Incremental Dynamic Updates with First-class Contexts

Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz

Software Composition Group
University of Bern, Switzerland
`http://scg.unibe.ch`

Abstract. Highly available software systems occasionally need to be updated while avoiding downtime. Dynamic software updates reduce downtime, but still require the system to reach a quiescent state in which a global update can be performed. This can be difficult for multi-threaded systems. We present a novel approach to dynamic updates using first-class contexts, called *Theseus*. First-class contexts make global updates unnecessary: existing threads run to termination in an old context, while new threads start in a new, updated context; consistency between contexts is ensured with the help of bidirectional transformations. We show how first-class contexts offer a practical and flexible approach to incremental dynamic updates, with acceptable overhead.

Keywords: dynamic language; dynamic software update; reflection

1 Introduction

Real software systems must be regularly updated to keep up with changing requirements. Downtime may not be tolerable for highly available systems, which must then be updated dynamically, *e.g.*, web servers. The key challenge for dynamically updating such systems is to ensure consistency and correctness while maximizing availability.

The most popular scheme for dynamic updates is to interrupt the application to perform a global update of both the code and the state of the program [19,26,25]. Such updates are inherently unsafe if performed at an arbitrary point in time: running threads might run both old and new code in an incoherent manner while old methods on the stack might presume type signatures that are no longer valid, possibly leading to run-time type errors. Quiescent global update points must be selected to ensure safe updates, but such points may be difficult to reach for multi-threaded systems [18,26]. More generally, a global update might not be possible due to the nature of the change, for example it would fail to update anonymous connections to an FTP server that mandates authentication after the update: the missing information cannot be provided *a posteriori* [19].

Instead of global updates, we propose *incremental* updates. During an incremental update, clients might see different versions of the system until the update eventually completes. Each version is represented by a first-class *context*, which can be manipulated reflectively and enables the update scheme to be tailored to the nature of the application. For instance, the update of a web application can be rolled out on a per-thread, or per-session basis. In the latter case, visitors always see a consistent version of the application. Such a scheme would not be possible with a global update: one would need to wait until all existing sessions have expired before starting new ones. The overall consistency of the data is maintained by running bidirectional transformations to synchronize the representations of objects shared across contexts. We show that the number of such shared objects is significantly smaller than the number of objects local to a context, and that this strategy fits well with the nature of the event-based systems we are interested in.

We introduced first-class contexts in a previous workshop paper [27], but this original proof-of-concept suffered from several practical limitations. In contrast to our earlier work, we support now class versioning, garbage collection and lazy transformations, and we rely on program transformations rather than changes to the virtual machine. Bidirectional transformations have been used to cope with version mismatches in other settings (namely C systems [4], databases [5], and type theory [7]). However, neither of these approaches modeled context explicitly, nor did they tackle object-oriented systems in their full complexity, taking into consideration type safety, performance, concurrency and garbage collection. The main contribution of this paper is to demonstrate that first-class contexts offer a practical means to dynamically update software.

First, we present our *Theseus* approach informally with the help of a running example in section 2. We present our model in detail in section 3 and our implementation in section 4. We validate our approach in section 5 and demonstrate that it is practical. We put our approach into perspective in section 6 and we compare it with related work in section 7 before we conclude in section 8.

2 Running example

To illustrate our approach let us consider the implementation of one of several available Smalltalk web servers[1]. Its architecture is simple; a web server listens to a port, and dispatches requests to so-called services that accept requests and produce responses. For the sake of our running example, let us assume that the server keeps count of the total number of requests that have been served. Figure 1 illustrates the relevant classes.

2.1 The problem with updates

Let us consider the evolution of the Response API, which introduces chunked data transfer[2], also depicted in Figure 1. Assume that instead of sending "Hello

[1] See `http://www.squeaksource.com/WebClient.html` (The name is misleading since the project contains both an HTTP client *and* server)

[2] See version 75 of the project.

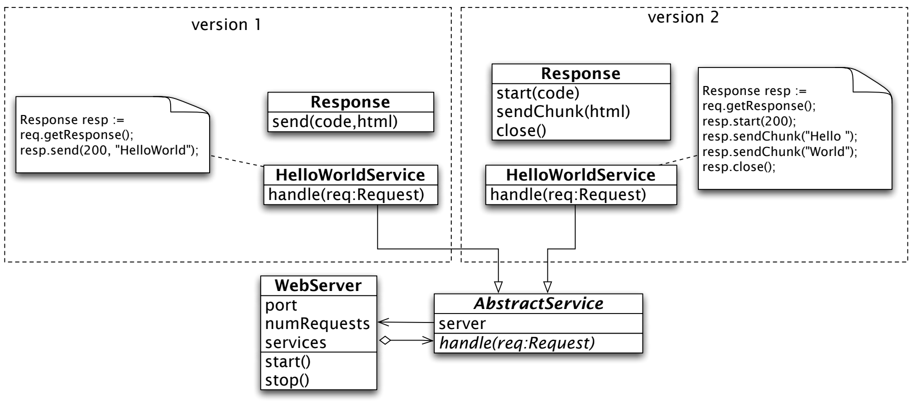

Fig. 1: Design of the web server and a simple behavioral update

World" over the wire we need to produce a sensible answer that takes some time. Installing such an update *globally* raises several challenges. First, both the `HelloWorldService` and `Response` classes must be installed together: *How can we install multiple related classes atomically?*

Second, the methods impacted by the update can be modified or added only when no request is being served: *When can we guarantee that the installation will not interfere with the processing of ongoing requests?*

Rather than performing a global update, it would be more appealing to do an *incremental* update, where ongoing requests continue to be processed according to the old code, and new requests are served using the new code. Note that the granularity of the increment might differ depending on the update. We could imagine that the modification of a check-out process spanning multiple pages would imply that the increment be the web session rather than the web request. Our solution to enable incremental updates is to reify the execution *context* into a first-class entity.

Not only the *behavior* but also the *structure* of classes can also change. Fields can be added or removed, and the type of a field can change. As a matter of fact, in a subsequent version of the project[3], the author added a field `siteUrl` to the `WebServer` class. Unfortunately, the server is an object shared between multiple requests, and each service holds a reference back to the server. If the object structure is updated globally while different versions of the code run to serve requests, old versions of methods might access fields at the wrong index. While the problem for field addition can be solved easily by ensuring new fields are added at the end, we need to consider type changes as well. For instance, one could imagine that in the future newest versions will store the `siteUrl` as an `HttpUrl` rather than a `String`. Therefore, the general problem remains: *How can we ensure consistent access to objects whose structure (position or type of fields) has changed?*

[3] See version 82 of the project.

Our solution to ensure consistent access is to keep one representation of the object per context and to synchronize the representations using bidirectional transformations. Once there is no reference any longer to a context, it is garbage-collected and the corresponding representations of objects as well.

2.2 Lifecycle of an incremental update

Let us consider the addition of the field `siteUrl` in the `WebServer` class in more detail. The following steps describe how an *incremental* update can be installed with Theseus[4], the implementation of our approach, while avoiding the problems presented above.

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 an event-based server system, such as a web server.

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 server 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. Fourth, the thread that listens to incoming connections in a loop is modified so that it is restarted periodically in the latest context. Note that the listening socket can be passed to the new thread without ever being closed.

After these preliminary modifications the system can be started, and now it supports dynamic updates. The life cycle of an update would be as follows:

1. *Bootstrap.* After the system bootstraps, the application runs in a default context named the *Root* context. The global variable `latestContext` is initialized to refer 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 `siteUrl` is added to `WebServer` and other related changes are installed.
3. *Update preparation.* The developer creates a class called `UpdatedContext`, which specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transformation that converts the program state between the *Root* context and the *Updated* context. Objects will be transformed one at a time. By default, the identity transformation is assumed, and only a custom transformation for the `WebServer` class is necessary in our case.
4. *Update push.* Using the administrative web interface, the developer uploads the class `UpdatedContext` as well as the other classes that will be required by the context. The application loads the code dynamically. It detects that one

[4] In reference to Theseus' paradox: if every part of a ship is replaced, is it still the same ship?

class is a context and instantiates it. Contexts are related to each other by a ancestor-successor relationship. The ancestor of the newly created context is the active context. The global variable `latestContext` is updated to refer to the newly created instance of the *Updated* context.

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request in the `latestContext` (which is now the *Updated* context) while existing threads terminate in the *Root* context.
6. *Incremental update.* When the web server is accessed in the *Updated* context for the first time, the new version of the class is dynamically loaded, and the instance is *migrated.* Migration is called when the object is accessed from a different context for the first time. In our case, this results in the fields `port` and `services` being copied, and the field `siteUrl` being initialized with a default value. Fields can be accessed safely from either the *Root* or *Updated* context, as each context has its own representation of the object. To ensure that the count of requests processed so far, `numRequests`, remains consistent in both contexts, bidirectional transformations between the representations are used. They are executed *lazily*: writing a new value in a field in one context only invalidates the representation of the object in the other context. The representation in the other context will be *synchronized* only when it is accessed again. Synchronization is called lazily when changes happen to objects that have already been migrated.
7. *Garbage collection.* Eventually the listener thread is restarted, and all requests in the old context terminate. A context only holds weakly onto its ancestor so when no code runs in the old context any longer, the context is finalized. The finalization forces the migration of all objects in the old context that have not been migrated yet. The old context and its object representations can then be garbage-collected. It must be noted that at the conceptual level, all objects in memory are migrated. In practice, only objects that are shared between contexts need to be migrated.

3 First-class context

Our approach relies on a simple, yet fundamental, language change: the state of an object is contextual. We assume, without loss of generality, throughout the rest of the paper that at most two contexts exist at a time, which we refer to as the “old” and “new” contexts. Clearly, the model could be generalized to support any number of co-existing contexts.

3.1 User-defined update strategy

Contexts are first-class entities in our system. Programmers have complete control over the dynamic update of objects and classes. Contexts are ordinary instances of the class `Context`, shown in Figure 2. A context is responsible for maintaining the consistency of the representations of the objects belonging to it. A context must implement methods `Context.migrate{To|From}` and

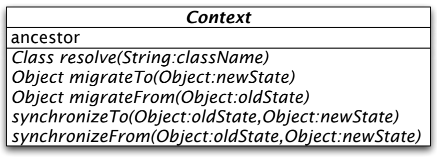
Context
ancestor
Class resolve(String:className)
Object migrateTo(Object:newState)
Object migrateFrom(Object:oldState)
synchronizeTo(Object:oldState,Object:newState)
synchronizeFrom(Object:oldState,Object:newState)

Fig. 2: The `Context` class.

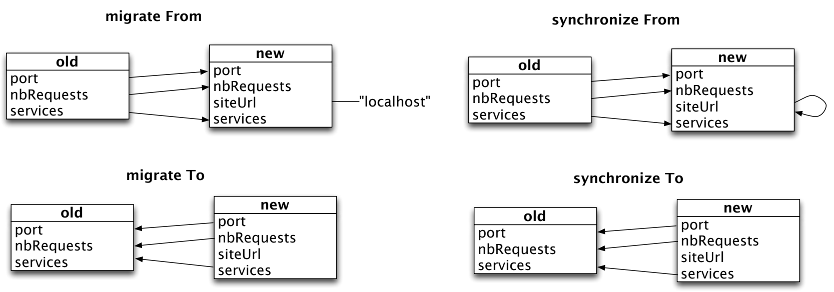

Fig. 3: The effects of the various methods that class `Context` mandates. Note that the arrow means a field copy operation and the method always applies to the new context.

`Context.synchronize{To|From}` to define the update strategy. We call "transformation" either the migration or the synchronization of the representations.

Each context has an ancestor. Since the contexts are loaded dynamically in an unanticipated fashion the update strategy is encoded in the newest context and expressed in terms of its ancestor, never in terms of its successor. Methods `Context.*From` assume the old representation is up-to-date, and transform the representation *from* the old context to the newest context; methods `Context.*To` assume the new representation is up-to-date, and update the representation from the new context *to* the old context. The *Root* context is the only context that does not encode any transformation and has no ancestor. User-defined contexts should default to the identity transformation for objects with no structural changes.

Figure 3 exemplifies the differences between the four methods using the running example. Methods `Context.migrate{To|From}` are responsible for creating the representation of an object upon the first access in the given context. In our case, the migration of the web server from the old context to the new context would copy the existing fields *as is* and initialize the new field `siteUrl` with a predefined value. Note that in this case, the object existed before the update and the migration from the new to the old context will never happen in practice[5]. Methods `Context.synchronize{To|From}` are responsible for subsequent updates of the state. In the case of our example, the field `siteUrl` must not be initialized again.

[5] This may not always be the case. It is possible for an object to be created in the new context and become reachable for objects in the old context

3.2 Reified state

From the application point of view, the state of an object will depend on the active context, and objects will have several representations. The transformations need to access both the old and new representations of an object. This requires the old and new representations to be *reified* into distinct objects, before they are passed to the transformations. Also, transformations are never called by the application, but by the run-time itself when necessary upon state read or write. Transformations run outside of any context.

Messages cannot be sent to contextual objects from within a transformation, as the system would not be able to decide what the "contextual" class of the object is in the absence of any context. This implies that certain objects must be primitive: they have a unique state in the system and are not subject to contextual variations. This is notably the case for the reified state, but also for contexts themselves. Immutable objects (string, numbers, *etc.*) are also considered to be primitive so that they can be used within transformations.

The reified state of an object can reference other contextual objects, however. If one has to query the state of such a dependent object from within a transformation, one would need first to obtain the reification of its state in either the old or new context.

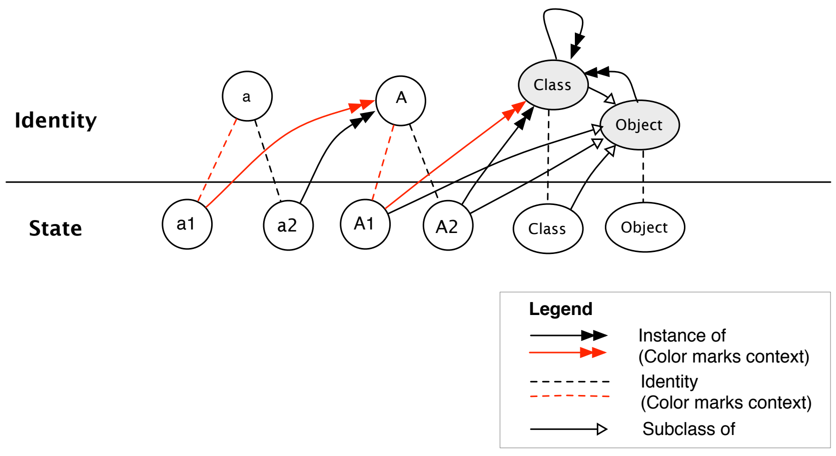

Fig. 4: Conceptual view of contextual objects and classes. `Object` and `Class` are primitive. Object `a` is contextual. Object `A` is a first-class class that is also contextual. The red path illustrates a reification: if the state of `a` is reified in the first context, one obtains `a1`, which is an instance of `A1`.

3.3 First-class classes

Classes are first-class in our model. They are contextual objects as well and a contextual class might have two versions, as depicted in Figure 4.

Conceptually, each contextual state is an instance of a contextual class, for example, the contextual state `a1` is an instance of the contextual class `A`. In practice, when the state is reified, the object that is obtained is not an instance of the contextual class, but of the reification of the contextual class: if the contextual state `a1` is reified, one obtains a primitive object that is an instance of `A1`.

When an object is migrated, a specific version of its state is reified and passed as a parameter to `migrate{To|From}`. The method must return the new version of its reified state, *e.g.*, migrate `a1`, which is an instance of `A1`, to `a2`, which is an instance of `A2`. The class can change only during migration. Indeed, methods `synchronize{From|to}` take as arguments the old reified state and the new reified state, but are not able to change the class they correspond to.

Classes are migrated similarly to regular objects. A specific version of the class is reified and passed as parameter to `migrate{To|From}`. The method must return the new version of the class, *e.g.*, migrate `A1` which is an instance of `Class` to `A2` which is also an instance of `Class`. Note that `Class` is a primitive in the system.

Classes are peculiar in that they can be resolved via a name, unlike "regular" objects. Contexts are responsible for class name resolution and must implement the method `Context.resolveClass(String)` which must return a specific version, *e.g.*, in Figure 4 "`A`" might resolve either to `A1` or `A2`. The way classes are migrated must correspond to the way classes are resolved for the system to be consistent.

3.4 Spawning thread

A thread can have one *active* context at a time. A predefined context exists, called the *Root*, which is the default context after startup. The runtime must be extended with a mechanism to query the active context, and also to specify a new context when a new thread is spawned. If none is specified, the thread will inherit the context of its parent thread.

4 Implementation

We report on the implementation of Theseus in Pharo Smalltalk. In contrast to our earlier work, this implementation does not require changes to the virtual machine. A unique aspect of our implementation is that it does not rely on proxies or wrappers, which do not properly support self-reference, do not support adding or changing public method signature, and break reflection [24,20].

During an incremental update, a contextual object corresponds concretely to two objects in memory, one per context. Figure 5 depicts such a setting. To maintain the illusion that the old and new representations of an object have the same identity, we adapt the references when necessary: for instance, if `b1` is assigned to a field of `a1` in the old context, this results in `b2` being assigned to the corresponding field of `a2` in the newest context.

Objects are migrated lazily, and can be either flagged as "clean" or "dirty". Dirty objects are out-of-date, and need to be synchronized upon the next access. Figure 5 shows the effect of an access to the dirty representation `b2`, which

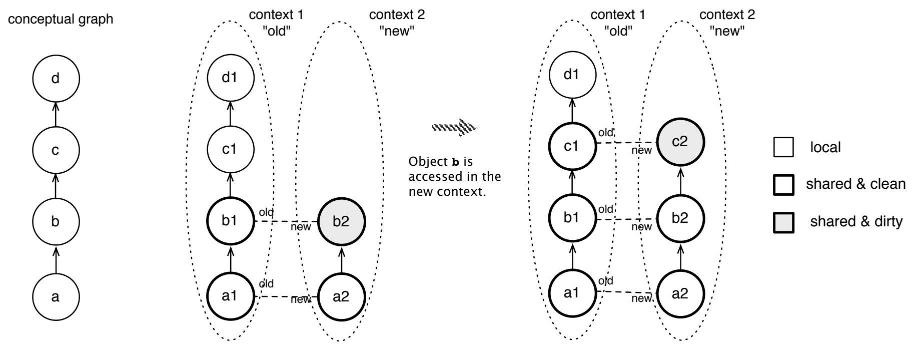

Fig. 5: The arrows between the four objects `a,b,c,d` represent references via a field. The objects exist in two contexts. Shared objects have one representation per context, which can be either "clean" or "dirty". Objects are migrated lazily. When object `b` is accessed in the new context for the first time, the representation `b2` is synchronized. Since `b` refers to `c`, this triggers the migration of `c` and the representation `c2` is created, originally considered "dirty". An access to `c` in the new context would create the representation `d2`, *etc.* Dashed lines represent relationships visible only to the implementation, not the application.

triggers the migration of the representation `c2` it references directly. After the synchronization, the two representations `b1` and `b2` of object `b` are clean. Subsequent writes to either representation would however result in the other one to be flagged as dirty. In the case of Figure 5, if `b2` is modified, `b1` would be marked as dirty.

We use bytecode rewriting to alter accesses to state and the way classes are resolved. Concretely, an extra check is added before each state read and state write to determine whether the object is shared between contexts. If it is, and the object is "dirty", it is synchronized and then marked as "clean". In case of state writes, the other representation is also invalidated and flagged as "dirty".

When the old context can be garbage-collected, we must ensure that all objects reachable from the new context have been migrated and are up-to-date. In the case of Figure 5, the system would force the migration of `d1` before garbage collection. If the graph of reachable objects is big, this operation can be relatively long, but can be conducted in background with low priority.

Concurrency. We assume that the subject program already correctly synchronizes concurrent reads and writes to thread-shared entities. Indeed, developers should neither make assumptions about the atomicity of read and write operations, nor about the visibility of side-effects between threads. Reads and writes are not atomic and concurrent accesses might trigger concurrent transformations.

Let us consider the web server of section 2. Field `numRequests` is synchronized with a lock, but `port` is not, as the value never changes after object creation. It is possible that `numRequests` is written and `port` is read concurrently. Two concurrent transformations might overlap and the new value of the field `numRequests` might be overwritten with the previous one. To ensure that the behavior of the original program is not altered, methods `synchronize{To|From}` take an additional parameter `field` in the full interface (not shown in subsection 3.1). This way, transformations can update fields selectively. There is also a per-field dirty flag.

When an object becomes shared, it is migrated. The migration `migrate{To|-From}` must however apply to the object as a whole (i.e. all its fields), as we cannot "partially" instantiate a representation. Also, a "forced" migration might happen due to the garbage collection of an old context, and despite a properly synchronized original program, concurrent migrations might occur. To resolve this situation, the migration of an object must be exclusive. Before a migration starts, it checks that the object was not migrated in the meantime. If this is the case it either (i) falls back to a `synchronize{To|From}` (normal migration) or (ii) is skipped (forced migration). If the object has already been migrated, the system must ensure that the new representation is up-to-date and might force synchronizations. Since these forced synchronizations might conflict with writes from the application, writes to shared objects must be exclusive to prevent lost writes. If the memory is not coherent (unlike with Pharo), reads to shared objects must also be exclusive to prevent stale reads, and the flag that indicates whether an object is shared must be defined so that it is not cached by the CPU (*e.g.*,`volatile` declaration in Java).

State relocation. Transformations can be more complex than one-to-one mappings. For instance, instead of keeping track of the number of requests in `num-Requests` using a primitive numeric type, the developer might introduce and use a class `Counter` for better encapsulation[6]. During the transformation, the actual count would be "relocated" from the web server object to the counter object that is now used. However, in this case, when the counter is incremented, the old representation of the web server with field `numRequests` needs to be invalidated. So far we have assumed that a write would invalidate only the representation of the object written to, which is not the case any longer. To support such transformations, the full interface enables custom invalidation on a per-field basis with `Context.invalidate{To|From}(Object oldState,Object newState,String field)`.

Further Details. We used a custom compiler to rewrite the bytecode of contextual classes. Primitive classes (see subsection 3.2) do not require any bytecode rewriting. In our scheme, contextual objects must have one representation per context, even if they are structurally equivalent. This applies to classes as well (see subsection 3.3). In Smalltalk, two instances of the same metaclass cannot be created, so we need to clone the metaclass as well. Closures are first-class in Smalltak. They encode offsets of bytecode in the `CompiledMethod` they reference. They are treated analogously to other objects. After migration, they

[6] This would be the refactoring "Replace Data Value with Object". See `http://www.refactoring.com/`

Request	B(ms)	T(ms)	# Read		# Write		# Reachable		Migrated
			Shared	Local	Shared	Local	Shared	Local	
1st request	30	60	-	128923	-	14674	-	-	-
2nd request	30	127	14535	130172	21	17901	1292	2781	585
3rd request	30	77	14547	120991	34	15539	1293	3311	588

Fig. 6: Time for three successive requests, one before the update, and two after the update. T=Theseus, B=Baseline. Migrated=cumulative number of migrated objects

reference the newest version of the corresponding `CompiledMethod`. The active context is stored in a thread-local variable and we add a new method to fork a closure in a specific context, *e.g.*,`[ ... ] forkWith: aContext`. When a closure is forked, it becomes a shared contextual object and is migrated. As the program proceeds, objects referenced by the closure are migrated lazily when accessed. Contexts hold only weak references to their ancestor and implement the method `Object>>#finalize`, which forces the migration of all reachable objects before the context becomes eligible for garbage collection. The class `Semaphore` is treated as primitive so that objects can be synchronized correctly. The `Object` class cannot be modified easily. To keep track of the necessary information we need about objects, we maintain a dictionary that maps objects to their extra information. Clearly, this level of indirection would need to be optimized in a full implementation.

5 Validation

Evolution. We conducted a first experiment whose goal was to assess whether our model could support long-term evolution, that is, whether it could sustain successive updates. We considered the small web server of section 2, which despite its simplicity cannot be updated easily with global updates. We selected the 4 last versions with effective changes: version 75 introduced chunked data transfer, version 78 fixed a bug in the encoding of URL, version 82 introduced `siteUrl`, and version 84 fixed a bug in MIME multipart support.

The listening thread that accepts incoming connections was modified to restart itself periodically. Only one update required us to write a custom transformation: the one that introduced the `siteUrl` field, which we initialized to a default value. We ran the 4 successive dynamic updates, and verified that once it was no longer used, the old context would be garbage-collected. In this way we validated that our implementation was coherent.

Run-time characteristics. For the second experiment, we picked a typical technology stack with well-known production projects: the Swazoo web server, the Seaside web framework, the Magritte meta-description framework, and the Pier CMS. This corresponds to several thousand classes. We were interested in the run-time characteristics and to assess (1) whether our assumptions about object sharing hold, and (2) what is the performance overhead. As a case study, we considered the default web site of the Pier demo. During maintenance, only few classes change. Most objects are migrated with the identity transformation, and only certain objects require custom transformations. The exact nature of the transformation is not significant. Therefore, for the sake of simplicity, we artificially updated the system and used the identity transformation for all objects.

We were interested to assess the overhead of our implementation in three different cases: (i) with only the old context when no object is shared, (ii) during the incremental update when objects are shared and migrated lazily, and (iii) after objects have been migrated but are still considered shared. To do so, we measured the time for three successive requests: one before the update, for case (i), and two after the update, for cases (ii) and (iii).

The results are presented in Figure 6. The overhead of our implementation is in the best case of factor two. In the worst case when many objects must be migrated, we have a degradation of factor four. We tracked the number of reads and writes to objects shared between contexts, and to objects local to a context. We clearly see that writes are one order of magnitude less frequent than reads. About 500 objects needed to be migrated and only a minority of accesses concern shared objects. The migrated objects and their direct references correspond to about 1300 reachable objects. These 1300 objects reference further about 3000 objects indirectly. These 3000 objects could be reached indirectly from both contexts, but are in practice local to a context. There are fewer than 50 writes to shared objects and we deduce that the code of the extra logic to invalidate representations is negligible (see section 4). In the first request, the system checks if objects are shared, which is never the case. In the third request, the system needs an additional check for dirtiness, which returns always false. This explains the difference between times (i) and (iii).

Our experiment did not simulate the run-time characteristics of a production system, however. We did not account for concurrent requests, which could cause objects to be synchronized back and forth. Further empirical validation is welcome. Also, our implementation is still relatively naive (see section 4). However, even with this implementation we achieve reasonable response time.

These results show that the approach can be made practical and fits well to the characteristics of real-world software.

6 Discussion

Performance. A drawback of our implementation is that shared objects need two representations, even if they are structurally identical and will use the identity transformation. Wrappers would make it possible to keep only one representation in such cases, but pose problems of self-reference, do not support adding or changing method signatures, and break reflection [24,20,22]. The benefit of our implementation is that object representations are really instances of their respective classes and avoid such problems. We plan to improve performance by not synchronizing state on each access, and instead synchronize groups of fields at precise locations, *e.g.*, synchronize all fields a method uses at once at the beginning and end of the method. Lock acquisitions/releases would force the synchronization of pending changes, similarly to *memory barriers* [9]. It could, at least, be done manually for heavily-used system classes. This would preserve concurrent behavior but increase significantly the performance.

Applicability. The impact on development is small. Developers must figure out the "increment" they wish, which results usually in a few well-located changes after which development proceeds as usual. Compared to other dynamic update mechanisms, there must exist a state mapping only for shared entities (not all entities), but the mapping must be bidirectional (not unidirectional). We can navigate the object graph during the transformation which seems to suffice for most evolution in practice [26,17,2]. Daemon threads must be adapted to restart periodically, but it is easy to do given their cyclic nature. Recent works showed that most of the transformation code can be generated automatically [21] and it would be interesting to assess whether we can generalize such results for bidirectional transformations as well.

7 Related work

A common technique to achieve hot updates is to use redundant hardware [11], possibly using "session affinity" to ensure that the traffic of a given client is always routed to the same server. Our approach is more lightweight and enables the migration of the state shared across contexts, notably persistent objects. Also, an advantage of being reflective is that the software can "patch itself" as soon as patches become available.

A large body of research has tackled the dynamic update of applications. Systems supporting *immediate and global dynamic updates* 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 of this kind have been devised for Java [6,20,15,10,3,28,23], with various levels of flexibility (a good comparison can be found in [10]). To be type-safe, HotSwap [6] imposes restrictions and only method bodies can be updated. The most recent approaches [28,23] are more flexible but can still lead to run-time errors if changes impact active methods. Most of these approaches rely on bytecode transformation [20,15,10,3,23] and do not address concurrency.

Several approaches have tackled the problem of safety by relying on temporal *update points* when it is safe to globally update the application. Such systems have been devised for C [11,19], and Java [26,17]. Update points might be hard to reach, especially in multi-threaded applications [18,26], and this compromises the timely installation of updates.

Some mechanisms diverge from a global update and enable different versions of the code or entities to coexist. In the most simple scheme, old entities are simply not migrated at all and only new entities use the updated type definition [13], or this burden might be left to the developer who must request the migration explicitly [8]. The granularity of the update for such approaches is the object; it is hard to guarantee *version consistency* and to ensure that mutually compatible versions of objects will always be used. When leveraged, transactions [2,22] provide version consistency but impede mutations of shared entities. Contexts enable mutations of shared entities and can be long-lived, thanks to the use of bidirectional transformations. With asynchronous communication between objects, the update of an object can wait until dependent objects have been upgraded in order to remain type-safe [14].

To the best of our knowledge, only three approaches rely on bidirectional transformations to ease dynamic updates. POLUS is a dynamic updating system for C [4] which maintains coherence between versions by running synchronizations on writes. We synchronize lazily on read, operate at the level of objects, and take garbage collection into account. Duggan [7] formalized a type system that adapts objects back and forth: when the run-time version tag of an object doesn't match the version expected statically, the system converts the object with an adapter. We do not rely on static typing but on dynamic scoping with first-class contexts, we address garbage collection, concurrency, and provide a working implementation. Oracle enables a table to have two versions that are kept consistent thanks to bidirectional "cross-edition triggers" [5].

Schema evolution addresses the update of persistent object stores, which closely relates to dynamic updates. To cope with the volume of data, migrations should happen lazily. To be type-safe, objects should be migrated in a valid order (*e.g.*, points of a rectangle must be migrated before the rectangle itself) [2,22]. Our approach migrates objects lazily, and avoids the problem of ordering by keeping both versions as long as necessary.

Class loaders [16] allow classes to be loaded dynamically in Java. Types seen within a class loader never change, which ensures type safety and version consistency, similarly to our notion of context. Two versions of a class loaded by two different class loaders are different types, which makes sharing objects between class loaders complicated. This is unlike our approach which supports the migration of classes and objects between contexts.

Context-oriented programming [12] enables fine-grained variations based on dynamic attributes, *e.g.*, dynamically activated "layers". It focuses on behavioral changes with multi-dimensional dispatch, and does not address changing the structure and state of objects as is necessary for dynamic updates. There exist many mechanisms to scope changes statically, *e.g.*, Classboxes [1], but they are not used to adapt software at run-time.

8 Conclusion

Existing approaches to dynamically update software systems entail trade-offs in terms of safety, practicality, and timeliness. We propose a novel, incremental approach to dynamic software updates. During an incremental update, clients might see different versions of the system, which avoids the need for the system to reach a quiescent, global update point.

Each version of the system is reified into a first-class context. Existing objects are gradually migrated to the new context, and objects that are shared between old and new contexts are kept consistent with the help of bidirectional transformations. Our validation with real-world systems indicates that only a fraction of accesses concern such objects.

In two experiments we have demonstrated that our current implementation is practical and flexible, with reasonable overhead. This work opens up several research directions: exploring different granularity of increments, providing developer tools to leverage contexts, and improving further the performance.

Acknowledgments 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. Bergel, A.: Classboxes — Controlling Visibility of Class Extensions. Ph.D. thesis, University of Bern (Nov 2005)
2. Boyapati, C., Liskov, B., Shrira, L., Moh, C.H., Richman, S.: Lazy modular upgrades in persistent object stores. SIGPLAN Not. 38(11), 403–417 (2003)
3. Cech Previtali, S., Gross, T.R.: Aspect-based dynamic software updating: a model and its empirical evaluation. In: Proceedings of the tenth international conference on Aspect-oriented software development. pp. 105–116. AOSD '11, ACM, New York, NY, USA (2011)
4. Chen, H., Yu, J., Hang, C., Zang, B., Yew, P.C.: Dynamic software updating using a relaxed consistency model. IEEE Trans. Software Eng. 37(5), 679–694 (2011)
5. Choi, A.: Online application upgrade using edition-based redefinition. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. pp. 4:1–4:5. HotSWUp '09, ACM, New York, NY, USA (2009)
6. Dmitriev, M.: 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)
7. Duggan, D.: Type-based hot swapping of running modules. In: Intl. Conf. on Functional Programming. pp. 62–73 (2001)
8. Gemstone/s programming guide (2007)
9. Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. Tech. rep., DEC (1995)
10. Gregersen, A.R., Jørgensen, B.N.: Dynamic update of Java applications — balancing change flexibility vs programming transparency. J. Softw. Maint. Evol. 21, 81–112 (mar 2009)
11. Hicks, M., Nettles, S.: Dynamic software updating. ACM Transactions on Programming Languages and Systems 27(6), 1049–1096 (nov 2005)
12. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3) (Mar 2008)

13. Hjálmtýsson, G., Gray, R.: Dynamic C++ classes: a lightweight mechanism to update code in a running program. In: Proceedings of the annual conference on USENIX Annual Technical Conference. pp. 6–6. ATEC '98, USENIX Association, Berkeley, CA, USA (1998)
14. Johnsen, E.B., Kyas, M., Yu, I.C.: Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In: Proceedings of the 2nd World Congress on Formal Methods. pp. 596–611. FM '09, Springer-Verlag, Berlin, Heidelberg (2009)
15. Kabanov, J.: Jrebel tool demo. Electron. Notes Theor. Comput. Sci. 264, 51–57 (feb 2011)
16. Liang, S., Bracha, G.: Dynamic class loading in the Java virtual machine. In: Proceedings of OOPSLA '98, ACM SIGPLAN Notices. pp. 36–44 (1998)
17. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic Java classes. In: Proceedings of the 14th European Conference on Object-Oriented Programming. pp. 337–361. Springer-Verlag (2000)
18. Neamtiu, I., Hicks, M.: Safe and timely updates to multi-threaded programs. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 13–24. PLDI '09, ACM, New York, NY, USA (2009)
19. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical dynamic software updating for C. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. pp. 72–83. PLDI '06, ACM, New York, NY, USA (2006)
20. Orso, A., Rao, A., Harrold, M.J.: A Technique for Dynamic Updating of Java Software. Software Maintenance, IEEE International Conference on 0, 0649+ (2002)
21. Piccioni, M., Oriol, M., Meyer, B., Schneider, T.: An ide-based, integrated solution to schema evolution of object-oriented software. In: ASE. pp. 650–654 (2009)
22. Pina, L., Cachopo, J.: Dustm - dynamic software upgrades using software transactional memory. Tech. rep., INESC-ID (2011)
23. Pukall, M., Kästner, C., Cazzola, W., Götz, S., Grebhahn, A., Schröter, R., Saake, G.: Flexible dynamic software updates of java applications: Tool support and case study. Tech. Rep. 04, School of Computer Science, University of Magdeburg (2011)
24. Pukall, M., Kästner, C., Saake, G.: Towards unanticipated runtime adaptation of java applications. In: APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference. pp. 85–92. IEEE Computer Society, Washington, DC, USA (2008)
25. Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION '96. pp. 21–38 (Apr 1996)
26. Subramanian, S., Hicks, M., McKinley, K.S.: Dynamic software updates: a VM-centric approach. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 1–12. PLDI '09, ACM, New York, NY, USA (2009)
27. Wernli, E., Gurtner, D., Nierstrasz, O.: Using first-class contexts to realize dynamic software updates. In: Proceedings of International Workshop on Smalltalk Technologies (IWST 2011). pp. 21–31 (2011), http://esug.org/data/ESUG2011/IWST/Proceedings.pdf
28. Würthinger, T., Wimmer, C., Stadler, L.: Unrestricted and safe dynamic code evolution for Java. Science of Computer Programming (Jul 2011)