Object Versioning for the Lively Kernel

Preserving Access to Previous System States in an Object-oriented Programming System

by

Lauritz Thamsen

A thesis submitted to the
Hasso-Plattner-Institute for Software Systems Engineering
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in Software Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Bastian Steinert

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

May 13, 2014

Abstract

In programming systems such as the Lively Kernel, programmers construct applications from objects. Dedicated tools allow them to manipulate the state and behavior of objects at runtime. Programmers are encouraged to make changes directly and receive immediate feedback on their actions.

When programmers, however, make mistakes in such programming systems, they need to undo the effects of their actions. Programmers either have to edit objects manually or re-load parts of their applications. Moreover, changes can spread across many objects. As a result, recovering previous states is often error-prone and time-consuming.

This thesis introduces an approach to object versioning for systems like the Lively Kernel. Access to previous versions of objects is preserved using *version-aware references*. These references can be resolved to multiple versions of objects and, thereby, allow re-establishing preserved states of the system.

This thesis presents a design based on proxies and an implementation in JavaScript. The evaluation of this implementation shows that the Lively Kernel can run with our version-aware references and that preserved system states can be re-established. While the memory overhead of the version-aware references is reasonable, the execution overhead is not yet practical. However, with performance improvements, the solution could be used to provide practical recovery support to programmers.

Zusammenfassung

Objekt Versionierung für den Lively Kernel: Erhaltung vorheriger Systemzustände in einem objekt-orientierten Programmiersystem

In Programmiersystemen wie dem Lively Kernel können Programmierer Anwendungen aus Objekten erstellen. Dabei erlauben dedizierte Werkzeuge den Zustand und das Verhalten von Objekten zur Laufzeit zu verändern. Programmierer werden ermutigt Änderungen direkt zu machen und erhalten umgehend Feedback.

Wenn Programmierer in solchen Programmiersystemen jedoch Fehler machen, müssen sie Änderungen rückgängig machen. Dazu müssen sie entweder Objekte manuell bearbeiten oder Teile ihrer Anwendungen neu laden. Die gemachten Änderungen können dabei über viele Objekte verteilt sein. Vorherige Zustände wiederherzustellen ist deshalb häufig schwierig und zeitaufwendig.

Diese Arbeit stellt einen Ansatz für die Versionierung von Objekten in Systemen wie dem Lively Kernel vor. Der Ansatz basiert auf *versionsbewusste Referenzen*. Diese können zu mehreren Versionen von Objekten aufgelöst werden und erlauben so vorherige Systemzustände wiederherzustellen.

Die Arbeit beschreibt einen auf Proxys basierenden Entwurf und eine Implementierung in JavaScript. Die Evaluierung der Implementierung zeigt, dass Systemzustände des Lively Kernels damit erhalten und wiederhergestellt werden können. Der zusätzlich nötige Arbeitsspeicher für die versionsbewussten Referenzen ist dabei vertretbar, während die Programmausführung erheblich verlangsamt wird. Mit Verbesserungen könnte die vorgestellte Lösung allerdings benutzt werden, um Entwickler mit praktikablen Wiederherstellungs-Werkzeugen zu unterstützen.

Acknowledgments

I am grateful to my supervisors Bastian Steinert and Professor Robert Hirschfeld for their guidance and support.

I am thankful to Jens Lincke, Marcel Taeumel, Tobias Pape, Tim Felgentreff, Michael Perscheid, Stephanie Platz, Carl Friedrich Bolz, Robert Krahn, and Dan Ingalls for fruitful discussions and valuable input.

I thank Oliver Buck for his comments on drafts of this thesis.

Contents

1 Introduction **1**
1.1 Contributions . . . 3
1.2 Thesis Structure . . . 4

2 Background **5**
2.1 Prototype-based Programming . . . 5
2.2 The Lively Kernel . . . 6
2.3 CoExist . . . 9

3 Motivation **13**
3.1 Part Development By Example . . . 13
3.2 Recovery Needs When Developing Parts . . . 15

4 Object Versioning **19**
4.1 Version-aware References . . . 19
4.2 Using Proxies as Version-aware References . . . 24

5 Implementation **31**
5.1 Using the ECMAScript 6 Proxies as Version-aware References . . . 31
5.1.1 ECMAScript 6 Proxies by Example . . . 31
5.1.2 Using the Proxies for Object Versioning . . . 34
5.2 Accessing All Mutable JavaScript Objects Through Proxies . . . 37
5.2.1 Transforming Literal Expressions . . . 38
5.2.2 Returning Proxies from Constructor Functions . . . 40
5.2.3 Wrapping Built-in Functions Into Proxies . . . 41
5.3 Workarounds for the Current State of the ECMAScript 6 Proxies . . . 44
5.3.1 Disabling Target Object Invariants . . . 45
5.3.2 Forwarding the *Instanceof* Operator . . . 46
5.3.3 Unwrapping Versions for Native Code . . . 47
5.4 Limitations of the Implementation . . . 48

6 Evaluation **51**
6.1 Test Setup . . . 51
6.1.1 Octane Benchmark Suite . . . 51
6.1.2 Lively Kernel . . . 52
6.1.3 Machine Configuration . . . 52

6.2 Functionality of Version-aware References . . . 53
6.2.1 Testing with Benchmarks . . . 53
6.2.2 Testing with the Lively Kernel . . . 54
6.3 Practicability: Memory Consumption . . . 55
6.3.1 Memory Overhead of Version-aware References . . . 55
6.3.2 Memory Consumption When Preserving Versions . . . 56
6.4 Practicability: Impact on Execution Speed . . . 58
6.4.1 Execution of Benchmarks . . . 58
6.4.2 Execution of the Lively Kernel . . . 60
6.4.3 Discussion of the Execution Overhead . . . 62

7 Related Work **63**
7.1 Recovering Previous System States . . . 63
7.1.1 CoExist . . . 63
7.1.2 Lively Kernel Offline Worlds . . . 64
7.1.3 Back-in-Time Debugging . . . 64
7.1.4 Software Transactional Memory . . . 65
7.2 Dynamically Scoping First-class Groups of Changes . . . 65
7.2.1 Worlds . . . 66
7.2.2 Object Graph Versioning . . . 66
7.2.3 Context-oriented Programming . . . 67
7.2.4 ChangeBoxes . . . 67
7.2.5 Practical Object-oriented Back-in-Time Debugging . . . 68

8 Future Work **69**
8.1 Improving the Performance . . . 69
8.2 Providing Recovery Tools . . . 72
8.2.1 Preserving Versions Automatically . . . 72
8.2.2 Tools For Finding and Managing Versions . . . 73

9 Summary **75**

List of Figures

2.1 The halo buttons of a basic morph. . . . 7
2.2 Three Lively Kernel's tools to manipulate morphs: the Inspector, the Style Editor, and the Object Editor. . . . 8
2.3 The Lively Kernel's Parts Bin opened on the *Tools* category. . . . 9
2.4 CoExist's tools to manage the preserved development states: the *Timeline* and the *Version Browser*. . . . 10

3.1 The Object Editor's magnifier button highlighted with a red outline. . . . 13
3.2 The Object Editor's magnifier button as it highlights the editor's target. . 14
3.3 Directly manipulating a button morph. . . . 15
3.4 The button's `onMouseMove` script with a text selection. . . . 17
3.5 Adding a submorph changes the state of a morph. . . . 17

4.1 An `address` object with three properties. . . . 19
4.2 Two versions of an `address` object in two versions of the system. . . . 20
4.3 Preserving the previous version of the `address` object. . . . 20
4.4 A reference refers to the previous version of the `address` object. . . . 21
4.5 A version-aware reference relates a `person` object to two versions of its `address` property. . . . 22
4.6 An object graph with version-aware references. . . . 23
4.7 Using a proxy as version-aware reference to connect a `person` object to two versions of an `address` object. . . . 25
4.8 Four versions of the system. . . . 27
4.9 A proxy with two object versions in context of the system versions. . . . 27
4.10 A new version of an object is created for a new version of the system. . . . 28

5.1 A `client` object has access to a `server` object via a proxy. . . . 32
5.2 A proxy with a handler that forwards to two versions of an `address` object. 34

6.1 Memory consumption when starting a Lively Kernel world with and without proxies. . . . 56
6.2 Memory consumed for three different states when the previous states are preserved in separate versions. . . . 57
6.3 Execution overhead for the Octane benchmark suite. . . . 59
6.4 Execution overhead for three user interactions in the Lively Kernel. . . . 61

1 Introduction

Programming systems such as Squeak/Smalltalk [11, 9] and REPLs for LISP or Python allow adapting programs at runtime. Changes to programs in such environments are effective immediately and programmers can see or test right away what differences their actions make. Thus, these systems provide immediate feedback to programmers.
A subset of such systems, which includes, for example, Self [34, 33] and the Lively Kernel [13, 15], are those built around prototype-based object-oriented languages [17]. In prototype-based systems programmers create applications using objects and without having to define classes first. In Self and the Lively Kernel, programmers can inspect and change the state and behavior of objects at runtime. Programmers create actual objects, not source code that only abstractly describes potential objects.

The Lively Kernel was designed to support this kind of development [22]. It provides tools to directly manipulate the style, composition, and scripts of graphical objects. For example, programmers can change the positions and composition of objects directly using the mouse. They can use temporary workspaces to manipulate objects programmatically. They can edit and try methods directly in the context of graphical objects.
For example, to add new functionality to a graphical application, a Lively Kernel user might copy an existing button object and then modify the new button object: move the new button to a sensible position, resize it, set a new label, and add a script to be executed on mouse clicks. The user makes all changes directly to one button object. How this button fits into the application's interface is visible at all time. Clicking the button allows to directly test its functionality. This way, the Lively Kernel allows for fast feedback, especially during the development of graphical applications.

Programmers' changes to objects can turn out to be inappropriate. Programmers can, for example, accidentally change positions or connect the wrong objects when manipulating applications with mouse interactions. They might try a couple of different alternatives such as different colors and layouts, only to realize that an earlier state was most appealing. Similarly, programmers might learn in hindsight that making a change to an object's scripts introduced an error or impacts the application's performance. They might make a mistake in a code snippet, which then manipulates many objects. Moreover, problematic changes can be introduced when code is evaluated only to understand or test behavior, not to permanently change state.

However, when changes turn out to be problematic, programmers often need to undo the changes manually. The Lively Kernel does not provide an undo for changes to objects. This is especially at odds with the Lively Kernel's support for trying ideas right away: Developers are able to make changes directly and receive immediate feedback, but do not get support when such changes turn out to be inappropriate. Thus, to recover a previous development state, programmers often need to manually reset the state to how it previously was—probably using the same tools the changes were initially made with. Furthermore, this potentially involves multiple properties of multiple objects changed by multiple developer actions.
The Lively Kernel provides tools to commit and load versions of objects. In case such commits exist, programmers can load earlier versions of objects to re-establish previous states. Nevertheless, depending on how far the latest version is from the actually desired state, manual changes might still be necessary. To keep the effort to re-establish *any* previous state low, programmers would need to commit many versions. However, this contradicts the goal as commiting many versions is also a significant effort. Some commits would be made only to protect intermediate states, not to share and document results. Especially when the preserved versions should be usable and documented, programmers would be required to test and describe many versions.
In summary, recovering previous states of objects in the Lively Kernel is currently a significant effort for programmers. They either have to manually re-set changed state or need to take time-consuming precautionary actions.

A typical approach to implementing multi-level undo for the changes to application state is the Command pattern [8]. The Command pattern packages changes into actions. These actions can then be recorded to be able to subsequentely undo them. This requires developers to implement undos for all possible actions. Therefore, an implementation of the command pattern—even when limited to the Lively Kernel tools that manipulate objects—would be rather comprehensive. Furthermore, using the Command pattern requires developers to follow the pattern when implementing new tools. The Command pattern is entirely impractical for undoing the effects of evaluating arbitrary code from the Lively Kernel's workspaces and editors.

Worlds [37, 36], in contrast, is a more generic approach for controlling the scope of side effects. Code is executed in *world* objects, which capture all side effects. The worlds can then be used to run code with particular sets of changes. Developers could create new worlds for all their actions and discard worlds to return to previous states when necessary. Therefore, it still requires programmers to explicitly take precautionary actions, similar to version control systems. In addition, the implementation of Worlds in JavaScript is not yet practical. For example, it currently prevents garbage collection.

CoExist [28, 29] provides automatic recovery support without requiring developers to take precautionary actions. CoExist automatically records versions for every change and, thereby, provides a fine-grained history of intermediate development states. Programmers

can review the changes chronological, examine the impact each change had, and re-establish previous versions. However, CoExist currently recognizes only changes made to the source code of classes. Its versions do not include the state of objects.

This thesis proposes an approach for versioning the entire state of programming systems as basis for automatic recovery support. In particular, this thesis introduces an approach to preserving and managing versions of all objects using alternative, *version-aware* references. Version-aware references are alternative references as they refer to multiple versions of objects. They resolve transparently to particular versions. Versions of objects are preserved together, so that version-aware references can be resolved transitively to the state of a particular moment. For this to be practical, versions of objects are kept in the application memory and the state of all versions is preserved incrementally on writes. To which versions the version-aware references resolve can be changed without significantly interrupting program execution: The version-aware references select the current versions dynamically instead of being hard-wired to specific versions.

We implemented our approach in JavaScript. The implementation does not require adaptions to established execution engines. Proxies are used to implement version-aware references: conventional references point to the proxies and the proxies delegate all object interactions transparently to particular versions of objects. Source transformations introduce proxies consistently for all objects. Therefore, programmers do not need to adapt their programs manually.

The approach supports fine-grained histories of development states. Not every state can be re-established, but versions that have been preserved. In this, the presented solution is a basis for recovery support that continuously preserves versions.

1.1 Contributions

The goal of this work is to provide object versioning for the Lively Kernel. To that effect, the main contributions of this thesis are the following:

- An approach to object versioning for systems like the Lively Kernel based on version-aware references that transparently delegate to one of multiple versions of an object (Section 4.1).
- A design that provides the proposed version-aware references through proxies for the Lively Kernel (Section 4.2).
- An implementation of the design in JavaScript that can be used to effectively preserve and re-establish development states of the Lively Kernel (Section 5).

1.2 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 describes prototype-based programming systems, CoExist, and the Lively Kernel. Chapter 3 illustrates how developers directly manipulate objects in the Lively Kernel and exemplifies recovery needs. Chapter 4 introduces our approach to object versioning and describes how proxies can be used for concrete solutions. Chapter 5 presents our implementation for the Lively Kernel, which Chapter 6 then evaluates in terms of functionality and practicability. Chapter 7 compares our solution to related work. Chapter 8 presents future work, while Chapter 9 concludes this thesis.

2 Background

This chapter describes prototype-based programming, the Lively Kernel, and CoExist. These works are the background of this thesis as we introduce an approach for providing CoExist-like recovery support in prototype-based programming systems, which we implemented for the Lively Kernel.

2.1 Prototype-based Programming

Prototype-based programming is object-oriented programming in which applications are created directly with objects, without requiring developers to define classes first.
Self, JavaScript, and Kevo [32] are prototype-based programming languages. Many end-user programming systems such as Scratch [25], Etoys [14], and Fabrik [12] also enable users to express programs using objects.

Prototype-based programming allows to build applications from particular objects. This is the fundamental difference to the class-based style of object-oriented programming, in which programs are expressed with classes. Each part of a prototype-based program has particular state.

There are different advantages associated with this kind of programming:

- [30] and [17] suggest that it might be easier for programmers to understand concrete examples than to grasp abstract classes. A concrete example provides particular values for its state and, in case of objects with a visual appearance, can be actually looked at.

- [34] and [2] describe how prototype-based programming makes it easier to introduce one-of-a-kind objects with their own structure or behavior.

- [2] and [24] argue that especially editing visual objects can be more concrete with prototypes. Instead of writing code that describes the appearance of objects, programmers can manipulate visual objects directly. Programmers could, for example, use the mouse to manipulate properties like the size, position, or to combine multi-

ple elements. This way, programmers always see intermediate states instead of only receiving feedback on explicit test runs in-between edit-compile-load cycles.

Editing Graphical Objects at Runtime Many prototype-based programming systems, including the examples given in this section, allow to manipulate objects at runtime. Scratch, Etoys, Fabrik, the Lively kernel, and Self all provide tools dedicated to manipulating graphical objects directly. Such graphical objects range from basic objects like primitive shapes to complete applications like presentation software or programming tools. Prototype-based programming, programming at runtime, and direct manipulation of graphical objects seem to be properties that suit each other.

Similar Objects Without Classes Different prototype-based programming systems provide different approaches for creating similar objects. Self and JavaScript incorporate *delegation* to allow for prototypical inheritance. Objects can inherit state and behavior directly from other objects: each object has a *prototype* to which it delegates whenever looking up a property in the object itself yields no results. In Self, the prototype of an object is set when objects are cloned: The clone's prototype is the object it was cloned from.
In JavaScript, objects are created from constructor functions. The constructor function's prototype becomes the prototype of created objects.
Kevo, in constrast, does not incorporate this notion of prototypical inheritance. It provides *concatenation* for incremental modification of objects [31]. Objects are copied to create objects with the same state and behavior as existing objects. These objects are self-contained. Changing an object only changes that particular object and a particular object can only be changed by directly changing it, not by changing any other object. To adapt many objects at once, programmers can use so-called *module operations* in Kevo. Module operations are evaluated for groups of objects.

2.2 The Lively Kernel

The Lively Kernel is a programming system in the tradition of Smalltalk and Self. Development happens at runtime. It incorporates tools and techniques to be completely self-sufficient. Thus, programmers can create versions of the Lively Kernel with the Lively Kernel.

The Lively Kernel is a browser-based system. It is implemented in JavaScript and renders to HyperText Markup Language (HTML).

Programming with Prototypes and Classes

As the Lively Kernel is based in JavaScript, the system and applications are expressed in a prototype-based object-oriented language that provides prototypical inheritance. At the same time, the Lively Kernel also provides a class system and considerable parts of the system are expressed using classes.

The Lively Kernel implements Morphic [24], a framework for developing graphical applications. The graphical objects of this framework are called *Morphs*. Each morph has a class but can also have object-specific behavior. They can be created by instantiating a class or by copying an existing morph. Morphs are often edited directly and not through adapting existing or creating new classes. This way, the Lively Kernel mixes the class-based with the prototype-based style of object-oriented programming.

The Lively Kernel's copy operation does not establish a prototypical inheritance relationship between the copy and the original. Instead, it creates a full copy of the original morph's properties, including its class. Therefore, even though JavaScript incorporates prototypical inheritance, the Lively Kernel encourages programmers to use classes to share behavior among objects.

Direct Manipulation of Morphs

Programmers can change the position of morphs by *dragging* and the composition by an alternative dragging, called *grabbing*. When a morph is grabbed, it can be added to another morph and becomes that morph's submorph. This way, a morph does not have to be a basic shapes or simple widgets, but can be the interface of any application.

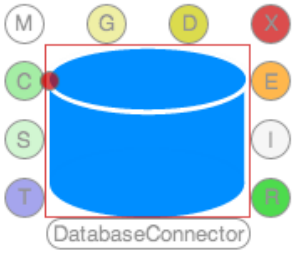

Figure 2.1: The halo buttons of a basic morph.

The Lively Kernel provides a set of manipulation tools, called *Halos*, as shown in Figure 2.1. Developers can bring up these tools for each morph. The different buttons of a morph's halo allow, for example, to resize, rotate, and copy morphs.

Other halo buttons open specific tools, which are shown in Figure 2.2:

1. The *Inspector* ① presents all the values that make up a morph's current state. It also has a small code pane at the bottom that can be used to manipulate the morph's properties programmatically.

2. The *Style Editor* ② allows to manipulate certain aspects of a morph's visual appearance. Programmers can use it to change, for example, a morph's color, border width, or the layout of its submorphs.

3. The *Object Editor* ③ is a tool to edit the object-specific behavior of morphs. It shows all scripts of a particular morph and allows programmers to add, remove, and edit scripts.

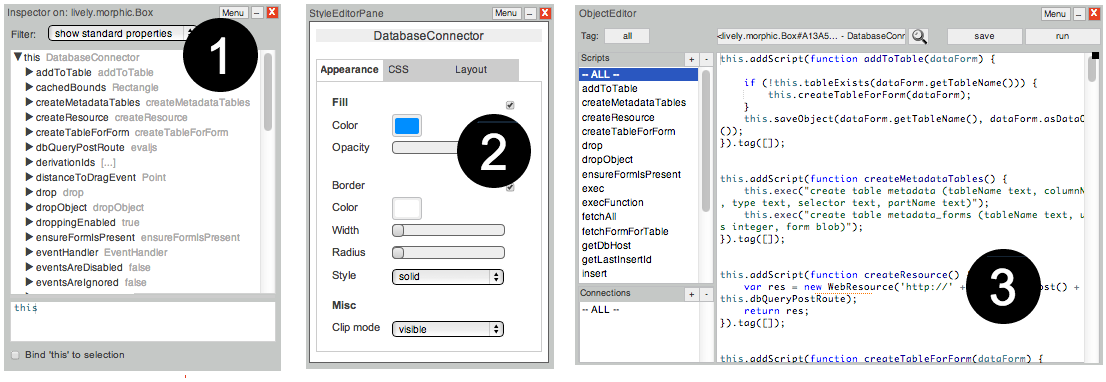

Figure 2.2: Three Lively Kernel's tools to manipulate morphs: the Inspector, the Style Editor, and the Object Editor.

Saving Morphs to the Shared Parts Bin Repository

A related tool is the Lively Kernel's *Parts Bin* [23], an object repository to commit and load specific versions of morphs. Morphs saved to the Parts Bin are called *parts* to emphasize the ability to reuse any of the morphs in the Parts Bin for other morphic applications. Figure 2.3 shows the Parts Bin, opened on the *Tools* category, which includes both the Style Editor and the Object Editor. Both these tools are examples for graphical applications developed from available parts. Their functionality is expressed in scripts and they are available to users through the Parts Bin.

The root of the scenegraph of visible morphs is called *World*. Worlds are not shared via the Parts Bin, but can be saved as a Web pages. A world stores the state of all visible morphs when saved and that state can be reloaded with the world.

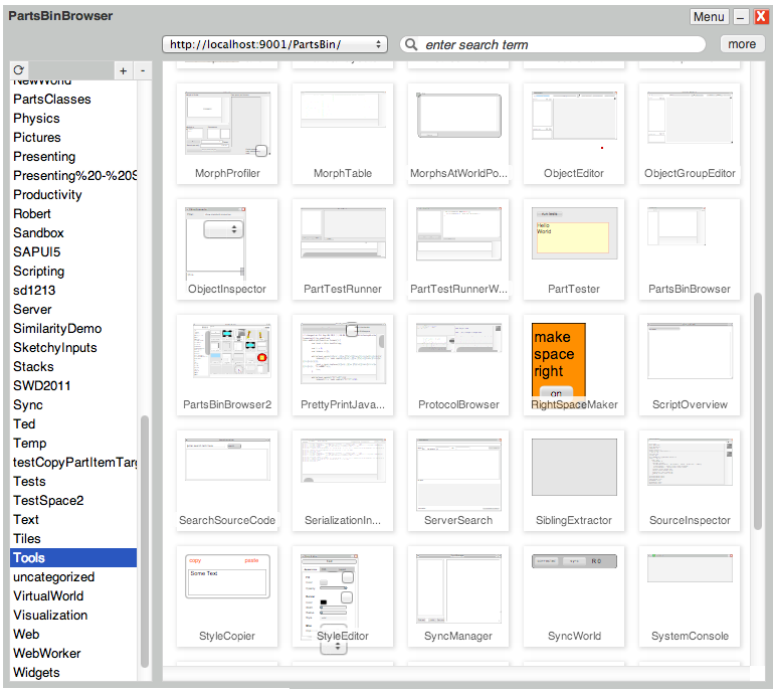

Figure 2.3: The Lively Kernel's Parts Bin opened on the *Tools* category.

2.3 CoExist

CoExist[1] provides recovery support to programmers. It continuously preserves access to intermediate development state. The states are recorded as separate version in their original order. For each version CoExist provides access to diffs, test results, and screenshots of the development environment. Programmers can review their programming sessions, inspect the impact changes had on test cases, and recover information from previous development states.

Tools to Recover Previous Development States

CoExist provides two tools to help programmers benefit from the preserved histories, shown in Figure 2.4.

Timeline CoExist's *Timeline* tool is located at the bottom of the development environment. It shows each intermediate version with a small rectangle. The color of the

[1] http://www.bastiansteinert.org/coexist.html, accessed February 28, 2014

rectangle's four corners indicate test results: the bottom of the rectangle shows how many test cases passed and failed absolutely, while the top highlights how the changes of a version affect test results.
Hovering over a rectangle shows which artifact was changed in the version.
The versions presented in the timeline can also be re-established.

Version Browser Besides the timeline of versions, a *Version Browser* tool provides an overview of the versions. For each package it shows the changes made in that package. It presents the same information on test cases, but also includes a diff view. Moreover, it provides a screenshot of the development environment for each version.

The tools support programmers in re-tracing their steps, understanding the impact of their actions, and in recovering previous development states. They can withdraw changes permanently or recover only specific information from previous versions.

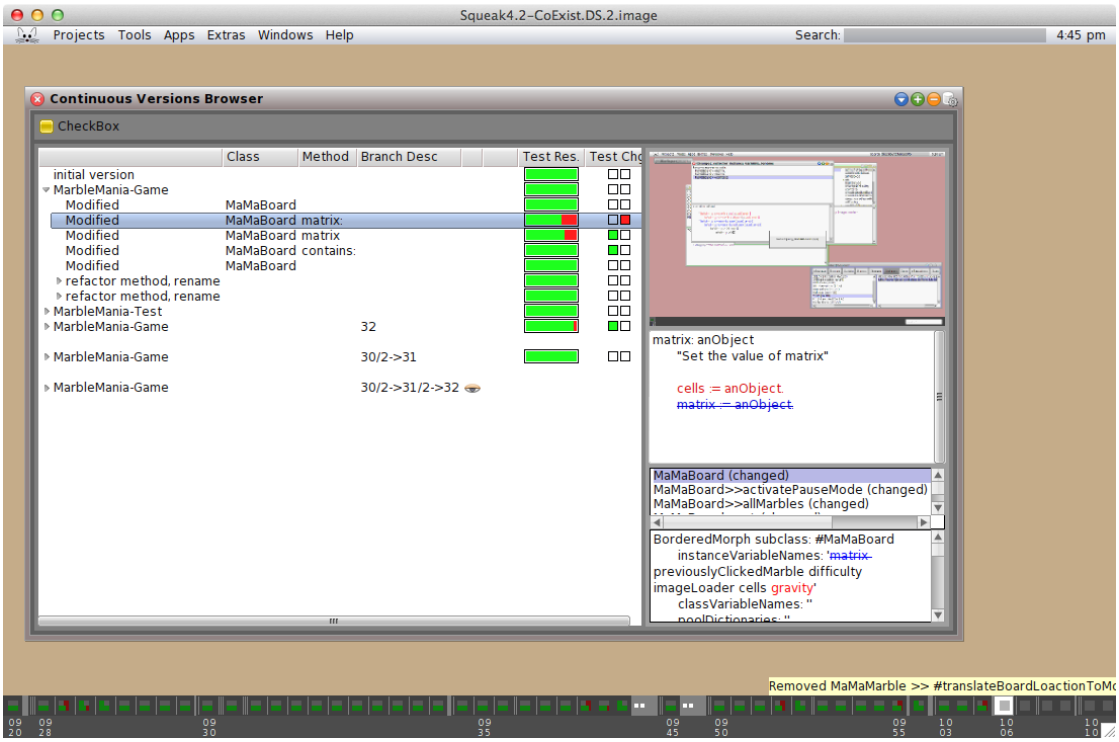

Figure 2.4: CoExist's tools to manage the preserved development states: the *Timeline* and the *Version Browser*.

Benefits of Continuous Versioning

CoExist aims at reducing the effort required to recover previous states and, thereby, at encouraging programmers to explore their ideas through making changes to the code.

Without CoExist, either compensational or precautionary actions are necessary for recovery. When programmers unintentionally introduce errors, decrease performance, or harm the program design, they have to repair changed code. They either need to edit the code again or load a previously commited version of the code. Editing code of potentially many methods across and many classes is obviously a significant effort and error-prone. Loading a previous version is only possible if a version has been commited previously. Therefore, programmers can reduce the cost of recovery by anticipating recovery needs beforehand. However, preserving versions is also an effort and especially so when revision histories are expected to be well-documented and immediately useful. For that, programmers need to assemble changes to meaningful increments, run tests, and write helpful commit messages.

CoExist, in contrast, makes recovery fast and easy. It is similar to the undo/redo of applications. Developers do not have to take explicit precautionary actions, but are still able to undo changes when necessary. However, CoExist provides convenient access to the previous states of entire systems not just to a particular source code view. It presents the preserved versions with additional information. Each version is associated with the static structure of the software system, related to other versions in a timeline, and accompanied by test results. Furthermore, making changes to a previous state in CoExist does not overwrite the history, but creates a branch.

In essence, instead of worrying about negative consequences, programmers can focus on implementing their ideas and rely on CoExist to help in case any action unexpectedly needs to be undone.

3 Motivation

In the Lively Kernel, programmers can create applications by manipulating and composing graphical parts. This chapter presents the development of such parts and related recovery needs by example.

3.1 Part Development By Example

To exemplify how developers work directly on objects in the Lively Kernel, we will outline how a Lively Kernel user adds a new feature to the Object Editor.

The editor has been developed by composing and editing graphical objects. Thus, the user does not adapt any source code modules to change the editor, but rather manipulates objects directly.

In this example, the user adds a magnifier tool to the Object Editor. The magnifier tool helps finding the editor's target, which is the object the editor currently presents scripts for. Implementing the new feature requires to create a new button morph and to add it to the editor, as shown in Figure 3.1.

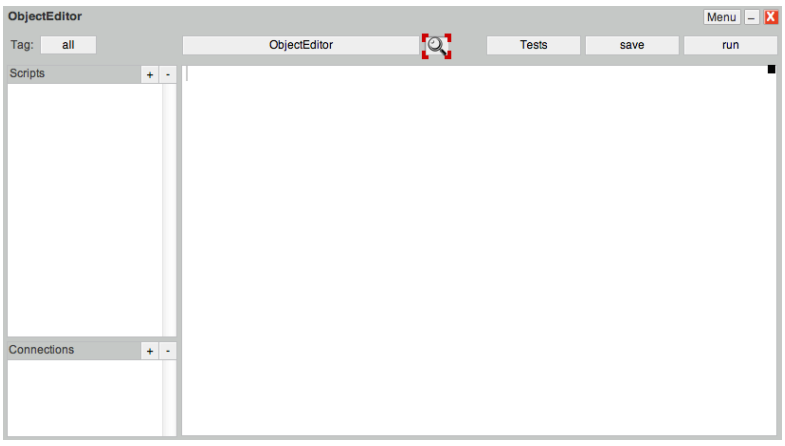

Figure 3.1: The Object Editor's magnifier button highlighted with a red outline.

The magnifier button has two features:

1. When a programmer hovers over the button, the Object Editor's current target is highlighted with a rectangular overlay.

2. When a programmer clicks the button, the current target selection is revoked and the programmer can select the new target of the editor.

The following covers the first of the two features, which is also shown in Figure 3.2 for an Object Editor currently targeting the character of a game.

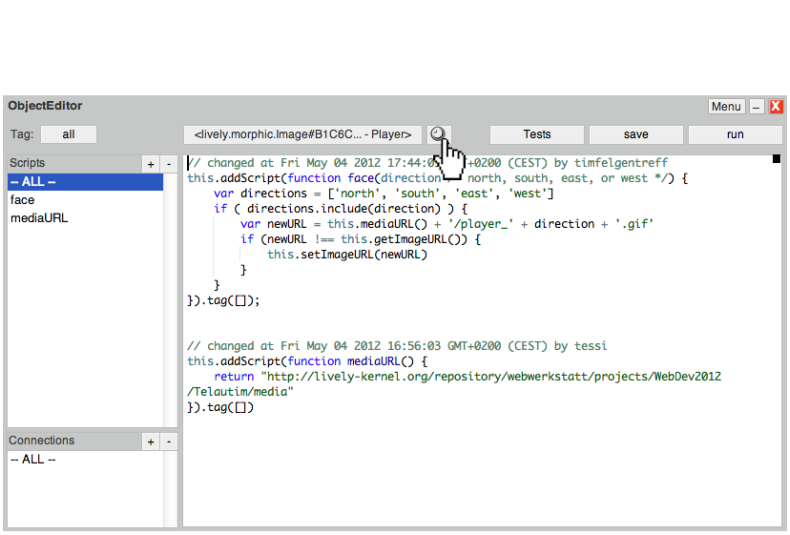

Figure 3.2: The Object Editor's magnifier button as it highlights the editor's target.

Manipulating the Button Morph Before implementing the button's behavior, the user first creates the button and manipulates its visual appearance. Figure 3.3 shows the steps in which the button is manipulated. A basic button, as visible in ①, can be found in the Parts Bin repository. In ②, the user resizes the button and gives it a square extent using the *Resize* halo button. Next, the user loads an image showing a magnifier icon. Using drag and drop, the image is added to the button in ③. Dropping a morph onto another connects the two morphs. Moving the button around will then move the image accordingly. Finally, the users adds the result of these manipulations, visible in ④, to the Object Editor.

All these changes are made directly to the state of objects: the button morph, the magnifier image morph, and the editor morph.
When programmers edit parts in this way, they often see the effects of their actions immediately. For example, when adding the new button to the Object Editor, the button

is visible at all times. Programmers do not need to run any code to see and test the button.

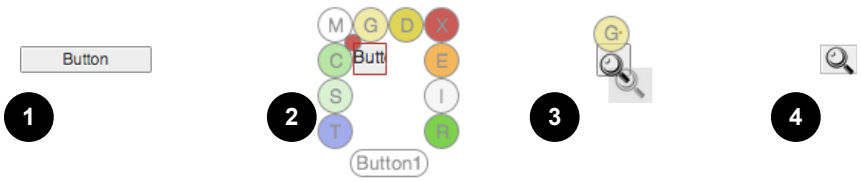

Figure 3.3: Directly manipulating a button morph.

Scripting the Button Morph Now the user implements the button's behavior. The user adds scripts to the button that lay a translucent rectangle over the current target. In particular, the button receives two scripts: `onMouseMove` and `onMouseOut`. The implementation of the behavior includes the following:

- The button holds a semitransparent rectangle morph.
- When the mouse enters the button (`onMouseMove`), the button resizes and adds the rectangle to the Lively Kernel world at the position of the target.
- When the mouse leaves the button (`onMouseOut`), the button removes the rectangle from the world again.

The Lively Kernel's scripting tools allow to evaluate code in the context of their target objects. Hence, when programmers want to test a script or even just specific lines of code, they can try the behavior directly for the actual target.

3.2 Recovery Needs When Developing Parts

While manipulating objects directly, developers might make changes which they later want to undo.

In the previous example the user could make *accidental changes*:

- **Accidental changes to state**: The user could accidentally grap and move a morph such as the new button and, thereby, change a carefully arranged layout. Similarly, meaningful state can be lost when a morph, for example the new button, is accidentally removed from the world.

- **Accidental changes to scripts**: The user could introduce a typographical error to or accidentally remove a script. Moreover, editing a script could introduce a defect or a decrease in performance.

Besides these accidental changes, well-intentioned changes can also turn out to be *inappropriate changes*:

- **Inappropriate changes through direct manipulation**: The user could make changes to the size, position, and colors of morphs to fine-tune the visual appearance of the editor's interface, only to decide later that a particular intermediate version was most appealing.

- **Inappropriate changes through scripts**: The user could make a mistake in a workspace snippet that is intended to manipulate morph properties programmatically. Such a snippet can change many properties of many objects.

Explorative Script Evaluation Undesirable changes can also be introduced when a programmer explores the behavior of objects by evaluating scripts. The Object Editor allows evaluating code directly for its target object. While such evaluation might help to understand the effects of particular code, it might also change the state of objects. For example, the user could be working on the button's `onMouseMove` script and could evaluate a few lines of code to quickly test them. These lines, as shown in Figure 3.4, would add the rectangle to the editor's current target. Only evaluating the selected lines would, however, neither check the conditions usually checked above nor set the state usually set below the selected lines. Therefore, evaluating this selection allows to test the highlighting behavior but leaves the system in a state it normally would not be in.

The examples show that there are many situations in which the user might want to undo previous actions. In programming systems like the Lively Kernel, where programmers work on objects, changes are always made to the state of objects. Functions are properties of objects. Even classes and modules are objects.

For example, evaluating the text selection in Figure 3.4 changes the `world` object's state. The `world` object has now one more submorph, as shown in Figure 3.5. Thus, the `world`'s collection of submorphs is changed.

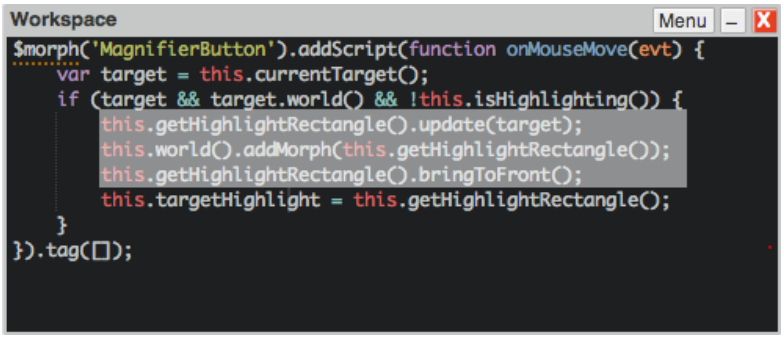

Figure 3.4: The button's `onMouseMove` script with a text selection.

To undo the side-effect of the script and re-establish the previous situation, the change to the `world` object needs to be undone. The `submorphs` property of `world` has to be as it previously was.
When the state of all objects is preserved and can be re-established, previous system states can be recovered when necessary.

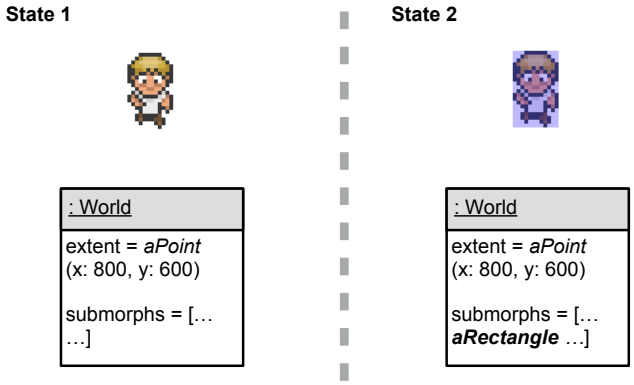

Figure 3.5: Adding a submorph changes the state of a morph.

4 Object Versioning

This chapter introduces our approach to preserving access to previous states in systems like the Lively Kernel. The approach is based on alternative, version-aware references that manage versions of objects transparently.
The chapter also presents a design that allows implementing version-aware references using proxies.

4.1 Version-aware References

In different versions of a system, objects have different states.

Versions of Objects

An object could represent an address. The state of such an *address* object could be as shown in Figure 4.1[1].

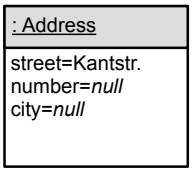

Figure 4.1: An `address` object with three properties.

If values are assigned to the `city` and `number` fields of the `address` object, the object's state is changed. As the `address` object's state is part of the system state, changing the object's state changes the system state as well. If we call the initial state version *v1* and the state after making changes to the object version *v2*, the state of the address object is different in the two versions of the system, as shown in Figure 4.2.

[1] The figures in Chapter 4 and 5 use the notation of Unified Modeling Language (UML) object diagrams. Extensions are explained in the figures.

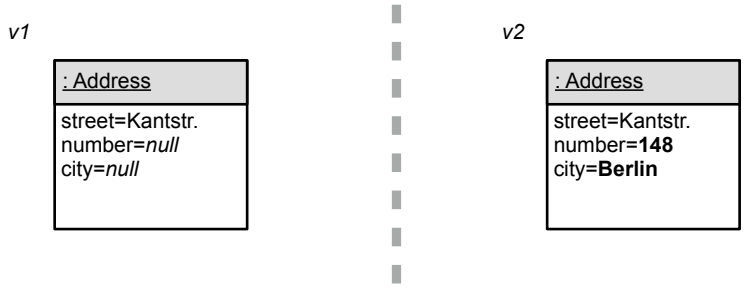

Figure 4.2: Two versions of an `address` object in two versions of the system.

To be able to recover previous versions after making changes, the previous states of objects need to be accessible. For this reason, versions of objects are preserved and changes are made to new versions of the objects. A version of an object is, in the simplest case, a copy of an object. When the `address` object is changed in version *v2* of the system, the system does not change the orginal `address` object but the copy.

As shown in Figure 4.3, there are now two versions of the `address` objects in version *v2* of the system. One of the objects holds the original state, while the other holds the state the object should have in version *v2* of the system. The two objects hold no information that indicates to which version of the system they belong. They also do not store any information showing that one object is a copy of the other.

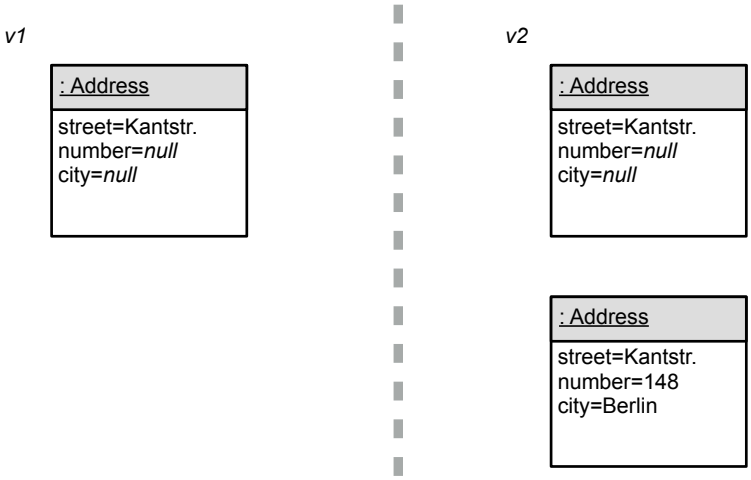

Figure 4.3: Preserving the previous version of the `address` object.

At the same time, references to objects remain unchanged. For example, there could have been a `person` object referring to the `address` object. This reference would still be

referring to the original `address` object, even in version *v2* of the system, as shown in Figure 4.4.

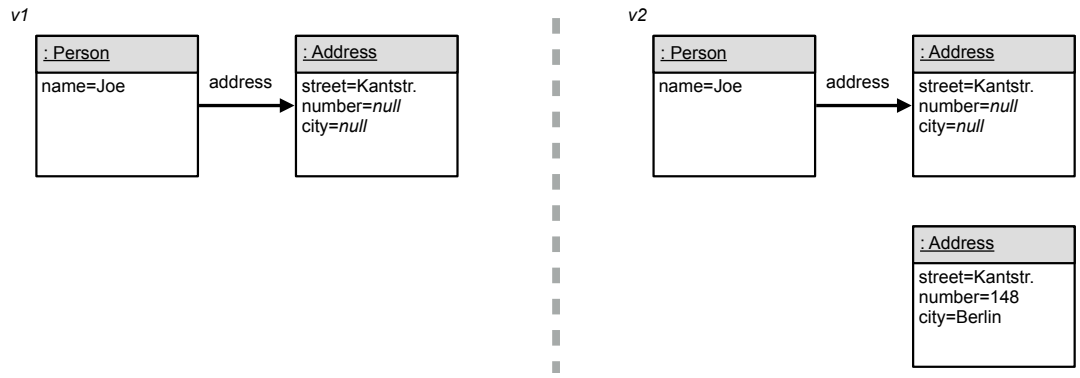

Figure 4.4: A reference refers to the previous version of the `address` object.

Even after adding values to the fields of the `address` object, the following statement would still return `true` when `aPerson` refers to the `person` object:

```
aPerson.address.city === null
```

Version-aware References

Our approach uses *version-aware references*. Version-aware references know the available versions of an object and always resolve to one of those. Furthermore, version-aware references know which object version belongs to which system version. None of the versions is hard-wired to be the active version. Instead, the version-aware references resolve dynamically to the correct versions using context information.

Apart from that, the version-aware references behave like ordinary references. They can be assigned to variables and object fields, and are passed around.

When the `person` object uses a version-aware reference to refer to its `address` property, it can resolve to the versions of its `address` object. The version-aware reference knows both versions of the address object. In version *v2* of the system, it resolves to the second version of the object, as shown in Figure 4.5.

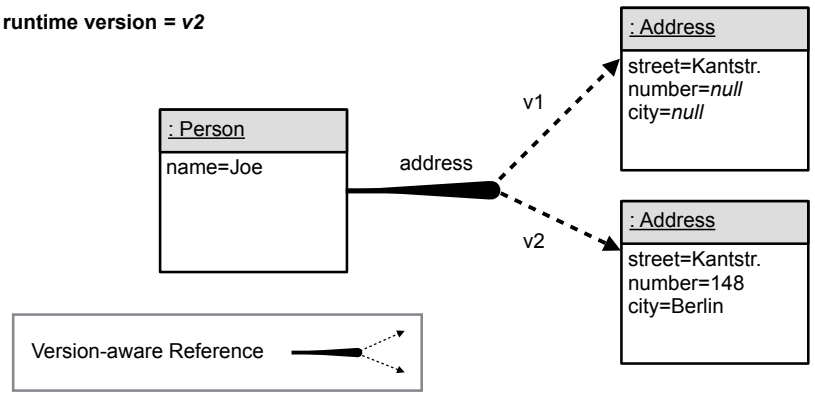

Figure 4.5: A version-aware reference relates a `person` object to two versions of its `address` property.

In the same way, multiple version-aware references can be resolved as one path through a graph of versions. The version-aware references all choose versions of objects that belong to the same system state and, thereby, form the object graph of that state.

Figure 4.6 shows an object graph that incorporates the previous example. The previously presented `person` object is a `company` object's `CEO` property. While the example shows that version *v2* is active, it also indicates a version *v1* and a version *v2* of the system. In version *v1*, the company's CEO has incomplete address information. In version *v3*, the company has a different CEO.

Versions of the System

To establish different versions of the system, the version-aware references have to resolve to different versions of objects. The version-aware references choose versions dynamically following a *version identifier*. Only this version identifier has to be changed to have version-aware references resolve to other versions of objects. For example, to undo the changes made with version *v2* of the system, the version identifier would need to be set to *v1* again.

Given the example situation from Figure 4.6 and given `aCompany` refers to the `company` object, the following statement would refer to three different values depending on the version identifier:

```
aCompany.CEO.address.number
```

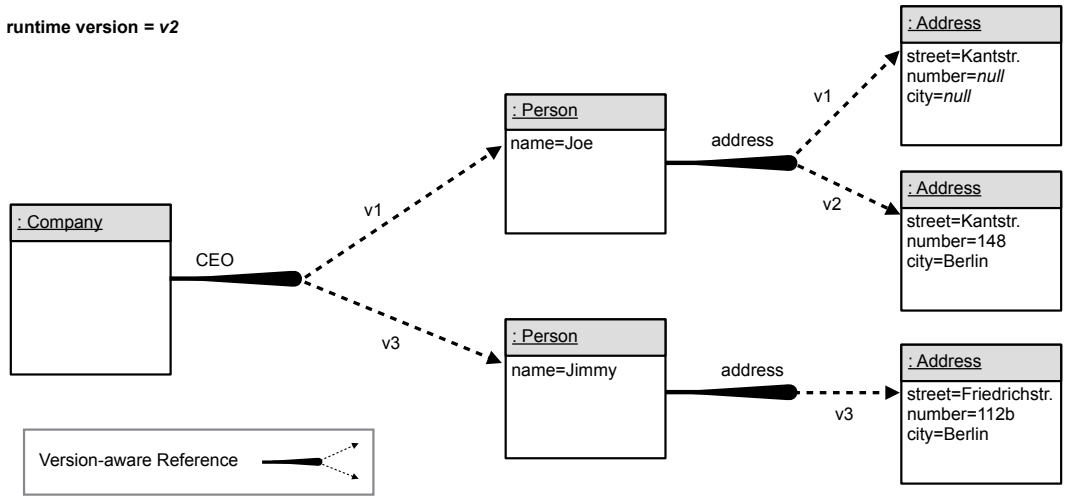

Figure 4.6: An object graph with version-aware references.

Evaluating the statement in version *v1* would return the value `null`, in version *v2* the value `148`, and in version *v3* the value `112b`.

The information that one version is the predecessor of another version can be used to resolve to an earlier object version when no current version is available. This allows to only create new versions of objects when necessary.

The version identifier needs to be accessible to the version-aware references. It could be available globally, to have a single active version of the system, but could also be scoped more locally such as thread-local or in the dynamic scope of a code block. It should, however, not be changed while multiple version-aware references of an object graph are resolved transitively. Consequentely, the version-aware references involved in evaluating the previous example statement should be resolved together for the same version identifier.

To be able to actually re-establish a particular version of the system with our approach, two requirements need to be fullfilled: First, all mutable objects of the programming runtime need to be accessed via version-aware references. Second, the particular version of the system needs to be available. Our approach does not allow re-establishing every state but specific states that have to been preserved. Programmers could preserve versions explicitly or the programming system could do so implicitly. When the programming system automatically preserves versions, each programmer action could implicitly yield a new version of the system. This way, programmers could undo and redo the changes

of their actions regardless of whether or not they preserved a version in anticipation of recovery needs.

Discussion

The presented approach is incremental, not a stop-the-world approach. The version-aware references allow to preserve and re-establish versions of the system without completely halting the program execution.
First, the version-aware references resolve dynamically to particular versions based on context information. Only this context information has to be changed to have all references resolve to another version. The version-aware references do not have to be re-configured individually.
Second, versions of the system are preserved incrementally. Instead of saving the state of all objects the moment a version is preserved, new versions of objects are created only when objects change. Before such writes, previous object versions continue to reflect the current state and can be read until they are changed.

4.2 Using Proxies as Version-aware References

We used proxies to implement version-aware references in JavaScript. Instead of actually requiring *alternative references*, proxies are referred to by *ordinary references* and transparently delegate to versions. This way, proxies allow a language-level implementation of version-aware references that works with existing JavaScript engines.

Proxies as Version-aware References

Figure 4.7 exemplifies how a proxy implements a version-aware reference in our solution. The proxy connects a `person` object to the two versions of its `address` property. The `person` holds an ordinary reference to the proxy in its `address` slot. The proxy in turn knows which versions are available for the `address` object.

When the `address` property of the `person` object is accessed, the proxy forwards the access transparently to a version. For example, in version *v2* of the system as indicated in Figure 4.7, even if the `address` property is a proxy, reading the proxy's `city` property returns the string `'Berlin'` . Given `aPerson` refers to the `person` object, evaluating the following statement returns `true` in version *v2*:

```
aPerson.address.city === 'Berlin'
```

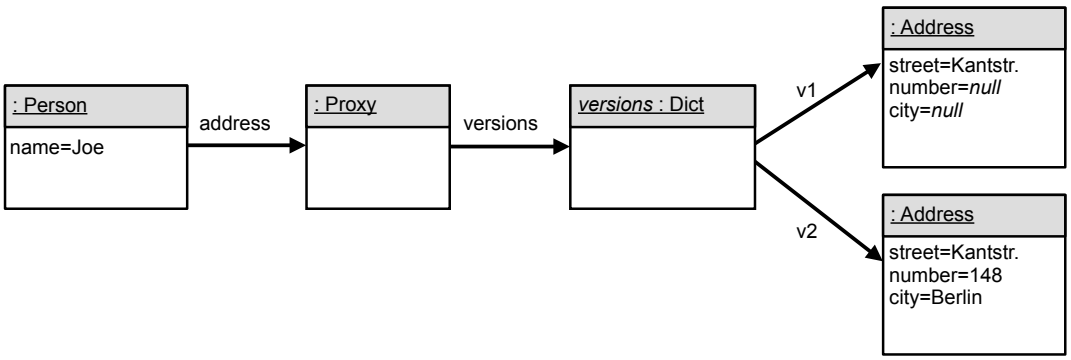

Figure 4.7: Using a proxy as version-aware reference to connect a `person` object to two versions of an `address` object.

The statement does not include any version information. In particular, it does not read a specific version from a table of available versions. Instead, the proxies intercept all object interactions and forward to specific versions transparently.

The proxies fulfill three responsibilities:

1. They know which versions are available for a particular object.

2. They choose a particular version among all available dynamically using context information.

3. They forward all interactions transparently to a chosen version.

The proxies in this design are *virtual objects* [35]. They do not stand in for specific objects, but can forward intercepted interactions to any object.

Using Proxies Consistently

The proxies need to be used consistently for all mutable state. Ordinary references that usually refer to an object need to refer to the proxy that stands in for the object.

To use proxies consistently, we create and return proxies for all new objects. All expressions that create new objects return proxies for those objects instead. This is achieved by transforming code before it is executed. The source transformations wrap object literals

and constructor functions into proxies. The proxies also always return proxies as return values. Thus, when proxied constructors are used, the constructors return proxies for the new objects.

The reference to the initial version of an object is only available to the proxy. The reference to the proxy is passed around instead. For this reason, all references that would usually point to the same object point to the same proxy. This way, proxies provide object identity. Checks that would usually compare an object to another objects now compare a proxy to another proxy.

As only the proxies hold references to the versions of objects, the versions get garbage collected with the proxies when the proxies are no longer reachable. For example, in the code of Listing 4.1, there would temporarily exist a version-aware reference—a proxy—connecting the `person` object to an `address` object, but the reference gets deleted before a version of the system is preserved. The `address` object is not required to re-establish either version *1* or version *2* of the system and nothing does prevent the garbage collector from reclaiming the proxy for the `address` object with the `address` object.

```
1   var person = {name: "Joe"};
2
3   \\ [preserve first version]
4
5   person.address = {street: "Kantstr.",
6                     number: "148",
7                     city: "Berlin"};
8
9   delete person.address;
10
11  \\ [preserve second version]
```

Listing 4.1: A newly created object is not preserved with any version.

Versions of the System

Proxies delegate to and create versions of an object using a *version of the system*.

A version of the system is an object that has a *version identifier*, a *predecessor* version, and a *successor* version. Figure 4.8 shows three system versions. In the example, version *v2* is the current version of the system.

The current version of the system is accessible to the proxies. Proxies use it to decide to which version of an object they currently should forward to. Figure 4.9 shows a proxy, versions of an object, and versions of the system. In this example, there are two object versions that correspond to the two system versions. The current version of the system is *v2* and, therefore, the version the proxy currently forwards to is version *v2* of the object.

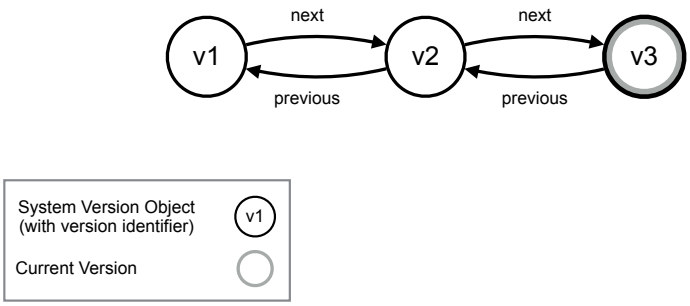

Figure 4.8: Four versions of the system.

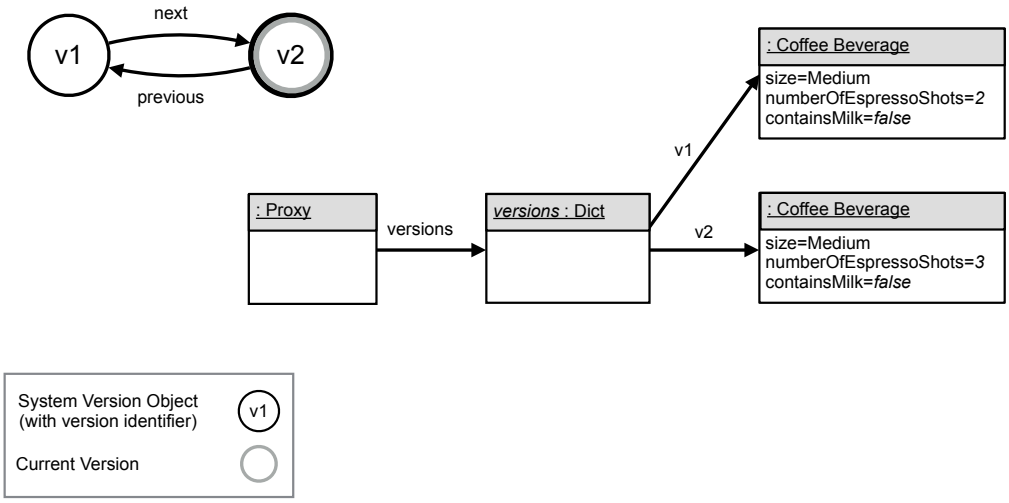

Figure 4.9: A proxy with two object versions in context of the system versions.

As long as the system version stays the same, the proxies forward to the same version of the object. Therefore, an object version is changed only as long as it matches the current system version.

To re-establish the previous version, the system version has to be set to its predecessor. In that case, proxies forward interactions to previous versions of the objects.

To preserve the current version, the system version has to be set to a different version. The proxies forward interactions to other object versions or, when no such version of the object exist, create new versions.

A situation in which a new version of an object is created is shown in Figure 4.10. In a new version *v3* of the system, the proxy intercepts a manipulation but has no object version it can forward to. It, therefore, copies the most recent version of the object and forwards to the copy.

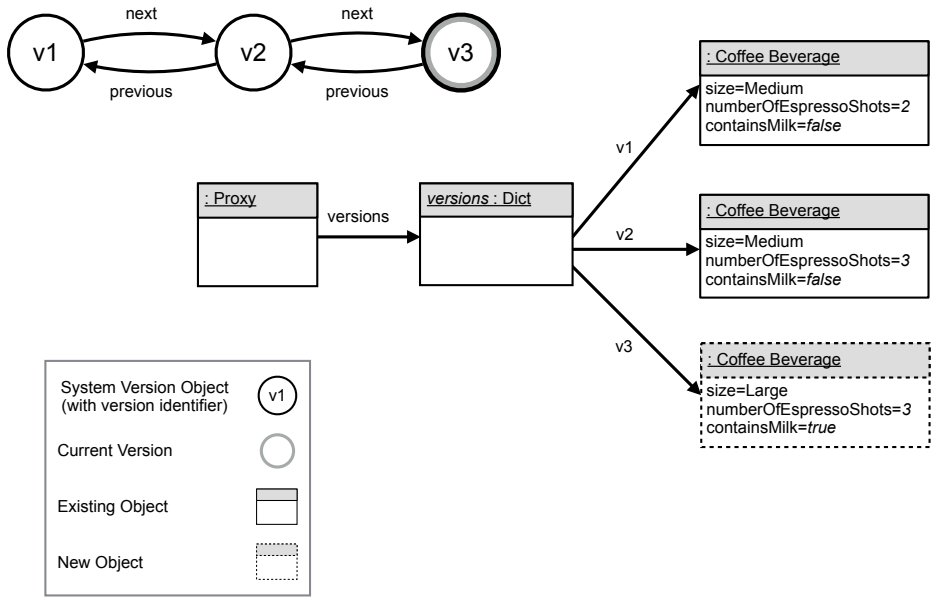

Figure 4.10: A new version of an object is created for a new version of the system.

New versions are only necessary when a proxy is about to delegate manipulations. As long as the state of an object is only read, the proxy reports values from a previous version as the old version of the object still reflects the current state. To create a new version, a proxy copies the most recent previous version of the object.

Limitations

The current design allows to preserve and re-establish versions of the system. Without further components, however, these versions only exist in memory and are not stored to disk.

Our current design does not support multiple predecessors or successors.

Another limitation of the current design is that the state of previous versions can be changed. New versions of objects are not affected by changes to previous versions, but

changes to object versions that have not been copied shine through in subsequent versions of the system.

In the future, the versioning might allow for branches and merging. Changes to previous states could then be handled in branches that programmers may or may not merge into future versions.

5 Implementation

This chapter describes how we used the ECMAScript 6 proxies[1] to implement version-aware references for the Lively Kernel. It presents the proxy's behavior and shows how proxies are inserted for ordinary references using source transformations. The chapter also presents workarounds for the current state of ECMAScript 6 proxies. It concludes with current limitations.

5.1 Using the ECMAScript 6 Proxies as Version-aware References

This section first describes the proxies proposed with the next version of JavaScript's standard. It then explains how the proxies are used in our implementation.

5.1.1 ECMAScript 6 Proxies by Example

The ECMAScript 6 proxies stand in for objects and intercept all kinds of interactions. For example, the proxies intercept property reads, enumerating over an object's properties, and calling a function.

The object a proxy stands in for is its *target*. The behavior of a proxy is controlled by a separate *handler* object. Target and handler are required when a proxy is created, as in the following example:

```
var proxy = new Proxy(target, handler);
```

The handler can implement *traps*, which are specific methods. Traps are called when a proxy intercepts corresponding object interactions. For example, the `get` trap is called for property reads. With these traps, the handler specifies how the proxy reacts on object interactions.

[1] http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies, accessed February 3rd, 2014

```
1  var client = {},
2      server = {openSecret: "I don't like Mondays"},
3      handler = {
4          get: function(target, name) {
5              console.log(name + ' was read at ' + Date());
6
7              return target[name];
8          }
9      }
10
11 client.server = new Proxy(server, handler);
```

Listing 5.1: Using a proxy to log property reads to an object.

Listing 5.1 shows an example, in which a proxy is used to log property reads. A `client` object is connected to a `server` object via a proxy, as shown in Figure 5.1. The `client` object's `server` property is a reference to a proxy and that proxy's target is the `server` object.

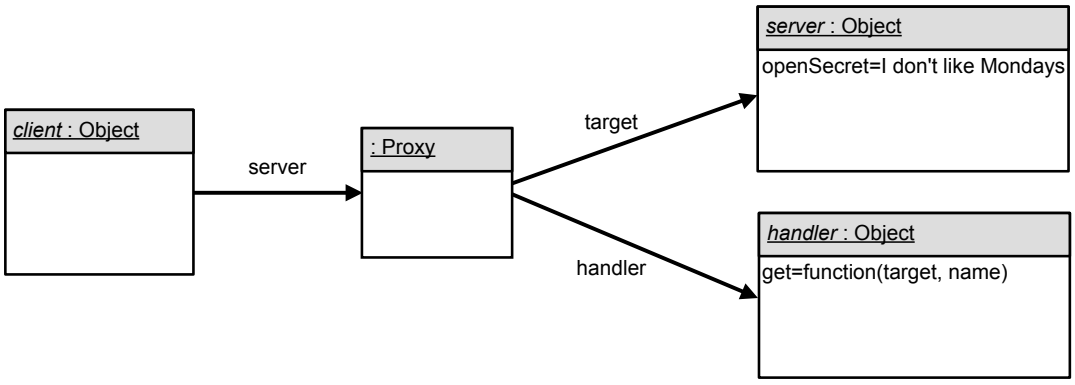

Figure 5.1: A `client` object has access to a `server` object via a proxy.

The proxy's handler implements the `get` trap. The `get` trap receives two arguments when called: `target` and `name`. The `target` parameter refers to the proxy's target object. The `name` parameter refers to the name of the property that was read. In Listing 5.1, the `get` trap logs the property read (Line 5), then forwards the read to the `target` object and returns the result (Line 7). Therefore, a log statement is printed whenever a property of the `server` object is read as in the following line of code:

```
client.server.openSecret;
```

The log statement would look similar to the following: "openSecret was read at Sat May 10 2014 23:00:54 GMT+0200 (CEST)".

A handler can implement traps for many kinds of object interactions. Table 5.1 lists all possible traps with their parameters. The `apply` trap and the `construct` traps are only called when the proxy's target is a function.

get: function(target, name, receiver)
set: function(target, name, value, receiver)
apply: function(target, thisArg, args)
construct: function(target, args)
has: function(target, name)
hasOwn: function(target, name)
defineProperty: function(target, name, desc)
deleteProperty: function(target, name)
getOwnPropertyDescriptor: function(target,name)
getOwnPropertyNames: function(target)
getPrototypeOf: function(target)
freeze: function(target)
seal: function(target)
preventExtensions: function(target)
isFrozen: function(target)
isSealed: function(target)
isExtensible: function(target)
enumerate: function(target)
keys: function(target)

Table 5.1: Traps that proxy handlers can provide.

The traps fire either when a proxy is accessed with JavaScript operators or when it is passed to meta-programming facilities. For example, the `apply` trap fires when a proxied function is applied as in the following statement:

```
proxy();
```

The `preventExtensions` trap fires when a proxy is passed to the respective function of the global `Object`. The `preventExtensions` function prevents subsequentely adding new properties to an object. The trap would be triggered by the following statement:

```
Object.preventExtensions(proxy);
```

When a proxy's handler does not implement a trap, the proxy forwards the intercepted interaction to the target.

Using the Proxies as Virtual Objects The proxies require target objects to which they forward by default. However, when a proxy's handler implements all traps, all intercepted interactions can be handled without forwarding to the target object. Therefore, even though proxies have target objects, the target objects do not have to be accessed with any object interactions.
This solution for using the proxies as virtual objects is also suggested by the official documentation[2].

5.1.2 Using the Proxies for Object Versioning

The proxies stand in for multiple versions of an object in our implementation. They forward all object interactions to one of those.

Figure 5.2 exemplifies our usage of the proxies. In the example, a proxy stands in for two versions of an `address` object: The proxy's handler holds a reference to a `versions` object, which in turn refers to the versions of the `address` object. The proxy's target is ommited from Figure 5.2 as we used the proxies as virtual objects. They do not forward to their target objects.

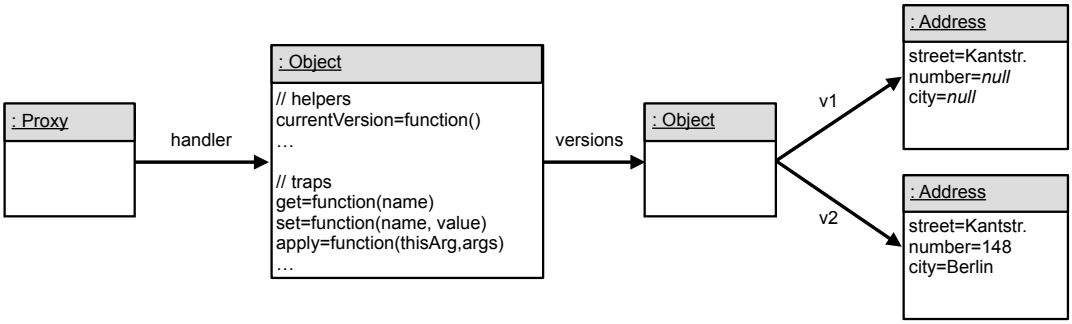

Figure 5.2: A proxy with a handler that forwards to two versions of an `address` object.

Our handler uses all traps to forward to the current version of an object. For example, its `get` trap is implemented as shown in Listing 5.2[3].

[2] http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies#virtual_objects, accessed May 11, 2014

[3] The code in this chapter is a simplified version of the actual code. It omits special cases for workarounds (Section 5.3) and debugging the implementation. The actual code is available at http://github.com/LivelyKernel/LivelyKernel/commits/50181548.

```
1 get: function(dummyTarget, name) {
2     var version = this.currentVersion();
3
4     return version[name];
5 }
```

Listing 5.2: The handler's `get` trap.

First, the trap retrieves the current version of the object using the `currentVersion` function (Line 2). Subsequentely, the trap reads the property from the version and returns the result (Line 4).

The `currentVersion` function chooses one of the versions of the object the proxy stands in for. It does so according to the version of the system. The system version is available globally as `lively.CurrentVersion`. It is an ordinary JavaScript object with three properties: an `ID`, a `predecessor`, and a `successor`. The `currentVersion` function uses the `ID` property to look up the correct version in its `versions` object.

Object versions of previous system versions are not allowed to change. However, when an object is not changed in versions of the system, a previous object version still reflects the current state. For this reason, our implementation does not copy objects that were not changed. Instead, the `currentVersion` function retrieves the latest available version as shown in Listing 5.3.

```
1  currentVersion: function() {
2      var objectVersion,
3          systemVersion = lively.CurrentVersion;
4
5      while(!objectVersion && systemVersion) {
6          objectVersion = this.versions[systemVersion.ID];
7
8          systemVersion = systemVersion.predecessor;
9      }
10
11     return objectVersion;
12 }
```

Listing 5.3: The handler's `currentVersion` method.

All traps that only read state select the version to forward to using the `currentVersion` function. However, traps that intercept changes are not allowed to forward to object versions of previous system versions. Instead, they need to make sure that a version of the object exists for the current system version. If such a version does not exist, the latest available version is copied and added to the `versions` object.

Listing 5.4 shows the `versionForWriteAccess` function that always returns an object version for the current system version. The function is used by all traps that intercept changes, which are listed in Table 5.2. The `apply` trap is included in this list because certain array functions such as `push` and `pop` are mutating.

```
versionForWriteAccess: function() {
    var newVersion;

    if (!this.versions[lively.CurrentVersion.ID]) {
        newVersion = this.copyObject(this.currentVersion());

        this.versions[lively.CurrentVersion.ID] = newVersion;
    }

    return this.currentVersion();
},
```

Listing 5.4: The handler's `versionForWriteAccess` method.

set: function(target, name, value, receiver)
defineProperty: function(target, name, desc)
deleteProperty: function(target, name)
freeze: function(target)
seal: function(target)
preventExtensions: function(target)
apply: function(target, thisArg, args)

Table 5.2: Traps that intercept changes.

Given the `currentVersion` and the `versionForWriteAccess` functions, the version of the system is written as long as it is referred to by `lively.CurrentVersion`. To re-establish a different version, only `lively.CurrentVersion` has to be changed. Changing the global version is an undo, redo, or commit depending on whether the version is set to a previous, following, or new version. The system provides the following three functions for this: `lively.undo` (Listing 5.5), `lively.redo` (Listing 5.6), and `lively.commit` (Listing 5.7).

Using a global version of the system is reasonable as JavaScript is executed single-threaded and scheduled cooperatively by the JavaScript engines. Therefore, while a script runs, the global version cannot be changed by another script.

```
undo: function() {
    var predecessor = lively.CurrentVersion.predecessor;

    if (!predecessor) {
        throw new Error('Can\'t undo: No previous version.');
    }

    lively.CurrentVersion = predecessor;
}
```

Listing 5.5: The `lively.undo` method.

```
1  redo: function() {
2      var successor = Lively.CurrentVersion.successor;
3
4      if (!successor) {
5          throw new Error('Can\'t redo: No next version.');
6      }
7
8      lively.CurrentVersion = successor;
9  }
```

Listing 5.6: The `lively.redo` method.

```
1  commit: function() {
2      var predecessor = lively.CurrentVersion,
3          newVersion;
4
5      newVersion = {
6          ID: predecessor.ID + 1,
7          predecessor: predecessor,
8          successor: null
9      };
10     predecessor.successor = newVersion;
11
12     lively.CurrentVersion = newVersion;
13 }
```

Listing 5.7: The `lively.commit` method.

Scope of the Versioning

Using the proxies allows multiple versions of JavaScript objects. However, certain host objects cannot be versioned with our implementation. These include the objects that represent the elements of the browser's Document Object Model (DOM). Some of the DOM objects cannot be copied. Therefore, it is not possible to create multiple versions of them. Furthermore, the DOM objects are referred to from the browser's DOM, which is external to the JavaScript runtime and which, thus, does not use proxies to access the objects. However, this is not a problem, because the state of the DOM can be derived from the Lively Kernel's morph objects. For this reason, we update the DOM from the current set of visible morphs when the system version changes. Besides these host objects, all objects that are accessed through our proxies are versioned with the system versions.

5.2 Accessing All Mutable JavaScript Objects Through Proxies

To be able to re-establish the system state with our versioning, our proxies need to be used to access all objects, arrays, and functions. This is necessary because objects, arrays,

and functions are mutable in JavaScript. In fact, functions and arrays are objects. They can have arbitrary properties.

Our implementation changes the return values of all expressions that create new objects. Instead of letting these expressions return references to the new objects, the expressions return references to proxies for the objects. As a result, references to proxies are passed around instead of references to objects so that all access goes through the proxies.

In JavaScript, there are three categories of expressions that create new objects:

- literal expressions: e.g. `{age: 12}`
- constructor functions: e.g. `new Person(12)`
- specific built-in functions: e.g. `Object.create(prototype, {age: 12})`

Our implementation uses source transformations and the proxy traps to have these expression return proxies.

5.2.1 Transforming Literal Expressions

We use source transformations to wrap literal expressions into calls to a `proxyFor` function. The function returns a proxy for its argument. The transformations for literal objects, arrays, and functions are shown in Table 5.3.

Type	Input	Output
Objects	`{name: 'James', age: 24}`	`proxyFor(name: 'James', age: 24)`
Arrays	`[person1, person2]`	`proxyFor([person1, person2])`
Functions	`function (a, b) {..}`	`proxyFor(function (a, b) {..})`

Table 5.3: Transforming literal objects, arrays, and functions.

However, some literal forms cannot be wrapped into function calls without introducing problems. In particular, function declarations and accessor functions need to be handled differently.

Function Declarations

A *function declaration* is a function literal that creates a named function and makes it available by the name. It does not need to be assigned to a variable to be available in the surrounding scope. The following statement is a function declaration:

```
function add(a, b) {return a + b}
```

In contrast, a *function expression* creates a function that needs to be assigned to a variable to be accessible. The following statement assigns a function expression to a variable:

```
var subtract = function(a, b) {return a - b}
```

Function expressions can create anonymous and named functions. The previous example creates an anonymous function, while the following example creates a named function:

```
var multiply = function multiply(a, b) {return a * b}
```

An anonymous function is always a function expression. A named function is either a function expression or a function declaration, depending on where it is expressed. A function declaration cannot be nested into other statements such as variable assignments. It has to start with the `function` keyword.

Therefore, when a function declaration is wrapped into a function call, it becomes a function expression. The function would no longer be available by its name in the surrounding scope. For this reason, the function declarations that are wrapped into calls to the `proxyFor` function are assigned to matching variable names. Table 5.4 shows an example for this transformation.

Input	Output
`function div() {}`	`var div = proxyFor(function div() {})`

Table 5.4: Transforming a function declaration.

In addition, because function declarations get hoisted in JavaScript, transformed function declarations are moved to the beginning of the defining scope.

Accessor Functions

Accessor functions are functions that are executed instead of property reads or writes. Listing 5.8 shows an example in which an accessor function is used to allow reading

a `person` object's `age` property even though the object actually only has a `birthdate` property.

```
var person = {
    birthdate: new Date(1984,27,5),
    get age() {
        return ageToday(this.birthdate);
    }
}
```

Listing 5.8: An object literal with accessor function.

Wrapping the accessor function into a call to the `proxyFor` function would not yield valid JavaScript syntax. However, accessor functions can also be defined using the `Object.defineProperty` function. For this reason, the object is first created without the accessor function and the function is added afterwards using `Object.defineProperty`. Listing 5.9 shows the result of transforming the example in Listing 5.8 in this way. The object literal and the call to `Object.defineProperty` are wrapped into an anonymous functions that is applied directly. This allows to have the object be available in a variable for the `Object.defineProperty` function call without polluting the variable bindings of the originally surrounding scope.

```
var person = function() {
    var newObject = lively.proxyFor({
        birthdate: new Date(1984,27,5);
    });
    Object.defineProperty(newObject, "age", {
        get: lively.proxyFor(function age() {
            return ageToday(this.birthdate);
        })
        enumerable: true,
        configurable: true
    });
    return newObject;
}();
```

Listing 5.9: The result of transforming an object literal with accessor function.

5.2.2 Returning Proxies from Constructor Functions

When functions are used as constructors, they need to return proxies. In JavaScript, all functions can be used as constructors and create objects when called with the **`new`** operator. Listing 5.10 shows how a literal function is used to construct a new object.

```
function Person() {}
var someone = new Person();
```

Listing 5.10: Applying a function with the `new` operator.

We use the `construct` trap to return proxies from proxied functions that are used as constructors. Listing 5.11 shows the `construct` trap of our proxy handlers.

```
1  construct: function(dummyTarget, args) {
2      var constructor, prototype, newObject, result;
3
4      constructor = this.currentVersion()
5
6      prototype = constructor.prototype ? constructor.prototype : {};
7      newObject = Object.create(prototype);
8
9      result = constructor.apply(newObject, args);
10
11     return proxyFor(result ? result : newObject);
12 }
```

Listing 5.11: The handler's `contruct` trap.

The `construct` trap does the following:

1. It retrieves the current version of the constructor (Line 4).
2. It creates a new object with the correct prototype (Line 7).
3. It calls the constructor with the new object as argument (Line 9).
4. It returns a proxy for either the return value of the constructor function or, in case the constructor did return a falsy value, the new object (Line 11).

This way, all proxied functions return proxies when used as constructors. With the previously presented transformations of literal expressions all literal functions are accessed through proxies.
However, there are also functions built into the JavaScript engines. These are not created from function literals and, therefore, cannot be proxied by transforming function literals.

5.2.3 Wrapping Built-in Functions Into Proxies

Some built-in functions can be used to create new objects. For example, the built-in constructors `Object` and `Array` can be used to create new objects and arrays. They return new objects when called with the **`new`** operator and when called without.
Other global functions that create new objects include, for example, `Object.create` and `eval`. The `Object.create` function takes an object as argument, uses the argument as prototype of a new object, and returns the new object. The `eval` function takes a string as argument. It evaluates the string as JavaScript code and, depending on the code string, can return new objects and even object graphs.

Wrapping Built-in Constructor Functions

We transform the built-in constructor functions by wrapping each into calls to the `proxyFor` function. Table 5.5 shows this with two examples for the global `Object` function.

Input	Output
`Object()`	`proxyFor(Object)()`
`new Object()`	`new proxyFor(Object)()`

Table 5.5: Transforming built-in constructors.

The global symbols that are wrapped into calls to the `proxyFor` are: `Array`, `Boolean`, `Date`, `Function`, `Iterator`, `Number`, `Object`, `RegExp`, `String`, `JSON`, `Math`, `Intl`, `XMLHttpRequest`, `Worker`, `XMLSerializer`, `window`, and `document`.

Therefore, when these function objects are used as constructors, the `construct` trap is called and returns proxies for the new objects as explained previously. As these functions also create objects when called without the `new` operator, we also have the `apply` trap return proxies. Therefore, the last line of our `apply` trap is:

```
this.ensureProxied(result);
```

When the argument to the `ensureProxied` function is already a proxy or an immutable value such as a string or a number, it returns the argument unchanged. When the argument is, however, an object, an array, or a function, the `ensureProxied` returns a proxy for that object. This distinction is necessary as the `apply` trap is called for all functions, not only built-in constructors.

Proxy Table With our solution each occurrence of a built-in constructor is wrapped into a separate call to the `proxyFor` function. Therefore, the same function objects are passed to the `proxyFor` function multiple times. To nevertheless return the same proxies for the same objects, we use a map to associate objects with their proxies. This *Proxy Table* is a weak-key map. It does not prevent the garbage collector from reclaiming the objects used as keys.
Using the same proxies for the same objects is not only an optimization, but necessary for identity checks. As a result of using the Proxy Table, the following statement returns `true` for an arbitrary `obj` object:

```
proxyFor(obj) === proxyFor(obj);
```

Wrapping Other Built-In Functions

Besides the built-in constructors, other built-in functions that create new objects need to return proxies as well. For example, the following statement needs to return a proxy:

```
Object.create(proto);
```

In addition to the `construct` trap and the `apply` trap, we also have the `get` trap return proxies. Analogous to our `apply` trap, the last line of our `get` trap is:

```
this.ensureProxied(result);
```

Therefore, proxies for objects always return proxies when properties are read. As a result, when the built-in constructor `Object` gets wrapped into a call to the `proxyFor` function as with the previously presented transformations, reading its `create` property returns a proxy for the property. Furthermore, this proxy's `construct` trap returns a proxy when applied. Thus, the following statement returns a proxy for the new object:

```
proxyFor(Object).create(proto);
```

Therefore, the previous transformations are sufficient to have all functions of the globals return proxies.

The `eval` function, however, is handled differently. It can return object graphs for the string argument provided. For example, the following statement returns an object with an `address` property, which is in turn an object:

```
eval("{age: 12, address: {street: 'Kantstr', city: 'Berlin'}}")
```

Therefore, it is not sufficient to ensure that the return value of the `eval` function is a proxy. Instead, the object in the example also needs to access its `address` property via a proxy.
Moreover, the code passed to `eval` could access built-in constructors or itself use the `eval` function. For this reason, we pass the string argument of the `eval` function to our source transformations before it is evaluated.

The built-in functions could be overwritten globally to return proxies for the new objects, but our implementation of object versioning is a JavaScript library and makes itself use of the built-in types. Additionally, at the time of writing, some JavaScript engines do not allow to overwrite particular built-in globals and we want our implementation of object versioning to work in every JavaScript engine that supports the ECMAScript 6 proxies.

Implementation of Source Transformations

Our implementation uses the *UglifyJS*[4] library for all source transformations. UglifyJS parses source code without relying on JavaScript exceptions. Therefore, when code is transformed, it does not yield exceptions that could be caught by an open debugger. In addition, UglifyJS supports Source Maps[5]. This allows the browser's developer tools to present the original sources, even though transformed code is executed.

5.3 Workarounds for the Current State of the ECMAScript 6 Proxies

Certain workarounds are required due to the preliminary implementation of ECMAScript 6 proxies in the JavaScript engines.

ECMAScript 6 Specification ECMAScript 6 will be the next version of JavaScript. Its specification has not yet been finalized. Drafts of it are released continuously with a target release date of December 2014[6]. The current draft is Revision 24 [6]. It includes the proposal of the proxies we used for our implementation.

ECMAScript 6 Implementation The JavaScript engines used by Chrome and Firefox provide preliminary implementations for some of ECMAScript 6's features. Among other features, they implement two different deprecated proposals of the proxy application programming interface (API). The *harmony-reflect* library[7] provides the current API on top of these. Our implementation uses the *harmony-reflect* library and, therefore, works in Chrome and Firefox.

However, even with the library, three issues need to be addressed with technical workarounds:

1. The proxies have to be provided with a target object, even when implementing virtual objects, and consistency invariants compare return values of the traps to the state of the target.

[4] http://github.com/mishoo/UglifyJS2, accessed March 12, 2014
[5] https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-_2gc6fAH0KY0k/edit#heading=h.ue4jskhddao6, accessed May 2, 2014
[6] http://github.com/rwaldron/tc39-notes/blob/48c5d285bf8bf0c4e6e8bb0c02a7c840c01cd2ff/es6/2013-03/mar-13.md#416-current-status-of-es6, accessed May 12, 2014
[7] http://github.com/tvcutsem/harmony-reflect, accessed February 3, 2014, used version 0.0.11

2. The proxies do not intercept the `instanceof` operator, but always delegate to the current target.

3. Certain built-in JavaScript functions don not handle proxies correctly.

These workarounds might no longer be necessary once the ECMAScript 6 specification gets released and fully implemented by the JavaScript engines.

5.3.1 Disabling Target Object Invariants

Problem

Even though the current ECMAScript 6 draft says otherwise[8], the proxies require a target object, as explained in Section . Even when the proxies are used as *virtual objects*, they still are connected to a particular object. Furthermore, the proxies are designed to ensure invariants between the return values of traps and the target's state [3]. For example, when an object's properties are made immutable through the `Object.freeze` function, invariants ensure that the target object has in fact been frozen, even if the trap delegates the operation to another object. Another invariant ensures that immutable values of the target object are reported by the traps. Therefore, the *get*-trap has to report the values of the target object when it previously has been frozen. As a result, in our case, configuring any property as immutable would effectively make that property immutable for all versions. Moreover, reading the property from object versions would potentially raise inconsistency errors.

Solution

For this reason, we adapted our copy of the *harmony-reflect* library. In particular, the `Proxy` constructor takes a third argument. This argument is a boolean that indicates whether a proxy is standing in for one target object or is a virtual object. Providing `true` as this argument effectively disables all consistency checks.

Discussion

Even with disabled consistency checks, the proxies still require actual objects as *target* objects. All trapped interactions are forwarded to the correct object versions, yet inter-

[8] http://people.mozilla.org/~jorendorff/es6-draft.html#sec-proxy-object-internal-methods-and-internal-slots, accessed April 15, 2014

actions that are not intercepted by any traps are still forwarded to the target objects. The current draft does not include traps for the **typeof** operator and the **instanceof** operator.

The **typeof** operator returns a certain set of values for different types. Proxies stand in for objects, arrays, and functions. It returns *"object"* for arrays and objects and *"function"* for functions. Therefore, the target object of our proxies is an object when a proxy stands in for the versions of an array or an object, while it is a function when a proxy stands for the versions of functions. Using a function object as target for a proxy that stands in for the version of a function is also necessary as the `apply` and `construct` traps are only called for proxies with function targets.

The **instanceof** operator needs to be handled differently as it does not necessarily return the same value for all versions of an object.

5.3.2 Forwarding the *Instanceof* Operator

Problem

The **instanceof** operator can be used to test whether an object has a specific type. In particular, it checks whether the `prototype` property of a function is in an object's prototype chain. For example, the following statement checks this for the `Person` function and the `me` object:

```
me instanceof Person
```

The prototype of an object is a property and can be changed at runtime. Moreover, the `prototype` property of a function is also mutable. As any other property, these properties can be different in different object versions. Thus, the **instanceof** operator needs to be delegated to the current object version. However, while trapping the **instanceof** operator is under discussion[9], there is currently no **instanceof** trap.

Solution

Our implementation provides a custom `Object.instanceof` function, which implements the semantics of the **instanceof** operator but delegates to object versions when applied with a proxy as argument. Consequentely, all usage of the **instanceof** operator is

[9] http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies#discussed_during_tc39_july_2012_meeting_microsoft_redmond, accessed May 1, 2014

transformed to the `Object.instanceof` function. Table 5.6 shows this transformation by example.

Input	Output
`me instanceof Person`	`Object.instanceof(me, Person)`

Table 5.6: Transforming the `instanceof` operator.

5.3.3 Unwrapping Versions for Native Code

Problem

Some built-in JavaScript functions do not work correctly with proxies. The functions react with errors, return wrong results, or silently ignore calls when applied with proxies as arguments or as their `this`-context. These built-in functions include, for example, the `concat` function of array instances and all functions that manipulate the browser's DOM. Moreover, string instance methods return wrong results when called with proxies for `RegExp` arguments.

Furthermore, the `onreadystatechanged` property of `XMLHttpRequest` objects is not allowed to be proxy. The `onreadystatechanged` property is expected to be a function, which is called when the server responds to asynchronous Hypertext Transfer Protocol (HTTP) requests. However, when a proxy is assigned as the property, the callback is not called with the response.

Solution

The problematic functions need to be provided with actual objects instead of proxies. Therefore, our implementation retrieves the current object versions from the proxies and provides these to the functions. The `apply` trap unwraps all arguments and the *thisContext* before applying built-in functions.

The `apply` trap detects built-in functions through their print strings. Built-in functions print to *"[native code]"*, while functions created from literals print to their function body. However, the `apply` trap does not unwrap the arguments to specific built-in functions. For example, the argument to an array's `indexOf` function is not unwrapped as the function compares the argument's identity to the identity of array elements. Additionally, the iterator functions of arrays handle proxies correctly.

Instances of strings are immutable and are not accessed through proxies in our implemention. Therefore, unwrapping arguments in the `apply` trap is not a solution for the string methods that do not handle proxies correctly. For this reason, our implementation patches these string functions explicitly. The functions are patched with functions that unwrap proxies before executing the original functions, as shown in Listing 5.12.

```
1 var originalStringMatch = String.prototype.match;
2 String.prototype.match = function match(regexp) {
3     var exp = Object.isProxy(regexp) ?
4         regexp.proxyTarget() : regexp;
5     return originalStringMatch.call(this, exp);
6 };
```

Listing 5.12: Patching a method of string instances.

The four string methods that are patched in this way are: `match`, `search`, `replace`, `split`.

For the `onreadystatechanged` property of `XMLHttpRequest` objects, we have the `set` trap unwrap the assigned proxy. However, unwrapping a particular version of the callback function is potentially problematic. Even though JavaScript does get executed with a single thread using cooperative scheduling, other scripts might get executed while the browser waits for the server's response. Such concurrently executed scripts can switch the global version before the callback is called. In a different version, the properties of the callback function can be different. Moreover, the callback function might not be available in a previous version. Switching the system version while the browser waits for a response is a problem for which our implementation currently does not provide a workaround.

5.4 Limitations of the Implementation

We are aware of three limitations of our implementation.

Availability of ECMAScript 6 Proxies Our implementation requires the ECMAScript 6 proxies to be available. The proxies are part of the next version of ECMAScript, which has currently neither been finalized nor completely implemented, as described in Section 5.3. For this reason, our implementation works only in versions of Firefox and Chrome that already implement preliminary versions of the proxies. Furthermore, Chrome users need to enable the proxies explicitly[10].

[10] http://plus.google.com/+PaulIrish/posts/T615Md5JPQG, accessed May 13, 2014

Proxies Impede Developer Tools The current implementation of ECMAScript 6 proxies impedes debugging. The proxies are partly implemented by a JavaScript library and every trapped object interaction is visible in multiple frames in the debugger. Consequentely, the stack of the debugger is cluttered with frames that belong to the proxy implementation, not to application code.
Moreover, the developer tools in Chrome do not handle proxies correctly under all circumstances. In particular, hovering over variable names that are bound to proxies yields errors. It is also not possible to step into proxied functions in Chrome's debugger. We did not test the developer tools of Firefox.

Concurrent JavaScript Even though JavaScript is executed with a single thread, scripts can be executed concurrently. In general, scripts can be started from events and from other scripts using the `setTimeout` or the `setInterval` function. Switching the system version can be problematic for such concurrently running scripts.
In the Lively Kernel, the `setInterval` function is used to repeatedly execute a method of an object. Re-establishing a previous version of the system can interfere with such ticking behavior. For example, the method that is called repeatedly can be unavailable in the previous version. For this reason, we stop scripts from future versions, when the system version changes. However, we did not succeed in finding a way to restart the scripts again when the future versions are re-established. Thus, subsequentely undoing and redoing changes can stop concurrently running scripts.

6 Evaluation

We evaluated the functionality and practicability of our implementation.
For this evaluation, we used the setup described in the following section.

6.1 Test Setup

We evaluated our implementation with the benchmark suite, the Lively Kernel version, and the machine configuration described in this section.

6.1.1 Octane Benchmark Suite

We used the *Octance* benchmark suite[1] to evaluate the behavior and performance of our implementation. Octane consists of eight JavaScript benchmarks. It is a suite of real programs such as the *DeltaBlue* [7] constraint solver. It does not test JavaScript's features systematically, but the benchmarks make use of many important language features, including primitive data types, operators, functions, objects, prototypical inheritance, and many built-in functions.
Octane is used for evaluating the performance of *v8*, the engine used by Chrome and Node.js. It is part of v8's official source code repository[2].

We use Octane for three reasons:

- Octane is a standard benchmark suite.
- Octane covers many JavaScript language features.
- Octane is used in tuning Chrome's JavaScript engine and Chrome is the browser in which the Lively Kernel works best.

[1] `http://code.google.com/p/octane-benchmark/`, accessed February 3, 2014, at version 26
[2] `http://v8.googlecode.com/svn/`, accessed April 23, 2014, at revision 20901

6.1.2 Lively Kernel

During the implementation of our approach, we tested our prototype continuously with the Lively Kernel.

We developed our system in the Lively Kernel repository. The commit we used for the evaluation is *ed0586d80*[3]. In this version, most of the Lively Kernel's code is passed through our transformations. Only the Lively Kernel's bootstrap code, its module system, extensions to built-in types, and our implementation are excluded from the source transformations. All modules loaded after these parts are transformed at load-time to enable versioning for them. This includes, for example, all classes of the Lively Kernel.

We tested our implementation with the Lively Kernel for two reasons:

- The Lively Kernel is a large JavaScript application that makes use of many features of the JavaScript language and the browser environment. The browser environment provides many built-in objects and functions. These are not part of the ECMAScript standard, but are nevertheless used by many applications. For example, the browser offers functions to manipulate its DOM, which the Lively Kernel uses for rendering. These built-ins are not covered by Octane or other popular JavaScript benchmark suites.

- The goal of this work is to provide object versioning for the Lively Kernel. Thus, we are particularly interested in evaluating our implementation for the Lively Kernel.

6.1.3 Machine Configuration

All tests and measurements were done on May 9, 2014 using a Macbook Air with a 2 GHz Intel Core i7 and 8 GB main memory, Mac OS X 10.9.2, and version 34.0.1847.131 of Chrome.

The presented measurement results were averaged over five runs.

We used Chrome for all experiments as the Lively Kernel currently works best in Chrome.

[3] http://github.com/LivelyKernel/LivelyKernel/commits/ed0586d80, accessed May 9, 2014

6.2 Functionality of Version-aware References

We tested whether the version-aware references forward correctly to versions of objects with benchmarks and with the Lively Kernel.

6.2.1 Testing with Benchmarks

We ran the Octane benchmark suite to test the functionality of our implementation.

Method We transformed the Octane benchmarks with our source transformations, executed the resulting code, and then checked for JavaScript errors and compared the results of the transformed benchmarks to their usual results.
We did this to test two aspects. First, to test whether our source transformations yield syntactically correct JavaScript code for the benchmarks. Second, to test whether our proxy-based version-aware references, inserted by the source transformations, allow to run the benchmarks without errors and with the expected results.

Results All benchmarks in this suite run without errors and return the same results as if executed without any source transformations. Therefore, at least for these tests, our source transformations produce working source code and our proxy-based version-aware references forward correctly to object versions.
During the development of our system, the *DeltaBlue* benchmark revealed a problem when proxies are used as prototypes of objects. We reported the issue to the *harmony-reflect* repository[4]. The problem was identified as an issue with the *v8* JavaScript engine[5]. We implemented a workaround for this problem, but the issue was subsequentely fixed, rendering the workaround redundant.

Discussion The proxies behave correctly like particular versions of objects in the situations tested by the benchmarks. While these benchmarks do not test JavaScript's features systematically, they cover a wide range of important language features.

[4] http://github.com/tvcutsem/harmony-reflect/issues/18, accessed April 23, 2014
[5] http://code.google.com/p/v8/issues/detail?id=2804, accessed April 23, 2014

6.2.2 Testing with the Lively Kernel

We tested whether the Lively Kernel loads and works with our version-aware references. Moreover, we tested whether versions of its state can be preserved with our implementation.

Method We transformed the JavaScript modules of the Lively Kernel at load-time to test whether it loads and works correctly with our proxy-based version-aware references. Furthermore, we tested whether the system allows re-establishing versions of the Lively Kernel's state in practice. Here, we tried multiple example scenarios, including the undo of changes to the state and behavior of basic morphs, morph compositions, and the state of more complicated graphical applications such as text editors and developer tools.
With this, we tested that the source transformations yield valid JavaScript code for the modules of the Lively Kernel, that the version-aware references delegate to the correct versions of objects, and that the version-aware references are used consistently.

Results The Lively Kernel loads when its modules are transformed to use our version-aware references. Most of its basic functionality works as expected and we were able to preserve and re-establish runtime states of multiple examples. However, not all functionality works as expected and we were, thus, not able to re-establish all preserved states. In particular, we learned about the many built-in functions that currently do not handle proxies correctly in Chrome and for which we implemented the workaround described in Section 5.3.3.

Discussion Most of the tested functionality of the the Lively Kernel works correctly. This includes the entire bootstrap process, rendering graphical objects, loading parts from the Lively Kernel's Parts Bin, and using the Lively Kernel's halo controls. However, certain functionality of the Lively Kernel is not yet working correctly or even yields errors. The remaining issues here are expected to be problems related to the built-in functions that do not work correctly when proxies are provided as arguments. Our implementation already unwraps object versions from proxies for many built-in functions, as explained in Section 5.3.3, but the configuration does not cover all problematic built-in functions yet. At the same time, the proxies are not yet fully supported by Chrome and we expect these issues not to be problematic anymore when proxies get fully implemented by Chrome's JavaScript engine.

6.3 Practicability: Memory Consumption

We measured the memory overhead imposed by the version-aware references and how much memory is consumed when versions of the Lively Kernel's state are preserved.

For the measurements, we used Chrome's built-in memory profiler[6]. It allows to take heap snapshots. These snapshots contain all reachable JavaScript objects. For each snapshot, Chrome shows the total size in Megabytes (10^6 Bytes) (MB).

6.3.1 Memory Overhead of Version-aware References

We measured how much more memory is required when loading the Lively Kernel with version-aware references.

Method We measured the memory required for loading a Lively Kernel world with and without version-aware references. We took heap snapshots right after the world was completely loaded without interacting with the system. We used an empty Lively Kernel world for this experiment and did not preserve any versions.

Results As shown in Figure 6.1, loading an empty Lively Kernel world requires three times more space with proxies than without proxies.

Discussion When loaded with proxies, the system requires space for the proxies. Even without preserving multiple versions of any object, the system uses a proxy for each object. These proxies require additional space: Each proxy comprises of at least a proxy object, a proxy handler object that specifies the proxy's behavior, and an object to hold all object versions.
We expect the memory overhead to increase linearly with the number of objects accessed through proxies. While the system creates proxies for most objects, it does not use proxies for all objects. In particular, it does not create proxies for objects that are present before our implementation of object versioning is loaded and all objects used by our implemenation itself. We expect the number of objects that are excluded from versioning to be relatively stable. All additional objects created at runtime will be accompanied by proxies.
The memory overhead does not appear to be problematic at the moment.

[6] http://developers.google.com/chrome-developer-tools/docs/heap-profiling, accessed May 8, 2014

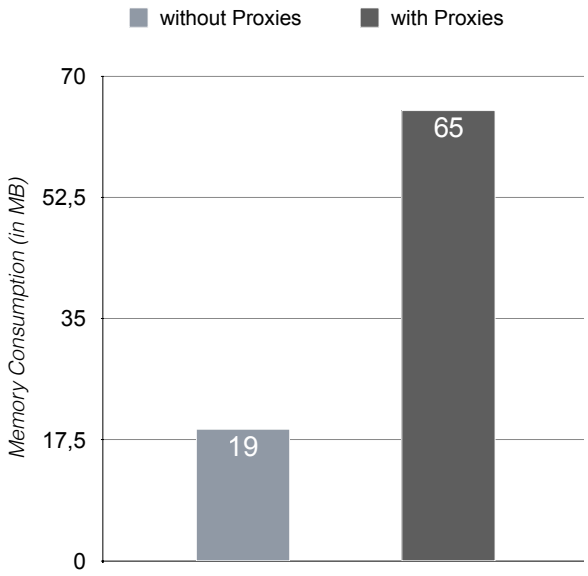

Figure 6.1: Memory consumption when starting a Lively Kernel world with and without proxies.

6.3.2 Memory Consumption When Preserving Versions

Besides the memory required for the version-aware references, memory is consumed when versions of the system are preserved.

Method

We measured how much memory is consumed when multiple versions of the system are preserved while working on a group of morphs. The three states for which we took snapshots are shown as ①, ②, and ③ in the upper half of Figure 6.2. In particular, we did the following in this experiment:

1. Version 1: We measured the memory consumed at State ① in the initial version of the system.

2. Version 2: We created a new version to preserve the initial state and then changed the state towards State ② in the new version. Subsequentely, we measured the memory consumption for this state.

3. Version 3: We preserved the previous state, changed the state to State ③ in a third version, and measured the memory consumption again.

This experiment does not show how much memory is required exactly for storing multiple versions of particular objects. Instead, the experiment shows the overall memory consumption of the entire Lively Kernel while our implementation is used realistically.

The snapshots include the size of all reachable JavaScript objects, not just the versions of the morph objects shown Figure 6.2. The reachable JavaScript objects in these snapshots are all objects of the Lively Kernel. For example, the tools we used to change the morph states between the snapshots are implemented in JavaScript. Their state is part of the system state.
We closed all tools before taking memory snapshots to exclude their state from the snapshots, but the Lively Kernel caches some state of these tools. The cached state might be different in the three states. Thus, the size of the cached state might be different in the three snapshots. Furthermore, previous versions of the cached state might get preserved with the versions of the system.

Results

Figure 6.2 shows the size of the three snapshots. State ① required the least memory. State ② requires 1.8 MB more memory. It also requires 0.3 MB more memory than State ③.

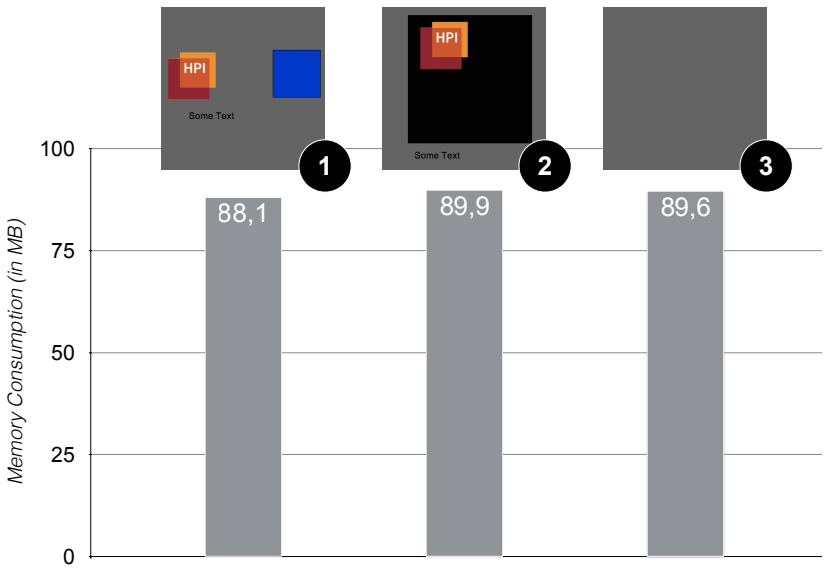

Figure 6.2: Memory consumed for three different states when the previous states are preserved in separate versions.

Discussion

The sizes of the three snapshots are not significantly different. Even though it is not clear how much space is used for preserving the previous states of just the morphs, the results show that preserving system states requires relatively little memory. Our implementation does not copy all objects for each version, but only creates copies when objects change from one version to another, effectively storing only the differences between system versions. Therefore, the memory required for preserving versions of the system depends on how objects change in each version. In the presented scenario, the space required for preserving the three states is insignificant to the space already required for running the Lively Kernel.

The results also show that the memory consumption does not always increase even when previous states are preserved: State ③ requires less memory than State ②. One explanation for this is that not all objects are preserved with the versions. One category of such objects are the objects that only provide access to the elements of the browser's DOM, as described in Section 5.1.2.

6.4 Practicability: Impact on Execution Speed

We measured the overhead our implementation of version-aware references imposes on running benchmarks and the Lively Kernel. A discussion of the results follows at the end of this section.

6.4.1 Execution of Benchmarks

The Octane benchmark suite shows how the proxies currently slow down a variety of different JavaScript programs.
A microbenchmarks shows the specific cost of having proxies forward object interactions.

Octane Benchmark Suite

Measuring the Octane benchmark suite highlights how the execution of eight JavaScript programs is affected by the proxies.

Method We ran the Octane benchmarks[7] with and without previous transformation of the benchmark code and, therefore, with and without version-aware references. The source transformations for this were done separately before measuring the execution times.

Results Figure 6.3 shows how much more time the benchmarks take when their source is transformed before execution and references are, therefore, version-aware. Executing individual benchmarks takes between 90 and 405 times longer with version-aware references than without. On average the execution is slowed down by a factor of 187.5.

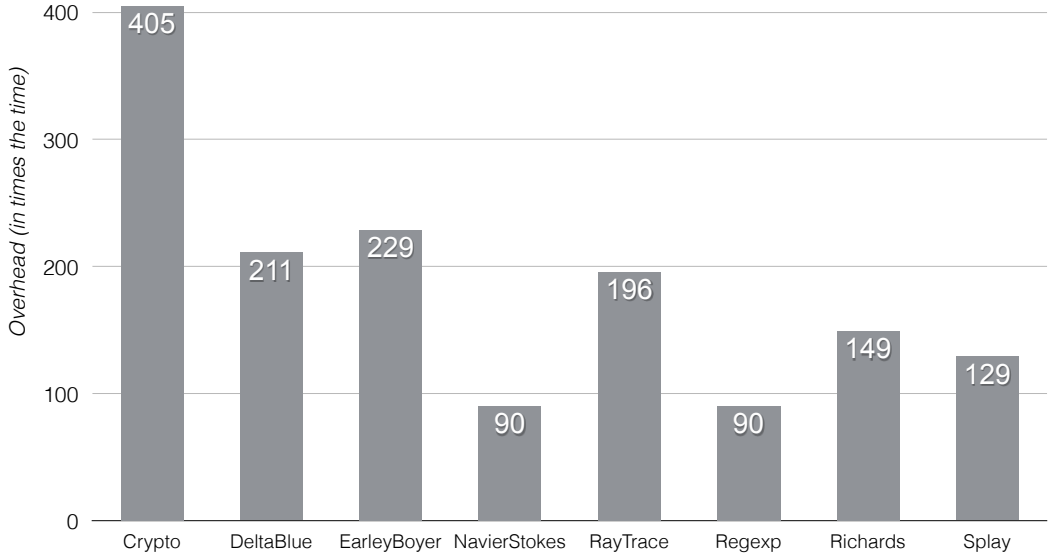

Figure 6.3: Execution overhead for the Octane benchmark suite.

Microbenchmarks

We implemented a microbenchmark to measure the overhead the proxies impose on resolving references. In particular, the microbenchmark shows how much time the proxies require to intercept and forward property reads to the single version of an object.

[7]Note: We reduced the input size of the Splay benchmark by an order of magnitude to prevent the browser from prompting for user input during the benchmark's execution. The prompt is triggered due to the long time required to run the benchmark. It cannot be disabled and would influence the benchmark result.

Method We measured how long it takes to resolve a reference as well as read and call a function property a million times. The reference connects a `client` object to a `server` object. The code of which we measured the execution time is shown in Listing 6.1.

```
for (var i=0; i < 1000000; i++) {
    client.server.foo();
}
```

Listing 6.1: Code we measured for the microbenchmark.

We compared the execution times of three different setups:

SETUP 1 The `client` object holds a reference directly to the `server`.
SETUP 2 The `client` object holds a proxy as its `server` property. In this setup, we used the proxy handler described in Section 5.1.2 for the proxy. The proxy has access to the actual `server` object as one of its version objects. It selects the `server` object when it intercepts the property read.
SETUP 3 The `client` object's `server` property is also a proxy but one created with a fixed target and without proxy handler. The fixed target is the `server` object to which the proxy then forwards by default.

In all setups, the `server` object holds a reference that directly refers to the `foo` function.

Results Table 6.1 shows the results of running the microbenchmark in the three setups. Using a proxy with our proxy handler takes three orders of magnitude more time than using an ordinary reference does: Instead of on average 10 milliseconds the test requires on average about 11000 milliseconds to finish. The difference between *Setup 3* and *Setup 1* is an order of magnitude less: 2000 milliseconds compared to 10 milliseconds. This shows that even a proxy with a fixed target and the default proxy behavior slows down the execution of the microbenchmark close to 200 times.

Setup 1	10 milliseconds
Setup 2	11000 milliseconds
Setup 3	2000 milliseconds

Table 6.1: Times to run the three setups of the microbenchmark.

6.4.2 Execution of the Lively Kernel

We measured the overhead imposed on three typical user interactions and how much longer it takes to load a Lively Kernel world with proxy-based version-aware references.

Typical Lively Kernel Interactions

As our goal is to provide recovery support for development in Lively Kernel, the overhead imposed on user interactions is especially interesting as it directly affects developers.

Method We measured the time three user interactions take when using proxies and compared this to the time the interactions usually take. We measured the time from the user events until the single-threaded JavaScript engine becomes responsive again programmatically. The three typical interaction we chose to investigate are: bringing up the halo buttons on a particular morph, opening the Lively Kernel's main menu, and opening the Lively Kernel's System Code Browser.
We chose these three interactions as we expect them to be more impacted by the version-aware references compared to interactions that are more browser-supported and less reliant on the execution of JavaScript code such as dragging elements around the screen. All three interaction trigger code from multiple different modules, including event handling code, rendering code, and tool-specific code.

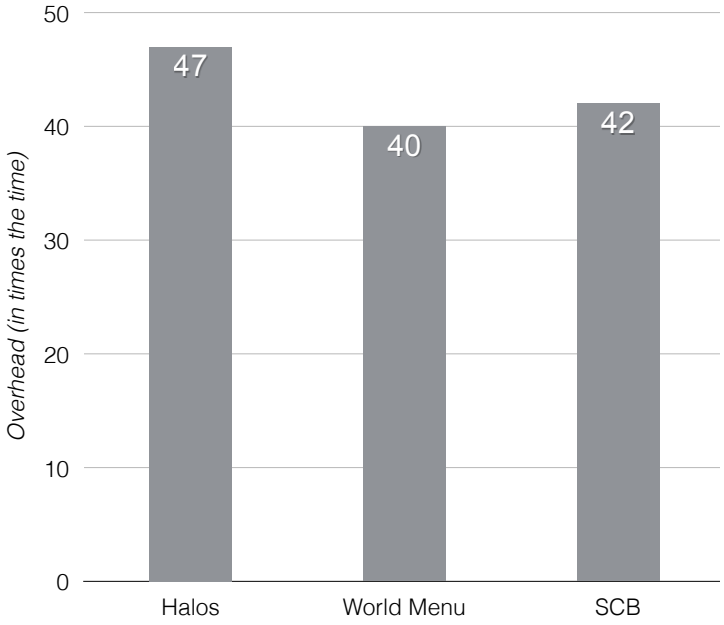

Figure 6.4: Execution overhead for three user interactions in the Lively Kernel.

Results Figure 6.4 shows the results. Each of the three interactions takes on average 43 times the time when triggered after the system was loaded with proxies.

Loading a Lively Kernel World

Another performance-related question specific to the Lively Kernel is how long it takes to load a world with proxies.

Method We measured how long it takes to load a specific Lively Kernel world with and without source transformations and, thus, proxies. Loading a world includes requesting the required modules from the Lively Kernel's server, client-side code to resolve dependencies among those modules, evaluating the code of the loaded modules, and deserializing the graphical state of the world's scenegraph. Additionally, in case proxies should be used, the sources of all modules also are transformed while loading the world.

Results It takes eight times more time to load a world with object versioning: instead of around 4 seconds, the user would have to wait around 32 seconds until the world becomes responsive.

6.4.3 Discussion of the Execution Overhead

The results of our evaluation show that the execution overhead is currently impractical. The Octane benchmarks indicate that executing real JavaScript programs takes two to three orders of magnitude more time. Similarly, the Lively Kernel tools are significantly less responsive.
Even though we expected a certain execution overhead with our approach, the current overhead is too high.

The microbenchmarks show that a considerable part of the overhead is introduced by using the ECMAScript 6 proxies. Even when these proxies are used to forward to a fixed target instead of a dynamically chosen target, they introduce a substantial overhead: It takes 200 times the time to have a proxy intercept and forward property reads than it takes to read a property after resolving an ordinary reference.
For this reason, we still consider our approach feasible for providing object versioning for the Lively Kernel. However, the performance of our current implementation needs to be improved before it provides practical recovery support to programmers.

7 Related Work

This chapter presents two categories of related work:

1. Approaches related to our motivation and, thus, to providing access to previous versions of the system state.

2. Approaches related to our technical solution and, thus, to combining changes into first-class objects that can be used to scope changes.

7.1 Recovering Previous System States

The approaches presented in this section support programmers in recovering previous states without requiring programmers to create snapshots in advance.

7.1.1 CoExist

CoExist [28] provides recovery support through continuous versioning in Squeak/Smalltalk. For each change made to source code, CoExist creates a new version of the system sources, resulting in a fine-grained history of changes. CoExist presents this history in a timeline tool and a dedicated browser. For each version, those tools show the changes, test results, and a screenshot. Developers can recover previous development states, even without taking precautionary actions beforehand. This way, developers can concentrate on implementing their ideas and let CoExist record the required versions to be able to recover when necessary.

Both CoExist and our approach to object versioning allow multiple versions of the development state to coexist. With both approaches, preserved versions are part of the program runtime and can be re-established easily. Currently only CoExist records versions continuously on the granularity of changes made by developers. CoExist provides much more tool support to find and recover changes from previous versions. However, CoExist

recognizes only changes to the source code of classes, while our system preserves the state and behavior of objects.

7.1.2 Lively Kernel Offline Worlds

Offline Worlds [4] is an approach to protect state against system failures by saving it periodically. In a fixed time interval the current Lively Kernel world is saved automatically as protection against unexpected crashes and network outages. The implementation serializes the state and object-specific behavior of all morphs. For each serialized state only the differences to the previous state is stored. Further, the implementation uses client-side storage for fast access and to safeguard against outages of both the client-side and the server-side. As only the differences to the last saved version is stored and previous versions, therefore, remain available, Offline Worlds could, conceivably, also be used to re-establish other versions than the latest, but does not provide support for this.

Offline Worlds preserves the latest state of a Lively Kernel world to support recovery from system failures, while our approach preserves multiple versions of the runtime to provide recovery when programmers make inappropriate changes. Offline Worlds only preserves the state of all graphical objects. In contrast, our system also recognizes changes to classes, globally accessible state, and morph state explicitly excluded from serialization. Further, our approach saves versions incrementally and re-establishes versions dynamically, while Offline Worlds saves and loads versions of the world in discrete, interruptive steps. Even when not used to recover from a system failure, Offline Worlds still requires to re-load the entire world, while our system only switches which versions of particular objects should be used and even preserves object identity of these through the version-aware references.

7.1.3 Back-in-Time Debugging

Back-in-time Debuggers [16], also known as *Omniscient Debuggers*, allow developers to inspect previous program states and step backwards in the control flow to undo the side effects of statements. Approaches for this are either based on logging or replay: either the debugger records information to be able to recreate particular previous situations, requiring mainly space for the different states, or the debugger re-executes the program up to a particular previous situation, requiring mainly time to re-run the program. While many logging-based approaches introduce significant execution overheads, replay-based approaches have to ensure that the program is re-executed deterministically, which can be a difficult problem when, for example, programs can rely on state outside of the program runtime such as the content of files or the state of other programs.

Our approach is more related to logging-based back-in-time debugging. It allows re-establishing a previous state through preserving information. However, back-in-time debuggers need to be able to undo the effects of each statement separately, while our system's versioning granularity is arbitrarily and can, for example, correspond to programmer interactions with the system. In general, back-in-time debuggers support a particular development task—debugging—and, thus, are also often only active when a program is started in a separate debugging mode. In contrast, the purpose of object versioning is more comprehensive. We expect object versioning to be active at least during all development tasks, but possibly even be enabled at all times.

7.1.4 Software Transactional Memory

Software Transactional Memory (STM) [27] captures changes to values in transactions, analogous to database transactions. Each transaction has its own view of the memory, which is unaffected by other concurrently running transactions. Multiple versions of the system state can coexist and which version is read and written to depends on the transaction. Transactions contain a number of program statements that are executed atomically. The changes from a transactions are only permanent when no conflicts occur with other transactions. On conflicts, all changes from the transaction are rolled back and undone.

STM and our approach are similar in that multiple versions of the system state can coexist and that a previous state can be re-established if necessary. However, STM provides concurrency control and an alternative to lock-based synchronization, while our approach provides recovery support to developers when changes turn out be inappropriate. STM transactions are automatically rolled back when changes conflict with changes from other concurrently running transactions, while our versions are offered to programmers to undo changes when necessary. Programmers can actively decide to undo changes when these, for example, negatively impact the functionality, design, or performance of programs. Our versions of the runtime are also first-class objects, which can be stored in variables and be re-established at any time, while transactions are always created implicitly through particular control structures and commited immediately upon success.

7.2 Dynamically Scoping First-class Groups of Changes

The approaches presented in this sections allow to combine changes into first-class objects and run code with particular sets of changes.

7.2.1 Worlds

Worlds provide a language construct for controlling the scope of side effects: changes to the state of objects are by default only effective in the *world* in which the changes occurred. These worlds are first-class values and can be used to execute statements with particular side effects being active. A new world can be spawned from an existing world, which establishes a child-parent relationship between the two worlds. Developers can commit changes from a child world to its parent world, thereby extending the scope of the captured side effects. The Worlds approach includes conditions that prevent commits that would potentially introduce inconsistencies.

In comparison, Worlds provides a language construct for experimenting with different states of the system, while object versioning allows to preserve versions of the system to recover previous states: Our approach does not include extensions to the host programming languages and no conditions for combining versions with their predecessor versions, but provides a basis for CoExist-like continuous versioning and recovery tools.

Other differences between Worlds and our approach regard the implementations. Our implementation in JavaScript does not prevent garbage collection as Worlds does. Further, both use different libraries for source transformations. Our source transformations are faster and do not use JavaScript exceptions.

7.2.2 Object Graph Versioning

Object Graph Versioning[26] allows programmers to preserve access to previous states of objects. Fields of objects can be marked as *selected* fields. When a snapshot is created, the values of these selected fields are preserved. Therefore, not every state can be re-established, but states that are part of global snapshots. The approach, thus, provides fine-grained control to programmers regarding which fields of which objects should be preserved when.

The technical solution is similar to our design. Analogous to our proxy-based version-aware references, selected fields do not refer directly to their actual values, but to chained arrays that manage multiple versions of the state of a field and delegate access to the current version transparently. The chained arrays decide which version to retrieve and when to create new versions using a global version identifier. In constrast to our sulution, individual fields are versioned and only when programmers explicitly mark them as selected.

Object Graph Versioning aims to support implementing application-specific undo/redo or tools like back-in-time debuggers. In contrast, our approach to object versioning

aims to support recovery of previous system states during the development of arbitrary applications.

7.2.3 Context-oriented Programming

Context-oriented Programming (COP) [10, 1] adds dedicated language constructs for dynamic behavior variations. Depending on context information, COP allows to enable and disable layers, which contain methods to be executed instead or around methods of the base programs. Context information can be any information which is computationally accessible. Layers can be enabled and disabled at runtime. Different implementations of COP provide different mechanisms to scope the activation of layers: for example, layers can be activated explicitly for a particular scope or globally for the entire runtime. ContextJS [20] it is possible to activate layers for specific objects.

In comparison to our approach, COP allows to activate combinations of layers, while our system executes code using a single active version. In COP layers are indepedent, while our versions are predecessors and successors of each other.

COP aims at supporting the separation of heterogeneous cross-cutting concerns, while object versioning aims at supporting developers with the recovery of previous states. However, [21] showed that COP can also be used to experiment with changes to a system: developers can implement experimental changes to behavior in layers, not to modularize context-dependent adaptions, but to be able to scope changes dynamically and recover the original system behavior easily. However, this requires programmers to make experiments explicitly. They need to use layers for their adaptions, enable the layers for test runs, and move code from layers back to the base system when experiments are successes and they want to maintain the original modularization of the system. COP also allows only behavior variations, while our approach recognizes changes to both state and behavior.

7.2.4 ChangeBoxes

ChangeBoxes [5] is an approach to capturing and scoping changes to a system using first-class entities, called ChangeBoxes. A ChangeBox can contain changes to multiple elements of a software system such as adding a field, removing a method, or renaming a class. The approach does not constrain how changes get grouped into ChangeBoxes, but every change has to be encapsulated by a ChangeBox. Each ChangeBox can be used for setting the set of active changes for the scope of a running process. This way, multiple running processes can view the system differently by using different ChangeBoxes. ChangeBoxes can have ancestor relations and merge changes from multiple ancestors.

With the ancestor relations, ChangeBoxes can be used to review the evolution of systems and to undo changes.

The ChangeBoxes approach is similar to our approach as changes to the system are grouped into first-class objects and these can be used to run code in different versions of the runtime. Furthermore, with both solutions there is no definite notion in how changes are grouped into versions. Our object versioning approach is intended to be used to group changes associated with developer actions and a simple global undo/redo mechanism to undo inappropriate actions is built into our solution. To actually undo changes ChangeBoxes, in contrast, is rather tedious [28]. Moreover, ChangeBoxes recognizes only changes to the static elements of a software system such as packages, the structure of classes, and methods. Object versioning, in contrast, preserves the state and behavior of objects.

7.2.5 Practical Object-oriented Back-in-Time Debugging

Practical Object-oriented Back-in-Time Debugging [19, 18] is a logging-based approach to back-in-time debugging that uses alternative references to preserve the history of objects. These alternative references, called *Aliases*, are actually objects and part of the application memory. These objects contain information about the history and origin of the values stored in fields. Aliases are not passed around, but instead are created for each read or write of a field and for each value passed as parameter. Each alias refers to an actual value, but also to another alias—its *predecessor*—representing the value previously stored by a field and to the alias that was used to create this new alias from—its *origin*. An alias and its origin both refer to the same value, but provide different information on their creation context, which is a particular method. The origin link can be used to follow the object's "flow" through the program. Each alias also records a timestamp on its creation and with this information the predecessor link can be followed to read a value as it was at a particular moment in time.

In comparison, with aliases it is possible to recreate all states the system was in and also retrace the flow of all values, while our system stores only particular versions. Such versions could, for example, correspond to programmer interactions, so that programmers can undo the effects of particular actions easily. Another difference between object versioning with version-aware references and reverse engineering with aliases is the existence of modes.The alias references are intended to be used in explicit debugging sessions, while our version-aware reference are intended to be used at all times.

8 Future Work

In the future, we would like our solution to become more practically useful. As described in this chapter, this could be achieved by improving the performance of our implementation and providing tool support.

8.1 Improving the Performance

Our current implementation introduces a significant execution overhead as presented in Section 6.4.

The version-aware references resolve to versions of objects dynamically: the correct version is selected the moment the version-aware references are resolved. Even though optimizations such as caching the current versions are possible, a certain execution overhead is to be expected with this approach.
However, our evaluation showed that most of the current overhead is introduced by the proxies we used for implementing version-aware references. Even when these proxies are configured to forward all interactions to a fixed target, it takes 200 times more time to have a proxy forward a property read than to read the property directly from the target.

There are three different approaches to this performance problem:

- **Waiting for faster proxies**: The proxies we used are not yet fully supported by the JavaScript engines and it seems reasonable to expect better performance in the future.

- **Using fewer proxies**: Proxies could be used only for the system parts for which state should be versioned.

- **Implementing an alternative to proxies**: Instead of using proxies, version-aware references could be implemented differently. A similar indirection could be provided using source transformations and ordinary JavaScript functions.

Waiting for Faster Proxies

Our implementation uses the proxies that the ECMAScript 6 standard will add to the JavaScript language. The ECMAScript 6 specification has not yet been finalized. The current draft can be used in Chrome and Firefox, but is not fully implemented by their respective JavaScript engines. Instead, the proxies are currently provided partly by a JavaScript library and partly by the JavaScript engines. In the future, the proxies will be implemented fully by the JavaScript engines. This will likely reduce the execution overhead.
Furthermore, it seems reasonable to assume that the parts already implemented by the engines have not yet been optimized. It is, after all, an experimental feature that has not yet been officially added to JavaScript.

Using Fewer Proxies

We could use proxies less deliberately. The state of some parts of the system could be excluded from versioning if access to previous states of such parts is not required. Moreover, there are even objects for which predictably only one version will exist.

For example, one system part that could be excluded from versioning is the Lively Kernel's *OMeta* [38] parser. The parser is, for example, used by to check for syntax errors before changes to code can be saved. It creates many objects while parsing code. Therefore, parsing takes much more time when all object interactions go through proxies. Many objects capture intermediate states of the parser, while in the end often only a success or failure needs to be returned. Given JavaScript's single-threaded, cooperatively scheduled execution it is not possible to switch versions during parsing. There would not be multiple versions of the objects that are only available while the parser runs.
However, the parser could return objects as results or otherwise make objects available to other system parts. These objects would have to be wrapped into proxies before becoming part of the versioned system state.

Another option would be to use object versioning only during development. This way, applications could run without the overhead of the proxies when versioning is not required. Our implementation introduces the proxies using source transformations on load-time. A Lively Kernel world can be loaded with source transformations to be able to preserve versions. The world can be saved and reloaded without source transformation to have it run at full speed. Appropriate tool support could allow users to switch whether versioning should be active for specific worlds.

Implementing An Alternative to Proxies

Version-aware references could be implemented without using proxies.

Ordinary JavaScript functions could be used to carry out object interactions on the right versions of objects. These functions could be similar to the traps of our proxy handlers, presented in Section 5.1.2. For example, a `get` function could allow reading a property from the current version of an object. The `get` function could be implemented similarly to the function shown in Listing 8.1.

```
1 function get(standIn, propertyName) {
2     var version = lively.getCurrentVersionOf(standIn);
3     return version[propertyName];
4 }
```

Listing 8.1: A function for reading a property from the correct version of an object.

The first parameter to this `get` function would be an ordinary object that stands in for the versions of an object. The `getCurrentVersionOf` function in Line 2 of Listing 8.1 uses this `standIn` parameter to retrieve the current version of an object. The `standIn` object could hold the versions of an object or be a key to a dictionary.

Functions like the `get` function could be inserted automatically by source transformations. The source transformation necessary to read an `age` property from a version of a `person` object could be as exemplified by Table 8.1.

Input	Output
`person.age`	`get(person, 'age')`

Table 8.1: Transforming a property read.

Other kinds of object interactions could be handled in similar functions. For example, an `apply` function could apply a version of a function.

To call a `dance` function of a `person` object in a version of the system, two steps are necessary. First, the `dance` property has to be read from the right version of the `person`. Second, the right version of the `dance` property, which is expected to be a function, needs to be applied. Therefore, calling a function of an object would require to insert the `get` function and the `apply` function, as exemplified by Table 8.2.

When the `dance` function is applied as method of the `person` object, the `this` keyword needs to refer to the right version of the `person` object. For this reason, the `apply` function is called with the `person` stand in in this example.

Input	Output
`person.dance()`	`apply(person, get(person, 'dance'))`

Table 8.2: Transforming a method call.

Discussion of the Approaches

Of the three approaches, using an alternative to proxies seems most promising.
Using proxies selectively for specific system parts or worlds would only be sufficient if these would not require performance improvements.
Waiting for a faster proxy implementations is an option, but there is not even an official release date for the ECMAScript 6 specification yet.
At the same time, early performance tests indicate that the alternative implementation of version-aware references could be faster. In particular, microbenchmarks show that going through a function to read a property of an object is only twice as expensive as reading the property directly.

8.2 Providing Recovery Tools

Our implementation allows to preserve and re-establish versions of the Lively Kernel's state. These versions currently still need to be created explicitly and there are no tools yet to find and manage versions.

8.2.1 Preserving Versions Automatically

With our implementation, programmers need to preserve versions to be able to re-establish them later. Preserving versions is an effort. It is difficult to assess the risk of upcoming changes when deciding whether a state needs to be preserved. Programmers could deliberately decide against preserving a version after underestimating the risk of changes. They might forget to preserve versions. Furthermore, it is time-consuming to run appropriate tests to ensure that the current state is a good state to preserve.
For these reasons, we want the system to preserve a fine-grained history automatically.

The system could create versions of the runtime for any change to an object. However, even if that were technically feasible, programmers need to be able to find and recognize relevant versions efficiently. Therefore, we propose that the system records versions automatically as proposed by CoExist: preserve a version for each action of a programmer.

The Lively Kernel could automatically preserve versions whenever a developer does any of the following:

- manipulate properties of a morph directly with a halo tool or through drag and drop
- add, remove, or edit a script of a morph or a method of a class
- evaluate a code snippet ("Do-It")
- trigger code execution through a mouse or keyboard interaction

This way, whenever programmers realize changes were inappropriate, they can undo their actions.

8.2.2 Tools For Finding and Managing Versions

The system should support developers in finding and re-establishing relevant states.

Finding Versions

Besides preserving versions continuously on a granularity helpful to developers, we want the system to present helpful information to each version. The system could present three categories of information:

WHEN Versions could be accompanied by a timestamp and be presented in a timelime as in CoExist.
HOW Versions could be annotated with the kind of action that triggered preserving the version such as whether a programmer used a halo button or evaluated a code snippet. This could be supported by recording screenshots or screencasts for versions.
WHAT Versions could store information on what was changed between two version: which objects did change, how these objects changed, and how this affected tests and benchmarks.

Changes can often be associated with static information such as the name of a class, a module, and a containing file. Some objects as, for example, morphs could be related to the scenegraph of visible morphs. Furthermore, morphs can have individual names in the Lively Kernel.

Managing Versions

When developers find a relevant previous state, they might want to use it for different purposes:

Revisiting previous states Programmers might want to re-establish a particular state of the system without making changes. For example, they might want to see how an application behaved at a particular moment to compare that to the current state.
Recovering previous states Programmers might want to recover state from one version in another version. For example, they could want to recover a particular version of an application or the state of a tool such as a browser.
Trying alternatives Programmers might want to try a new idea in an earlier version without loosing neither that version nor any following versions. Therefore, they might want to create a branch as an alternative to the main line of version history.

We want programmers to be able to re-visit versions of the system and to be able to create, merge, and delete lines of history. Additionally, programmers should be able to copy particular objects from one version to another.

9 Summary

This work introduced an approach to preserving access to previous states of programming systems such as the Lively Kernel. The approach is based on version-aware references. These references manage different versions of objects transparently. They resolve to one of multiple versions of objects; to which ones in particular can easily be changed. Thereby, different preserved states can be re-established.

We presented a design for our approach that uses proxies for version-aware references. Instead of actually using alternative references, ordinary references refer to proxies and proxies forward all interactions transparently to the right versions. The design allows implementing version-aware references without any adaptions to existing execution engines—neither for alternative references nor for the garbage collection of versions.
For each object that is created, a proxy is created and returned instead of the object. Thus, references to proxies are passed around and all access goes through the proxies. Moreover, only the proxies refer to the versions of an object. Consequentely, the versions of an object are reclaimed together with their proxy by the ordinary garbage collector.
Returning proxies for new objects is achieved using source transformations. The program sources are transformed when loaded and do not have to be adapted manually.

We implemented our approach to object versioning in JavaScript. The implementation allows preserving and re-establishing versions of the system's state. It is optimized for fine-grained histories.
To switch the version of the system, only a global version identifier has to be changed. Using this identifier, the proxies choose versions of objects dynamically. This way, it is not necessary to re-configure all proxies to switch versions.
To preserve versions of the system, the proxies copy object versions on writes: When a proxy intercepts a mutating operation to an object for which no current version exists, it copies a previous version of the object. Until then, proxies reuse object versions from previous system versions.

We integrated our implementation into the Lively Kernel and made it work with our version-aware references. Users can commit versions of the system state. Using these versions, they can undo and redo changes. This shows that our implementation works correctly in all situations tested. The memory overhead is reasonable. The execution overhead is not yet practical: the execution of eight JavaScript benchmarks takes currently three orders of magnitude more time with our version-aware references.

In the future, the implementation could be improved by reducing the execution overhead. In addition, the system should preserve relevant versions automatically and provide dedicated tools to developers.

Bibliography

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. "A Comparison of Context-oriented Programming Languages". In: *International Workshop on Context-Oriented Programming*. COP '09. ACM, July 2009, 6:1–6:6.

[2] Alan Borning. "Classes Versus Prototypes in Object-oriented Languages". In: *Proceedings of 1986 ACM Fall Joint Computer Conference*. ACM '86. IEEE, Nov. 1986, pp. 36–40.

[3] Tom Cutsem and Mark S. Miller. "Trustworthy Proxies: Virtualizing Objects with Invariants". In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP '13. Springer, July 2013, pp. 154–178.

[4] Martin A. Czuchra. "Offine Worlds: Automated Client–Side Persistence in Lively Kernel". Master Thesis. Hasso-Plattner-Institute, University of Potsdam, Jan. 2012.

[5] 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 '07. ACM, Aug. 2007, pp. 25–49.

[6] Ecma/TS39. *ECMAScript Language Specification (Draft for 6th Edition)*. Published April 27, 2014 (Draft, Revision 24). Available at `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`. Accessed May 11, 2014. 2014.

[7] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. "An Incremental Constraint Solver". In: *Commununications of the ACM* 33.1 (Jan. 1990), pp. 54–63.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Jan. 1995.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Jan. 1983.

[10] Hirschfeld, Robert and Costanza, Pascal and Nierstrasz, Oscar. "Context-oriented Programming". In: *Journal of Object Technology* 7.3 (Mar. 2008), pp. 125–151.

[11] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '97. ACM, Oct. 1997, pp. 318–326.

[12] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. "Fabrik: A Visual Programming Environment". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '88. ACM, Jan. 1988, pp. 176–190.

[13] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The Lively Kernel–A Self-supporting System on a Web Page". In: *Self-Sustaining Systems*. S3. Springer, May 2008, pp. 31–50.

[14] Alan Kay. *Squeak Etoys Authoring and Media*. Tech. rep. Published February 2005. Available at http://www.vpri.org/pdf/rn2005002_authoring.pdf. Accessed March 7, 2014. Feb. 2005.

[15] Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. "Lively Wiki: A Development Environment for Creating and Sharing Active Web Content". In: *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*. WikiSym '09. ACM, Oct. 2009, 9:1–9:10.

[16] Bill Lewis. "Debugging Backwards in Time". In: *Proceedings of the Fifth International Workshop on Automated Debugging*. AADEBUG'03. Springer, Sept. 2003, pp. 225–235.

[17] Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPLSA '86. ACM, June 1986, pp. 214–223.

[18] Adrian Lienhard. "Dynamic Object Flow Analysis". PhD thesis. University of Bern, Dec. 2008.

[19] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming*. ECOOP '08. Springer, July 2008, pp. 592–615.

[20] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. "An Open Implementation for Context-oriented Layer Composition in ContextJS". In: *Science of Computer Programming* 76.12 (Dec. 2011), pp. 1194–1209.

[21] Jens Lincke and Robert Hirschfeld. "Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming". In: *Proceedings of the International Workshop on Context-Oriented Programming*. COP '12. ACM, June 2012, 2:1–2:6.

[22] Jens Lincke and Robert Hirschfeld. "User-evolvable Tools in the Web". In: *Proceedings of the 9th International Symposium on Open Collaboration*. WikiSym '13. ACM, Aug. 2013, 19:1–19:8.

[23] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Röder, and Robert Hirschfeld. "The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content". In: *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*. HICSS '12. IEEE, Jan. 2012, pp. 693–701.

[24] John H. Maloney and Randall B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. ACM, Dec. 1995, pp. 21–28.

[25] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment". In: *Transactions on Computing Education* 10.4 (Nov. 2010), 16:1–16:15.

[26] Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. "Executing Code in the Past: Efficient In-memory Object Graph Versioning". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. ACM, Oct. 2009, pp. 391–408.

[27] Nir Shavit and Dan Touitou. "Software Transactional Memory". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. ACM, June 1995, pp. 204–213.

[28] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. "CoExist: Overcoming Aversion to Change". In: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS '12. ACM, Jan. 2012, pp. 107–118.

[29] Bastian Steinert and Robert Hirschfeld. "How to Compare Performance in Program Design Activities: Towards an Empirical Evaluation of CoExist". In: *Design Thinking Research*. Understanding Innovation. Springer, Jan. 2014, pp. 219–238.

[30] Antero Taivalsaari. "Classes vs. Prototypes-Some Philosophical and Historical Observations". In: *Journal of Object-Oriented Programming* 10.7 (Apr. 1996), pp. 44–50.

[31] Antero Taivalsaari. "Delegation Versus Concatenation or Cloning is Inheritance Too". In: *SIGPLAN OOPS Messenger* 6.3 (July 1995), pp. 20–49. ISSN: 1055-6400.

[32] Antero Taivalsaari. *Kevo, a Prototype-based Object-oriented Language Based on Concatenation and Modules Operations*. Tech. rep. Technical Report LACIR 92-02, University of Victoria, 1992.

[33] David Ungar and Randall B. Smith. "Self". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, June 2007, 9:1–9:50.

[34] David Ungar and Randall B. Smith. "Self: The Power of Simplicity". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '87. ACM, Dec. 1987, pp. 227–242.

[35] Tom Van Cutsem and Mark S. Miller. "Proxies: Design Principles for Robust Object-oriented Intercession APIs". In: *SIGPLAN Notices* 45.12 (Oct. 2010), pp. 59–72.

[36] Alessandro Warth. "Experimenting with Programming Languages". PhD thesis. University of California, Los Angeles, Dec. 2009.

[37] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. "Worlds: Controlling the Scope of Side Effects". In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. Springer, July 2011, pp. 179–203.

[38] Alessandro Warth and Ian Piumarta. "OMeta: An Object-oriented Language for Pattern Matching". In: *Proceedings of the 2007 Symposium on Dynamic Languages*. DLS '07. ACM, Oct. 2007, pp. 11–19.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den 13. Mai 2014 _____

Lauritz Thamsen