Technical Report RT/7/2009

Atomic Dynamic Software Upgrades

Luis Pina
INESC-ID/IST
luis.pina@ist.utl.pt

João Cachopo
INESC-ID/IST
joao.cachopo@ist.utl.pt

Jan 2009
Upgrading software still is a problem that lacks any automated and practical solution. Systems that do not allow any downtime implement upgrades through a combination of application-specific software and redundant hardware.

The novel feature of the proposed upgrade system is to use a Software Transactional Memory (STM) to deal with software upgrades, just as it deals with program data. This feature has an important advantage: the ability to upgrade the program in an atomic fashion without disrupting the application.

This paper also presents a conceptual model of an abstract upgrade system based on STMs, an overview of the existing systems that support dynamic software upgrading, as well as a framework that allows us to reason, classify and compare such systems. It also proposes an upgrade system based on STMs that does not need the redundant hardware and is embedded in the Fénix Framework.

**Keywords:** Software Transactional Memory, Dynamic Software Upgrades.
Atomic Dynamic Software Upgrades *

Luís Pina and João Cachopo
INESC-ID/IST

Abstract. Upgrading software still is a problem that lacks any automated and practical solution. Systems that do not allow any downtime implement upgrades through a combination of application-specific software and redundant hardware.

The novel feature of the proposed upgrade system is to use a Software Transactional Memory (STM) to deal with software upgrades, just as it deals with program data. This feature has an important advantage: the ability to upgrade the program in an atomic fashion without disrupting the application.

This paper also presents a conceptual model of an abstract upgrade system based on STMs, an overview of the existing systems that support dynamic software upgrading, as well as a framework that allows us to reason, classify and compare such systems. It also proposes an upgrade system based on STMs that does not need the redundant hardware and is embedded in the Fénix Framework.

1 Introduction

With the ubiquity of internet, software upgrading met a new development. The logistics of distributing applications to its customers was simplified. Instead of shipping them in a physical medium, like a floppy disk or a CD, new upgrades can be published on a web site and the user, or the application itself, can download and install those upgrades.

Yet, upgrading software is not a trivial task. Typically, it requires restarting the application that needs to be upgraded. Several services on the internet run continuously, without allowing any downtime at all. Upgrading such programs cannot involve its restart, since it would mean user noticeable downtime. Thus, upgrading these programs is a quite complex task.

Software upgrades are also used to introduce new features that add value to the application as a commercial product. Customers want to be able to use the new features as soon as they are released. Quite often, such applications have bugs that need to be fixed. Internet also brought a new type of bugs: security related. Software users do not notice such bugs that need to be corrected as soon as they are detected.

* This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006), which is funded by the Portuguese FCT (Fundação para a Ciência e a Tecnologia).
As the internet grows and evolves, reaching more people and more devices, so does the need of upgrading software. As the next scenarios show, upgrading software still is a problem that lacks any automated and practical solution. An upgrade system that supports Dynamic Software Upgrades is able to upgrade applications “on the fly”, without any disruption nor data loss, as we shall see in the following scenarios.

1.1 Scenario 1: Desktop Application

Consider a desktop application that allows the user to have several open documents, some not yet saved. By saved, I mean committed to a persistent representation that allows the application to restore the exact state of the documents after being closed. This application can be a common office productivity application or a web browser, available to the average user.

Now consider that the previous application is running on a laptop that the user hibernates when he is moving from one place to another. This application is, in this condition, able to be kept open for quite a while (spanning from days to weeks). Even if all the documents are saved, there is some state of the application itself that cannot be saved. For instance, consider a file editor that allows the user to split the screen and edit a different file in each split. Although an user is able to save all the files, he cannot restore the split by just opening all files again. Or consider an office productivity application. Although an user can save a document, he cannot save the exact task he was performing before closing the document (for instance, a complex search and replace that requires the user to read a phrase before performing each change).

So far I have defined the application and its environment. Consider also that this application detects a new version on its vendor site and asks the user to upgrade. Knowing that the upgrade will close the application and lose all that “unsavable” state, the user is most likely to prevent the application from upgrading during a long time.

On the other hand, if the application asks the user to upgrade but guarantees that it will restore the precise state after upgrading, the user is most likely to allow the upgrade to happen, even if the application cannot be used during the upgrade.

1.2 Scenario 2: Server Application

Consider a server application that is used by hundreds of users at any moment. If such application is running on a single server, performing an upgrade means that the application must be restarted. Meanwhile that application cannot be used, resulting in an user noticeable downtime.

Using Dynamic Software Upgrades, the application’s upgrade can be achieved without any downtime at all. Nevertheless, given the high concurrency of such applications, we must upgrade the application in an atomic way. Without this atomicity, users can get conflicting results and data may get corrupted.

As pointed out in [19], these applications often overcome the upgrading problem
with a combination of application-specific software and redundant hardware. The redundant hardware may already be present to provide fault tolerance and/or load balancing. Taking advantage of this redundant hardware, there are some techniques to upgrade the application gradually [8, 4]. However, maintaining the state across all redundant hardware using these techniques is a complex and mostly unsolved problem. Please note that, as in scenario 1, not all the state may be savable. For instance, user sessions may not be saved and an upgrade system must aim at preserving them in order to be transparent for the user.

Using Dynamic Software Upgrades, this upgrade can be done at each node just as another operation:

- Without adding any noticeable overhead (at least, when compared to the alternative techniques)
- Without needing any redundant hardware
- Maintaining the state that is not savable

But we must always keep in mind that distributing Dynamic Software Upgrades is not so straightforward if we consider atomicity.

The remainder of this document is organized as follows: Section 2 establishes the goals that my work must achieve and introduces the problem using an abstract conceptual model. Section 3 introduces concepts that are useful to classify systems that support Dynamic Software Upgrades and presents several systems using those concepts to compare them. Section 4 describes an overview of the upgrade system that I propose. Section 5 presents several benchmarks able to check if the solution achieves the established goals. Finally, I present the conclusions on Section 6.

2 Goals

The main goal of my work is to develop an upgrade system that allows the application to be used while it is being upgraded, without losing any part of its state due to the upgrade process, thus increasing the availability of the application. A similar system is presented by Hicks et al [19]. They also define four goals that any upgrade system supporting Dynamic Software Upgrades should ideally meet. Given that these goals also apply to any general-purpose upgrade system, I will establish them as goals of my work:

**Flexibility** Any part of a running application should be updateable without disruption.

**Robustness** The upgrade system should minimize the risk of errors and crashes due to an upgrade, using automated means to promote upgrade correctness.

**Ease of Use** The less complicated the upgrading process is, the less error-prone it will tend to be. The upgrade system should therefore be easy to use.

**Low Overhead** Making a program upgradable should impact its performance as little as possible.
Fig. 1. A running program, as described by the conceptual model

Besides these goals for general-purpose upgrade systems, I will establish the following additional goals:

**Timing** Any part of a running application must be updateable at any moment, even if that part is in execution when the update is performed. This goal appears as an extension of Flexibility.

**Multithreaded Environments** The upgrade system must be able to run correctly in a multithreaded environment, without causing any data corruption nor crash induced by the operation of the upgrade system in such condition.

Due to implementation constraints, I will establish also the following goals:

**Extend JVSTM** My work will be merged with the Fénix Framework project, which uses JVSTM as its Software Transactional Memory implementation.

**Java implementation** The proposed upgrade system must be integrated in Fénix Framework. This framework is implemented in Java, so the upgrade system must be written in Java and deal with its limitations as well.

### 2.1 Conceptual Model

This section describes the problem of upgrading dynamically an application in an abstract and simple way, introducing some concepts that will be useful throughout this document. Figure 1 summarizes the conceptual model presented in this section.

**Program** The program is a set of sequential instructions that define the behaviour of the application. There is another sequence of instructions that define the behaviour of the execution environment itself — the execution platform.
**Execution Environment** The execution environment runs the application by executing the program. It receives a *stimulus* from the exterior and reacts accordingly, executing the program and sending the result of that computation — a *response*.

The current state of the application is kept by the execution environment — the *program state*. The program reads and writes the program state at each handled request.

Each request is handled by its own control flow — a *thread*. The execution environment is able to support more than one thread. Each thread executes code concurrently with the remaining ones.

For simplicity, I assume that a thread handles only one stimulus. Such thread is launched by the execution environment when a stimulus arrives. After sending the response, the thread dies and the execution environment reclaims the resources used by the late thread. All this behaviour is defined by the execution platform.

Although each thread has its own private data — the *local program state* — in which it keeps local variables and local copies of portions of the program state, all threads share some portion of the program state — the *shared program state*.

Since there is a portion of the program state is shared by several threads, we require a synchronization mechanism in order to avoid concurrency related problems and resulting data corruption. I consider that a transactional memory is used — a *Software Transactional Memory*, or *STM* — to keep the shared program state correct. Each thread has access to a private local state that does not need any synchronization at all. Each thread executes its request on its own transaction, so the thread always has a coherent vision of the program state. Any change made to the shared program state by a thread is written to the shared program state only when the thread commits the transaction. This way, each request is handled by the execution environment on an atomic fashion.

**Software Upgrades** An *upgrade* is a special kind of stimulus. It contains a new program to replace the program currently in execution. Like all other stimuli, the upgrade is handled by a thread and must be perceived by the application as an atomic action.

However, unlike other requests, the handling of an upgrade means that the program itself must be written. So far, the program has been considered read only (immutable), thus each thread could read it without any synchronization mechanism. Introducing dynamic upgrades means that the thread’s accesses to the program must be synchronized also, just like the accesses to the program state.

Since we are using an STM, we have *equality between data and code*: different threads may be executing different programs just as different threads execute over different program states. In this case, each thread is running a *program version*. In order to support this behaviour in an atomic way, the program must also be kept in an STM.
Given the tight coupling of the program with the state it manipulates, the upgrade may need transform the program state to allow the correct operation of the new program. This transformation is done by a transform function\cite{7,12}: a function that maps the current program state to a new program state, compatible with the new program.

The execution platform is immutable during the lifetime of the execution environment. Therefore, it does not need any synchronization mechanism to remain correct.

3 Related Work

Several systems have already approached the dynamic upgrade problem. However, none of those systems combines all the features required to achieve the goals established in section 2.

3.1 Concepts

Before presenting the related work, we must consider some concepts that are useful to understand, classify and compare the features of different systems that support dynamic upgrading. Then, we can define the relevant features of the existing solutions by composing the concepts presented in this section.

**Partial Upgrades** Section 2.1 defined an upgrade as a special stimulus that modifies the program, replacing it with a new one. Although simple, to present replacing the entire program may cause a great amount of overhead. Besides, the upgrade system can take advantage of the modular structure of applications and programming languages and detect the bounds of the upgrade.

Typically, upgrades change only a portion of the program. I refer to these upgrades as partial upgrades. This section addresses the problem of defining partial upgrades, that is:

- Identifying the relevant portions of the program
- Defining what operation (delete, replace, user defined, . . .) should be applied to each relevant portion identified

For instance, an upgrade system for an object oriented programming language may detect the set of classes that an upgrade modified and replace only those classes.

**Program Comparison** The first approach to define the program upgrade is very similar to what source code revision control systems do. The programmer works on top of the current program. When he finishes, he submits the new program to the upgrade system. Like revision control systems, the upgrade system compares the two versions of the program (the current and the new) and incorporates the changes on the current program.
With this approach, the programmer can reason about the new program without bothering about the current one, manipulating it as a whole new consistent program instead of portions of new code patched into old one.

**Domain Specific Language (DSL)** The previous approach has some limitations: the upgrade system may not detect some type of upgrades. For instance, consider an object oriented application. On an upgrade, the name of a class is changed. To the upgrade system, the previous class was deleted and a new one was inserted. What will happen to the instances of this class? Will they be deleted?

A direct approach to this problem is quite simple: create a domain specific language in which the programmer can define explicitly:

- What portion of the program is to be upgraded
- What operation should be performed to the program and program state

With this language, the programmer could define completely an upgrade. Making this language small and simple enables a programmer to learn it quickly and use it easily. Besides the new program, the programmer also provides a file that describes the upgrade, written in the DSL.

**Programming Language Support** The previous approach may use an external DSL [17]: written in a language separate to the base language of an application. This kind of DSLs suffer from several disadvantages:

- The programmer must learn the DSL. One could argue that this language is sufficiently small and simple that it places no burden on the programmer at all, but it does not eliminate the need for a programmer to learn and use this language.
- Tools for supporting the change specification language must be developed. Even the smallest, simplest language that one can think of requires the effort of creating tools to parse it.
- Poor integration with the application’s programming language. This is the worst problem of external DSLs, rendering them error-prone. A completely new language is defined, detached from the programming language used to develop the application.

All these problems could be avoided by specifying the upgrades in the base language itself. There are three approaches possible:

**Define the DSL inside the application’s language** : The programmer still needs to learn the DSL. But, with an internal DSL, the poor integration previously described does not exist anymore. The DSL is fully merged inside the application programming language.

**Use the programming language to define the upgrades** : Some languages allow the programmer to redefine portions of a program directly, using the language itself.

**Use an API defined by the upgrade system** : The upgrade system must define the granularity of the program that it is able to upgrade as a partial upgrade. Using an API, the program can interact with the upgrade system as it interacts with other libraries. To the programmer, the upgrade system behaves as if it was defined on the programming language itself.
Converting the Program State  Applications structure their data in higher abstraction level entities (abstract data types, objects, etc). Thus, as I have described in the conceptual model, often evolving the program also means converting the program state, in order to make it compatible with the new program. This section addresses what support the upgrade system may offer to convert the program state to cope with the new program. Please note that there is still another problem to solve: when will the conversion happen. That problem will be addressed on the following section.

Automatic Conversion  There are several situations where the upgrade system can gather enough information to make the right decision about the correct conversion of the program state.

The upgrade system can use some rules for performing default conversion. For instance, consider that a program upgrade increases the precision of an integer, changing it from 32 to 64 bits. It is easy to write a rule that automatically deals with increasing the precision of arithmetic portions of the program state, freeing the programmer from this task.

An upgrade can also change the structure of the high abstraction level entities, inserting new data in them. In this case, the current program state does not have that new data in it. An upgrade system can initialize this data with some default value defined by the upgrade system. For instance, an upgrade system written in Java could initialize new primitive type fields to 0 and new reference type fields to null.

Custom Conversion  The correct decision about the conversion that the program state must suffer in order to be compatible with the new program may be arbitrarily complex. In this case, the upgrade system alone is not able to convert the program state correctly. It must provide means for the programmer to describe the custom conversion that must be applied to the program state.

Please note that, even in this case, the simple automatic conversion that the upgrade system provides frees the programmer from tedious, repetitive, and error-prone program state conversion.

Executing Upgrades  Consider an upgrade that is ready to be submitted to the upgrade system. The upgrade system must take into account new concerns:

- Define when will the upgrade be put in execution
- Define when the program state will be converted

This section will address these aspects of the upgrade system.

Offline Upgrades  This upgrade model require that the application is completely stopped (shut down and restarted) before the new program can be put into execution.

is model of upgrade has the following disadvantages:
- Shutting down the application means that some data cannot be saved, as it was pointed out on Section 1.1.
The service that the application provides is completely disrupted while the upgrade is taking place. Despite these disadvantages, this model of upgrade has also some good properties:

- Only persisted program state must be converted.
- The semantics of program state conversion is clear and simple.

**Immediate Upgrades** Also known as ”stop the world upgrades” [12, 7], this upgrade model prevents the application from executing other code while it is upgrading. It brings several benefits when compared to off-line upgrades:

- The application does not shut down completely. No data is lost during the upgrade and we skip the overhead of restarting the application is avoided.
- The semantics of upgrading the program is also clear and simple.

On the other hand, we still have some disadvantages:

- Navigating through the program state to identify the exact portion that must be converted may take a long time.
- As above, the service provided by the application is unavailable while the upgrade is processing.

**Lazy Upgrades** In this approach, once the new program is copied to the execution environment, the application can resume its execution. The program state is converted lazily: the upgrade system converts a portion of the program state only when the program tries to manipulate it.

This approach brings the following advantages:

- The disruption caused by the upgrade is minimal. The application is interrupted only as long as it needs to convert the portion of the program state immediately needed after the upgrade.
- As before, no data is lost due to the upgrading process.

This approach also introduces some disadvantages:

- The semantics of the upgrade process is not clear anymore. The upgrade system must be ever vigilant during the execution of the application, converting the program state just in time. As a result, programmers do not know exactly when the conversion code will be executed.
- At any instant, the application can have more than one program version. In both previous approaches, the application only dealt with two version of the program while upgrading. After the upgrade, only one version of the program was being executed. Now, the upgrade system must cope with several versions of the program.

**Multithreaded Environments** If an upgrade system follows off-line or immediate upgrades, it must stop all the threads before executing the upgrade. So, a multithreaded environment does not add other problems to the upgrade system.

On the other hand, upgrade systems that follow a lazy upgrade approach must deal with new problems. Interleaving application and program state upgrade threads is not trivial.

Consider the example shown in Figure 2. A and B are two application transactions, T(x) and T(y) are two upgrade transactions that convert object x.
and $y$, respectively. $A$ and $B$ use objects $x$ and $y$, respectively, triggering their conversion. $T(y)$ needs to read object $x$ in order to convert $y$. The figure shows an interleaving that results in $T(y)$ finding an unexpected interface for object $x$.

The question of ordering the upgrade of different portions of the program state is crucial when we use lazy upgrades, since the upgrade system cannot control the order that the application will follow when manipulating dependant portions. And there is an even worse problem: how should we deal with mutually dependant portions?

### 3.2 Persistent Object Data Stores

The program state in object oriented languages consists of a set of objects. These objects have an internal state — data — and behaviour — code. Each object is an instance of (at least) one class. All the classes are arranged in a hierarchy. Each object can reference other objects.

Considering all these aspects, we find new problems on upgrading applications developed using an object oriented language. These problems have been identified by previous work on persistent object data stores. This section will describe the problems found, as well as the proposed solutions, considering several persistent object data store systems.

**PJama** PJama [15] is an experimental prototype that implements Orthogonal Persistence for the Java platform (OPJ). It also provides a tool that supports the evolution of persistent classes and instances. PJama detects the classes that must be replaced using a simple change specification language. This language identifies clearly the classes that are to be modified and describes also how the instances should be migrated. Besides this language, PJama’s upgrade tool also detects all the classes that must be modified in order to keep the store consistent after modifying the interface of any class. Once the upgrade is detected, the upgrade system checks if two versions of the same class — the original in the store and the upgraded — are substitutable.
It follows a set of rules in order to perform this verification. The rules can be found in [15].

PJama’s upgrade tool supports a mixture of automatic and custom instance conversion, arranged in three types:

**Default Conversion** Given a pair of classes, the existing class in the store and its upgrade, this conversion performs a set of simple tasks, like copying of the unchanged fields, default initialization of new fields, and trivial automatic conversions (like widening primitive conversion, as defined in [18]).

**Bulk Conversion** This is a kind of custom conversion. It is supposed to be used when all the instances of any upgraded class should be converted in the same way. Other upgrade systems call these transform functions [7, 12]. When using bulk conversion, the programmer can also take advantage of the default conversion, as the upgrade system runs bulk conversion code only after performing the automatic default conversion.

**Fully Controlled Custom Conversion** Gives the programmer total control about how and in what order the instances are converted. The programmer writes a method `conversionMain` and provides it to the upgrade system. Then, the upgrade system will call this method, after the verification of substitutability, and will ignore any other conversion methods. The programmer gets total freedom and full responsibility for the results of such conversion.

PJama’s upgrade tool supports only (at least currently) offline upgrades. All the applications that use an object store cannot be running while the upgrade tool is running. The provided upgrade tool replaces the classes existing in the store and migrates all the instances that need to be migrated.

**OODMBS** There are several commercial OODBMS solutions that have already considered dynamic upgrading the objects contained in the data store they persist. In his work, Dmitriev [15] presents a survey about these systems that is still up-to-date. In this section, I resume briefly the approaches used by each solution, as well as the limitations.

Objectivity/DB [3] supports C++, Java and Smalltalk bindings. It supports custom object conversion through a custom reflection API to access fields of both the current and the new version of the evolving object. The documentation does not say anything about calling methods of evolving objects and recommends limiting the use of conversion functions to the objects being converted. It supports three kinds of object conversion:

- Offline, allowing arbitrary changes
- Lazy, supported with several restrictions:
  - Custom lazy conversion is allowed to set only primitive fields of objects
  - A lazily converted instance is made persistent only if it was converted inside an update transaction
  - Lazy conversion cannot be combined with class changes that affect the hierarchy
– On-Demand, means eagerly performing previously defined lazy conversion on a subset of objects

Versant Developer Suite [14] offers multiple language bindings, including C/C++ and Java. It supports lazy conversion, although it is limited to only default conversion. It also supports immediate conversion that allows arbitrary changes (except adding/dropping non-leaf classes) in a cumbersome fashion. For instance, in order to change a class \( C \) we must follow the following steps:

1. Create a new class \( D \) with the same definition, then run a program to create an instance of \( D \) per existing instance of \( C \), copy information between them and repair all references of \( C \) from other objects. This process must be all coded manually
2. Delete class \( C \). This process also deletes all of its instances
3. Create a new class \( C \), repeating step 1 in order to replace \( D \) by the new \( C \) and perform the required conversion when copying information between old class \( D \)
4. Delete class \( D \)

O2 [16] is an OODBMS offering C/C++ bindings with one of the most sophisticated upgrade system. It supports any modification (except adding/removing non-leaf classes) using two approaches:

– Incrementally, through primitives such as adding/deleting fields, described in a DSL
– By comparison with a new class definition, also written in the DSL

Conversion of instances can be performed immediately or lazily, with both default and custom conversion functions. O2 also supports versioning of classes and instances, requiring the programmer to provide forward and backward conversion functions for each version.

3.3 Chimera

Chimera [5] is an object-oriented deductive active data model. Besides the concepts commonly ascribed to object-oriented data models it also provides capabilities for defining deductive rules that can be used for a variety of purposes: define views and integrity constrains, formulate queries, specify methods to compute derived information.

Objects in Chimera can \textbf{migrate}: become direct members of a class which is different from the class from which the object has been created.

Migrating an instance to a sublass is trivial since the instance will maintain its interface. Consistency problems arise when we consider moving an instance to a superclass. Another instance may use some fields or methods that were lost. The typesafety of the application is at stake. Chimera uses two approaches to support this kind of migration:
Global Type Modification The object is modified directly, changing its state and deleting some methods or fields. All objects that had a reference to a migrated instance must be notified that the instance is no longer a member of the expected class. Using a programming language with static typing, we can avoid such problem by detecting which classes are affected by this upgrade and accept only upgrades that also make these classes compatible.

Local Type Modification The object is not modified directly, instead a new view of the object is created. In Chimera’s model, the local type modification makes sense: the system is creating yet another view of the data. Nevertheless, outside Chimera, this approach adds complexity when reasoning about the upgrade, rendering the upgrade system error prone and vulnerable to data corruption.

3.4 Lazy Modular Upgrades

In their work, Boyapati et al [7] describe how an upgrade system based on transform functions can provide good semantics that let programmers reason about the transform functions locally while running upgrades efficiently, both in space and time.

The authors assume that applications access objects within atomic transactions. A transaction terminates either by committing or aborting. If the commit succeeds, changes become persistent. If instead the transaction aborts, none of its changes affect the objects. The assumed system model is very similar to what transactional memories provide, as I describe in Section 3.6. Upgrades are also performed inside their own transaction. When receiving an upgrade, the system checks it for consistency and, if it is consistent, installs it.

To run the upgrades efficiently in time, the authors suggest running transform functions lazily. Multithreaded environments are also considered and the authors define three conditions that the upgrade system must enforce when ordering upgrade and application transactions. These conditions guarantee the modularity of the upgrades given the lazy execution of transform functions, i.e., transform functions encounter only the interfaces that existed at the time the upgrade was installed. The problem showed in Figure 2 is solved by these conditions.

Finally, the authors discuss how such upgrade system can run upgrades efficiently in space by taking advantage of the modularity of the applications, avoiding versioning unless strictly necessary. However in my work I will combine the upgrade system with a transactional memory that already uses versioning [10].

Converting the program state This upgrade system supports custom instance conversion only, using transform functions. A transform function converts the instances from the old representation to the new representation introduced by an upgrade.

The semantics of such transform functions is simple and meant to keep it local to the class being upgraded. The programmer must write them as additional
methods of each of the old classes, considering the same assumptions he did to write the old methods. Thus, a transform function accesses only the old version of the object and the old version of other instances modified by the upgrade.

**Partial Upgrades** Boyapati et al define an upgrade as a set of one or more class-upgrades. A class-upgrade identifies a class that will be upgraded, couples it with the new class that will replace it and a transform function for instances of that class. Thus, a class-upgrade is a tuple <Old-class, New-class, TF>.

The upgrade system allows arbitrary modifications to the program. As a consequence, an upgrade may change the interface of a given class incompatibly (for instance, by deleting a method or changing the superclass/implemented interfaces). To maintain the program correctness, all classes that are affected by such incompatible modification must be upgraded as well. They define a class as affected if it depends on properties of an upgraded object that are not supported by the new type.

A complete upgrade contains class upgrades for all the classes that need to change due to some class-upgrade already in the upgrade. The proposed upgrade system accepts only complete upgrades. Once accepted, the upgrade becomes installed. Moreover, conceptually, an installed upgrade has already converted all the objects it modifies.

**Executing Upgrades** This upgrade system supports lazy upgrades without any restriction and works in a multithreaded environment. It also achieves modularity, ensuring that the custom conversion code encounters the expected interface before converting any object.

To better understand the modularity problem, please recall the example illustrated in Figure 2. To avoid such error to occur, the authors define three conditions that must be enforced in order to guarantee the modularity property.

1. All application transactions that are serialized after an upgrade is installed must use the new program and the converted program version. Upgrade systems that support offline or immediate upgrades trivially ensure this property, but upgrade systems that support lazy upgrades must ensure this condition when it runs an application transaction before a transform function.

2. In situations similar to the example given, where a transform function $T(y)$ uses object $x$ and runs after $T(x)$, this ordering must have the same effect as running $T(x)$ after $T(y)$. A straightforward implementation could enforce that $T(y)$ has access only to the previous version of $x$, thus verifying this condition. This can be achieved using versions or controlling the order of transforms.

3. For unrelated objects, the behaviour of the application must be independent of the order in which their transformation occurs. The upgrade system is free to choose the order it wants for the two transforms.

An upgrade system that enforces these conditions can use lazy upgrades in multithreaded environments and still provide the transform functions with strong
semantics about the execution of such conversion code. The formal definition of such conditions can be found in [7]. The application always operates using the last upgrade installed. Since lazy upgrades are supported, the upgrade system also keeps old versions of the instances until it detects an attempt to manipulate them, converting them just in time. But the combination of lazy upgrades with data dependence from several instances can result in the need of keeping old versions of already converted instances. Recall the example presented in Figure 2. After upgrading \( x \), the upgrade system had to keep a copy of the old version to provide it to \( TF(y) \).

The authors argue that this approach is not space efficient and try to take advantage of a good practice of object oriented programming: object encapsulation. In the example, if \( y \) encapsulates \( x \) then the application must always convert \( y \) before \( x \), because there is no other way to access \( x \) without accessing \( y \). Nevertheless, this is not always possible and, in the worst case, they fall back to versions. Using versions, we could provide \( TF(y) \) with the previous version of \( x \).

Please note that, to avoid data corruption, the old versions of converted objects must become immutable after installing an upgrade.

3.5 Programming Languages Supporting Dynamic Code Changes

There are some programming languages that provide mechanisms to support dynamic code changes, such as Common LISP Object System (CLOS) and Smalltalk. Besides these languages, there is also a proposal to an extension to the Java programming language that enables dynamic code changes: UpgradeJ [6].

CLOS has an interesting set of mechanisms that I will describe on this section. I will also present UpgradeJ, since it is meant for Java.

**CLOS** The Common Lisp Object System (CLOS) is an object-oriented extension to Common Lisp. The full reference of LISP and CLOS can be found in [24]. Its dynamic upgrade features can be briefly described as follows:

**Partial Upgrades** Upgrades are defined at the class level. The programmer upgrades a class by redefining it. CLOS detects this situation and converts all instances of the class (and all instances of subclasses of the class) without creating new instances nor changing their identity. All the class upgrade mechanism is defined in LISP, as well as CLOS. Thus, the partial upgrades are supported at the programming language level. Nevertheless, CLOS detects which slots were added and which slots were deleted, performing some program comparison as well.

**Converting the Program State** When a programmer defines a class (a completely new class or a redefinition of an existing class) or when he creates new instances, he may also define an initialization for each slot. This automatic initialization can be used when converting instances of redefined classes. CLOS also allows custom instance conversion, giving the programmer access to the: instance featuring the new structure, the added slots, the deleted ones and their respective value.
Executing Upgrades The exact moment when instances are converted is left
implementation dependent. The standard states only that converting any
instance occurs no later than the next time a slot of that instance is read or
written. An implementation of CLOS is free to choose between immediate
and lazy upgrades. Allowing lazy upgrades introduces some problems. For
instance, consider a class \( C_0 \) that is redefined twice, \( C_1 \) and \( C_2 \). Some of
the existing instances may keep the structure defined by \( C_0 \) even after \( C_2 \) has
been defined. The programmer must write intermediate conversion code that
deal with this problem. As for multithreaded environments, the standard
does not discuss them.

UpgradeJ The UpgradeJ extension provides language support for upgrading
a Java application in a variety of ways, while checking upgrades for consistency
with the running program:

Partial Upgrades In UpgradeJ, upgrades are defined at the class level. Three
kinds of upgrades are supported, all at runtime:

- **New Class Upgrades** Add a new class definition to the program
- **Revision Upgrades** Change the method bodies of a class. The signature
  of all methods must remain the same and the revision class must have
  the same name and implement the same interfaces as the revised one.
- **Evolution Upgrades** Add methods and/or fields to a class. The evolution
  must keep all the existing methods and fields, and must it also have the
  same name as the evolved class.

Any class may have one revision and one evolution, and only one (a revi-
sion/evolution also is a class and can be revised/evolved). A programmer
can define instances as upgradable. Such instances use the latest revision
upgrades of their class automatically. When creating instances of a class, a
programmer may ask for the latest revision of the latest evolution of that
class. This way, the program can use the evolution upgrades. As for hierarchy,
all upgradable instances also see revisions made to their superclasses.

Converting the Program State UpgradeJ does not convert any program state
at all. It is focused on adding new classes to the application and performing
minor and major upgrades to existing classes. The program has access to
all the versions of a class since they were introduced. Combining that with
the required explicit version control from the programmer, UpgradeJ can
avoid upgrading instances and still achieve type correctness. Nevertheless,
requiring the programmer to use explicit versions may pose a big burden.

Executing Upgrades Although UpgradeJ does not convert any program state,
it may change the behaviour of upgradable instances. Such change is intro-
duced on all upgradable instances instantly. Thus, we can say that UpgradeJ
features instantly applied immediate upgrades.

3.6 Software Transactional Memories

In the Conceptual Model, I assumed that a Software Transactional Memory
(STM) is used to control the concurrent access to the program state shared by
all the threads. Using this technology, we can achieve data synchronization without resorting to locks: an error-prone mechanism which can easily lead to data corruption. Using STMs, the programmer can group several operations with a new language construct: a transaction. This means that the execution of such operations will appear as an atomic operation to the program state. Moreover, each single operation will execute on a consistent view of the program state, even if that state was written by other operations/transactions. If a transaction has seen a consistent view of the shared program state, it can commit and its changes can be written to the shared program state, becoming visible to other transactions. On the other hand, if other transaction has changed the shared program state so that the view of the committing transaction is not consistent anymore, the committing transaction must be rolled back and retried (or aborted).

**JVSTM** The STM that the Fénix Framework uses is JVSTM [10]. It is based on the concept of versioned boxes: containers that keep a sequence of values — the history of the versioned box. Each of the history’s value corresponds to a change made to the box by a successfully committed transaction. JVSTM checks for conflicts only at commit time.

A transaction is associated with exactly one thread and lasts until that thread either aborts or commits the transaction. Each transaction keeps the following information:

- A transaction number
- A set of boxes that were read in the context of the transaction
- A local-values map, that maps boxes to values

At commit time, JVSTM checks if a committing transaction $T$ conflicts with any other. According to the behaviour of $T$, it can be classified in:

**Read-only** No boxes were changed, and all boxes read returned the value they had when $T$ started. Thus, $T$ has seen a consistent view of the data and can always be safely commit. To the rest of the program, $T$ behaves as it executed instantly when it started.

**Read-write** JVSTM system must ensure that none of the boxes that $T$ read changed after it started. In this case, we say that $T$ is valid.

**Write-only** This type of transaction is always valid because $T$ have not read any box.

After a successful commit, $T$’s number is increased by one and the value of each modified box is added to the corresponding box. Only at this point the boxes are changed. Before this point, the modification was saved as a mapping from the box to the new value.
3.7 Java Implementation Mechanisms

The Java environment, consisting of the Java programming language [18] and the Java Virtual Machine (JVM) [22], does not provide native support for upgrading applications dynamically. Nevertheless, this environment has several features that can be used to implement an upgrade system without changing the Java programming language nor the JVM. I will discuss such features in this section.

**Classloaders** Java supports dynamic class loading using classloaders. In [21], the authors describe the dynamic class loading features of Java. A classloader is an ordinary object that is instance of subclasses of the class `java.lang.ClassLoader`. Assuming that a class $C$ is loaded by a classloader $L$, we say that $L$ is the **defining loader** of $C$. $L$ will also be used to load all classes referenced by $C$. These references are actually symbolic references that $C$ has to other classes, which are resolved at link time to actual classes.

Please note that classes do not have to be stored in actual files. They can be stored on memory buffers or obtained from a network stream. A classloader must be able to obtain a byte array given a binary name, as defined by [18].

Consider the example show in Figure 3, related to a Java enabled web browser. It illustrates the use of several classloaders. The system classes are loaded by a system classloader, directly supported by the Java Virtual Machine (JVM). The arrows indicate delegation: a classloader $L_1$ may ask another classloader $L_2$ to load a class $C$ on its behalf. In this example, applets and browser classloaders delegate the loading of system classes to the system classloader.

The use of delegating classloaders has several advantages:

- In Java, a class type is uniquely determined by the combination of the class name and the classloader that loaded that class. This guarantees that all system classes are unique
- Two classes with the same name loaded by different classloaders are considered different types. Thus, classes loaded by one applet are unable to interfere with classes loaded by another applet.
This technology can be used to implement the proposed upgrade system for the Fénix Framework. Nevertheless, there are some issues identified in [21] for which the Java classloaders do not provide any direct solution:

- Migrating existing instances from one class version to another
- Map static field values from one class version to another
- The application may be executing a method that belongs to a class we want to reload

Java Platform Debugger Architecture The Java Platform Debugger Architecture (JPDA)[2] provides the infrastructure needed to build end-user debugger applications for the Java Platform. The version 1.4 of the Java SDK included two JPDA enhancements, which may be useful to implement an upgrade system for long running applications:

”HotSwap” Class File Replacement This feature encapsulates the ability to substitute modified code in a running application through the defined APIs. It allows a class to be updated while under the control of a JPDA debugger. Although it may support arbitrary changes to a class, currently the new version of the class must maintain the same signature as the old version. Only method bodies can change.

Full Speed Debugging In previous versions, when debugging was enabled, the program executed using only the interpreter. Now, the full performance advantage of the JVM is available to programs running with debugging enabled.

Using these two features, we could define an upgrade system as a debugger. At first glance, this approach may appear too restrictive, defeating the flexibility goal. However, it is possible to overcome this constraint using binary rewriting [20]. Through automated tools that change the bytecode as it is loaded by the JVM [13, 9], we can introduce a new level of indirection without changing the program semantics nor introducing significant overhead. The upgrading process can be explained by the example shown in Figure 4. On version 1, upgradeable class A is divided by the upgrade system into virtual superclass and proxy. The actual implementation of method foo is defined in the superclass. All the classes that refer to A in the source code (A’s clients) will refer to the proxy. Thus, we can say that the proxy keeps the identity of each instance.

On version 2, a new method bar is inserted into the source code. The upgrade system will create a new HelperClass object that will contain the implementation of bar. The superclass has access to all helper objects. Besides, the proxy was featured with an invoke method that is able to invoke methods over the helper objects. All invocations of bar present in A’s client classes are replaced by an invocation to A’s invoke method, specifying bar as the target of the invocation. Helper objects can also be used to add new fields. In this case, we can choose from:

- Use a new helper object to contain all fields introduced by each new version
- Use a single class that contains a mapping data structure that represents all the added fields
4 Proposed Solution Overview

At this point, I cannot compromise my work to any final decision. After studying all the related work, I can define the semantics of the upgrade system that I propose. But I will leave open all the questions related to specific implementation dependencies.

4.1 Conceptual Model instantiation

In the Conceptual Model section, I defined an abstract model that allowed us to understand the problem and reason about the possible solutions. To start defining the architecture of the solution that I propose, I will start by instantiating the abstract conceptual model to the specific problem that I aim to solve:

Execution Platform Composed by the Operating System, the Java Virtual Machine, a container for the application (Tomcat, for instance), Fénix Framework and the libraries it uses (including JVSTM)

Program The application itself, running on top of the Execution Platform

Local Program State The thread’s stack, the data kept in order to maintain user sessions (using the container) and the transaction related data
Shared Program State The data kept in the heap of the application. This data is accessible through a Root Domain Object, that aggregates all accessible objects and allows us to navigate through them.

Stimulus/Response HTTP request, intercepted by the container that runs the program to produce an HTTP response to send to the client.

Classification Throughout the related work, I used the concepts that I presented to classify the existing upgrade systems. In this section, I will classify the upgrade system that I propose using those same concepts.

Partial Upgrades All the implementation mechanisms available for the Java environment use a class as the upgrade unit. The upgrade system I will develop is bound to this constraint as well. It will only accept complete upgrades, like defined in Section 3.4, thus achieving the robustness goal.

Partial Upgrades All the implementation mechanisms available for the Java environment use a class as the upgrade unit. The upgrade system I will develop is bound to this constraint as well. It will only accept complete upgrades, like defined in Section 3.4, thus achieving the robustness goal.

Any class belonging to the application may be upgraded at any instant. I will not consider classes belonging to the execution platform. This way, I achieve the flexibility and timing goals.

As for the technology for defining the upgrades, I have showed that program comparison was too restrictive for certain kinds of upgrades. Nevertheless, I do not yet have a final decision about the technology that I will use. The options I am considering are external or internal DSLs, the latter in the form of a class that describes the upgrade process in a programmatic way.

Converting the Program State The upgrade system will convert the old state using a mechanism similar to transform functions, like in Section 3.4, or bulk conversion, like in Section 3.2. These functions will be written in Java and will be able to perform arbitrary computation. Although these functions will be defined inside a class, they will be able to access objects belonging to other classes. The modular conditions defined in Section 3.4 will be enforced by the upgrade system. This fully custom approach is the main mechanism of program state conversion. In the whole program, only these upgrade functions are aware of the upgrade process, dealing with two versions of the program and its state. Limiting such complex code to appear only in the upgrade functions, which have well defined semantics, allows the upgrade system to achieve the ease of use goal.

As for automatic conversion, the upgrade system that I propose will use Java’s automatic initialization of new fields, introduced by upgrades. Other automatic conversions are also possible, such as widening reference conversion as defined in [18]. Nevertheless, I consider such additional automatic conversions as an optional feature.

Executing Upgrades The upgrade system will convert the program state lazily, thus avoiding most of the introduced overhead. This way, the upgrade system achieves the low overhead goal. It will also operate on highly concurrent environments, supporting multithreading without crashing the application nor corrupting data, thus achieving the multithreaded environments goal. The application will always work with the latest installed upgrade. Since it will use a lazy upgrade mechanism, the upgrade system must keep copies of...
objects that are yet to be upgraded. Actually, JVSTM already keeps several copies of the objects through the versioned boxes. Since my work will extend JVSTM, I can take advantage of such feature.

4.2 Upgrade Examples

Most of the behaviour that the upgrade system must implement in order to support the features that I described can be illustrated with simple examples.

Fig. 5. Two examples of upgrades supported by the proposed upgrade system

**Simple Upgrades** The example shown on the left of Figure 5 illustrates a simple scenario. An upgrade that changes class $X$ is installed. Then, a transaction $A$ tries to use a not yet converted instance $x$ of class $X$. The upgrade system freezes transaction $A$, converts instance $x$ through the execution of the transform function defined in class $X$ for this upgrade — $TF(x)$ and finally returns, allowing $A$ to resume and commit. Consider that the execution of $TF(x)$ modifies a value that $A$ previously read. Such conflict would be detected on commit time, and $A$ would be retried with the new version of $x$ from the start.

**Cascading Upgrades** In the scenario shown on the right of Figure 5, two upgrades for class $X$ are installed. After that, transaction $A$ tries to access an instance $x$ of class $X$ that was not converted by any of the upgrades. The upgrade system will behave as expected, converting $x$ according to $TF1(x)$, defined in the first upgrade, then converting $x$ according to $TF2(x)$, defined in the second upgrade, and finally resuming transaction $A$.

**Complex Upgrades** If we recall the problem shown in Figure 2, previously discussed, we find a simple solution through the usage of versions. When transaction $TF(y)$ tries to use $y$, the upgrade system provides $TF(y)$ with the previous version of $x$. This way, $TF(y)$ does not find any unexpected interfaces and $y$ is properly converted, without crashing the application.
5 Evaluation

The proposed upgrade system will be integrated with the Fénix Framework project. It already features a versioned STM: JVSTM. The novel aspect of my work will be extending JVSTM to deal with program code the same way it deals with program data.

Several commercial benchmarks have been ported from other systems to Fénix Framework:

**Apache Daytrader** [1] DayTrader is a benchmark application built around the paradigm of an online stock trading system.


**TPC-W** [23] This benchmark is based on a business model that employs a shopping scenario typical of an online bookstore.

The benchmarks presented in this section allow the testing of the following goals:

**Low Overhead** The overhead introduced can be accurately measured.

**Robustness** A crash induced by the upgrade system can be detected running these benchmarks for a considerable amount of time.

**Flexibility, Timing and Multithreaded Environments** An highly concurrent workload can be used to upgrade any part of the program, even if that part is being executed by some threads when the upgrade is installed.

As for the remaining goals, they are related with static parts of the upgrade system and cannot be easily tested using these benchmarks.

6 Conclusion

This document introduced the problem of atomic dynamic upgrades. Dynamic upgrades consist on upgrading an application without stopping it. Performing such task on multithreaded environments requires that this action appears atomic to the application. After describing the problem using an abstract conceptual model, some systems systems that already implement dynamic upgrades were presented. After that, a solution in which the upgrade system will extend the JVSTM existing in Fénix Framework was proposed, allowing it to deal with code changes on a multithreaded environment in an atomic fashion. Then, several goals that such upgrade system must achieve were established and 3 benchmarks that allow us to test the system against such goals were presented.

The novel feature of the proposed upgrade system is to use a STM to deal with Dynamic Software Upgrades. This way of dealing with the program code as we deal with the program data has an important advantage: the ability to upgrade the program in an atomic fashion, even in a multithreaded environment.
References


