

Safe Dynamic Software Updates in Multi-Threaded Systems with ActiveContext

Masterarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

David Wendelin Gurtner

April 2011

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Erwann Wernli

Toon Verwaest

Institut für Informatik und angewandte Mathematik

ACTIVE
C2NTEXT

Copyright © 2011 by David Gurtner



This thesis is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/> for more information.

Abstract

Applications need to be updated. The traditional approach to stop and consequently restart an application for an update is not a valid scenario in the case of high availability environments—they need to be updated at runtime. The reflective capabilities of dynamic languages offer a convenient way to install updates at runtime, but do not provide adequate mechanisms to ensure safety. They suffer from (a) the lack of a state transfer mechanism, and (b) possible inconsistent data accesses from old code to new data structures.

Existing dynamic software update systems provide solutions for safe dynamic updates but do not scale well to multi-threaded systems. To address these issues, we propose ActiveContext, a programming model to enable dynamic software updates in multi-threaded systems. ActiveContext builds on the reflective capabilities of dynamic languages and adds first-class contexts to support the co-existence and synchronization of multiple versions of objects in memory to ensure safety.

We implemented ActiveContext in Pinocchio, a variant of Smalltalk, and built a proof-of-concept dynamic update system leveraging ActiveContext. We demonstrate the viability of our approach with a running example.

Acknowledgements

First and foremost I thank Erwann Wernli who made this thesis possible with his ideas, his work, and his continued support. I enjoyed the many pair programming sessions and the discussions held over many a cup of coffee—you not only helped me realize this project, but you made it fun. I thank Toon Verwaest, who holds an equal part in making this thesis possible. You always had an open door for me and were always ready to help me with your amazing technical knowledge.

I wish to thank Prof. Oscar Nierstrasz as much for giving me the opportunity to write this thesis at the Software Composition Group as for the support I received throughout my studies. His inspirational lectures are the reason for my ongoing interest in software engineering and programming languages.

My gratitude extends to the entire SCG and all my colleagues, you all contributed to this work with your knowledge, advice and help. I would like to specifically thank Philipp Bunge for his help with grammar and on style.

To my friends, I am deeply grateful and happy to call you my friends. Without you I would have never been able to make it through uni. You are what made my life on and off campus a great experience and incredibly fun.

To my parents, my siblings, Seul and all of my family. There is no way to adequately express my gratitude for the love and unconditional support I received not only in my studies but my entire life.

Thank you!

Contents

1. Introduction	1
1.1. Problem Statement	1
1.2. Contributions and Thesis Structure	2
2. Dynamic Software Update	3
2.1. Motivating Dynamic Software Update	3
2.2. An Introduction to Dynamic Software Update	4
3. Related Work	7
3.1. Approaches in C	7
3.2. Approaches in Java	8
3.3. Approaches in Smalltalk	11
4. ActiveContext in a Nutshell	15
4.1. A Motivational Example	15
4.2. A Step by Step Update	16
5. The ActiveContext Model	19
5.1. Identity, State and Contexts	19
5.2. State Synchronization	20
5.3. Reflective Hook	21
6. Implementation	23
6.1. Pinocchio	24
6.2. Implementation	25
7. Validation	35
7.1. A Running Example: Telnet Server	35
7.2. The Code of the Telnet Server	38
8. Discussion	43
8.1. Analysis of ActiveContext	43
8.2. Limitations of the Implementation	45
9. Conclusion	47
9.1. Future Work	47

Contents

9.2. Final Remarks	48
A. Installation	49
A.1. Prerequisites	49
A.2. ActiveContext	50
List of Tables	53
List of Figures	55
Bibliography	57

Chapter 1

Introduction

1.1. Problem Statement

Software needs to be updated: Apart from the need to continuously evolve to support new and possibly unanticipated features, there is also a need to fix existing bugs.

The process of updating software consists of two parts. First the update is implemented, and then the update is installed. Typically such updates are installed while the application is not running, i.e. the application is stopped, the update is deployed and the application is then restarted.

For certain applications this is not a viable scenario. For ISPs, banks, online stores, online news outlets and others, downtime creates an immediate financial loss. In the case of life-support systems or air traffic control it might even lead to a danger for human life [Segal and Frieder, 1993]. Hence, changes have to be rolled out at runtime.

While reflective languages like Smalltalk give the programmer capabilities to change a program on the fly by changing the meta-level, these techniques are not appropriate for production systems. The two main shortcomings are:

- Fields can be added or removed from existing classes, but corresponding objects need to be updated accordingly. So the old state needs to transition into a new, valid state. This lack of support for *state transfer* prevents arbitrary software evolution.
- Old code on the stack might get out of sync with updated data structures, leading to inconsistent data access. This can result in *unsafe* program execution.

There is a lot of existing research in the field of dynamic update systems which

solves these issues for non-reflective systems. These solutions usually revolve around creating tools that provide a way to replace behavior (methods, functions or procedures depending on the language), and then finding update points where it is safe to update the program. From a high level perspective, these systems work according to the following scheme [Ebraert *et al.*, 2005]: (1) preparation of the change, (2) dynamic addition of new code, (3) deactivation of impacted entities, (4) transformation to new behaviour and transformation of state, (5) verification and (6) reactivation of the entities concerned.

Because of this general approach to finding update points where the impacted entities could be deactivated, most of these systems are ill suited to support updates for multi-threaded applications, as it becomes incrementally harder to find a common update point for all the running threads.

In this work we present ActiveContext, a reflective approach to safe dynamic software updates in multi-threaded systems. Key to our approach is the scoping of updates on a per-thread level which alleviates the need to find update points, and the clear separation of identity and state with built-in support for state transition. ActiveContext is a generic framework providing scoping and state transition.

1.2. Contributions and Thesis Structure

The main contributions of this thesis are as follows:

1. An Introduction to dynamic software update systems and a review of existing research.
2. The presentation of ActiveContext, an approach to dynamic software updates in multi-threaded environments.
3. A proof-of-concept implementation of ActiveContext together with a running example to show the viability of ActiveContext.
4. The discussion and an outlook for further research around ActiveContext

We will start with an in-depth look at dynamic update systems in Chapter 2 and review existing research in Chapter 3. We will give a motivational example how ActiveContext enables dynamic software updates, by going through a step by step dynamic software update, in Chapter 4. Following in Chapter 5 will be a more formal introduction to ActiveContext and its model. The implementation of ActiveContext will be presented in Chapter 6. Next, we continue with a validation of our approach, based on a running example in Chapter 7. In the following Chapter 8 we discuss our approach. Finally we conclude in Chapter 9.

Chapter 2

Dynamic Software Update

After sketching out the idea of dynamic software update systems in the introduction, we present an in depth look at dynamic software systems. We motivate the use of dynamic software update systems by looking at where they can be used and what the alternatives are. We present the different possible features of such dynamic update systems, together with resulting drawbacks and considerations.

2.1. Motivating Dynamic Software Update

Software is not a static product. Software constantly changes, new features are implemented as well as bugs fixed. New versions of software containing new features or bug-fixes need to be installed to leverage those features.

Installed software is a static artifact residing on a hard disk, while the running application is held in memory. To gain the benefit of a new version of the software, it is not enough to install the update and replace the static artifact on the hard disk. The application running in memory needs to be updated too. Traditionally this is achieved by stopping and restarting the application, which reloads the software from the hard disk into the memory, loading the newest version.

While this is a valid scenario for many use cases there exist at least two kinds of systems where this is not viable: (1) software available over the internet, possibly being used by a multitude of different users simultaneously such as in search engines, banking software, online news outlets or the software of an internet service provider itself, (2) critical steering or guidance systems like life support systems, air traffic control, power plant steering and similar systems, where downtime could even pose a threat to human life [Segal and Frieder, 1993].

Such systems are in need of updates too, possibly even more so than other software systems, and a possible way to deploy updates is necessary. One possible solution

is a hardware based approach where multiple hardware systems run the same software, offering the same service in parallel. In such a scenario it is possible to take part of the hardware off-line, install the update and turn it back on, without having a downtime in the overall system. This solution is not always feasible because not all software is designed to be distributed over multiple machines. Specifically if the application depends on shared memory which needs to be synchronized among multiple instances, as opposed to having a transaction based, hierarchical architecture where no memory is directly shared among parallel instances of a system. Another downside of this approach is the increased cost of hardware as well as electrical power, together with increased cost for additional hardware operation staff. The second solution is to change software systems, or on a more basic level, programming languages, to support dynamic update of software, where the update can be loaded directly into memory and the already running system can be updated without a need for a restart.

2.2. An Introduction to Dynamic Software Update

The minimal requirement for dynamic software update is the ability of the host programming language to load new code at runtime. Such a feature is available in most modern programming languages. In C/C++ this is done via the `dlopen` library in UNIX-like operating systems like Mac OS X, Linux and Solaris, while the Windows operating system provides dynamic loading through the Windows API. Java allows programmers to dynamically load classes via the `ClassLoader` object. Fully reflective languages like Smalltalk provide an integrated development environment (IDE) as part of every running instance and adding code dynamically is the default way to introduce changes.

In dynamic software updates there exist different degrees of possible *behavior change*. The simplest way of adapting an application is to change the implementation of a *method body*, *i.e.*, updating the method body to a new version without changing the overall application, for example if a bug inside a single method gets fixed, or a faster algorithm with the same functionality gets deployed. A next step towards arbitrary updates is the ability to change the *method signature*, where not only the internals of a method, but also the method signature itself, *i.e.*, the number and types of parameters, the return type or the method name get changed. The last step towards arbitrary changes and fully dynamically updateable systems is the support for changing global fields and fields inside of structures, for example in the case of class-based systems the fields of objects as defined by their respective classes.

If fields are added or removed from existing classes, corresponding objects need to be updated accordingly. Otherwise the existing fields of the object in memory

might differ from the fields the application expects upon checking the definition of fields in the class. As a result, it might come to *inconsistent data access*, where objects might end up in a state which violates the class invariant. To keep this from occurring, the old state needs to transition into a new, valid state. This is called *state transfer*. The two possible levels of state transfer are *basic state transfer* and *custom state transfer*. Basic state transfer means that new fields get initialized to `nil`, to prevent access to undefined memory. This might be sufficient for changes where new fields get added and the fields can have `nil` values, but basic state transfer does not generally ensure the class invariant. Custom state transfer allows the program to have arbitrary, possibly user provided functionality to initialize new fields. This makes changes like a merge, split, or a renaming of a field between two versions of an application possible and guarantees the class invariant to hold. In non-class-based systems a similar mechanism is needed if global fields or fields in structures like `structs` in the C programming language are changed.

Another issue resulting from the possibility to change the shape of methods and objects is *unsafe access*. If there is code on the call stack corresponding to methods or objects which get updated, this code might point to methods which no longer exist or have different names and signatures, incompatible return types, or try to access fields which no longer exist. This can lead to program failure in the worst case. This is even more of a problem in multi-threaded systems, where all threads have their respective call stacks and the chances for unsafe access are much higher. Dynamic software update systems provide different solutions to solve the issue of unsafe access.

The most simple one is to wait until none of the code to be updated is on the call stack, a so called *update point*, and then pause the application, deploy the update and transfer the state, before continuing program execution. A drawback of this approach is that such an update point might never be reached because some methods are always on the call stack, for example if the code contains long running loops which are never exited. As a consequence the update might *fail to deploy* altogether, because it is waiting for an update point infinitely. This approach is also not optimal for multi-threaded applications, because the chances for code to be on the call stack of any of the threads is much higher.

An alternative approach is to make the update atomic and do an *immediate activation*, where all code, including the one on the call stack, gets replaced at once and the call stack is reconstructed to represent a valid program state of the updated software system. The drawback here is the high complexity to generate a valid program state.

Some systems also allow for *reflective activation*, where the update is triggered from inside the application, rather than by an outside modification. Reflective activation is a kind of white box approach, where the update system knows the

Chapter 2. Dynamic Software Update

running application and can anticipate possible update points and ensure safety and consistent data access. A variation of this strategy is the manual reflective activation of the update by the programmer at a point in the application where he knows the update to be safe, because he has intimate knowledge of the internals of the application.

It is also possible to use *scoping* to permit changes to parts of the application as opposed to only allow for changes to the application as a whole, which can be used to leverage the previous approaches.

We will look at existing research next and discuss it with regard to the concepts introduced above.

Chapter 3

Related Work

In this chapter we present an overview of what we believe to be the most influential research in the field of dynamic software update. We give a short overview of the different approaches, introducing their core concepts and discuss what kind of dynamic software updates they support. Table 3.1 and Table 3.2 present a full comparison.

Different programming languages provide a rather different set of features and infrastructure, which leads to different requirements for dynamic update systems. We will therefore introduce the existing research separated by programming language.

3.1. Approaches in C

Ginseng

Ginseng [Neamtiu *et al.*, 2006] is an approach to dynamically update single threaded C programs, and consists of three parts: a *runtime*, a *compiler* and a *patch generator*. The compiler generates a special updateable executable which will run on top of the runtime. The patch generator can compare the currently running version of the program with the next version, and output a special patch. The compiler compiles this into a dynamic patch which the runtime can load dynamically to update the application it is running.

The Ginseng compiler analyses the program to discover changes of type definitions and constrains the updates to certain specific update points, when none of these definitions are in use. The runtime updates the program at the next available update point.

Ginseng allows for arbitrary method and signature change. Furthermore, custom type transformations can be specified by programmers. Updates on the other hand

are not immediate.

UpStare

UpStare [Makris and Bazzi, 2009] is geared towards updating single and multi-threaded C programs. Similar to Ginseng, UpStare consists of a runtime, a compiler and a patch generator. Additionally UpStare provides an *update control tool*, to initiate the update.

UpStare is based on *stack reconstruction*: it allows applications to unroll the call stack when an update is triggered and reconstitute it by replacing functions with their updated versions.

The stack reconstruction approach allows for updates to be immediately active, with no possible activation failure. Updates also allow for arbitrary updates of method code and signatures and support custom transformation functions for state.

POLUS

The focus of POLUS [Chen *et al.*, 2007] is to support updates to multi-threaded systems, where the changes involve state. POLUS achieves this by not having update points, but by allowing the coexistence of old and new state and calling synchronization functions on the state whenever a write access happens.

POLUS uses a *patch constructor* to generate a patch out of the current and new version, and compiles it with a standard compiler. Furthermore it uses a *patch injector* to insert the patch into the running system. Patches contain the necessary code to maintain state consistency among threads when they manipulate shared data.

POLUS allows for immediate updates, by allowing the coexistence of old and new versions after an update, possibly using custom state transformation functions to ensure consistency. It is possible to update method implementations and signatures.

3.2. Approaches in Java

Hotswap

HotSwap [Dmitriev, 2001] allows programmers to *hot swap* currently executing classes with new ones. The new version of a class is developed as usual. The source

code is edited and then compiled by an ordinary Java compiler. A GUI client allows the developer to upload the new class files and the necessary code to perform the hot swap via a socket connection. A Java Virtual Machine (JVM) internal call `RedefineClasses()` switches the byte-codes with the new versions.

HotSwap is the initial step towards a dynamic update system for the HotSpot JVM, the primary virtual machine for Java. HotSwap only allows to change method bodies. Everything else in the updated class needs to stay exactly the same, and as a result there is no need for any state transfer. Because there is no change in state the activation can happen instantly. All new calls go to new methods, while currently active calls to old methods complete normally.

Iguana/J

Iguana/J [Redmond and Cahill, 2002] introduces *adaptation classes* to redefine application behavior. In Iguana/J the original application is considered to be the base level, and adaptation classes make up a meta-level. Calls to a method in the original application get intercepted and redirected to the meta-level.

Classes in the meta-level do not redefine method behaviour on a per method basis, but allow additional code to hook into different parts of the original method execution in an Aspect Oriented Programming (AOP) way. The `MExecute` meta-class for example defines an execute method which is invoked on method execution and could introduce logging functionality before returning the result of the original method execution.

In Iguana/J the association between application classes and adaptation classes can be statically defined via an association declarations file or dynamically via association method calls which allow to change the behavior of an application reflectively from within the application.

Iguana/J is geared towards AOP support and therefore only concerned with changing the method behavior. It does not support modifications to the external interface, *i.e.*, the method signatures. Furthermore the handling of state and state transformations are of no concern in Iguana/J.

Dusc

Dusc [Orso *et al.*, 2002] enables dynamic updates through a system of proxy classes. The original class gets replaced with a proxy, containing an equivalent interface to the original class and a pointer to the current implementation. The proxy delegates all calls to the current version of the implementation. The proxy class also keeps

track which of its methods are active, and only allows updates of the corresponding implementation class when none of its methods are on the call stack.

Dusc is a somewhat limited approach, as the proxy class itself cannot be updated dynamically, forcing the method signature to stay the same and allowing only behavioral changes. Dusc does not address safety during an update and expects the developer to only deploy updates which lead to a safe state. Another drawback is that the initial application needs to be adapted and proxy classes need to be provided for all classes that should support dynamic software updates in the future.

DVM

In DVM [Malabarba *et al.*, 2000] a *dynamic class loader* is introduced. The dynamic class loader is an extension of the default JVM class loader. In addition to the default methods for class loading, the dynamic class loader allows programmers to reload an active class and replace it with a new version. Classes loaded by this dynamic class loader are called *dynamic classes*. The dynamic class loader was implemented as a custom VM, the dynamic-classes enabled virtual machine (DVM).

DVM uses an algorithm similar to garbage collection mark-and-sweep algorithms to find and transform object instances which need updating. This algorithm does not allow for custom state transformation and initializes new fields to `nil`.

DVM uses a rather aggressive approach towards active methods: if a method will be updated and is found on the call stack of a thread, an exception is raised and the thread aborted. While this prevents inconsistent data access, the abortion of threads can lead to overall program failure and makes DVM ill-suited for multi-threaded applications where the chances for such an event to occur are higher.

JVolve

JVolve [Subramanian *et al.*, 2009] provides an *Update Preparation Tool* (UPT), which generates mappings for classes and objects between different versions. The UPT also identifies safe update points in the running application by restricting which methods are allowed to reside on the call stack if an update were to take place. The application is stopped and the modified classes are loaded and JIT compiled, effectively replacing the old classes. The mappings, so called *state transformers* generated by the UPT, get applied to return to a valid state before the application is reactivated. The state transformers can also be customized beforehand to allow custom state transformations.

With the replacement of classes and the state transformers, Jvolve allows method body and signature changes as well as transforming state. Updates require safe update points, which might defer them indefinitely. Jvolve requires a special virtual machine and does not run on the default JVM.

3.3. Approaches in Smalltalk

Smalltalk

Smalltalk [Rivard, 1996] is a fully reflective language where the entire meta-level is reified in the language itself. All changes are done from within the running application. Naturally, the Smalltalk language allows for arbitrary changes to methods and method signatures, as this is the default way to introduce changes in the language. On the other hand, only basic state transfer is supported, as all newly created objects are initialized with `nil`.

It is important to note that updates in Smalltalk do not prohibit inconsistent data access and it is the programmers' responsibility to ensure that no changes to currently active code are made.

ChangeBoxes

ChangeBoxes [Denker *et al.*, 2007b] are a mechanism to scope changes in a Smalltalk application by making them first-class, *i.e.*, exposing the changes as normal objects to the application. ChangeBoxes allow programmers to provide multiple, possibly incompatible changes for an application and to switch between them at runtime. Changes get compiled into a change set, and the lookup is redirected to the desired change set at runtime.

The reflective approach of ChangeBoxes allows for changes in the behavior of an application from within the application itself, and it also allows changes to the method signature. ChangeBoxes do not tackle the issue of state transfer and do not allow the shape of classes to change dynamically. The issue of safety and safe access is not discussed in the work about ChangeBoxes.

	Ginseng	UpStare	POLUS	HotSwap	Iguana/J
Supported changes	Supports all possible changes to method bodies and method signatures, as well as global fields, but changes as global fields must be behavioral equivalent.	Allows changes to method bodies and method signatures, as well as global fields, but changes must be behavioral equivalent.	Supports all possible changes to method bodies and method signatures, as well as global fields.	Only allows changing the method bodies. Object fields and method signatures cannot be changed.	Only allows programmers to introduce additional functionality on a per method basis in an AOP fashion.
State transfer capabilities	Application wide state transfer, automatically generated as well as user provided.	Application wide state transfer, automatically generated as well as user provided.	Application wide state transfer, automatically generated as well as user provided.	No support for changing of fields, no state transfer required.	No support for changing of fields, no state transfer required.
Update activation	Safe updates, but requires programmer to specify update points, minimises delay between update points.	Safe and immediate updates.	Safe and immediate state updates, active methods finish with old code, state synchronization between old and new.	Immediate but not fully safe as active methods finish with old code. Programmer can manually ensure safety by recursively activating updates at safe update points.	Immediate, requires programmer to recursively activate updates and manually ensuring safety.
Access consistency	Update points and state transfer functions ensure no inconsistent data access.	Immediate atomic updates ensure no inconsistent data access.	No inconsistent data access, as old and new code exist in parallel but do not get mixed.	No inconsistent data access, because data fields cannot change.	No inconsistent data access, because data fields cannot change.
Support for multithreading	Use of update points makes this ill-suited for multithreaded systems.	Instant, update stacks provides good support for multi-threaded systems.	Good support for multi-threaded systems, old threads continue with old code, new threads with new code.	Manual approach to safety is ill-suited for multithreaded systems.	Manual approach to safety is ill-suited for multithreaded systems.

Table 3.1. Comparison of dynamic software update systems 1/2

	Dusc	DVM	JVolve	Smalltalk	Changebox
Supported changes	Freezes the public interfaces and allows only changing method implementations and private class fields.	Supports all possible changes to method bodies and method signatures, as well as global fields.	Supports all possible changes to method bodies and method signatures, as well as global fields.	Supports all possible changes to method bodies and method signatures, as well as global fields.	Supports all possible changes to method bodies and method signatures, as well as global fields.
State transfer capabilities	Generated and user provided state transfer on a per class basis.	Automatic state transfer. New fields get set to nil.	Generated and user provided state transfer on a per class basis.	Automatic state transfer. New fields get set to nil.	No support for state transfer.
Update activation	Safe, but possibly delays updates indefinitely until a valid update point is reached.	Immediate but not safe, prevents the update of active methods and aborts threads with active methods.	Safe, but delays updates possibly indefinitely until valid update point is reached.	Immediate, requires programmer to recursively install updates and ensure safety manually.	Inherits activation properties from Smalltalk and has the same shortcomings.
Access consistency	Update points and state transfer functions ensure no inconsistent data access.	No inconsistent data access. Updated methods can not be on the call stack, or the respective thread gets aborted, and new fields get initialized to valid values.	Update points and state transfer functions ensure no inconsistent data access.	Inconsistent data access possible.	Inherits access consistency properties from Smalltalk and has the same shortcomings.
Support for multi-threading	Manual approach to safety is ill-suited for multi-threaded systems.	Abortion of threads with active code leads to bad support for multi-threaded systems.	Use of update points makes this ill-suited for multi-threaded systems.	Manual approach to safety is ill-suited for multi-threaded systems.	Same shortcomings as Smalltalk.

Table 3.2. Comparison of dynamic software update systems 2/2

Chapter 3. Related Work

Chapter 4

ActiveContext in a Nutshell

In this chapter we motivate the research of ActiveContext, our approach to dynamic software updates. In Section 4.1 we show a common scenario for a dynamic update, which we believe cannot be achieved satisfactorily with existing methods. We propose how to update such a system by providing a step by step guide in Section 4.2.

4.1. A Motivational Example

While there exist dynamic update systems which support updates in multi-threaded environments, most of them are not designed specifically with support for multi-threaded environments as a main goal. Furthermore all the promising approaches are for the C programming language and there is a lack of good approaches for class-based languages. We propose a novel approach for safe dynamic updates in class-based systems, specifically tailored towards multi-threaded applications, which we call ActiveContext.

ActiveContext is a programming model to facilitate the design of dynamically updateable systems. In our model, different variations of a program can run in different execution contexts concurrently. An execution context is explicit: it can be loaded, manipulated, and specified dynamically when a new thread is started. As a consequence, threads can run different variants of the program.

To illustrate our approach for dynamic software updates, let's consider a web based address book like the Google Contacts application¹. The main entity of the domain model of such a system is the `Contact` which represents the entry of a person in the address book. The domain model is held in memory, globally accessible, and shared amongst all threads using the address book.

¹<http://www.google.com/contacts>

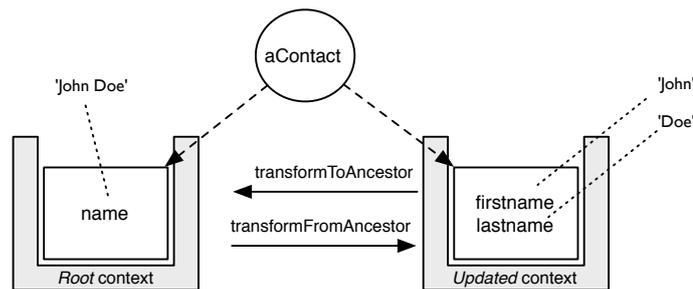


Figure 4.1. An instance of a `Contact` object has different states in different contexts. There are transformation functions between the two contexts.

Furthermore we consider the evolution shown in Figure 4.1. The `Contact` class is refactored to not only store the name in a single field `name`, but allow for separate fields `firstname` and `lastname`.

Such an evolution would not be easily achieved with the reflective facilities of a dynamic language such as Smalltalk: it would require an “intermediate” version of the class with all three fields `name`, `firstname`, `lastname` in order to allow the state of the impacted object instances to be migrated incrementally, for instance with `Contact allInstances do: [...]`. Only then could the `name` field be removed. Such an update is not only complicated to execute, but is also not atomic, possibly leading to consistency issues.

Even with dynamic software update mechanisms such an update would still be hard to achieve: they would either require that all requests complete prior to a global update of the system state, or that all `Contact` instances be updated immediately and face the risk that some existing thread running old code attempt to access the `name` field which no longer exists.

4.2. A Step by Step Update

The following steps describe how such an update can be installed with `ActiveContext` while avoiding these issues. First, the application must be adapted so that we can “push” an update to the system and activate it. Here is how one would typically adapt a server-side software such as an online address book or another system serving requests to leverage the programming model.

0. *Preparation.* First, an administrative interface is added to the online address book where an administrator can push updates into the system; the uploaded code will be loaded dynamically. Second, a global variable `latestContext` is

added to track the latest execution context that was loaded by the administrator. Third, the main loop that listens to incoming requests is modified so that when a new thread is spawned to handle the incoming request, the latest execution context is used.

After these preliminary modifications the system can be started, and now supports dynamic updates. The life-cycle of the system is now the following:

1. *Bootstrap.* After the system bootstraps, the application runs in a default context named the *Root* context. The global variable `latestContext` refers to the *Root* context. At this stage, only one context exists and the system is similar to a non-contextual system.
2. *Offline evolution.* During development, the field `name` is replaced with the two fields `firstname` and `lastname`. Figure 4.1 shows the impact on the state of a contact.
3. *Update preparation.* The developer creates a class `UpdatedContext` that specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transfer function which transforms the program state between the *Root* context and the *Updated* context. Objects will be transformed individually, one at a time.

In our case, the field `name` is split into `firstname` and `lastname` in one direction, and the fields `firstname` and `lastname` are joined into `name` in the other direction. The class of an object is considered part of the object's state and the transfer function also specifies that an updated version of the `Contact` class will be used in the *Updated* context.

Contexts may co-exist at run-time for an arbitrary period of time. It is therefore necessary that the object representations stay globally consistent with one another, which explains the need for a *bidirectional* transformation. If the state of an object is modified in one context, the effect propagates to the representation in the other contexts as well. Only fields that make sense must be updated though; fields that have been added or removed and have no counterpart in another context can be omitted from the transformations.

4. *Update push.* Using the administrative web interface, the developer uploads the updated `Contact` class and the `UpdatedContext` class. The application loads the code dynamically. It detects that one class is a context and instantiates it. This results in the generation of the new representation of all contact objects in the system. Objects now have two representations in memory. The global variable `latestContext` is finally updated and now refers to the newly created instance of the *Updated* context.

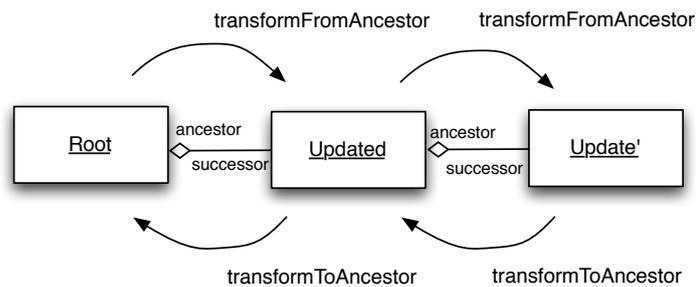


Figure 4.2. Context instances form a list

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request whose execution context will be the context referenced in `latestContext`, which is now the *Updated* context.
6. *Stabilization.* Since the execution context can be changed per thread, existing threads serving ongoing requests will finish their execution in the *Root* context, while new threads will use the *Updated* context. Assuming that requests always terminate, the system will eventually stabilize. A context can always be accessed safely from one execution context or another as the programming model maintains the consistency of various representations with each other using the bidirectional transformation. This alleviates the need for global, temporally synchronized update points which are hard to reach in multi-threaded systems.

Subsequent updates will be rolled out following the same scheme. For each update, a context class is created, loaded and instantiated dynamically. Contexts are related to each other with an ancestor/successor relationship. They form a list, with the *Root* context as the oldest ancestor, as shown in Figure 4.2.

We presented existing dynamic software systems and showed why we decided to design our own system. We gave a step by step walk-through of how we plan to model updates with our system, *ActiveContext*. We will give a more formal overview of *ActiveContext* by introducing the model and the implementation in the next chapters.

Chapter 5

The ActiveContext Model

In this chapter we would like to introduce ActiveContext formally by presenting a detailed view of the model.

As illustrated in the previous chapter, ActiveContext makes a clear distinction between the identity and the state of an object. An object can have several representations which remain consistent with one another thanks to state transformations. Behavior can change as well, since the class of an object is part of its state. ActiveContext is a programming model that supports scoping of state and migration of state between contexts.

We will discuss an implementation of the model in the following chapters, and evaluate it by providing a running example of an updateable system.

5.1. Identity, State and Contexts

The identity of an object is an identifier that unambiguously references an object in the system. The state of an object is comprised of (i) a set of fields and their corresponding values, and (ii) the class of the object. The fields of the object must of course match the fields declared in the class description.

An object can have as many states as there are contexts. A context can be seen as a mapping between the objects' global identity and the corresponding state that is relevant for that context. A thread has one *active* context at a time, which defines the state to be used for this specific thread of execution. This is illustrated in Figure 5.1.

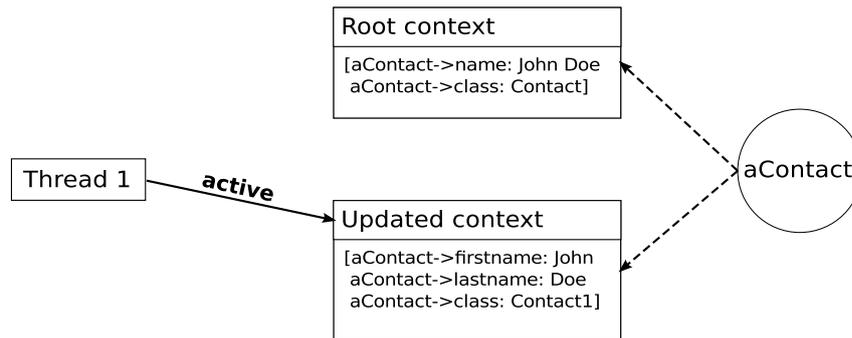


Figure 5.1. In Thread 1 Updated context is active, aContact has fields `firstname` and `lastname` and is of class `Contact1`

5.1.1. Contextual Objects

The interpreter or virtual machine has intimate knowledge of contexts, like classes or other internal abstractions. Contexts are however explicit in our model and reified as *first-class* entities at the application level. Contexts can be instantiated and manipulated dynamically like other objects. When a new thread is created, a context can be specified dynamically. It will be the *active* context for that thread of execution as soon as it starts running ¹.

The class of an object is part of its state. Behavioral variations are therefore achieved by changing the class of the object between contexts and scoping behavioral changes is reduced to a special case of scoping state.

5.2. State Synchronization

Context instances have a special structure that the virtual machine or interpreter expects. As shown in Figure 4.2 they form a list at run-time and contexts must (i) have a field `ancestor` pointing to a valid context which realizes the ancestor/-successor relationship, and (ii) implement two methods `transformFromAncestor` and `transformToAncestor` which realize the bidirectional transformation.

The role of the bidirectional transformation is to maintain the consistency between several representations of an object in various contexts. A change to an object in a context will “propagate” to its ancestor and successor—which in turn will propagate it further—so as to keep the representations of an object consistent in all contexts. This propagation happens upon the following events: (i) the state of

¹Context switches per method call (`aContext do: [...]`) are not addressed in this work as their exact implication deserves further research.

an object is changed in a context, (ii) a new object is instantiated, and (iii) a new context is instantiated. The most frequent case is the first one, which corresponds to a regular field write `o.field := value`.

The state of an object can only be changed once it is synchronized across all contexts. The same holds true when a new context is instantiated, so no objects can be changed until the synchronization of objects to the new context is complete. Therefore the state propagation following any of the operations (i) – (iii) have to be mutually exclusive and a locking mechanism is needed to ensure safety during the propagation. Reading object state also has to be mutual exclusive from state propagation. On the other hand if the state is consistent and no propagation is happening, read operations can happen in parallel.

As contexts are loaded dynamically in an un-anticipated fashion, the transformation is encoded in the newest context and expressed in terms of its ancestor, never in terms of its successor. We have one method to transform *from* the ancestor to the newest context, and one method to transform from the newest context *to* its ancestor. The *Root* context is the only context that does not encode any transformation.

Such a sample one-way transformation is shown in Figure 5.2. It corresponds to the transformation from the *Root* context to the *Updated* context of Figure 4.1: `self` refers to the *Updated* context, and `ancestor` to the *Root* context. A change to the `name` of an instance of `Contact` in the *Root* context would be propagated to the *Updated* context and the fields `firstname` and `lastname` would be updated accordingly.

5.3. Reflective Hook

Contexts are *reflective hooks* which blend into the semantics of the programming language. Field write and object instantiation will be evaluated differently depending on the set of such contexts and their corresponding transformations: context instances become extension points of the interpreter itself.

While contexts are regular objects with state, their state is accessed by the interpreter or virtual machine itself (not only other application objects) which means they cannot be contextual in the same way as other objects. A solution would be to introduce *meta-contexts*, *i.e.*, contexts at the meta-level, containing the state of contexts. While this seems like an elegant solution, it does not fully solve the issue. The meta-contexts themselves would still need to store their state in yet another level of meta-contexts. To avoid dealing with such infinitive meta-regression issues, we decided that code running at the interpreter level should run *outside* of any context. As a consequence, some objects in the system must be *primitive*: they

```

transformFromAncestor: id
  | cls name firstname lastname |
  cls := ancestor readClassFor: id.
  ( cls = Contact ) ifTrue: [
    name := ancestor readField: 'name' for: id.
    name isNil iffFalse: [
      firstname:= name before: ' '.
      lastname:= name after: ' '.
    ].
    self writeClassFor: id value: Contact1.
    self writeField: 'firstname' for: id value: firstname.
    self writeField: 'lastname' for: id value: lastname.
  ]
  ( cls = AnotherClass ) ifTrue: [
    ...
  ]
  ...

```

Figure 5.2. State transfer

have a unique state in the system and are not subject to contextual variations. This is notably the case with context instances.

Transformations run at the interpreter level outside of any context. As a consequence, during a transformation only primitive objects can be accessed and manipulated *implicitly*. Contextual objects must be manipulated *reflectively* with `readClassFor:`, `writeClassFor:value:`, `readField:for:` and `writeField:for:value:` meta-facilities as shown in Figure 5.2. These are not user methods, but special language constructs. This way, the state of an object (class or fields) in an arbitrary context can be updated without extra transformation(s) being triggered, and independently of the active context. Obviously it is unsafe to access these meta-facilities from the application level. To ensure safety, the meta-facilities are encapsulated in two kinds of mirrors [Bracha and Ungar, 2004], which reify the state an object has in a particular context at the interpreter level and the application level respectively. The interpreter level mirrors access the meta-facilities directly, while the application level mirrors access them via reflective method calls on the interpreter and ensure that transformations are triggered.

After this formal introduction to ActiveContext, the basis for our dynamic update system, we will present the implementation thereof in the following chapter. This will be followed up by the implementation of an example which will be the basis for the discussion of our approach.

Chapter 6

Implementation

After having presented how to model dynamic software updates with `ActiveContext`, we would like to go into some of the implementation specific details. First we discuss the choice to use Pinocchio [Verwaest, 2009] as a platform, and give an introduction to Pinocchio. Next we present the implementation of `ActiveContext`, by showing some preliminary changes we did to Pinocchio, and then the full implementation in detail.

`ActiveContext` changes the default way object state and behaviour is accessed in a given programming language. To implement this, specific reflective functionality to change the way the interpreter handles lookups is necessary. There are two possible ways to implement such reflective capability: (i) complex program transformations or (ii) a custom virtual machine [Verwaest *et al.*, 2010]:

Reflective capabilities are typically fixed by the Virtual Machine (VM). Unanticipated reflective features must either be simulated by complex program transformations, or they require the development of a specially tailored VM.

The second approach allows developers to leverage existing applications with `ActiveContext` without having to change their implementation or adapt them in any way, and therefore we decided to implement `ActiveContext` as a specially tailored VM. Another consideration in the choice of a language platform was the fact that we wanted to be compatible with standard Smalltalk. We decided to use the Pinocchio [Verwaest, 2009] programming language as a platform, as it is designed to support easy development of custom VM features and has a Smalltalk syntax.

Pinocchio allows developers to provide custom interpreters by making them first-class. Interpreters in Pinocchio are normal objects in the language, which can be instantiated, inherited from and passed around like any other object in the language. Applications freely flow from interpreter to interpreter depending on the required semantics. Applications specify their own interpreters *inside* the runtime as subclasses of the default `Interpreter` class, a reification of the core interpreter.

This is an easy and natural way to get custom behaviour into a VM.

6.1. Pinocchio

Instead of interpreting byte-codes, Pinocchio directly interprets abstract syntax trees (ASTs) that more faithfully represent Smalltalk-80 code. The core interpreter is implemented in C, and is reified in the runtime as a first-class interpreter. The interpreter provides a basic meta-object protocol (MOP) to support structural reflection. Unlike most interpreters that are based on the assumption that the VM is a black box isolated from the runtime system, Pinocchio supports behavioral reflection by opening the interpretation of code to the runtime. Behavioral reflection is supported by explicitly instantiating first-class interpreters that subclass the reified core interpreter. Extending interpreters is facilitated since AST nodes are semantically closer to the original source code than byte-code [Denker *et al.*, 2007a; D’Hondt, 2008].

To construct a new variant of the Pinocchio interpreter it suffices to subclass the `Interpreter` class and override a part of its interface (see Figure 6.1). The `Interpreter` class defines a meta-circular interpreter implemented as an AST visitor that manages its own environment but relies on recursion to automatically manage the runtime stack. The meta-circular interpreter reifies the core interpreter written in C, so its methods are actually implemented as native functions that hook into the underlying C interpreter code. From the user’s point of view the `Interpreter` is fully written in Pinocchio itself.

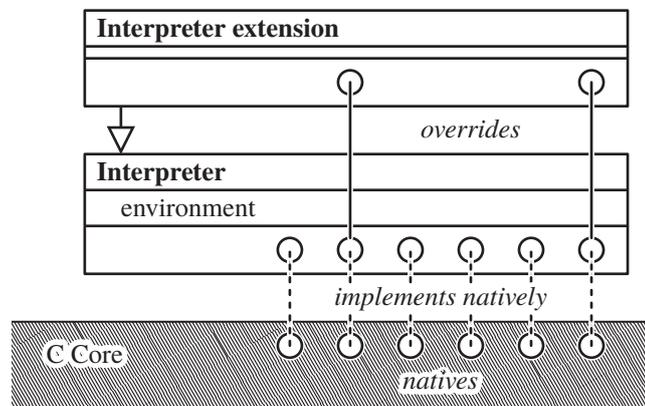


Figure 6.1. Native methods in the `Interpreter` and interpreter extension through sub-classing

Application code is evaluated by a new interpreter by sending the `interpret:` message to the desired interpreter class with a closure representing the code as its argument. For example, the expression

```
ActiveContextInterpreter interpret: [ self runApplication ].
```

will cause the closure `[self runApplication]` to be evaluated by the `ActiveContextInterpreter` interpreter.

As usual, closures encapsulate an *environment* and an *expression* object. When starting up a specialized interpreter the continuation of the interpreted application is empty. The interpreter installs the enclosed environment and starts evaluating the expression in this environment. Since the expression passed to the default interpreter is a closure, it is evaluated by sending the message `value` to the closure on top of the interpreter:

```
Interpreter>>interpret: aClosure
  ↑ self send: (Message new selector: #value)
    to: aClosure.
```

Pinocchio provides the following core features for compatibility with Smalltalk:

- *the object model and meta-model*: Pinocchio uses the same object model as Smalltalk.
- *syntax*: Pinocchio uses a Smalltalk compatible syntax for most parts. A main difference is the syntax to define classes, which currently is not yet available in Pinocchio.
- *doesNotUnderstand*: Pinocchio's core interpreter sends the `doesNotUnderstand:` message to any object that does not implement a method corresponding to the selector of a message sent to it, as defined in the Smalltalk-80 specification.

6.2. Implementation

6.2.1. Adaptations of Pinocchio

The idea behind `ActiveContext` is to make dynamic software updates feasible in a multi-threaded environment with concurrent access to the same objects. To create a scenario with concurrent access, a way to asynchronously interact with applications was necessary. This requires support for threads in the language, as well as an interactive, non-blocking way for input and output. Pinocchio is still a rather new

platform and cannot be considered complete just yet. It was specifically missing these features for asynchronous access, so we started our implementation by adding them to Pinocchio. Apart from threads, we supplied support for non-blocking web-sockets for asynchronous interaction.

Threads in Pinocchio

Threading support in Pinocchio is based around a scheduler thread. The scheduler thread keeps a list of all other threads in the application, and schedules them in a circular manner. Normal threads can voluntarily give up execution by yielding or will automatically be preempted after a fixed number of method invocations. Whenever a thread yields or is preempted, control is passed over to the scheduler thread. This has to happen atomically and is therefore implemented as a primitive function in Pinocchio, *i.e.*, it is implemented as a C procedure invoked by a single method call in a Pinocchio application.

```
Thread>>primYield
  <pPrimitive: #primYield plugin: #'Runtime.Thread'>
```

The yield procedure backs up the state of the current thread together with the program counter and switches control to the scheduler thread. The call stack of the scheduler thread is reset every time control is passed to it, and only the method to add the previous thread to the end of the queue of waiting threads and activate the next is pushed onto the call stack.

```
void yield() {
  _thread->backup_pc = pc;
  Thread previous   = _thread_;
  _thread_         = _scheduler_thread_;
  reset_thread(_scheduler_thread_);
  Class_direct_dispatch((Optr)Thread_Class, HEADER(Thread_Class),
                      (Optr)SMB_yield_, 1, (Optr)previous);
}
```

The method performing the actual scheduling is implemented in Pinocchio, as all the threads are native Pinocchio objects, and managing them from C code is very cumbersome compared to the higher level functions available in Pinocchio. This is possible due to the fact that the scheduler thread is active, and none of the states of the previous or next thread are touched.

```
Thread>>yield: previousThread
  threads addFirst: previousThread.
  self resumeNext.
  ↑ nil.
```

```
Thread>>resumeNext
  threads removeLast resume
```

The activation of the following thread on the other hand has to be a primitive procedure once again, and restores the previously backed up state and program counter of the following thread before passing control over to it.

```
static void NM_Thread_resume(Optr self, Class class, uns_int argc) {
  Thread next_thread = (Thread) self;
  _thread->backup_pc = pc;
  RETURN_FROM_NATIVE(nil);
  pc                = next_thread->backup_pc;
  _thread_          = next_thread;
}
```

All threads have their own interpreter instance. If new threads are spawned, their interpreters need to be instantiated and initialized from the interpreter level as opposed to the application level from which the threads are spawned. Otherwise the interpreters will run as subprocesses of the original interpreter and not independent from it. Forking of new threads is implemented as a reflective procedure on the interpreter. The `fork` method on `BlockClosure` invokes a reflective method call installed on the interpreter level.

```
BlockClosure>>fork
  <pinocchioReflective: #blockClosureFork:message:>

Interpreter>>blockClosureFork: aBlockClosure message: aMsg
  | interpreter |
  interpreter := Interpreter new.
  ↑ Thread primFor: [ i interpret: [ aBlockClosure value ] ].
```

Sockets

The support for sockets in Pinocchio is very similar to Smalltalk. The Pinocchio classes for `Socket` and `SocketStream` are a direct port of the Smalltalk implementation, with minimal adaptations as not all of the used `String` manipulation functions were available in Pinocchio yet.

We implemented the C level primitive functions to connect to UNIX sockets according to the Linux Programmer's Manual on `socket` and `select` to enable non-blocking system calls.

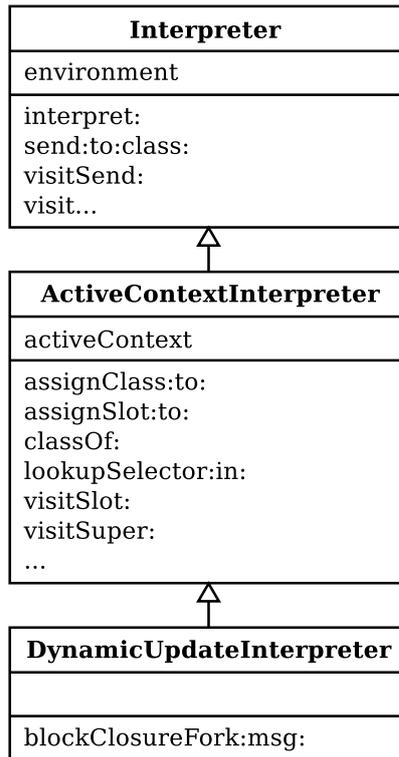


Figure 6.2. The interpreter hierarchy of the ActiveContext implementation.

6.2.2. Implementation Details

We implemented our ActiveContext dynamic software update system as two different interpreters. First we created the `ActiveContextInterpreter` and implemented the base ActiveContext model, which could also be reused for purposes other than dynamic software update. We added the `DynamicUpdateInterpreter` to add the features specifically needed for dynamic software updates. The relationship between the interpreters can be seen in Figure 6.2. The `DynamicUpdateInterpreter` is actually very lightweight and only provides the functionality to set the newest updated context as active if a new thread is forked, by setting the `activeContext` of the interpreter of that thread to the `latestContext`:

```

DynamicUpdateInterpreter>>blockClosureFork: aBlockClosure message: aMsg
| interpreter |
interpreter := DynamicUpdateInterpreter new.
interpreter activeContext: DynamicUpdate latestContext.
↑ Thread primFor: [ i interpret: [ aBlockClosure value ] ].
  
```

Contextual Objects

The `ActiveContextInterpreter` changes the way state and memory are managed, in particular the treatment of object fields. The read and write access to object fields needs to be changed away from the default way in which it is handled in Pinocchio to a contextual model.

Pinocchio knows two different kinds of object fields: slots and arrayed fields. Slots are “normal” fields in that they have a fixed name by which they can be accessed. The number of slots an object has is statically defined. Arrayed fields on the other hand do not have a name and are accessed instead by a number representing their position inside of an object. The number of arrayed fields an object has is dynamically defined at object instantiation. Objects can contain slots as well as arrayed fields.

All the visitor methods that access slots or arrayed fields have been modified to check the object type (primitive or contextual) and access the object accordingly.

It was decided that primitive objects would “delegate” to the native memory management of Pinocchio. The set of primitive objects was adapted to match the reality of a fully reflective system. `Object`, `Behaviour`, `Class`, `Metaclass` and other special classes needed during bootstrapping cannot be contextual, as they need to exist in order for the `ActiveContext` classes to be defined, so for them to be contextual would lead to a chicken-and-egg problem. The same also holds true for the basic, immutable objects `nil`, `true` and `false`, as well as for numbers, characters, strings and symbols.

To distinguish between primitive and contextual objects, we implemented a method `isPrimitive` on `Object` which returns if an object is primitive. This method looks up the object in two pools of object references we maintain internally on the root context as `Sets`, one for primitive and one for contextual objects. This approach is rather naive—tagging the pointer would be much more efficient, for instance.

```
Object>>isPrimitive
  ↑ Context root primitives includes: self.
```

To be able to change the access to slots, the `ActiveContextInterpreter` overrides the `visit` method for slots, `visitSlot:`, and `assignSlot:to:`, the method to assign values to slots. The methods delegate to the meta-facilities for reading and writing implemented on the context and indicate that a locking mechanism is in place. Both will be explained in more detail further down. The `assignSlot:to:` method has a call to `propagate:interpreter:` after writing the value, which triggers the synchronization with the other contexts available in the system.

Chapter 6. Implementation

```
ActiveContextInterpreter>>visitSlot: aSlot
| res |
( self currentSelf isPrimitive ) ifTrue: [
    ↑ super visitSlot: aSlot .
].
self class lock critical: [
    res := self activeContext read: aSlot name for: self currentSelf.
].
↑ res.
```

```
ActiveContextInterpreter>>assignSlot: aSlot to: value
self class lock critical: [
    self activeContext write: aSlot name for: self currentSelf value: value.
    self activeContext propagate: self currentSelf interpreter: self.
].
↑ value
```

For the arrayed fields the `ActiveContextInterpreter` intercepts the `at:` and `at:put:` native method calls which implement the read and write access in default Pinocchio, by overriding the `invokeNativeMethod:on:message:alternative: message send` which dispatches all native methods on the standard meta-circular `Interpreter`. The calls get redirected to the methods `invokeAtPut:on:message:alternative:` and `invokeAt:↵on:message:alternative`, which trigger a contextual lookup or delegate back to the native memory model for primitive objects:

```
ActiveContextInterpreter>>invokeNativeMethod: aClosure on: receiver message: ↵
aMessage alternative: aBlock

(aMessage selector = #'at:put:') ifTrue: [
    ↑ self invokeAtPut: aClosure on: receiver message: aMessage alternative: ↵
    aBlock
].

(aMessage selector = #'at:') ifTrue: [
    ↑ self invokeAt: aClosure on: receiver message: aMessage alternative: ↵
    aBlock
].
[...]
```

To access the class in a contextual way, the special field `class` is used. Method calls to the native accessor for classes get also intercepted in `invokeNativeMethod:↵on:message:alternative:.` The `classOf:` on `ActiveContextInterpreter` switches between primitive and contextual objects and the `classOf:` on `Context` does the lookup using the `class` field.

```
ActiveContextInterpreter>>invokeNativeMethod: aClosure on: receiver message: ←
  aMessage alternative: aBlock
  (aMessage selector = #'class') ifTrue: [
    ↑ self classOf: receiver .
  ].
[...]
```

```
ActiveContextInterpreter>>classOf: anObject
  ( anObject isPrimitive ) ifTrue: [
    ↑ anObject class.
  ].
↑ self activeContext classOf: anObject.
```

```
Context>>classOf: anObject
  ↑ self read: #'class' for: anObject .
```

The visitor method that implements the method lookup inside a class' method dictionary, `lookupSelector:in:` has been modified to access the class contextually via the special `class` field.

Similarly to `class`, a special field `superclass` has been introduced because the superclass and super message sends have been changed to access the superclass contextually via this field.

```
ActiveContextInterpreter>>visitSuper: aSuper
[...]
```

```
↑ self
  send: message
  to: receiver
  class: ( self superclassOf: self currentClass)
  for: aSuper
```

```
ActiveContextInterpreter>>superclassOf: aClass
  ( aClass isPrimitive ) ifTrue: [
    ↑ aClass superclass.
  ].
↑ self activeContext superClassOf: aClass.
```

```
Context>>superClassOf: aClass
  ↑ self read: #'superclass' for: aClass.
```

To summarise which methods the `ActiveContextInterpreter` overrides, Table 6.1 shows the relevant visitor methods of the default `Pinocchio Interpreter`, and indicates which ones were overridden by the `ActiveContextInterpreter`.

Objects are instantiated by sending `new` to a class as usual. The default implementation of the `new` command in `Pinocchio` calls the native method `basicInstantiate` to create objects. This native call gets intercepted by the `ActiveContextInterpreter` in the `invokeNativeMethod:on:message:alternative:` similar to the `at:` and `at:put:` before.

Visitor method	Overridden
assignVariable: aVariable to: value	.
assignSlot: aSlot to: value	✓
classOf: anObject	✓
invokeNativeMethod: aClosure on: receiver	✓
message: aMessage alternative: aBlock	
lookupSelector: selector in: class	✓
visitAssign: anAssign	.
visitConstant: aConstant	.
visitClassReference: aClassReference	.
visitVariable: aVariable	.
visitSlot: aSlot	✓
visitSelf: aSelf	.
visitSend: aSend	.
visitSuper: aSuper	✓

Table 6.1. The visit methods of the interpreter

```

ActiveContextInterpreter>>invokeNativeMethod: aClosure on: receiver message: ←
  aMessage alternative: aBlock

  (aMessage selector = #'basicInstantiate:') ifTrue: [
    ↑ self invokeBasicInstantiate: aClosure on: receiver message: aMessage ←
      alternative: aBlock
  ].

  (aMessage selector = #'basicInstantiate:sized:') ifTrue: [
    ↑ self invokeBasicInstantiateWithSize: aClosure on: receiver message: ←
      aMessage alternative: aBlock
  ].
[...]
```

The `invokeBasicInstantiate:on:message:alternative:` method for objects with slots only, or the alternative `invokeBasicInstantiateWithSize:on:message:alternative:` for objects with arrayed fields decides if a contextual or primitive object is instantiated.

Context and Meta-Facilities

Internally, the interpreter uses several `Dictionary` instances to implement the memory model for contextual objects: one dictionary per context instance is created and maps $\langle \text{object identity, field} \rangle$ to the corresponding value. This implies one level of indirection to access the state of a contextual object.

The keywords `readField:for:`, `writeField:for:value:` for meta-facilities to access the state dictionary as introduced in the code example Figure 5.2 have been implemented with regular message sends on the context instances. They have been renamed to the simpler `read:for:` and `write:for:value:`. No keyword for `readClassFor:`

and `writeClassFor:value:` was required as we use `read:for:` and `write:for:value:` with the special `class` field.

All context classes must inherit from the `Context` class, which implements the methods for meta-facilities, and the `Dictionary` to store the contextual objects. `Context` also implements the default identity transformation. Because the transformation function needs to provide code to transform every single object, this is a convenience function making it possible to delegate to the superclass with the `super` keyword at the end of the transformation function as a catch-all for objects which do not need custom transformation code because they did not change between two contexts.

Transformation

In the case of a write access, the interpreter propagates the change and executes the necessary transformation to keep the object representation consistent in the other contexts.

To access objects reflectively from the interpreter level as stated in Section 5.3, two object representation classes were introduced to interact with the objects in a convenient way during transformation. `ObjectState` for the interpreter level, and `ReifiedObjectState` for the application level. Both classes encapsulate the meta-facilities to access the state as implemented on contexts and provide a simple `at:` and `at:put:` interface to access the fields of the object.

The read and write methods of `ReifiedObjectState`, the representation of object state in the application level, are implemented as reflective method calls:

```
at: instVarName
  <pinocchioReflective: #objectStateAt:message:>

at: instVarName put: value
  <pinocchioReflective: #objectStateAtPut:message:>
```

The reflective procedures then access an object of type `ObjectState`, which mirrors the state of the `ReifiedObjectState` at the interpreter level.

As mentioned in Section 5.2, access to objects needs to be mutually exclusive no matter whether they trigger a transformation or not. To implement this constraint, a unique global lock is used for reads, writes and the state transformations by protecting them in a critical block of a global `Mutex`. The `Mutex` allows reentrant access to a context from within a single thread and blocks access from other threads. Other threads enter a Spin-Lock until they are granted access.

Bootstrapping

The system bootstraps initially in the *Root* context. It was decided that when this context is active the interpreter would delegate to the native memory management of Pinocchio for both primitive *and* contextual objects. This decision was taken to (i) facilitate bootstrapping, (ii) ease early development, and (iii) emulate the original programming model for applications which do not leverage `ActiveContext`.

Chapter 7

Validation

In this chapter we present an example application making use of the ActiveContext dynamic update system we explained so far. The sources of the example are available, and instructions on how to obtain and install them are available in Appendix A.

7.1. A Running Example: Telnet Server

To validate our approach, we adapted an application following the update scheme presented Chapter 4. The application that was adapted is a Telnet server that provides a simple command-line interface to manipulate a single user entry shared between different, simultaneous connections. A client can connect to the server and run a few simple commands to print and edit the name field(s) of a user. We provide two different versions of the interface. The first only allows clients to manipulate a single name field of the user, and the domain model used is a `Contact` object with a single `name` field. The updated version provides the ability to enter the first and last names individually and the updated `Contact1` class contains the two fields `firstname` and `lastname`.

Below is the interface of our simple telnet server, as displayed by the help function of the updated version:

```
This is the example telnet server
Possible commands:
  version -- show version information
  user show -- show current user information
  user set -- set current user information
  help -- print this help
  quit -- disconnect and close
```

While simpler than a full-fledged online address book as mentioned in Section 4.1,

```
TelnetServer>>run: aPort
  | interpreter |
  DynamicUpdateInterpreter reset.
  interpreter := DynamicUpdateInterpreter new.
  DynamicUpdateController updateInterface: 5678 interpreter: interpreter.
  interpreter interpret: [
    TelnetServer runServer: aPort
  ].
```

Figure 7.1. The adapted Telnet server interface to support dynamic software updates with ActiveContext.

this system exhibits the same characteristics in terms of design and difficulties with respect to dynamic updates.

To enable dynamic updates, the initialization of the Telnet server was adapted according to step *Preparation* in Section 4.2. First, the Telnet server was made to run on top of the `DynamicUpdateInterpreter`, so that the main loop that listens for incoming TCP connections, spawns the connections in a thread using the `latestContext` as active context as explained in the implementation details in Section 6.2. Second, a controller interface was started on an alternative port, where a user can use special commands to “push” an update. This controller should also be able to upload the code of the update, but Pinocchio does not yet provide a syntax to define classes and this is not possible in the current version of our example. The code for the steps above can be seen in Figure 7.1.

Upon bootstrapping the system initially runs in the *Root* context. Connections to the Telnet server get handled by the version 0 of the Telnet server as shown in Figure 7.2. An administrator can connect to the controller and use a special command to activate the update. Further connections will get handled by the version 1 of the Telnet server as shown in Figure 7.3. The user contact gets synchronized in the background. The object is the same in both versions, but with a different state representation, showing that the update is indeed safe.

A thread handling a specific client connection keeps running as long as the connection is established. Clients that are using the original version are not influenced by the update and multiple clients connected to the server might see different versions of the command-line interface at the same time.

Upon disconnection of a client, its server-side thread terminates. The system stabilizes eventually when all clients have disconnected.

Early benchmarking showed a significant performance loss in the order of up to a magnitude of 4, and this is one of our main concerns at the moment. The

7.1. A Running Example: Telnet Server

```
guru@plumeis: ~  
guru@plumeis:~$ telnet localhost 9090  
Trying ::1...  
Trying 127.0.0.1...  
Connected to localhost.  
Escape character is '^]'.  
> version  
This is the telnet server handler version 0  
> user set  
Enter user name  
> John Doe  
current user: John Doe  
> 
```

Figure 7.2. Version 0 of the Telnet server.

```
guru@plumeis: ~  
guru@plumeis:~$ telnet localhost 9090  
Trying ::1...  
Trying 127.0.0.1...  
Connected to localhost.  
Escape character is '^]'.  
> version  
This is the telnet server handler version 1  
> user show  
Current user:  
First name: John  
Last name: Doe  
> user set  
Enter first name  
> Jane  
Enter last name  
> Foo  
Current user:  
First name: Jane  
Last name: Foo  
> 
```

Figure 7.3. Version 1 of the Telnet server.

performance loss and other concerns will be addressed in Chapter 8. This running example nonetheless demonstrates the viability of our approach.

7.2. The Code of the Telnet Server

The two different versions of the Telnet server interface are implemented in two versions of the handler classes `Handler` and `Handler1`. They both implement a `run` method to which new requests to the interface get dispatched. To be able to access the updated handler which has a different class name, a factory is needed to instantiate the handlers indirectly.

```
TelnetServer>>runServer: aPort
| sock conn |
factory := HandlerFactory new.
user := Contact new.
[
    sock := self new.
    sock primListenOn: aPort.
    true whileTrue: [
        conn := sock accept.
        [
            handler := factory on: conn.
            handler run.
        ] fork
    ] fork.
] fork.
```

The `run` method is a simple switch-case statement to handle different commands entered. Both versions of the method support a `user set` and a `user show` command to interact with the contact object stored in memory. The original version interacts with a contact object of class `Contact` with only a single `name` field and corresponding setter and getter methods, while the updated version in `Handler1` can interact with the updated object with `Contact1` class with setter and getter methods for `firstname` and `lastname`:

```
Handler>>run
| data |
socket write: '> '.
self receiveDataIfAvailable.
data := self nextAllInBuffer.
[ data size > 0 and: [ data = ('quit', String crlf) ]] whileFalse: [
    data = ('user show', String crlf) ifTrue: [
        socket write: 'Current user: ', TelnetServer user name, String crlf, '↵
        > '.
    ] ifFalse: [
        data = ('user set', String crlf) ifTrue: [
```

```

        socket write: 'Enter user name', String crlf.
        self receiveDataIfAvailable.
        data := self nextAllInBuffer.
        TelnetServer user name: (data before: Character cr).
        socket write: 'current user: ', TelnetServer user name, String crlf.
    ]
]]

Handler1>>run
| data |
socket write: '> '.
self receiveDataIfAvailable.
data := self nextAllInBuffer.
[ data size > 0 and: [ data = ('quit', String crlf) ]] whileFalse: [
    data = ('user show', String crlf) ifTrue: [
        socket write: 'Current user: ', String crlf,
            'First name: ', TelnetServer user firstname, String crlf,
            'Last name: ', TelnetServer user lastname, String crlf.
    ] ifFalse: [
        data = ('user set', String crlf) ifTrue: [
            socket write: 'Enter first name', String crlf, '> '.
            self receiveDataIfAvailable.
            data := self nextAllInBuffer.
            TelnetServer user firstname: (data before: Character cr).
            socket write: 'Enter last name', String crlf, '> '.
            self receiveDataIfAvailable.
            data := self nextAllInBuffer.
            TelnetServer user lastname: (data before: Character cr).
            socket write: 'Current user: ', String crlf,
                'First name: ', TelnetServer user firstname, String crlf,
                'Last name: ', TelnetServer user lastname, String crlf.
        ]
    ]
]]

```

The `ActiveContext` model in the background ensures that the user information gets synchronized and can be changed concurrently from both versions of the Telnet server and from multiple concurrent connections. The user fields are kept synchronized by the transformation functions of the `Update1` context, where the name gets split from the *Root* context to the `Update1` context, and assembled from the `Update1` context to the *Root* context. The transformation functions also show how the handler class gets updated to a newer version. The factory to instantiate the handler is also updated to a version which can instantiate the new handler.

```

Update1>>transformFrom: objStateFrom to: objStateTo
| cls |
cls := objStateFrom at: #'class'.
(cls = Contact) ifTrue: [
    objStateTo at: #'firstname' put: ((objStateFrom at: #'name') before: $ ).
    objStateTo at: #'lastname' put: ((objStateFrom at: #'name') after: $ ).
    objStateTo at: #'class' put: Contact1.
]

```

Chapter 7. Validation

```
] ifFalse: [
  (cls = Handler) ifTrue: [
    objStateTo at: #'class' put: Handler1.
  ] ifFalse: [
  (cls = HandlerFactory) ifTrue: [
    objStateTo at: #'class' put: HandlerFactory1.
  ] ifFalse: [
    super transformFrom: objStateFrom to: objStateTo
  ]
]
```

```
Update1>>transformTo: objStateTo from: objStateFrom
| cls |
cls := objStateTo at: #'class'.
(cls = Contact1) ifTrue: [
  (((objStateTo at: #'firstname') ~= nil)
   and: [(objStateTo at: #'lastname') ~= nil]) ifTrue: [
    objStateFrom at: #'name' put: ((objStateTo at: #'firstname') ↔
    asString, ' ', (objStateTo at: #'lastname')).
  ].
] ifFalse: [
  super transformTo: objStateTo from: objStateFrom.
]
```

The `DynamicUpdateController` first saves the existing context as `oldCtx` by extracting it from the main application, in this case the Telnet server. This needs to be done from inside the interpreter running the main application, because the `DynamicUpdateController` itself is not contextual and runs under a different interpreter. The `DynamicUpdateController` is laid out similarly to the handler classes, except that it contains the whole logic to listen for socket connection on a port. It uses a switch-case statement to interpret the different commands entered. Upon calling `installUpdate1` the `Update1` context gets instantiated inside the interpreter of the main application. The `Update1` context instance is then set as the `latestContext` so new connections to the Telnet server will run on interpreters which have the `Update1` context set as the `activeContext`.

```
DynamicUpdateController>>updateInterface: aPort interpreter: interpreter
| sock oldCtx ctx1 |
interpreter interpret: [
  oldCtx := CurrentContext instance.
].
sock := self new.
sock primListenOn: aPort.
[true whileTrue: [
  | conn stream data |
  conn := sock accept.
  [
    stream := SocketStream on: conn.
    [
      conn write: '> '.
    ]
  ]
]
```

```

stream receiveDataIfAvailable.
data := stream nextAllInBuffer.
[ data size > 0 and: [data = ('quit' , String crlf)]] whileFalse: ←
[
  data = ('install Update1' , String crlf) ifTrue: [
    self warn: 'Installing context: Update1' , String crlf.
    interpreter interpret: [ ctx1 := Update1 newFrom: oldCtx. ←
    ].
    DynamicUpdate latestContext: ctx1.
    conn write: 'Installed Update1', String crlf.
  ].
  conn write: '> '.
  stream receiveDataIfAvailable.
  data := stream nextAllInBuffer
]] on: ConnectionClosed do: [:ignore | nil].
conn closeAndDestroy: 0
] fork
]] fork

```

We showed the viability of `ActiveContext` by presenting a running application. We also presented the whole code of the application showing that the different versions really interface with two different versions of the user object. We also showed the transformation functions as well as how the update is activated. We will discuss `ActiveContext` in the next chapter, using this Telnet server example to highlight features and shortcomings.

Chapter 8

Discussion

This work presents a conceptual model for systems to support dynamic software updates in multithreaded environments. After demonstrating our approach with a running example in the previous chapter, we would like to discuss some general points about `ActiveContext`, as well as what additionally needs to be considered for a serious implementation.

8.1. Analysis of `ActiveContext`

If we recall the different features of dynamic software update systems introduced in Chapter 2, we can classify `ActiveContext` as we did the approaches presented in Chapter 3 in Table 3.1 and Table 3.2. `ActiveContext` supports arbitrary changes, and it allows developers to change method implementations and method signatures as well as the fields of objects. State transfer is a main feature of `ActiveContext` and there is support for custom transformation functions as well as automatic transformation through the identity transformation functions provided in the `Contact` base class. The updates are immediate for new threads, while old threads continue to run on the old code, and safety is ensured because the active methods on the call stack are not updated. There is also no inconsistent data access as the old and new code do not get mixed, but state is kept in sync through the transformation functions.

Following are some additional considerations on the approach with `ActiveContext`:

Class naming

Different versions of a class can coexist at run-time. While the link between an object and its class is contextual in the current implementation of `ActiveContext`, the link between a class reference and the implementation of a class is not yet

contextual. As a consequence the classes need to have different names as seen in the example in the previous chapter where the updated version of the `Contact` class is called `Contact1`. This is not very convenient and it would be interesting to have a true support for class versioning. A consequence of having different names for versions of a class, is the need for factory classes or a similar indirect way to instantiate objects, because the name of the updated class cannot be easily anticipated.

Preparing applications for ActiveContext

Initially applications can be implemented without any special considerations towards support for dynamic software updates with ActiveContext. Only at deploy-time is there a need to include ActiveContext and to deploy the application on top of the `ActiveContextInterpreter`. In the current implementation of ActiveContext this is not fully correct though. Because of the class naming issue discussed above, existing applications need to be adapted to instantiate objects indirectly via a factory. This can also be seen in the Telnet server implementation discussed in Section 7.2, where a factory is used to instantiate the two different handler versions.

Updates on the other hand need to be deployed via an updated context, which requires the programmer to supply the state transformation functions and write the update code specifically for ActiveContext. This could be improved by providing tool support to auto generate the transformation functions by comparing the old and new versions and only require the programmer to provide transformations manually when custom behaviour is desired.

Support for long running threads

ActiveContext is designed to support dynamic updates in multi-threaded applications and only supports updates on a per thread level. Furthermore only new threads can profit from an update. The main application thread which is started initially cannot be updated with ActiveContext. A possibility to support updates to the initial thread would be to allow the activation of an updated context via method call and not only on thread initialization. This would require further research into the consequences of such updates, and the safety of them.

Updates to primitive classes

The contexts in ActiveContext are designed as first-class entities and to instantiate the first context, a set of primitive classes needs to be present. As a consequence ActiveContext does not support updates to those primitive classes. To

support updates to primitive classes a more advanced bootstrapping system could be envisioned, where primitive classes can be loaded into the contextual model of `ActiveContext` and the system would become fully contextual.

Alternatively the design of `ActiveContext` could be adapted to support contexts at the meta-level, where the system would be fully contextual by design. This would resolve the need for primitive classes altogether.

8.2. Limitations of the Implementation

The implementation presented in this work was intended as a proof of concept and several further points would need to be considered in a serious implementation:

Support for class definition

As mentioned previously `Pinocchio` does not provide syntax for class definitions yet. As a consequence, the code for the updates cannot be supplied at runtime. We had to provide the code for the update of the Telnet server example presented in Chapter 7 before initially starting the system, and could only show the dynamical activation thereof. To have a fully working system the code for the updates needs to be loadable at runtime. A system could be imagined where the programmer uses an update controller interface similar to the one in the Telnet server example where he could upload the code for updated classes as well as the code for the updated context itself in a first step, and then activate it in a second step.

Performance overhead

The naive implementation of this paper entails a significant overhead. The slowdown of four orders of magnitude makes the current implementation unusable for productive environments. A set of performance optimizations to improve the speed drastically would be needed to that end. While a minimal overhead resulting from the indirect access to the object state will remain, other factors could be improved or removed completely and even the indirect overhead could be compiled away with a JIT compiler.

Rather than copying all non-primitive objects for each context, a copy-on-write strategy could be envisioned, removing the overhead to synchronize objects which are not edited. Also the overhead to synchronize state on each write could be reduced by using time-stamps and lazy transformations on read. Assuming that most objects “die young” and are accessed from one context only, the combination

of these optimizations could improve the performance significantly by reducing transformation to only those objects which are actually accessed concurrently from different versions of an application.

The lookup could also be improved by storing the information if an object is primitive or contextual on the object itself, and not in a `Set` as done in the current implementation.

Long-term evolution

No matter how optimized the implementation is, the system would inevitably run out of memory and performance would degrade over time as no context instance is ever discarded. All context instances are indeed connected to each other in a list which prevents them from being garbage collected. To support long-term evolution, the model should be improved with a mechanism to discard unused context instances either automatically or manually. For instance, after the system has stabilized and runs entirely in the latest context, the old context could be discarded completely to restore close-to-native performance.

Chapter 9

Conclusion

In this work we presented an overview of dynamic software update and existing research. We detected a lack of systems supporting dynamic updates in multi-threaded environments, especially for class based languages. We presented a programming model and an approach for dynamic software updates in reflective, dynamic languages aimed specifically at multi-threaded systems.

We argue that scoping state between the old and new version of a program with first-class execution contexts is an elegant way to solve two problems at once: (1) scoping state solves the safety problem of inconsistent access to new data structures from old code, and (2) first-class contexts provide an intuitive way to provide state transfer capabilities in a language.

We demonstrated the viability of the approach with a proof-of-concept implementation of our system in the Pinocchio programming language, together with an in depth tour of an example application which is dynamically updateable.

The main shortcoming of our current approach is the considerable overhead in performance.

We have sketched implementation optimizations to alleviate the overhead in performance, which should not impact the model as we have formalized it. We expect to be able to reduce the performance overhead to a level that is viable for real world applications.

9.1. Future Work

Further work could be done by implementing the suggested performance improvements to show the feasibility for real world environments.

Another promising research direction would be to analyze the impact of switching

context *inside* of a running thread and investigate possibilities to ensure safety in such a scenario.

For `ActiveContext` to be broadly usable it should be ported to other programming languages. Good candidates would be class based, dynamically typed languages. Specifically Python would seem a good choice, as it uses a VM and is used broadly. We believe `ActiveContext` could also be extended to statically-typed programming languages, following an approach similar to DVM [Malabarba *et al.*, 2000] or Jvolve [Subramanian *et al.*, 2009] which support class replacement in Java, but rely on update points. The main difficulty would be to store the type of a field while storing the field itself in a container allowing to store all kind of types at the same time.

9.2. Final Remarks

Pinocchio proved to be a very good platform to quickly prototype the `ActiveContext` model. Implementing a running example which leverages dynamic software updates with `ActiveContext` on the other hand proved to be difficult, because much of the base functionality needed was not available. Other than the telnet example we presented in this work, we started to port a full fledged web server as a more complex example, which would be closer to real world applications. The porting proved to be a lot more time consuming than anticipated as not only the functionality discussed in Section 6.2.1 but much of the `String` and `Stream` functionality is missing from Pinocchio. Also missing was the complete set of `Date`, `Time`, `DateAndTime` and all related classes which we have ported now. Once finished, the web server example could make for a much more compelling demo of `ActiveContext`.

The current implementation of Pinocchio has a hard-coded layout for basic classes like `Object`, which kept us from storing the information if a field is primitive or contextual on the objects themselves and forced us to choose a much slower implementation and store this information in a `Set`. This stands in contrast to the idea behind Pinocchio to change the language without changing the C level VM code.

As a consequence, we conclude that Pinocchio is a good choice for prototyping and for proof-of-concept implementations, but already reaches its limit when trying to implement more complex examples for such prototypes. For implementations aimed towards production systems we suggest to use a more mature programming language.

Appendix A

Installation

This appendix provides the necessary information to obtain and install the implementation referenced in this work, together with the example from Chapter 7.

At the time of writing Pinocchio requires a 64-bit UNIX like operating system like Linux or Mac OS X.

A.1. Prerequisites

The Pinocchio VM is developed in C, while the class library is developed in Pharo, a fork of the Squeak open-source Smalltalk platform. To run Pinocchio the class library needs to be exported to C sources from within a Pharo image. The exported C sources together with the Pinocchio VM sources can be compiled into a running Pinocchio instance.

To be able to compile the sources a C compiler, like the GNU Compiler Collection <http://gcc.gnu.org/> is necessary. Furthermore the Boehm garbage collector http://www.hpl.hp.com/personal/Hans_Boehm/gc/ and ncurses <http://www.gnu.org/software/ncurses/> need to be installed and the corresponding C header files need to be available. Make sure to have them installed in the 64-bit version.

To get started with Pinocchio download the VM sources. The sources are available from github <https://github.com/aldavud/p>. If you have installed git, get the sources from a shell with:

```
git clone git://github.com/aldavud/p.git
```

This will create a directory called `p` with the 3 subdirectories `src`, `pharo` and `meta`.

Appendix A. Installation

Now to get started with Pharo, download the latest “OneClick” image from the Pharo homepage <http://www.pharo-project.org/pharo-download>. Extract the “OneClick” image to `p/pharo/Pharo.app`. Please make sure to use the “OneClick” image and store it in the correct location, so the generated Pinocchio class library sources will end up in the right place.

A.2. ActiveContext

To load ActiveContext and the Pinocchio class library, start Pharo and click on the background to select the “Monticello Browser” from the world menu as shown in Figure A.1.

Add a new HTTP-based repository by clicking the “+Repository” button and selecting “HTTP”. Enter the following information:

```
MCHttpRepository
  location: 'http://www.squeaksource.com/ActiveContextP'
  user: ''
  password: ''
```

Click “OK” to save the repository and return to the main Monticello screen where you can click on “Open” to open the newly created repository. Load ActiveContext by selecting ActiveContextP and the newest version and clicking “Load” as shown in Figure A.2. Pinocchio will be pulled in as a dependency and loaded automatically.

The Pinocchio class library is available under the “Pinocchio” category, the ActiveContext sources reside under the “ActiveContextP” category.

To export the Pinocchio class library sources as well as the ActiveContext sources open the world menu again by clicking on the Pharo background and selecting “Pinocchio” and then “Export All” as shown in Figure A.3.

To compile Pinocchio navigate to the `p/src` directory in a shell and start the compilation with `make`.

After the compilation is finished the example from Chapter 7 can be run with:

```
./pinocchio telnetServer.p
```

To connect to the server open a telnet connection on port 9090, to connect to the controller open a telnet connection on port 5678. To install the updated context type `install Update1`, as seen in Figure A.4.

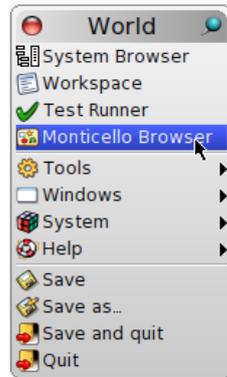


Figure A.1. Starting the Monticello Browser.

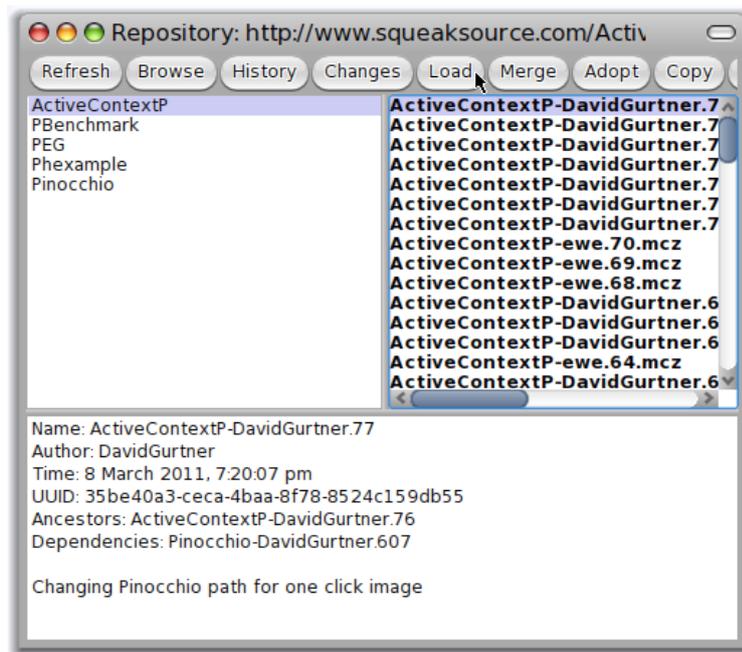


Figure A.2. Loading ActiveContext in Pharo.

Appendix A. Installation

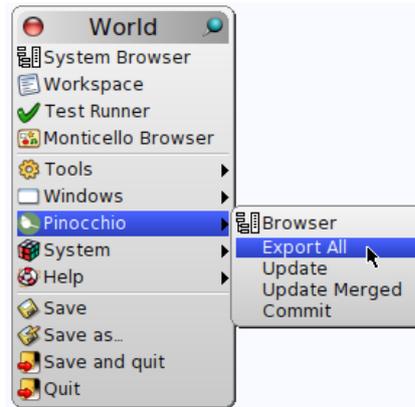


Figure A.3. Exporting the Pinocchio class library sources.

```
guru@plumeis: ~  
guru@plumeis:~$ telnet localhost 5678  
Trying ::1...  
Trying 127.0.0.1...  
Connected to localhost.  
Escape character is '^]'.  
> install Update1  
Installed Update1  
> █
```

Figure A.4. Installing the updated context.

List of Tables

3.1. Comparison of dynamic software update systems 1/2	12
3.2. Comparison of dynamic software update systems 2/2	13
6.1. The visit methods of the interpreter	32

List of Tables

List of Figures

4.1. An instance of a <code>Contact</code> object has different states in different contexts. There are transformation functions between the two contexts.	16
4.2. Context instances form a list	18
5.1. In Thread 1 Updated context is active, a <code>Contact</code> has fields <code>firstname</code> and <code>lastname</code> and is of class <code>Contact1</code>	20
5.2. State transfer	22
6.1. Native methods in the <code>Interpreter</code> and interpreter extension through sub-classing	24
6.2. The interpreter hierarchy of the <code>ActiveContext</code> implementation.	28
7.1. The adapted Telnet server interface to support dynamic software updates with <code>ActiveContext</code>	36
7.2. Version 0 of the Telnet server.	37
7.3. Version 1 of the Telnet server.	37
A.1. Starting the Monticello Browser.	51
A.2. Loading <code>ActiveContext</code> in Pharo.	51
A.3. Exporting the Pinocchio class library sources.	52
A.4. Installing the updated context.	52

List of Figures

Bibliography

- [Bracha and Ungar, 2004] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.
- [Chen *et al.*, 2007] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.
- [Denker *et al.*, 2007a] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
- [Denker *et al.*, 2007b] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.
- [D’Hondt, 2008] Theo D’Hondt. Are bytecodes an atavism? In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*, pages 140–155. Springer-Verlag, Berlin, Heidelberg, 2008.
- [Dmitriev, 2001] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, October 2001.
- [Ebraert *et al.*, 2005] Peter Ebraert, Yves V, and E Berbers. Pitfalls in unanticipated dynamic software evolution. In *Cazolla W., Ed., In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution in conjunction with the 18th European Conference on Object-Oriented Programming*, pages 41–49, 2005.

Bibliography

- [Makris and Bazzi, 2009] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference, USENIX'09*, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association.
- [Malabarba *et al.*, 2000] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.
- [Neamtiu *et al.*, 2006] Iulian Neamtiu, Michael Hicks, Gareth Stoye, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06*, pages 72–83, New York, NY, USA, 2006. ACM.
- [Orso *et al.*, 2002] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002.
- [Redmond and Cahill, 2002] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.
- [Rivard, 1996] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.
- [Segal and Frieder, 1993] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10:53–65, 1993.
- [Subramanian *et al.*, 2009] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09*, pages 1–12, New York, NY, USA, 2009. ACM.
- [Verwaest *et al.*, 2010] Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard, and Oscar Nierstrasz. Pinocchio: Bringing reflection to life with first-class interpreters. In *OOPSLA Onward! '10*, 2010.
- [Verwaest, 2009] Toon Verwaest. Pinocchio — an open system for language experimentation, June 2009. <http://scg.unibe.ch/pinocchio>.

