Using First-class Contexts to realize Dynamic Software Updates

Erwann Wernli David Gurtner Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland

`http://scg.unibe.ch/`

Abstract

Applications that need to be updated but cannot be easily restarted must be updated at run-time. We evaluate the reflective facilities of Smalltalk with respect to dynamic software and the state-of-the-art in this field. We conclude that while fine for debugging, the existing reflective facilities are not appropriate for dynamically updating production systems under constant load. We propose to enable dynamic updates by introducing first-class contexts as a mechanism to allow multiple versions of objects to coexist. Object states can be dynamically migrated from one context to another, and can be kept in sync with the help of bidirectional transformations. We demonstrate our approach with *ActiveContext*, an extension of Smalltalk with first-class contexts. ActiveContext eliminates the need for a system to be quiescent for it to be updated. ActiveContext is realized in Pinocchio, an experimental Smalltalk implementation that fully reifies the VM to enable radical extensions. We illustrate dynamic updates in ActiveContext with a typical use case, present initial benchmarks, and discuss future performance improvements.

1. Introduction

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

Changing the system at run-time for debugging purposes has long been a common practice in dynamic object-oriented languages such as JavaScript, Ruby and Smalltalk. In the case of Smalltalk, as it is fully reflective, there is actually no other way to change the system than to adapt it at run-time, and development in Smalltalk is conducted with this in mind.

In this paper we discuss the requirements for the dynamic update of production systems, and evaluate the reflective capabilities of Smalltalk according to these requirements. We consider three main dimensions during this evaluation: *safety*, the ability to ensure that dynamic updates will not lead to abnormal executions, *timeliness*, the ability to install the update quickly, and *practicality*, the fact that the dynamic software update mechanism must not constrain developers. The outcome of this analysis is that while the existing reflective mechanisms are fine for debugging, they are not adequate to update production systems under constant load, notably because of safety issues.

As a remedy to the problems we have identified, we propose ActiveContext, an extension of Smalltalk with first-class contexts. Contexts have two roles: they dynamically scope software versions, and they mediate access to objects that can be migrated back and forth between versions. As a consequence, the single abstraction of a first-class context enables not only the *isolation* of software versions, but also the *transition* from one version to another.

Making contexts *first-class* empowers the developers with more control over how dynamic updates should happen; it shifts part of the responsibility of the update from the system to the application. This way, it becomes possible to tailor the update scheme to the nature of the application, *e.g.*, rolling out new code on a per-thread basis for an FTP server, or on a per-session basis for a web application.

Entities shared between several versions are mediated by the system and code running entirely in one version is guaranteed to be always consistent. The abstraction of context is intuitive and can be implemented with a reasonable overhead. ActiveContext qualifies as a safe and practical approach to dynamic software update.

As a validation, we have implemented ActiveContext in Pinocchio, an experimental Smalltalk platform that fully reifies the runtime to enable invasive language extensions. We demonstrate a typical use case with a Telnet server.

First we discuss the challenges of dynamic software update and review existing literature in section 2. In section 3 we present our approach with the help of a running example. In section 4 we present the model in more detail, and in section 5 we present an implementation. We discuss future work in section 6 before we conclude in section 7.

2. Why dynamic updates are challenging

The main challenge of dynamic updates is achieving an optimal compromise between safety, timeliness and practicality. Safety means that dynamic updates are guaranteed to not lead to abnormal executions ; timeliness means that updates are installed immediately and instantly; practicality means that the system does not impose additional constraints during development or operation. We resort to common sense for why these characteristics are desirable.

To understand why there are tensions between these properties, let's consider an update that alters a method signature. Once installed, subsequent invocations of the method will use the updated method body that expects the newest list of arguments. It is clear that installing the change immediately is unsafe, as *active methods* on the stack might still presume the old signature, which will lead to type errors [21]. To increase safety, type errors can be prevented with the proper timing of the update using automated checks, but the behavior of the program might still be incorrect depending on the change in the program's logic [23]. Manual assistance to define safe update times is required [12, 22]. This affects negatively both practicality and timeliness.

Obviously, the layout of classes (the set of fields and methods) can change as well, which means the program state (object instances) must be adapted correspondingly during the update. If the update is immediate, active methods might presume the old type and lead to inconsistent accesses to state. Automated checks to delay the update can prevent such type errors, but are not enough. In the case of state, not only should we consider when to migrate the state, but how : existing invariants in the program state must be preserved and safe updates require manual assistance to provide adapters that will migrate the old state to a new, valid, state. This impacts practicality negatively.

A way to reconcile safety and timeliness is to restrict the update to only certain changes, *e.g.*, changes that do not alter types, or changes that are behavior-preserving [19], but this impedes practicality.

Note that transferring the state for large heaps has the same tensions between safety, timeliness and practicality : state transfer in a stop-the-world fashion is safe and practical but compromises timeliness, while lazy state transfer is timely but either unsafe, or less practical [2] depending on the design.

2.1 Assessment of Smalltalk

Now that we have explored the core reasons of these tensions, let's focus on Smalltalk and assess its reflective capabilities according to safety, timeliness and practicality:

Safety. Smalltalk initializes new fields to `nil` and does not allow to customize state transfer such that it maintains existing invariants in the program state.

Run-time errors can occur after an update. When a field is removed, all methods of the class are recompiled. Accesses to the suppressed field return `nil` instead, and assignment to the suppressed field are ignored. Old versions of the method existing on the stack might continue to run, which can lead to severe errors such as the mutation of instance variable at the wrong index, or even the crash of the entire image. Method suppression does not suffer such severe symptoms, as method resolution is dynamic, and at worst raises a `doesNotUnderstand` error.

Smalltalk does not support the atomic installation of corelated changes to multiple classes. An execution that uses a mix of old and new versions of the classes might be incorrect.

Timeliness. Changes are installed immediately. Object instances are migrated in a stop-the-world fashion. Changes on classes that are higher in the class hierarchy might result in the recompilation of many subclasses, as well as the migration of their object instances, which might take long.

Practicality. Arbitrary changes to method signature, method body, class hierarchy or class shape are supported. There is no overhead after the installation of the update.

As this analysis shows, the reflective capabilities of Smalltalk are timely and practical, but not safe, which makes them inadequate to update production systems on the fly. In practice, developers rely on ad-hoc scripts and techniques to update their Smalltalk image in production.

2.2 Other approaches

A large body of research has tackled the dynamic update of applications, but no mechanism resolved all three tensions previously presented. Existing approaches can be classified into three categories:

1. Systems supporting *immediate and global dynamic update* have been devised with various levels of safety and practicality. Dynamic languages other than Smalltalk belong naturally to this category; they are very practical but not safe. Dynamic AOP and meta-object protocols also fit into this category. Systems for Java [5, 8, 11, 16, 24, 26] of this kind have been devised. However, they are less practical and impose restrictions on the kinds of changes supported, due to Java's type system [30]. For example, only method bodies can be updated with HotSwap [8]. Other systems have tried to reconcile practicality and static typing, at the expense of timeliness or safety. For example, some updates will be rejected if they are not provably type-safe [20] or might produce run-time errors [33].
2. Several approaches have tackled the problem of safety by relying on *update points* to define temporal point when it is safe to globally update the application. Such systems have been devised for C [14, 22], and Java [28]. Update points might be hard to reach, especially in multi-

threaded applications [21], and this compromises the timely installation of updates.

3. Some approaches do not carry out the update globally and allow different versions of the entities to coexist at run-time. Different variations of this scheme exist. With dynamic C++ classes [15], the structure of existing objects is not altered, and only new objects have the new structure ; the code is however updated globally which is unsafe. With Gemstone [10], class histories enable different versions of classes to coexist, and objects can be migrated on demand from one version to another ; this is more flexible than an immediate migration but is still unsafe. A strategy that is safe is to adapt the entities back and forth when accessed from different versions of the code using bi-directional transformations. To the best of our knowledge, only two approaches have pursued the latter one: a dynamic update system for C [6], and a type system extended with the notion of run-time version tags that enables hot swapping modules [9].

More generally, dynamic software updating relates to techniques that promote *late binding*. Three main categories of such techniques can be listed: support for virtual classes [18], isolation of software versions (Java class loader [17], ChangeBox [7], and ObjectSpace [4]), and fine-grained scoping of variations (selector namespaces, context-oriented programming [29], classbox [1]). Finally, the migration of instances relates to the problem of *schema evolution* [25] and techniques to convert between types (expanders [32], translation polymorphism [13], implicit conversion [27]). None of these techniques is in itself sufficient to enable dynamic software update though.

3. Our approach — ActiveContext

We believe that dynamic software updates should be addressed explicitly (*i.e.*, reflectively), so as to give developers control over when new code becomes active. Developers should be able to implement an appropriate update scheme for the application, such as roll out new code on a per-thread basis for an FTP server, or on a per-session basis for a web application.

Most approaches for dynamic software updates are either non-reflective, or reflective but lack safety. One notable exception is the work by Duggan [9], which is both reflective and safe. However, it relies on static typing, and the update of a module impacts all modules depending on it: they must all be updated and use the new type, which is not practical. It also fails to support the atomic change of multiple types at once.

ActiveContext is an approach that aims to introduce an ideal dynamic software update mechanism that is explicit and satisfies all requirements established previously. It introduces first-class contexts into the language in a way that reifies software versions. Contexts can be loaded, instantiated, and manipulated explicitly. A context defines a dynamic scope, within which code can be executed, which is just as simple as `aContext do: [ .... ]` .

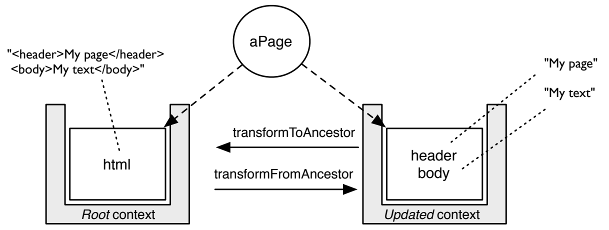

Figure 1. An instance of a `Page` object has different states in different contexts. There are transformation functions between the two contexts.

Contexts also encode bi-directional transformations for objects whose representations differ between contexts, but whose consistency is maintained by the system. Thanks to the mediation of objects using bi-directional transformations, code running in different contexts can coexist at run-time, yet safely operate on shared data. ActiveContext belongs to the third kind of approach described in subsection 2.2.

3.1 ActiveContext examplified

To illustrate our approach to dynamic software updates, let us consider the evolution of a class in a Web Content Management System like Pier[1]. The main entity of the domain model of such a system is the `Page`. It represents the source code of an HTML document. The domain model is held in memory, globally accessible, and shared amongst all threads serving HTTP requests. Let us consider the evolution shown in Figure 1. The `Page` class is refactored, and the data stored originally in the `html` field is now stored in two individual fields `body` and `header`.

Such an evolution cannot be easily achieved with the reflective facilities of Smalltalk: it would require an "intermediate" version of the class with all three fields `html`, `body`, `header` in order to allow the state of the concerned object instances to be migrated incrementally, for instance with `Page allInstances do: [...]`. Only then could the `html` field be removed. Such an update is not only complicated to install, but is also not atomic, possibly leading to consistency issues.

Dynamic software update mechanisms that carry out immediate updates (first category in subsection 2.2) face the risk that some existing thread running old code may access the `html` field which no longer exists. Those which use update points (2nd category in subsection 2.2) would still have to wait until all requests complete prior to a global update of the system state.

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"

[1] `http://www.piercms.com`

an update to the system and activate it. Here is how one would typically adapt a Web Content Management System or any server-side software serving requests.

0. *Preparation.* First, a global variable `latestContext` is added to track the latest execution context to be used. Second, an administrative page is added to the Web Content Management System where an administrator can push updates to the system; the uploaded code will be loaded dynamically. 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:

```
latestContext do: [
    [ anIncomingRequest process ] fork.
]
```

After these preliminary modifications the system can be started, and now supports dynamic updates. The lifecycle 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 `html` is replaced with the two fields `body` and `header`. Figure 1 shows the impact on the state of a page.
3. *Update preparation.* The developer creates a class, say called `UpdatedContext`, that specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transformation which converts the program state between the *Root* context and the *Updated* context. Objects will be transformed one at a time.

   In our example, the field `html` is split into `body` and `header` in one direction, and the fields `body` and `header` are joined into `html` in the other direction. The class of an object is considered to be part of the object's state and the transfer function also specifies that an updated version of the `Page` class will be used in the *Updated* context.

   Contexts may coexist at run-time for any length 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 need to be updated though; fields that have been added or removed and have no counterpart in another context can naturally be omitted from the transformations.
4. *Update push.* Using the administrative web interface, the developer uploads the updated `Page` 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 pages in the system. Objects now have two representations in memory. Last, the global variable `latestContext` is updated and refers to the newly created instance of the *Updated* context.

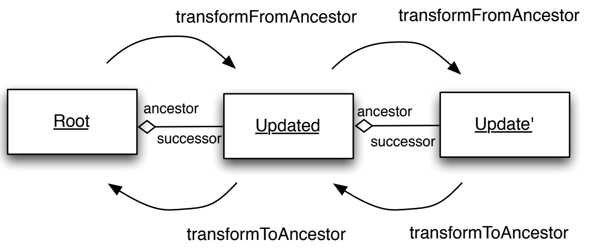

Figure 2. Context instances form a list

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request. The active context that was dynamically changed in the listener thread (see point 0) propagates to the spawned thread. The execution context will be the context referenced in `latestContext`, which is now the *Updated* context.
6. *Stabilization.* This update scheme changes the execution context 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 page can always be accessed safely from one execution context or another as the programming model maintains the consistency of various representations using the bidirectional transformations. This alleviates the need for global, temporally synchronized update points which can be 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, and then loaded and instantiated dynamically. Contexts are related to each other with an ancestor/successor relationship. They form a list, with the *Root* context as oldest ancestor, as shown in Figure 2.

7. *Garbage collection.* When no code runs in the oldest context any longer, the context can be removed from the list and be garbage collected, just as the representation of objects in it. (This step is not implemented yet)

4. The ActiveContext Model

We now present the ActiveContext model in more detail. ActiveContext makes a clear distinction between the identity and the *contextual 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 dynamic scoping of state and migration of state between contexts.

4.1 Identity, State and Contexts

An object identifier uniquely identifies an object in any given context (where that object exists). The state of an object may, however, vary from context to context. The contextual state of an object consists of (i) a set of fields and their corresponding values, and (ii) its class, which depends on the context. Of course, the fields of the object must match the fields declared in the (contextual) class description.

An object can have as many states as there are contexts. A context can be seen as a mapping between the global identities of all objects and their corresponding states in that context. A thread can have one *active* context at a time. However, the active context of a thread can be switched any time. Contexts consequently constitute dynamic scopes: `aContext do: [...]`.

The interpreter or virtual machine has an 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, it inherits the context of its parent thread, which will be the *active* context for that thread of execution as soon as it starts running.

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 reduces to a special case of scoping state.

4.2 Transformations

As shown in Figure 2, contexts form a list at run-time. They must have an `ancestor`, and they must implement two methods `transformFromAncestor` and `transformToAncestor` that realize the bidirectional transformation.

The role of the bidirectional transformation is to maintain 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.

As contexts are loaded dynamically in an unanticipated 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 another method to transform from the newest context *to* its ancestor. The *Root* context is the only context that does not encode any transformation.

A sample one-way transformation is shown in Figure 3. It corresponds to the transformation from the *Root* context to the *Updated* context of Figure 1. `self` refers to the *Updated* context, and `ancestor` to the *Root* context. Line 5 reads the `html` of the `Page` in the *Root* context. Lines 7–8 split the html into `body` and `header`, and the representation of the `Page` in the *Updated* context is updated accordingly in lines 11–12.

```
1.  transformFromAncestor: id
2.      | cls html body header |
3.      cls := ancestor readClassFor: id.
4.      ( cls = Page ) ifTrue: [
5.          html := ancestor readField: 'html' for: id.
6.          html isNil ifFalse: [
7.              body:= html regex: '<body>(.*)</body>'.
8.              header:= html regex: '<header>(.*)</header>'.
9.          ].
10.         self writeClassFor: id value: Page2.
11.         self writeField: 'body' for: id value: body.
12.         self writeField: 'header' for: id value: header.
13.     ]
14.     ( cls = AnotherClass ) ifTrue: [
15.         ...
16.     ]
17.     ...
```

Figure 3. State transfer—one-to-one mapping between two versions of a class.

4.3 Meta levels

Contexts are meta-objects that are causally connected with the runtime: field writes and object instantiations will be evaluated differently depending on the set of contexts and their corresponding transformations.

Before a context can be used, it must first be registered via `aContext register`. This establishes the causal connection between a context and the runtime. The registration will also create the new representation of all contextual objects in the newly registered context. After this step, the context can be used and only of objects that are created or modified will need to be synchronized later on. This way, all contextual objects have a valid representation in all existing contexts anytime.

Transformations are never called directly by the application, but by the run-time itself because of the causal connection. This corresponds to two distinct meta-levels, that we refer to as the *application* level and the *interpreter* level: user-written code runs at the application level, except for transformations that run at the interpreter level.

4.4 Primitive

Contexts are connected with the run-time, and their state is accessed by the interpreter or virtual machine itself (not only other application objects), which means contexts can't be contextual. If they were, then the code of the interpreter would also be contextual, and it would need to be interpreted by another interpreter. To avoid the infinitive meta-regression, code running at the interpreter level runs outside of any context. As a consequence, some objects in the system must be *primitive*: they have a unique state in the system and are not subject to contextual variations. Context objects are an example.

```
1.  transformFromAncestor: id
2.      | cls holder email body header |
3.      cls := ancestor readClassFor: id.
4.      ( cls = Contact ) ifTrue: [
5.          email := ancestor readField: 'email' for: id.
6.          email isNil ifFalse: [
7.              alias := email regex: '(.*)@'.
8.              host := email regex: '@(.*)'.
9.              self interpret: [
10.                 holder := Email new.
11.                 holder alias: alias.
12.                 holder host: host.
13.                 holder contact: id.
13.             ].
14.         ].
15.         self writeClassFor: id value: Contact2.
16.         self writeField: 'email' for: id value: holder.
17.     ]
18.     ( cls = AnotherClass ) ifTrue: [
19.         ...
20.     ]
21.     ...
```

Figure 4. State transfer—refactoring and usage of the `interpret` keyword to switch between levels.

4.5 Mirror

The fact that transformations run at the interpreter level, outside of any context, implies that one can send messages only to primitive objects in the transformations. Contextual objects must be manipulated *reflectively* via a context with `readClassFor:`, `writeClassFor:value:`, `readField:for:` and `writeField:for:value:` as shown in Figure 3. With these language constructs, a context acts as a mirror [3] that reifies the state of an object in this particular context. This way, the state of an object (class or fields) in an arbitrary context can be updated without subsequent transformations being triggered, and independently of the active context.

4.6 Interpretation

In complex transformations, it can be necessary to evaluate a whole block in a given context to manipulate contextual objects. This can be achieved with `aContext interpret: [...]`, which will evaluate the block in the given context as if it was executed at the application level.

Unlike with mirrors, subsequent transformations will be triggered in this case when contextual objects are modified. This is necessary in particular to instantiate a new contextual object from within a transformation. If subsequent transformations were not triggered, the new object would be missing representations in other contexts.

The evaluation of `do:` and `interpret:` are similar. The difference between `do:` and `interpret:` is that `do:` expects to switch from a current context to another and must be called from the application level, while `interpret:` must be called from the interpreter level, and expects that there is no current context.

Transformations can be more complex than one-to-one mappings and Figure 4 shows the usage of `interpret:` in the case of the refactoring of an email string of the form

Keyword and description
// Creates the causal connection between a context and the runtime
`aCtx register`
// Evaluate the block in the given context (used from the interpreter level)
`aCtx interpret: [ ... ]`
// Evaluate the block in the given context (used from the application level)
`aCtx do: [ ... ]`
// Read a value from a field reflectively
`aCtx readField: aFieldName for: aCtxObj`
// Write a value to a field reflectively
`aCtx writeField: aFieldName for: aCtxObj value: aValue`
// Read the class reflectively
`aCtx readClassFor: aCtxObj`
// Write the class reflectively
`aCtx writeClassFor: aCtxObj value: aClass`

Table 1. Keywords to manipulate contexts and contextual objects.

"alias@host" into a dedicated `Email` holder object with field `alias` and `host` (inspired by a case found in practice [28]). Code between lines 10–13 runs at the application level in the context referred by `self`, and line 10 instantiates a new contextual object.

4.7 The big picture

Table 1 recapitulates the syntax to deal with contexts and contextual objects, and Figure 5 visually summarizes the abstractions presented earlier.

5. Implementation

We report on the implementation of ActiveContext in Pinocchio [31][2], an experimental Smalltalk system designed to enable invasive changes to the runtime system. Interpreters are first-class in Pinocchio. The default meta-circular interpreter of Pinocchio can be subclassed to create custom interpreters and experiment with programming language variations. Pinocchio is otherwise similar to conventional Smalltalk systems. It supports in particular the object model and reflective architecture of Smalltalk 80.

5.1 Implementation Details

Pinocchio represents code as abstract syntax trees, or ASTs. Code is evaluated by an interpreter that visits nodes of the AST. The state of a Pinocchio object is stored in slots (first-class fields), which are represented as AST nodes.

Table 2 shows the relevant visitor methods of the interpreter, and indicates which ones were overridden to implement ActiveContext. The ActiveContext interpreter changes the way state and memory is managed, in particular the treatment of slots and message sends, whose corresponding visit methods have been overridden accordingly. Only contextual objects are treated specially. Primitive objects delegate to the native memory management of Pinocchio to avoid any performance overhead. A similar decision was taken for the

[2] `http://scg.unibe.ch/pinocchio`

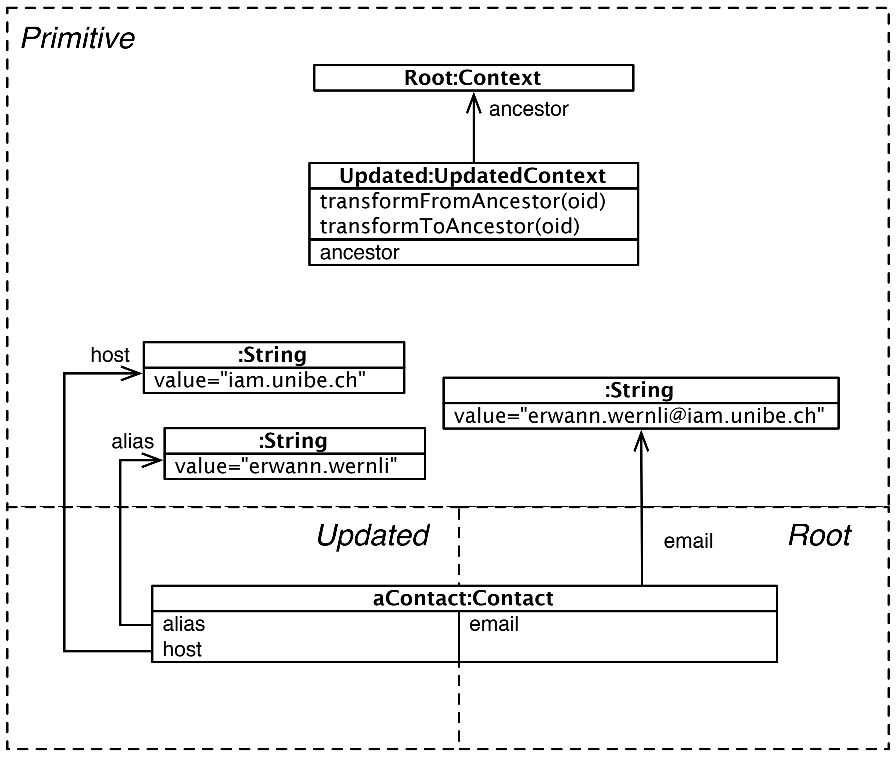

Figure 5. Conceptual overview of the system at run-time. It shows a system with the *Root* context, the *Updated* context, three primitive `String` objects, and one contextual `Contact` object. The memory is conceptually divided into three segments, one for primitive objects, one for the objects' representation in the *Root* context, and one for objects' representation in the *Updated* context.

Visitor method	Overriden
`visitConstant: aConstant`	·
`visitClassReference: aClassReference`	·
`visitVariable: aVariable`	·
`visitSlot: aSlot`	✓
`assignVariable: aVariable to: value`	·
`assignSlot: aSlot to: value`	✓
`visitAssign: anAssign`	·
`visitSelf: aSelf`	·
`visitSend: aSend`	✓
`visitSuper: aSuper`	✓

Table 2. The visitor methods of the interpreter.

Root context, which also delegates to the native memory management even for contextual objects.

Internally, the interpreter uses several `Dictionary` instances to implement the memory model for contextual objects: one dictionary per registered context is created and maps {`object identity, field`} to the corresponding value. A special field `class` is used for the class of the object. This implies one level of indirection to access the state of a contextual object.

The active context is referenced in a field of the interpreter and each thread has a dedicated instance of the interpreter. To distinguish between primitive or contextual objects, we maintain two pools of references represented internally with two `Sets` (this is admittedly a naive approach: tagging the pointer would be faster).

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 `BaseContext` class to be defined, so for them to be contextual would lead to a chicken-and-egg problem. The same also holds true for basic, immutable objects like `nil`, `true` and `false`, as well as numbers, characters, strings and symbols.

The keywords in Table 1 have been implemented with regular message sends that the interpreter handles in a special way.

5.2 Demonstration

To validate our approach, we have implemented a canonical server-side application that illustrates our approach and follows the use case of section 3. The application we im-

		new	**send**	**read**	**write**
Primitive	Metacircular	191	215	137	137
	Root	393	297	215	219
	Updated	492	294	217	283
Contextual	Metacircular	192	146	137	200
	Root	347	229	292	294
	Updated	**935**	341	259	**598**

Table 3. Benchmark—Milliseconds for 500 executions of specific instructions for contextual and primitive objects using the meta-circular interpreter and the ActiveContext interpreter with one context (*Root*) and two contexts (*Root* and *Updated*).

plemented is a Telnet server. A client can connect to the server and run a few simple commands to update a list of contacts stored in memory. While simpler than a web-based Content Management system, such a system exhibits comparable characteristics in term of design and difficulties with respect to dynamic updates.

The telnet server was adapted according to step *Preparation* in section 3. First, a global variable `latestContext` was introduced. Second, the main loop that listens for incoming TCP connections was modified so that when a new connection is accepted, the corresponding thread that is spawned to handle the connection is executed in the latest context. Third, a client connects to the server and usees special commands to upload code and "push" an update.

The system can then be bootstrapped and runs initially in the *Root* context. An administrator can connect to the server and use a special command to upload an update. The classes implementing the logic to process commands can in particular be changed to change the logic of an existing command, or to add new ones. All commands executed as well as their versions are logged in a file. A thread handling a specific client connection keeps running as long as the connection is established. Already connected clients that use the original version are not impacted by the update and multiple clients connected to the server might see different versions of the command-line interface.

When a client disconnects, its server-side thread terminates. The system stabilizes eventually when all clients have disconnected. The log shows the various commands executed over the time and the migration of the server from one version to another.

We benchmarked object creation, message send, field read and field write for primitive and contextual objects under the three configurations of the system that are described in Table 3.

The meta-circular interpreter serves as the baseline for comparison. When running the benchmark with this interpreter, contextual and primitive objects are treated in the same way and results are then similar. When running the benchmark with the ActiveContext interpreter with solely the *Root* context, our implementation delegates to the native memory for primitive and contextual object. Results for both kinds of object are in the same range, but slower than on the meta-circular interpreter due to the overhead of our interpreter. When running the benchmark with the ActiveContext interpreter and two contexts (*Root* and *Updated*), we perceive a small performance drop for primitive objects, but a significative performance drop for contextual objects, notably for operations new and write (in bold in the table). This can be explained easily: (1) send and read operations for contextual objects need to look up data in internal dictionaries, and (2) in addition to the lookup in dictionaries, operations new and write need to trigger transformations to synchronize the data in the *Root* context.

Looking at these results, the worst performance drop is in the range of a factor 5 for contextual object creation (935 ms vs. 191 ms). These results will vary depending on the structure of the objects, the number of objects created and maintained in the pool of references, and the number of registered contexts that need to be synchronized, as well as the complexity of the transformations. This suffices however to estimate the maximum performance degradation to one order of magnitude. We believe this performance degradation can be reduced by a smarter implementation. We consider this to be a validation of the conceptual contribution and a positive feasibility study.

5.3 Assessment of ActiveContext

Let us assess ActiveContext according to the safety, timeliness, and practicality, as we did for vanilla Smalltalk in section 2:

Safety. Custom state transfer can be specified to transition from one version to the other. Code running in a given version will not produce run-time type errors due to dynamic updates. Contexts also help address safety that is beyond typing errors: it provides version consistency. Contexts enable the atomic installation of co-related changes to class, and ensure that code running in a context always corresponds to one precise version of the software.

Timeliness. If the synchronization is performed lazily, the creation of a new software version entails no overhead, and it can be used immediately after creation.

Practicality. Contexts are simple abstractions that are easy to use. They extend the language and do not impose restrictions. Writing the transformations manually is extra work, but it is acceptable if updates are not too frequent, *e.g.*, during maintenance phase. The overhead for synchronizing objects is significant, but it can be dramatically improved by (1) synchronizing only objects that are actually shared, and (2) synchronizing lazily.

ActiveContext extends the reflective architecture with features that enable the update of production system safely.

6. Future work

This paper presents a conceptual model for systems to support dynamic software updates, as well as a prototype to demonstrate the soundness of the approach. Several further points would need to be considered in a full implementation:

Lazy transformation and garbage collection The model that we have presented and implemented uses eager transformations: the state of objects is synchronized after each write. This entails significant overhead for objects whose lifetime is short, and are never accessed from another context than the one in which they were created. This also entails high memory consumption as we keep as many representations for an object as we have contexts. All context instances are connected to each other in a list which prevents them from being garbage collected. With eager transformations, long-lived objects consume more and more memory and become slower and slower to synchronize.

More appealing are *lazy transformations*: instead of synchronizing their state eagerly on write, it is synchronized lazily on read, in a way similar to how caches work. Not only would this reduce the performance overhead, but also reduce memory consumption as only the most up-to-date representation would be kept in memory. There should be a significant overhead only for objects whose structure has changed and has been accessed from several contexts.

Keeping only the most up-to-date representation assumes that the transformation is *lossless*, that is, one representation can be computed out of another one without loss of data. This is not always the case, *e.g.*, in case of field addition or removal with no counterpart in the other context. Such transformations are said to be *lossy*. One idea would be to track which transformations are lossy or not, and only keep multiple versions of objects impacted by lossy transformations.

We plan to implement lazy transformations, to distinguish between lossy and lossless transformations for further optimizations, and to enable garbage collection of unused contexts using weak references in our implementation.

Interdependent class evolution The object graph can be navigated during the transformation, which makes our approach very flexible to support arbitrary forms of evolution and interdependent class evolution, as was shown in the refactoring of Figure 4. Other approaches with similar facilities to navigate the object graph proved to support most scenarios of evolution in practice [2, 20, 28]. Keeping several versions of objects in memory is necessary until an update has been installed completely [2]. This puts memory pressure on the system, regardless of whether the transformations happen lazily or eagerly. One promising aspect of our approach with bi-directional transformations is that the old representation can in principle be recovered at any time; we could avoid keeping multiple representations (at least for objects subject to lossless transformations) and thus relieve the memory pressure.

Versioning of class hierarchies In our current implementation, classes are not contextual objects and this implies that two versions of a class have distinct names across contexts (see line 10 in Figure 3). In a more elaborate implementation, the same class name could be used and would resolve to a different representation of the class. The contextual class state would include `methodDict` and `super`. This would enable the fine-grained evolution of class hierarchies: the superclass of a class could differ in two contexts (without the subclass being modified), and conversely, two versions of a subclass could have different superclasses in two contexts. Metaclasses could possibly also be contextual but some classes would need to be primitive and would not be resolved contextually, for the same reasons that we distinguish between primitive objects and contextual objects (see subsection 4.3).

7. Conclusion

We have presented a novel approach to dynamically update software systems written in dynamic languages. ActiveContext is a programming model that extends the reflective capabilities of a dynamic language with first-class contexts to support the coexistence and synchronization of alternative representations of objects in memory. With ActiveContext, existing threads run to termination in the old context while new threads run in a new context. Program state will eventually migrate from the old to the new context, and during the transition period the state will be synchronized between contexts with the help of bi-directional transformations. We showed that ActiveContext is safe, practical, and timely. It empowers the developer with more control over dynamic updates, and does not require that the system be quiescent to be updated. We have demonstrated how to build a dynamically updatable system with a typical use case. The next step is to introduce lazy transformation and enable garbage collection, which should improve performance and further reduce memory consumption.

Acknowledgments

We would like to thank Mircea Lungu, Adrian Lienhard, Niko Schwarz and Toon Verwaest for kindly reviewing earlier drafts of our paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012).

References

[1] A. Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Bern, Nov. 2005. URL `http://scg.unibe.ch/archive/phd/bergel-phd.pdf`.

[2] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003. ISSN 0362-1340. doi: 10.1145/949343.949341. URL `10.1145/949343.949341`.

[3] G. Bracha and D. 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. URL `http://bracha.org/mirrors.pdf`.

[4] G. Casaccio, D. Pollet, M. Denker, and S. Ducasse. Object spaces for safe image surgery. In *IWST '09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 77–81, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5. doi: 10.1145/1735935.1735948.

[5] S. Cech Previtali and T. R. Gross. Aspect-based dynamic software updating: a model and its empirical evaluation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 105–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0605-8. doi: 10.1145/1960275.1960289. URL `http://doi.acm.org/10.1145/1960275.1960289`.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. 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. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.65.

[7] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. 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. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352681. URL `http://scg.unibe.ch/archive/papers/Denk07cChangeboxes.pdf`.

[8] 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*, Oct. 2001.

[9] D. Duggan. Type-based hot swapping of running modules. In *Intl. Conf. on Functional Programming*, pages 62–73, 2001.

[10] Gemstone. Gemstone/s programming guide, 2007. URL `http://seaside.gemstone.com/docs/GS64-ProgGuide-2.2.pdf`.

[11] A. R. Gregersen and B. N. Jørgensen. Dynamic update of Java applications — balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21:81–112, mar 2009. ISSN 1532-060X. doi: 10.1002/smr.v21:2. URL `http://portal.acm.org/citation.cfm?id=1526497.1526501`.

[12] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996. ISSN 0098-5589. doi: 10.1109/32.485222. URL `http://portal.acm.org/citation.cfm?id=229583.229586`.

[13] S. Herrmann, S. Herrmann, C. Hundt, C. Hundt, K. Mehner, and K. Mehner. Translation polymorphism in object teams. Technical report, Technical University Berlin, 2004.

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

[15] G. Hjálmtýsson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1268256.1268262`.

[16] J. Kabanov. Jrebel tool demo. *Electron. Notes Theor. Comput. Sci.*, 264:51–57, feb 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.02.005. URL `http://dx.doi.org/10.1016/j.entcs.2011.02.005`.

[17] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998. doi: 10.1145/286936.286945.

[18] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, Oct. 1989.

[19] K. Makris and R. 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. URL `http://portal.acm.org/citation.cfm?id=1855807.1855838`.

[20] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. 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. ISBN 3-540-67660-0. doi: 10.1007/3-540-45102-1_17.

[21] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1543135.1542479.

[22] I. Neamtiu, M. Hicks, G. Stoyle, and M. 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. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133991. URL `http://doi.acm.org/10.1145/1133981.1133991`.

[23] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1):37–49, 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328447.

[24] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002. doi: 10.1109/ICSM.2002.1167829. URL `http://dx.doi.org/10.1109/ICSM.2002.1167829`.

[25] M. Piccioni, M. Oriol, B. Meyer, and T. Schneider. An ide-based, integrated solution to schema evolution of object-

oriented software. In *ASE*, pages 650–654, 2009.

[26] B. Redmond and V. 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. doi: 10.1007/3-540-47993-7_9.

[27] scala. The scala programming language. URL `http://lamp.epfl.ch/scala/`. http://lamp.epfl.ch/scala/.

[28] S. Subramanian, M. Hicks, and K. 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. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542478. URL `http://doi.acm.org/10.1145/1542476.1542478`.

[29] E. Tanter. Contextual values. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: 10.1145/1408681.1408684. URL `http://doi.acm.org/10.1145/1408681.1408684`.

[30] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Influence of type systems on dynamic software evolution. CW Reports CW415, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2005. URL `https://lirias.kuleuven.be/handle/123456789/131703`.

[31] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Nierstrasz. Pinocchio: Bringing reflection to life with first-class interpreters. In *OOPSLA Onward! '10*, 2010. doi: 10.1145/1869459.1869522. URL `http://scg.unibe.ch/archive/papers/Verw10aPinocchio.pdf`.

[32] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167477.

[33] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852764.