs the results in the standard output of the terminal. Listing 8 shows the code to evaluate to run the benchmark.

```
ChatUpdateMeter >> doTest
  ((0 to: 7) collect:[:e | 10 ** e]) collect:[:q |
      Stdio stdout << q asString.
      Stdio stdout << (q -> (ChatUpdateMeter new testWith:q)) asString.
      Stdio stdout crlf ; flush.
  ].

  Smalltalk garbageCollect.

ChatUpdateMeter >> testWith: numberOfInstances
  | user room duration |

  ChUser initialize.
  ChRoom initialize.

  (1 to: 3 do: [:e | Smalltalk garbageCollect ]).

  user := ChUser registerUser: 'username' firstName: 'firstName' lastName: 'lastName'.
  room := ChRoom addRoom: 'roomName'.

  instances := OrderedCollection new: numberOfInstances.
  1 to:numberOfInstances do:[:i |
    instances add:(i % 2 = 0 ifTrue:[
      ChMessage from: user to: room text: 'Generic user message'
    ] ifFalse:[
      ChMessage in: room text: 'Generic info message'.
    ])
  ].

  self assert: instances size = numberOfInstances.
  duration := [ChatUpdate new updateV1ToV2] timeToRun .
  "The garbage collector runs many times to force the instances
  to move to the old space. Doing so we test a scenario that
  is closer to a long running application."
  (1 to: 3 do: [:e | Smalltalk garbageCollect ]).

  "Perform the update"
  ChatUpdate new updateV2ToV1.
  ^ duration.
```

Listing 7 – Memory Consumption Benchmark

```
"Executing memory and time benchmark per instance quantity. (Long to execute). It outputs in the
   Standard Output the different test executed, listing number of instances and time consumed.
   From 1 instance to 10.000.000"

ChatUpdateMeter new doTest.
```

Listing 8 – Launching Memory Consumption Benchmark

A.3.2 Server Response Time

The second benchmark is designed to validate the downtime of the application during an update. In this benchmark, the application is running in server mode. It implements a REST server to receive request through HTTP.

The server is launched and stopped from the image. Listing 9 shows the Smalltalk code to start and stop the server instance.

REST URL	Action
http://localhost:1701/updateV1ToV2	Updates from V1 to V2.
http://localhost:1701/updateV2ToV1	Updates from V2 to V1.
http://localhost:1701/updateKernel	Updates the Kernel implementation of OrderedCollection.
http://localhost:1701/revertUpdateKernel	Reverts the Kernel implementation of OrderedCollection.
http://localhost:1701/updateAtomicProcess	Updates the DSU implementation.
http://localhost:1701/revertUpdateAtomicProcess	Reverts the DSU implementation.

Table 4 – Update REST Entry points

```
"Starting the HTTP test to run the benchmark with JMeter. The REST server is listening in port 1701
    "
ChatServer uniqueInstance start.

"Stopping server"
ChatServer uniqueInstance stop.
```

Listing 9 – Controlling the Bench Server

Once the server is started, the update process can be requested via REST calls. We implemented a different REST request for each of the updates. Table 4 shows the REST entry points that can be used to perform different updates while the server is running.

To simulate the load of the server we use a JMeter[8] script. This script is designed to perform 10 concurrent requests during 2 minutes. It generates an average of 700 requests per second. The script is located in the root of the git repository, and it is named chatServer.jmx. We refer to the documentation of JMeter on how to run the given script.

In the benchmark, we execute the JMeter script and after one minute we perform one of the update REST calls. Once the JMeter scripts ends it presents the results of the benchmark.

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://pharobyexample.org/ ,http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf.

[CJ16] Walter Cazzola and Mehdi Jalili. Dodging unsafe update points in java dynamic software updating systems. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 332–341. IEEE, 2016. doi:10.1109/ISSRE.2016.17.

[8] http://jmeter.apache.org/

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. URL: http://www.iam.unibe.ch/~scg/OORP, doi:10.1109/ICSM.2005.67.

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007. URL: http://rmod.inria.fr/archives/papers/Denk07c-ICDL07-Changeboxes.pdf, doi:10.1145/1352678.1352681.

[GJK+12] Allan Raundahl Gregersen, Bo Nørregaard Jørgensen, Kai Koskimies, et al. Javeleon: An integrated platform for dynamic software updating and its application in self-* systems. In *Engineering and Technology (S-CET), 2012 Spring Congress on*, pages 1–9. IEEE, 2012. doi:10.1109/SCET.2012.6341955.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. URL: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005. doi:10.1145/1108970.1108971.

[HN12] Christopher M Hayden and Iulian Neamtiu. Report on the third workshop on hot topics in software upgrades (hotswup'11). *ACM SIGOPS Operating Systems Review*, 46(1):93–99, 2012. doi:10.1145/2146382.2146399.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KM85] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985. doi:10.1109/TSE.1985.232231.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in pcl. In *Proceedings of ACM conference on Lisp and Functional Programming*, pages 99–105, Nice, 1990. doi:10.1145/91556.91600.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985. URL: ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf.

[MB09] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, volume 2009, 2009.

[MB15] Eliot Miranda and Clément Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management (ISMM '15)*, pages 93–104, Portland, United States, June 2015. URL: https://hal.inria.fr/hal-01152610, doi:10.1145/2754169.2754186.

[MDB15] Sébastien Martinez, Fabien DAGNAT, and Jérémy Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015 : 10th International Conference on Software Engineering Advances*, pages 80 –

85, Barcelone, Spain, nov 2015. URL: https://hal.archives-ouvertes.fr/hal-01247603.

[MME12] Emili Miedes and Francesc D Munoz-Escoı. Dynamic software update. Technical report, Technical Report ITI-SIDI-2012/004, 2012.

[MRH17] Toni Mattis, Patrick Rein, and Robert Hirschfeld. Edit transactions: Dynamically scoped change sets for controlled updates in live programming. *The Art, Science, and Engineering of Programming*, 1, 2017. URL: http://arxiv.org/abs/1703.10862, doi:10.22152/programming-journal.org/2017/1/13.

[NH09] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1542476.1542479, doi:10.1145/1542476.1542479.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=302163.302181, doi:10.1109/ICSE.1998.671114.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658. IEEE, 2002. doi:10.1109/ICSM.2002.1167829.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: architecture for component trading and dynamic updating. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*, pages 43–51, May 1998. doi:10.1109/CDS.1998.675757.

[PC11] Luis Pina and Joao Cachopo. Dust'm-dynamic upgrades using software transactional memory. 2011. doi:10.1109/hotswup.2012.6226612.

[PDF+15] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Max Mattone. Virtualization support for dynamic core library update. In *Onward! 2015*, 2015. URL: http://rmod.inria.fr/archives/papers/Poli15b-Onward-CoreLibrariesHotUpdate.pdf, doi:10.1145/2814228.2814233.

[PH13] Luis Pina and Michael Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. In *HotSWUp*, 2013. doi:10.1.1.711.3281.

[PH16] Luís Pina and Michael Hicks. Tedsuto: A general framework for testing dynamic software updates. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 278–287. IEEE, 2016. doi:10.1109/ICST.2016.27.

[PKC+13] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor-flexible runtime updates of java applications. *Software: Practice and Experience*, 43(2):153–185, 2013. doi:10.1002/spe.2107.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the gemstone object-oriented DBMS. In *Proceedings OOPSLA '87, ACM SIGPLAN*

Notices, volume 22, pages 111–117, December 1987. doi:10.1145/38807.38817.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996. doi:10.1.1.111.5354.

[SAM13] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013. doi:10.1002/smr.1556.

[San78] Erik Sandewall. Programming in an interactive environment: The "lisp" experience. *ACM Comput. Surv.*, 10(1):35–71, March 1978. URL: http://doi.acm.org/10.1145/356715.356719, doi:10.1145/356715.356719.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, June 2009. URL: http://doi.acm.org/10.1145/1543135.1542478, doi:10.1145/1543135.1542478.

[Ste90] Guy L. Steele. *Common Lisp The Language*. Digital Press, second edition, 1990.

[WLN13] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. *Journal of Object Technology*, 12(3):1:1–27, August 2013. URL: http://www.jot.fm/contents/issue_2013_08/article1.html, doi:10.5381/jot.2013.12.3.a1.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1852761.1852764, doi:10.1145/1852761.1852764.

[Zer12] ZeroTurnAround. What developers want: The end of application redeployes. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012.

About the authors

Pablo Tesone is a PhD student at IMT Lille Douai, France. He is studying *Dynamic Software Update Solutions* applied to Live programming environments, distributed systems and robotic applications. He is interested in improving the tools and the daily development process. He is an enthusiast of the object oriented programming and their tools. He collaborates with different open source projects like the ones in the Pharo Community[9] and the Uqbar Foundation[10].

Guillermo Polito is a coding enthusiast, software engineer and researcher. He is research engineer at CNRS working currently in the RMoD[11] and Emeraude[12] teams. His research targets programming language abstractions and tool support for modular long-lived systems. For this, he studies how reflective systems can evolve

[9] http://pharo.org/
[10] http://www.uqbar-project.org/
[11] http://rmod.lille.inria.fr
[12] http://www.cristal.univ-lille.fr/emeraude/

while maintaining these properties. He is interested in how these concepts combine with distribution and concurrency. He obtained the 13 of April 2015 his PhD entitled *Virtualization support for application runtime specialization and extension* under the direction of Stéphane Ducasse (Inria Rmod team) and the supervision of Noury Bouraqadi and Luc Fabresse (CAR team of Mines Douai). He loves coding and spends a lot of my free time helping the amazing community of Pharo. He also participates in several projects such as the Pharo's database driver suite (DBXTalk), its shortcut framework, or the static web page generator Ecstatic.

Noury Bouraqadi is a full professor at IMT Lille Douai, France since 2001. His research addresses mobile and autonomous robots from two complementary perspectives: Software Engineering (SE) and artificial intelligence (AI). From the SE perspective, he studies software architectures, languages and tools for controlling individual robots. He mainly focuses on reflective and dynamic languages, as well as component models, for a modular and agile development of robotic software architectures. From the AI perspective, he studies coordination and cooperation in robotic fleets. He mainly focuses on communication models as well as emerging or predefined organizations for multi-agent robotic systems.

Stéphane Ducasse is directeur de recherche at Inria. He leads the RMoD[13] team. He is expert in two domains: object-oriented language design and reengineering. He worked on traits, composable groups of methods. Traits have been introduced in Pharo, Perl, PHP and under a variant into Scala and Fortress. He is also expert on software quality, program understanding, program visualisations, reengineering and metamodeling. He is one of the developer of Moose[14], an open-source software analysis platform. He created Synectique[15] a company building dedicated tools for advanced software analyses. He is one of the leader of Pharo[16] a dynamic reflective object-oriented language supporting live programming. The objective of Pharo is to create an ecosystem where innovation and business bloom. He wrote several books such as Functional Programming in Scheme, Pharo by Example, Deep into Pharo, Object-oriented Reengineering Patterns, Dynamic web development with Seaside. According to Google his h-index is 53 for more than 12000 citations. He would like to thanks all the researchers referencing his work!

Luc Fabresse is associate professor at IMT Lille Douai, France. His researches aims at easing the development of mobile and constrained software using dynamic and reflective languages such as Pharo. One of his goal is to support live programming of mobile and autonomous robots in an efficient way. He is the co-author of multiple research papers (http://car.mines-douai.fr/luc) and he concretizes all these ideas (models and tools) in Pharo to develop, debug, test, deploy, execute and benchmark robotics applications. Each year, Luc also gives computer science lectures, co-organizes events (technical days, conferences, ...) and promotes Smalltalk as an ESUG (European Smalltlak User Group) board member.

[13] http://rmod.lille.inria.fr
[14] http://www.moosetechnology.org/
[15] http://www.synectique.eu/
[16] http://www.pharo.org/

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.