Encapsulating and Exploiting Change with Changeboxes

Marcus Denker, Tudor Girba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, Pascal Zumkehr

Encapsulating and Exploiting Change with Changeboxes*

Marcus Denker, Tudor Gîrba, Adrian Lienhard,
Oscar Nierstrasz, Lukas Renggli, Pascal Zumkehr

Software Composition Group, University of Bern
www.iam.unibe.ch/∼scg

Abstract. Real world software systems change continuously to meet new demands. Most programming languages and development environments, however, are more concerned with limiting the effects of change rather than enabling and exploiting change. Various techniques and technologies to exploit change have been developed over the years, but there exists no common support for these approaches. We propose Changeboxes as a general-purpose mechanism for encapsulating change as a first-class entity in a running software system. Changeboxes support multiple, concurrent and possibly inconsistent views of software artifacts within the same running system. Since Changeboxes are first-class, they can be manipulated to control the scope of change in a running system. Furthermore, Changeboxes capture the semantics of change. Changeboxes can be used, for example, to encapsulate refactorings, or to replay or analyze the history of changes. In this paper we introduce Changeboxes by means of a prototype implementation. We illustrate the benefits that Changeboxes offer for evolving software systems, and we present the results of a preliminary performance evaluation that assesses the costs associated with Changeboxes while suggesting possible strategies for improvement.

1 Introduction

It is well-established that so-called *E-type* systems, *i.e.*, real-world applications that are *embedded* in the environment where they are used, *must* change continuously or else become less useful over time [31].

Oddly enough, most programming languages and development environments have traditionally invested more effort into providing mechanisms that *limit* change than into those that enable or exploit change [39,40]. Some typical symptoms of this phenomenon include:

– A name for a software artifact, such as a class, a type or a module, is generally assumed to have a globally consistent meaning within a single running system. Different versions of the same artifact cannot be simultaneously active within the same system, *e.g.*, a single virtual machine (VM).

* Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 25-49. DOI 10.1145/1352678.1352681.

- Whenever an artifact evolves, this must be done in such a way that existing clients are not adversely affected. Although interfaces may be *deprecated*, it can be hard or impossible to definitely remove them.
- Frameworks provide not only black-box but also white-box components. Refactoring may preserve the public interface of a subsystem while breaking implicit contracts visible only to subclasses in client code [45]. As a consequence, developers may be reluctant to modify framework components that may break client application code.

The fact that software evolution needs to be effectively managed can be observed by the range of tools and techniques developed over the years. Consider some of the following examples. Versioning systems are used to keep track of changes made to a software system, and configuration management systems manage the different versions that need to be deployed [38]. In Model-Driven Engineering, model transformations are used to ensure that different views of a software system remain synchronized when changes to requirements occur [26]. Refactoring operations can be mined from versioning repositories or stored when they happen, and then replayed to ease migration at multiple client sites [15,16]. Adaptations of third-party software can be conveniently expressed as locally visible extensions [8]. OpenModules [3] allow clients to adapt a module by providing advice code for methods and pointcuts declared in the interface. Each of these techniques, however, manages and exploits change from its own perspective and at different levels of abstractions, and little effort is made to benefit from a common infrastructure. In each case we have thoroughly different mechanisms to express and manage change. Furthermore, these approaches generally focus on managing change to static software artifacts, rather than on enabling change to a running system.

Although change is fundamental to software, we are lacking the means to encapsulate and express change in such a way that it can be effectively controlled and exploited. Essentially we need to address the following three points:

- *Encapsulating change.* In order to fully exploit the history of changes, we need a mechanism to *capture change as it occurs*. In this way the full history of a system is accessible as it evolves. Furthermore, the semantics rather than simply the syntactic effects of change need to be captured so that we can reason about change and manipulate it in a meaningful way. Depending on the application, a change may be encapsulated as, for example, an edit, a refactoring, or a more general kind of transformation.
- *Scoping change.* Different versions of the same software may be active simultaneously, even within the same virtual machine (VM). Therefore we need means to control the scope of change within a running system.
- *Deploying change.* Running systems evolve, therefore we need means to merge and deploy changes on the fly and without restarting the system.

We propose Changeboxes as a mechanism to address these issues. A Changebox is a first-class entity that encapsulates change while providing an execution scope for controlling the visibility of changes to the running system. A

Changebox provides a unifying mechanism for various kinds of change. Multiple Changeboxes can be simultaneously active *within the same running system.* Since Changeboxes capture the entire history of change, it is possible to exploit these historical views at run-time. Furthermore Changeboxes can be composed by merging. A fine degree of control over the semantics of merging is possible — even necessary — to allow Changeboxes to be fully exploited.

Structure of this paper: In the following section we present a running application of Changeboxes that illustrates several key scenarios for encapsulating and exploiting change. In Section 3 we provide a capsule summary of the Changebox model. Section 4 presents a more detailed description of the prototype implementation of the Changebox model. In particular, we explain how change is encapsulated as Changeboxes, how Changeboxes provide a scope for execution, and how different strategies for merging are supported. We evaluate the prototype in Section 5 with the help of a number of benchmarks. In Section 6 we discuss new avenues that are opened by this work. We discuss related work in Section 7, and we conclude in Section 8 with some remarks about ongoing and future work.

2 Motivating example

In this section we present different scenarios that hamper evolution when using traditional static versioning systems such as CVS or SVN. As an example we have chosen Pier [43], an industrial strength Content Management System (CMS) built on top of the Seaside web application framework [18]. Web applications offer a good example of a domain in which changes occur frequently, yet the applications need to be up and running virtually all the time.

In Figure 1 we see the evolution of a Pier application along three development branches. The arrows point backwards in time to their respective ancestor versions. Each snapshot of the system is defined by a commit to the versioning system of all the sources of this particular version.

2.1 Traditional Versioning System

We see that the code has been split from the release branch and deployed to create a customized version (1) for a customer. A third development branch leads to a defective version (2) where the current development for next version is going on.

To fix a bug (3) in the released branch the developer needs to either update his working copy to this particular version or check out a new working copy at a different location. Either way, he is required to recompile the system and restart the development server, which can take a significant amount of time for large applications. Moreover, if developers want to have different versions of the same application running at the same time, they need a complicated setup, presumably with different back-ends and multiple server entry points.

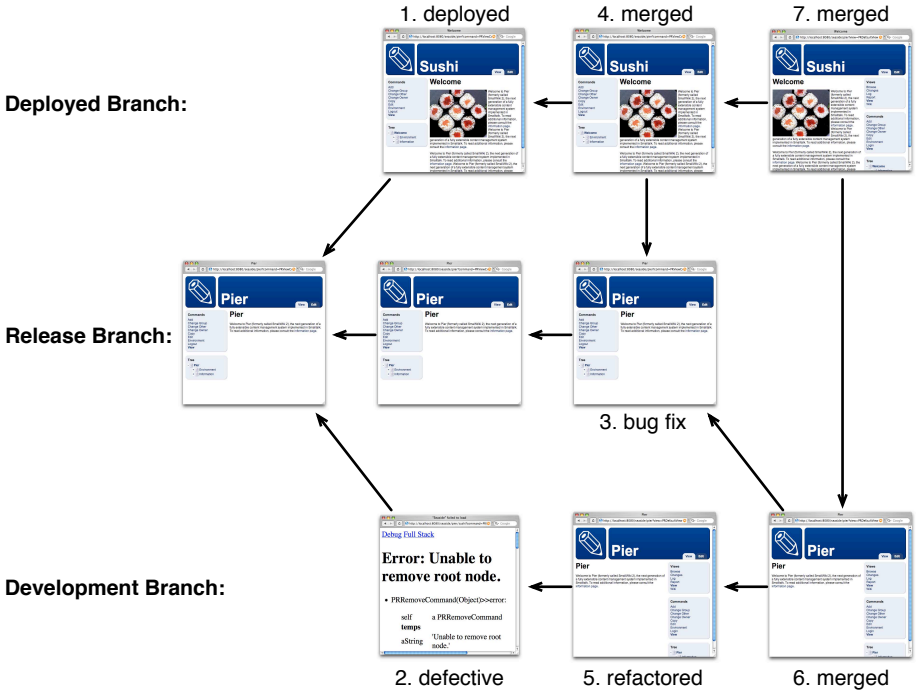

Fig. 1. The evolution of a web application: every screenshot represents the running system in the context of its history. The arrows points back to the ancestor versions.

Merging and deploying the bug fix (4) on the server is difficult. Since a downtime of the application is often not acceptable, the developer has to install the new version separately. Then the server configuration has to be changed so that new user session will open in the new server instance. Old sessions continue to live in the old version or, if possible, are redirected and migrated to the new one.

In the meantime some heavy refactorings (5) — like class-renames affecting many different places in the code — were applied to the development branch. Merging of the bug fix into the other two branches is not trivial anymore. Unfortunately the versioning system does not know what refactorings were performed and is therefore unable to automatically transform the code to be merged (6, 7). The developer is required to manually perform the refactorings.

2.2 Advantages when using Changeboxes

We have also developed the previously described scenario within a Changeboxes-aware system. While doing so, the drawbacks described above are overcome.

As Changeboxes keep track of different versions of the source code available within the development environment there is no need for manual checkout and recompilation. Moreover, Changeboxes also define the mechanisms to execute any version of this code at any time, so it is just a matter of swapping the scope by selecting the version to be used. Every process of the system runs within its own scope defining a specific view on source code, classes and executable methods.

When deploying an update of the application on a specific server (4, 7), there is no need to set up a backup server. New code can be directly loaded and compiled in the running system without affecting any currently existing user session. Within the scope of the new code, developers can even run unit- and functional-tests on the server itself while the productive part is still using the old code. When the developer is confident about deploying the revised code, it can be activated for new sessions. Existing sessions will run to completion with the old code, since every session references the specific version it has been started with. Unless a developer decides to change this reference manually, existing sessions will stick with the same code even though changes have been applied to the system in the meantime.

Although it may seem highly unusual to develop, deploy and test within the same environment, numerous advantages can be gained if this is done in a disciplined way. Having the possibility to work on the same machine for development, testing and deployment eliminates the need to set up multiple machines with similar but not necessarily identical environments. Problems the customer might experience can be investigated and possibly fixed directly on the server and do not have to be reproduced in a different environment, thus speeding up corrective maintenance tasks.

Refactorings such as class renamings affect many places in the code. The Changeboxes system tracks what refactorings were performed and transforms the code to be merged as well, so that the fix can be added to the 1.1 branch without any difficulty (6). Thus, backwards compatibility is rendered irrelevant.

Evolution of the Changebox-aware Pier application benefits from the following points:

- Instead of checking out code and manually recompiling the system, all versions are always available in an executable form. Every process can define its own view on the system and therefore multiple versions can be running on the same VM at the same time.
- Instead of having to reproduce bugs within a different development environment, they can be quickly fixed and tested on the running server. Users of the application benefit by immediately profiting from fixes without even having to login again.
- Instead of having the versions available in a textual form like in traditional versioning systems, the versions are available in an executable form. By running tests in different versions developers can determine exactly what change caused a test to fail.

– Refactorings performed in one branch and encapsulated as Changeboxes can be replayed in another development branch by merging the corresponding Changeboxes.

3 Changeboxes in a nutshell

In Figure 2 we show the core of the Changebox metamodel (at the bottom) and several versions of a Changebox-aware application conforming to this metamodel (on top).

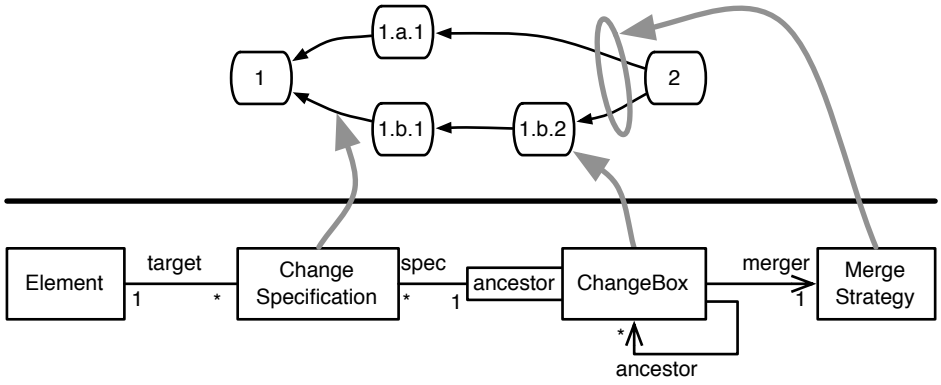

Fig. 2. The Changebox metamodel and an example of Changebox dependencies.

A software system is composed of various *elements*, such as classes, methods and packages. Many different versions of a system may exist and may be run at the same time. Each of these versions consists of incremental changes to a previous *ancestor* version.

A Changebox is an immutable entity that defines a snapshot of a system by:

– encapsulating a set of change specifications,
– specifying a set of ancestor Changeboxes to which these changes apply, and
– providing *a scope for dynamic execution.*

A *change specification* describes how one version of an element may be transformed into another version of that element. The system can only be changed by creating a new Changebox that encapsulates a change specification.

In Figure 2, Changebox 1 is empty. Changebox 1.a.1 encapsulates a single change specification, thus defining a new version of the system and a new scope for dynamic execution. Changebox 2 has two change specifications which are applied to elements found in ancestors 1.a.1 and 1.b.2. Changebox 2 performs a

merge. As merges can introduce possible conflicts, a strategy is used to implement the different ways of solving the conflicts.

In Figure 2 we also introduce the core of a Changebox-system (a more detailed diagram can be found in Figure 3). An element represents a structural entity in the programming language (*e.g.*, package, class, method). A Changebox has several ancestors and encapsulates a set of ChangeSpecifications. The possible conflicts that appear due to merging are resolved with a MergeStrategy.

At first glance, a Changebox-system may appear to be yet another versioning system. However the key difference is that a Changebox-system maintains versions of a running system, not just static software artifacts. Furthermore, any and all versions may be executable at the same time in the same VM, by providing for each Changebox a scope for dynamic execution.

Each process is executed within the scope of a Changebox which defines a consistent, flat view of the system. This view is determined by the changes of the selected Changebox and all its ancestors. Rather than compiling the whole system for a specific version to be executed, the actual version of a method or a class is looked up at runtime. This allows for compiled methods to be shared between different versions of a system. Furthermore, since methods are compiled incrementally, execution of a new version starts instantaneously.

4 Implementing Changeboxes in Smalltalk

The Changebox prototype is implemented in Squeak[1], an open source Smalltalk dialect.

We decided to use the reflective features of Smalltalk to implement Changeboxes rather than attempting to modify the underlying VM. Although an implementation at the level of the VM would certainly have performance advantages, we felt that the development effort would be too great for a first prototype. This strategy enabled us to quickly obtain a proof-of-concept prototype of Changeboxes with adequate performance to carry out real experiments. At the same time we were able to gain insights into the implications of various implementation approaches.

To implement Changeboxes in Smalltalk, the following problems must be solved:

- The Changebox metamodel needs to be implemented and the Smalltalk reflective kernel needs to be extended to capture structural changes to software entities as they occur (Section 4.1).
- The Smalltalk runtime environment needs to be extended with execution scopes so that different versions of the same program may execute concurrently (Section 4.2).
- A flexible approach is needed to merging Changeboxes that can easily be adapted to the needs of different applications (Section 4.3).

We summarize our approach in the rest of this section. Full details are to be found in Zumkehr's Masters thesis [47].

[1] http://www.squeak.org

4.1 Modeling and capturing changes

Figure 3 illustrates the key classes of the Changebox implementation. In addition to the three key classes `ChangeBox`, `ChangeSpecification` and `Element` we see:

- the concrete element and change specification subclasses,
- the system classes `Class` and `CompiledMethod` representing specific versions of classes and methods used by the VM,
- the class `MergeStrategy` to define the merge semantics,
- the class `WorkSession` which tracks a sequence of changes of a user, and
- the class `Scope`, the common superclass of `ChangeBox` and `WorkSession`.

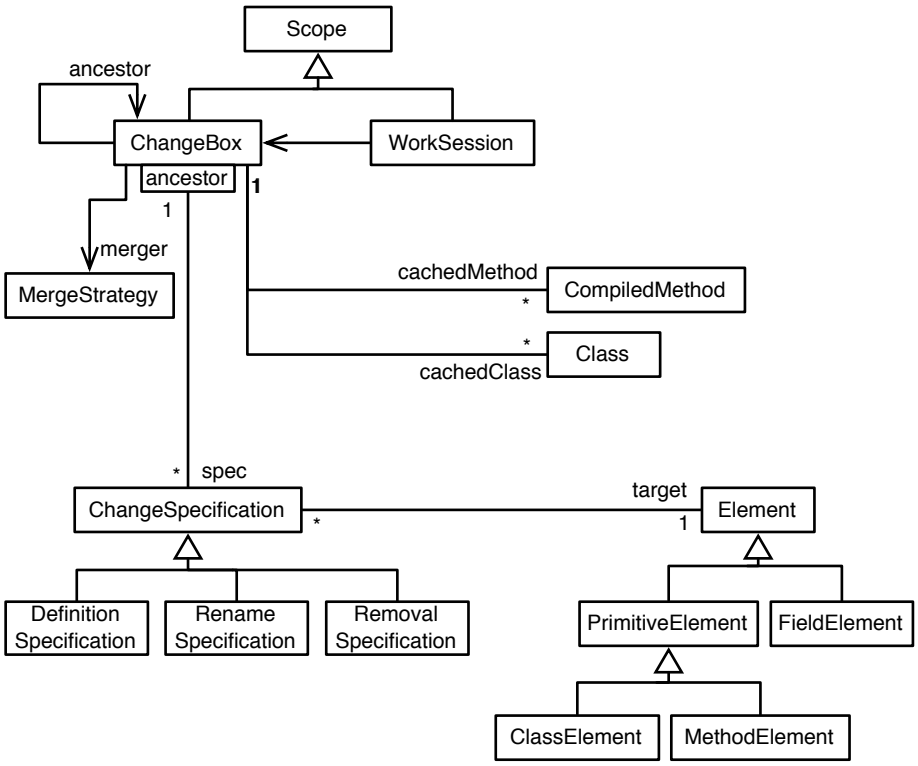

Fig. 3. Changebox implementation class diagram.

Elements represent the historical perspective of static entities of the programming language, such as classes, methods, and fields. Only within a specific scope can a version of an actual software entity be produced from an element (*e.g.*, `CompiledMethod`).

Since Smalltalk is a reflective system implemented in itself, instances of the system classes `Class` and `CompiledMethod` are the meta-objects used by the VM as it executes. They provide the necessary information for the VM to instantiate new objects (the class format) and execute methods (bytecode). Fields, however, are not reified in the Smalltalk runtime, so only classes and methods are considered to be primitive elements. (`ClassElement` and `MethodElement` inherit from `PrimitiveElement`, but `FieldElement` does not.) As a consequence, renaming a field will entail a change to the class that defines it.

The actual changes between two points in the evolution of a system are defined by change specifications which are encapsulated in Changeboxes. A change specification implements the process to change the form of an element (*e.g.*, (re-)definition, rename, or removal) within a certain system snapshot. For example, a definition specification holds the new properties of the changed element (as the source code of a method, or the superclass of a class), and a rename specification specifies the new name.

The changed element is stored in the Changebox which encapsulates the change (*i.e.*, in the form of a `Class` or a `CompiledMethod`). In this way the right version of an element can be efficiently retrieved when it is looked up in a specific execution scope. Lookup is discussed in detail in Section 4.2 and merging is discussed in Section 4.3.

In the remaining part of this section we discuss how changes are captured, how work sessions keep track of a sequence of changes and how specifications are generated from changes.

Capturing changes. In a conventional Smalltalk system, there exists exactly one version of each method and class at a time. Changes, *i.e.*, the recompilation of methods and classes, are performed through the reflective kernel; the previous version of a compiled method or class is replaced with the new one and is eventually garbage collected.

In the Changebox model, the system can always be viewed exactly as it existed at an arbitrary moment in the past. Therefore, a Changebox system needs to keep track of all changes. Our implementation modifies the appropriate structural reflection entry points of the kernel to capture changes and to install the appropriate mechanisms to make execution aware of scopes (see Section 4.2).

A *work session* manages a sequence of changes being produced by a developer with a Changebox-aware development tool. While a developer is modifying the system within a particular work session, the work session keeps track of the most recent Changebox in effect. A work session, like a Changebox, can be used to define an execution scope. The difference is that the Changebox defines a fixed execution scope, while a work session takes the most recently produced Changebox to define its scope.

A work session is often used as a common execution scope of a set of IDE tools. A change produced in one of the tools is then automatically reflected in the other tools running within the same scope. Until now, the main tools that have been made aware of Changebox are the code browser and its various helpers

(to browse senders, implementors, variables and so on), the debugger and the test runner. Additionally, the version control system was adapted to be able to load code from and into specific work sessions. A work session browser manages the various work sessions present in the system.

As each new change is captured, a change specification is generated. Each change specification encapsulates the process rather than the result of the change. Change specifications are generated by comparing the resulting `Class` or `CompiledMethod` instances of a change to their previous version.

The advantage of encapsulating the change process is that the semantics of a change can be preserved at a more finely grained level. For example, renaming a field in Smalltalk results in a change of its class and yields a new `Class` instance. A change specification to rename a field element, however, captures the intent of renaming the field (rather than redefining a whole class) by only encapsulating the field element and its new name.

This *bottom-up* approach captures all structural changes at a fine grain of detail independent of their high level intent.

Capturing higher-level changes. The approach to capturing changes described above captures each *single* change separately. In the case of changes performed by a refactoring tool, however, a single high-level change may entail a large number of individual finer-grained changes. A simple example is renaming a method. This change results in the removal of the method and its re-compilation under a new name.

As Changeboxes are to capture the semantics of changes as precisely as possible, we complement the bottom-up approach with a *top-down* mechanism to capture changes. The top-down mechanism hooks into the IDE tools and captures a change at the higher level where the change semantics are expressed. Tools can directly instantiate change specifications and pass them to the current work session.

For example in the case of a method rename, a `RenameSpecification` operating on a `MethodElement` can be created, specifying the new method name. As a result, only one Changebox is created to encapsulate this specification. Apart from the advantage that there is only one Changebox generated, this strategy also allows for capturing refactorings which may have different effects depending on the system snapshot they are applied in. This is possible because the purpose rather than the effect is captured at a more abstract level. A method rename specification, for example, can then be applied to different snapshots to appropriately update the senders of the renamed method (*e.g.*, library refactorings applied to different client applications).

4.2 Scoping execution

Processes are always executed within the scope of a Changebox. The Changebox in effect is determined by dynamic variable scoping, that is, it is accessed by searching down from the top of the execution stack for the most recent definition.

The active Changebox can be set at run-time for a block closure which then executes its code in this new scope.

During execution, each message send and each access to a class reference results in a lookup of the most recent version of the element (as described below). While the versions of methods and of class references depend on the active Changebox, the version of an object's class is never changed. The consequence is that an object instantiated in the execution scope of one Changebox cannot be moved to the scope of another Changebox in which the shape of its class is different, *i.e.*, where fields were added or removed.

An early version of our prototype made use of Smalltalk's exception mechanism to search down the stack for the active Changebox. To improve performance we then enhanced the implementation to cache the active Changebox within the executing process. We added an instance variable to the system class `Process` to store the active execution scope. This variable is changed when starting to execute a block for which a new Changebox is defined. On leaving the execution of the block the previous value is restored. This approach avoids expensive stack lookups while perserving the semantics.

We now provide the details of how the standard method lookup and access to class references are modified to reflect the runtime scope.

Method lookup. Since multiple versions of a given method may be simultaneously active in a Changebox-aware system, in our prototype implementation we modify the method lookup to find the right version in the currently active Changebox.

In Smalltalk, several techniques for message passing control exist [17]. Our implementation is based on method substitution [7], thus making use of the following reflective capability of the VM. The method dictionary of a class associates selector names to compiled methods. If a selector maps to an object which is not an instance of the class `CompiledMethod`, then the VM sends another special message to this object (*i.e.*, `#run:with:in:`). We use this mechanism to dispatch the message send to the actual version of a method which depends on the active execution scope.

The dispatching process is illustrated by Figure 4. As an example, let's consider the classes `A` and `B` and a set of Changeboxes which define and remove several methods. The message `#m` sent to an instance of class `B` now triggers the following sequence of events.

1. The VM performs a normal method lookup and finds the associated object of selector `#m` in the method dictionary of `B`. The object is a method element for `B>>m`.
2. Since the associated object is not of type `CompiledMethod`, the VM sends the message `#run:with:in:` to this object (the method selector `#m`, an empty collection for the arguments, and the original receiver instance `b` are passed as arguments).

3. The method element gets the active scope from the currently executing process. Within this execution scope a lookup of the Changebox which last changed the element is started.
4. The matching Changebox is the one that has a definition change specification for the method element `B>>m`.
5. This Changebox then returns the actual compiled method, which is eventually executed on the instance of `B`.

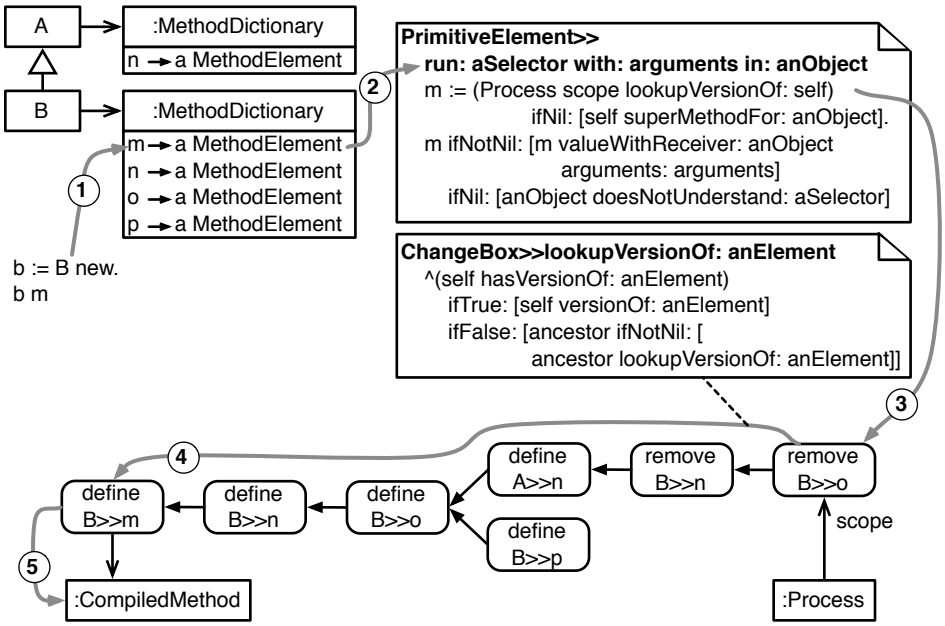

Fig. 4. Scoped Method Lookup.

If in step 4 no Changebox was found or if the method was not found, the lookup would continue in the superclass. The message `#n` sent to an instance of `B` is an example of this case. Since method `#n` was deleted from `B` in the second last Changebox, the method `#lookupVersionOf:` returns nil, which forces a second lookup in the superclass `A`. This lookup of the method element `A>>n` then succeeds.

If the superclass is nil a `DoesNotUnderstand` exception is raised, simulating the normal behavior of the VM. This case is exemplified by sending the message `#o` to an instance of `B`. Since the element `B>>o` was removed and there is no definition in any superclass, the message `#doesNotUnderstand:` is sent to the instance of `B`.

Sending the message `#p` results in an exception as well because the method for `#p` is defined in a different branch and cannot be found in the current execution scope.

To improve performance, in each Changebox all previous lookup results are cached. On each subsequent request for the same element in the same execution scope, the value from the cache can be returned directly. This leads to a dispatching overhead which does not depend on the number of Changeboxes (*i.e.*, number of versions in the system). As with a compilation phase, the caches for a specific execution scope can be filled up front. Benchmarks are discussed in Section 5.

Class reference resolution. In Smalltalk, class references in the source code are resolved at compile time and are stored directly in the `CompiledMethod` instance. Whenever a class is rebuilt, all references to this class are updated by the VM using object identity swapping. Additionally, all classes are stored in the global `SystemDictionary` instance.

The global identity swapping mechanism conflicts with the requirement of having different versions of the same class simultaneously and selecting one depending on the active execution scope. Hence, class references should not be resolved at compile time, but rather at runtime.

Our implementation solves this problem by modifying the compiler. The bytecode output for class references is changed to perform a lookup instead of pushing the class reference from the literal frame of the method onto the stack. With this level of indirection the reference to the appropriate instance of the class can be obtained by sending a message to the global `SystemDictionary` instance which then returns the appropriate version of the class.

Furthermore, the system dictionary is modified to store elements instead of classes in case they are under control of Changeboxes. Those elements then determine the actual version of the class using the same lookup mechanism as for the method lookup discussed above.

4.3 Merging

A key element of Changeboxes is that different strategies for merging system snapshots can be provided, depending on the application domain.

Several Changeboxes can be joined together, combining all the changes they contain. This may lead to various conflicts about which version of an element to use in the merged snapshot. A conflict occurs when each of two merged branches contains a Changebox with a different change specification but for the same element. Additionally, removal specifications for certain elements can conflict with specifications for dependent elements. For example, the removal of a class conflicts with the definition of a new method for that class.

There is no universal strategy for conflict resolution that would be appropriate for all possible applications of Changeboxes. Various approaches for conflict resolution in merge processes have been proposed for different application domains. For example:

- A wide variety of popular software revision control systems such as RCS [46], CVS [10] or Subversion [13] work with line-based three-way merging. These tools do not consider any semantical information [34], neither of the versions to be merged nor of the changes that led to these versions. Furthermore, they are entirely text-based and therefore do not take any objects into account.
- A different approach is operation-based merging, which is designed for object systems and works on the information about the changes performed on the objects to be merged [21,33,37]. Operation-based merging works on arbitrary object types and allows one to detect semantical conflicts unrecognized by textual algorithms [34].
- Finally, depending on the application, very specific conflict resolution strategies can be demanded as well. Maybe the specifications from one branch should always overwrite the ones from the other, or the change that was performed later in time should be chosen.

Since the merge strategy to be used will depend on the application domain, we have kept the implementation of Changeboxes open to any possible algorithm. Different algorithms can be implemented by subclassing `MergeStrategy` (see Figure 3). Changeboxes that merge two or more ancestors delegate the conflict resolution to their instance of merge strategy. A merge strategy also orders the selected specifications following the concept of operation-based merging taking the dependencies between specifications into account. For example, class definitions are applied before the definitions of their methods.

As a proof of concept we implemented three different merge strategies. Two simple ones which run without user interaction; one based on the specification order of the system snapshots to merge, the other based on the point in time a change was performed. A third, more sophisticated strategy asks the user for interaction by presenting a list of conflicting changes to be resolved manually.

5 Performance evaluation

Although the Changebox model offers attractive possibilities for managing software evolution, the runtime cost must be acceptable or it will never be used in practice. The proof-of-concept prototype presented in the previous section adopted a very straightforward implementation strategy based on the existing Smalltalk runtime architecture. The only optimization adopted is to cache lookup results, thus dramatically improving the cost of repeated lookups.

The goal of this section is to investigate further the actual cost of our simple implementation strategy, and thus to identify areas where performance improvements would be needed for a practical implementation of Changeboxes.

We report on two kinds of benchmarks. First we evaluate the runtime overhead of two different kinds of real application by comparing runtime in plain Smalltalk versus Changebox-aware Smalltalk. Second, we carry out micro benchmarks to identify more precisely the overhead of method lookups and class reference resolution with two different implementation strategies.

5.1 Benchmarking Real Applications

We benchmarked[2] two applications, Hessian and Pier. In each case we loaded all available versions of these applications into a Changebox-aware Smalltalk image, thus generating a large sequence of Changeboxes representing the development trail of the applications. We then compared the performance of these applications in the Changebox-aware Smalltalk against that in a regular image.

Hessian. Hessian is a binary web service protocol application with 28 classes and 468 methods. Hessian comes with 107 test cases, which we used to compare performance with and without Changeboxes. In a plain Smalltalk image, these tests run in 3.85 seconds.

The source code repository contained 13 versions for this project, which generated a total of 570 Changeboxes. In order to evaluate the cost of lookup through different numbers of Changeboxes, we also evaluated the performance of several different versions of Hessian with additional, dummy Changeboxes introduced in between the Changeboxes representing the real changes to the application. The results of the experiment are shown in Table 1.

Number of Changeboxes	**1st execution**		**2nd execution**	
	Time	**Ratio**	**Time**	**Ratio**
570	4.33 s	1.12	3.96 s	1.03
29070	67.73 s	17.57	4.09 s	1.06
57570	112.13 s	29.09	3.99 s	1.03
86070	173.12 s	44.92	4.02 s	1.04

Table 1. Hessian: runtime in seconds of 107 tests with 570 Changeboxes and artificially added Changeboxes. Runtime without Changeboxes: 3.85 s.

The first row shows the values for the original 570 Changeboxes. The first time the tests are run they take 4.33 seconds, representing an overhead of 12% (*i.e.*, a ratio of 1.12 compared to the time they take in an unaltered image). The second time the tests are run, the message and class lookup results have been cached, so the overhead drops to just 3% (*i.e.*, a ratio of 1.03).

In the subsequent rows 50, 100 and 150 dummy Changeboxes have been inserted in between each connected pair, respectively. When we add large numbers of dummy Changeboxes, the ancestry becomes larger, and the lookup has to go deeper and thus takes a longer time to complete. The time increases linearly with the number of Changeboxes in the ancestry and becomes up to 45 times slower (*i.e.*, 173.12 s) with over 86000 Changeboxes in place. For the second and subsequent runs, however, the overhead is negligible, between 3% and 6%.

Pier. The second case study is Pier, the content management system we use as motivating example in Section 2. We loaded 115 versions of Pier into a

[2] All benchmarks were performed on an Apple MacBook Pro, 2 GHz Intel Core Duo

Changebox-aware Smalltalk image, the most recent of which consisted of 194 classes and 1883 methods. In total, 6283 Changeboxes were created from these versions. As test bed we used 1057 tests, which run in a total of 1.01 seconds in a plain Smalltalk image (average time over 100 runs).

As with the Hessian case study, we inserted dummy Changeboxes to stress test the lookup. The values in Table 2 show similar results as for the Hessian implementation: a linear growth of the running time for the first execution and a constant time once the caches are filled. The overhead for running the Pier tests with a cache is much bigger with a ratio of about 4.9 compared to a system running without Changeboxes.

Number of Changeboxes	**1st execution**		**2nd execution**	
	Time	**Ratio**	**Time**	**Ratio**
6283	24.28 s	23.96	4.87 s	4.83
56547	305.40 s	301.37	5.00 s	4.94
106811	489.63 s	483.18	5.02 s	4.95

Table 2. Pier: runtime in seconds of 1057 tests with 6283 Changeboxes and artificially added intermediate boxes. Runtime without Changeboxes: 1.01 s

As a second performance evaluation we examined how many requests a Pier application server can handle with and without Changeboxes. The benchmark was performed with the tool ApacheBench to request 1000 times in sequence the same page. In a first run we requested a small page, in a second run a large page (see Table 3).

Page size	**without Changeboxes**	**with Changeboxes**	**Ratio**
8 kB	10.36	4.83	2.14
179 kB	4.60	1.82	2.53

Table 3. Pier: requests/second with and without Changeboxes, average over 1000 requests.

Compared to the unit tests benchmarks (Table 2) the slowdown for this benchmark considerably smaller (*i.e.*, 2.53 compared to 4.83). The reason is that only part of the execution time is effected by Changeboxes, as the web server and web application framework Pier is based on are not under control by Changeboxes.

Comparing Hessian and Pier. The difference in overhead between the two applications is quite striking, but can be easily explained. Only methods and classes belonging to the loaded application are encapsulated in Changeboxes. Those belonging to the Smalltalk base system itself are unaffected, and experience no overhead. The Hessian protocol implementation makes heavy use of

standard Smalltalk classes (*e.g.*, `Array`). As a consequence relatively few messages are sent during the test run to objects affected by Changebox-based lookup.

Pier, on the other hand, is a much larger framework, and the test runs cause a much larger proportion of messages to be sent to objects coming from Pier itself. As a result, a far greater proportion of message sends must be resolved with the Changebox-based lookup, and the overhead is consequently much higher.

5.2 Micro Benchmarks

In a conventional Smalltalk image, message sends are directly performed by the VM, and class references are resolved at compile time, resulting in no run time costs (apart from pushing the class on the stack). In our prototype, both lookups are performed at runtime, without special support from the VM. We performed several micro benchmarks to assess the cost of message sending and class lookups in a Changebox-aware image using two different implementation strategies.

The benchmarks are shown in Table 4. We compared the times for message sending and class lookup (i) in the global scope (*i.e.*, without Changeboxes), (ii) using the exception handling mechanism for the execution scope lookup (*i.e.*, dynamic variables), and (iii) with the scope cached in an instance variable of the current Smalltalk Process object.

Operation	**Global Scope**	**Dynamic Variables**		**Process Variable**	
	Time	**Time**	**Ratio**	**Time**	**Ratio**
10^6 sends	100 ms	14023 ms	140.0	6510 ms	65.1
10^6 lookups	2030 ms	13724 ms	6.8	6053 ms	3.0

Table 4. Benchmarks for message sending and class lookup, average over three runs.

The micro benchmarks are all done with only one Changebox and with a filled cache (*i.e.*, corresponding to the second execution) since we wanted to measure the pure overhead of the basic mechanisms used for the message send and class access. As all benchmarks are run in a Changebox-aware Smalltalk, class reference resolution is always performed at run-time. This causes the high class lookup time measured in the global scope.

The major difference between the execution times is not surprising since it merely reflects the difference between executing compiled basic operations and interpreting them. More interesting is the difference between the two implementation strategies. There is a factor of two difference between using dynamic variables and the Process instance variable.

We have further investigated this difference by also varying the stack depth. Table 5 shows the access times for the scope value based on different method context stack depths. This is important because dynamic variables have to search the stack upwards for the definition of their value.

The difference of the two approaches is even more significant when measured separately. Even without intermediate method calls, the access of the `Process`

Scope access	Dynamic Variables	Process Variable
Stack depth 0	7455 ms	258 ms
Stack depth 100	7967 ms	258 ms
Stack depth 1000	17883 ms	255 ms

Table 5. Benchmarks for scope access (10^6 times each), average over three runs.

instance variable is 29 times faster than the access of the dynamic variable. With growing method context stacks, dynamic variables get increasingly slower, while the access to the `Process` instance variable remains at a stable value. The results of these experiments led us to the decision to use the Process instance variable for the prototype implementation of Changeboxes.

As shown in Table 4, 10^6 message sends take 6510 ms to run with the Process instance variable. How expensive are the different parts of the modified method lookup as described in Section 4.2? We split up the lookup into the following three measurable steps:

1. Dispatch the message send on the object in the method dictionary.
2. Determine the scope with Process instance variable strategy.
3. Lookup the version of the method in the scope and execute it.

The first step only takes a fraction of the total time with 146 ms (2.2%). The scope lookup in the second step takes 258 ms (4.0%) as shown in Table 5. The last step uses the most time with 6106 ms (93.8%).

The first step is fast because it uses a reflective capability of the VM. The second step only accesses instance variables and is not very time consuming as well. The third step, in contrast, is much slower because it consists of many message sends including a dictionary lookup implemented in Smalltalk. We discuss possible ways to further improve performance in Section 6.

6 Discussion

First, we consider possible directions for applying Changeboxes, and then we discuss some of the problems and challenges for developing Changeboxes into a truly effective tool for managing software evolution.

We have already seen how Changeboxes can be used to manage multiple development branches of an application within a single running application. A Changebox can therefore package bug fixes and refactorings as well as new features. Furthermore, Changeboxes can be dynamically deployed, thus avoiding the need to shut down and restart the application. In some cases, non-invasive changes can be applied to running sessions, thus affecting the behaviour of existing objects, not just newly created ones. As such, Changeboxes are intended to serve as a general-purpose mechanism for controlling the scope of software evolution.

Another application for Changeboxes would be to package extension to third-party software. Class extensions, for example, add or override methods to existing classes. A potentially serious problem with class extensions is that the

newly defined behaviour may break existing clients. A Changebox tackles this by providing a well-defined execution scope within which these extensions are visible. (See also the discussion of Classboxes in Section 7.)

A Changebox does not simply specify a set of software elements, but rather how some existing software elements from an existing system snapshot are *changed* to yield a new snapshot. How exactly these changes are specified and how sets of changes are merged to resolve eventual conflicts will depend on the application domain. One particularly interesting track is to consider change specifications as general *transformations* which may be applied to a variety of software elements. By capturing high-level changes directly in the tools that produce them, we are able to to encapsulate general, reusable transformations rather than low-level edits. With appropriate tool support, such change specifications could be composed, merged and replayed to yield new system snapshots. This can be used for example to ease framework refactoring: all refactoring transformations would be stored as high-level change specifications. Clients that want to migrate to a new version can use these change specifications to transform their own code.

In the current implementation, the Changebox which provides the execution scope for method lookup is set explicitly. This may be done deliberately by a developer, as outlined in the scenario of Section 2. Alternatively, the current Changebox could be set implicitly depending on properties of the current runtime context. Such a mechanism would enable applications to adapt their behaviour at runtime. This would be useful in particular for implementing context-aware, mobile applications, but it would also be interesting for other domains. Consider, for example, collaborative work applications where the same objects may exhibit different behaviour or features depending on who is interacting with them. For each user there would be a Changebox providing those features that should be in effect when that person is using the system [22].

The current implementation is performant enough to serve as a proof-of-concept prototype, and to offer a platform for realistic experiments. A real implementation, however, would need to address several problems:

- To improve performance to meet the requirements for real usage and for a system in which also library classes are under control by Changeboxes would require a scope-aware execution at the VM level.
- The current implementation keeps all Changeboxes in memory, even though they may not all be used. Having a policy to transparently swap-out unused Changeboxes and flush lookup caches would heavily reduce the memory footprint.
- Currently, the class of an object remains that from which it was instantiated in its execution scope. Changeboxes do not offer any mechanism for migrating to new internal representations or new classes. Such support would be necessary to fully enable dynamic adaptation of running objects as they are accessed from the execution scope of different Changeboxes.

One problem which we face with the Smalltalk implementation is that the notion of *execution scope* is not explicit, so it must be simulated by modify-

ing the method lookup. An implementation based on an explicit and efficient scoping mechanism would lead to a more natural implementation strategy. The performance analysis (Section 5) has shown that most time is spent in image level Smalltalk code. We plan to experiment with moving the lookup logic into the VM. This alone should provide a substantial improvement in performance. Other ideas to explore are more elaborate caching schemes, for example one cache per send side that takes the current execution scope into account, similar to a *Polymorphic Inline Cache* (PIC) [28].

Current versioning systems support collaboration by allowing a programmer to checkout a snapshot from the server, perform changes locally and then commit them. In agile environments, programmers are suggested to perform frequent commits to stay as much in contact with the overall development as possible. In a Changebox-aware system, we can envision a server which is the development platform for all developers, and eventually even the deployment platform. In such a system, programmers transparently and concurrently perform changes. They can effectively see who is changing the system when and where, and can choose to integrate at any point.

The above example would pose a challenge to the way we should navigate and even think about Changebox-based systems. We currently have difficulty managing large information spaces — the space to navigate becomes much larger if we have access to every possible version of software elements. Changeboxes should make it easier rather than more difficulty to navigate this space, but we are currently lacking good metaphors for representing and navigating the version space.

Having the entire history of changes provides us with new kinds of data for understanding how systems evolve. For example, currently, a large body of work is invested into recovering the meaning of the past changes (*e.g.*, refactorings). However, given that the meaning of a change is encapsulated in Changeboxes we can concentrate more effort on understanding why the changes have happened. We can encapsulate in Changeboxes the actions of a programmer to create the changes (*e.g.*, clicks, menu activations, keys pressed). Such information can reveal which tools are used for development tasks and how they are used.

7 Related work

Changeboxes superficially resemble Classboxes [6,8,9], a module system that provides scoped access to class extensions. Within the scope of a Classbox, new classes may be defined, or classes may be imported from other Classboxes and extended with new or overridden methods. Extensions are only visible from within a Classbox, or another Classbox that imports classes from it. As a consequence, within a single running system, different versions of the same class may be active. Classboxes only support addition and overriding of methods; they do not support removal and thus cannot model changes. Classboxes also do not support high-level change specifications — only new method definitions. They also do

not support any general merging operations — method extension simply override existing methods of the same name.

Virtual classes [20,36] allow class names to be looked up dynamically. Virtuality of classes, however, is associated with a hierarchy of encapsulating entities, rather than with a particular version of the system as it evolves.

Piccola [2] is a language for specifying applications as compositions of software components. The key mechanism in Piccola is the notion of a first-class namespace (or *form*) which is used to encapsulate the services of a component [1]. Forms also serve as the execution context for scripts. In particular, within a single running application, different execution contexts can be simultaneously active. Like Changeboxes, forms are immutable. Piccola does not provide any special support for encapsulating or merging changes.

PIE [11,23,24,25] was an experiment to extend the Smalltalk object model with the notion of *views* coming from frame-based languages. PIE is implemented in itself, and therefore, the code does not consist of regular Smalltalk classes, but of PIE *nodes*. PIE provides code with multiple views, *i.e.*, representing design decisions from the perpectives of different developers. PIE does not support the possibility of multiple views to be simultaneously active. Before execution, code is flattended to regular Smalltalk classes.

ContextL [14] is a language to support Context-Oriented Programming (COP). The language provides a notion of *layers*, which package context-dependent behavioural variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. Layers are dynamically enabled or disabled based on the current execution context. ContextL does not support a more general notion of change specification.

Us [44] is a system based on Self that supports subjective programming. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContexL and does not provide a first-class representation of change.

Aspect-Oriented Programming (AOP) [29] provides a general model for modularising cross cutting concerns. *Join points* define points in the execution of a program that trigger the execution of additional cross-cutting code called *advice*. Join points can be defined on the runtime model (*i.e.*, dependent on control flow). Although AOP is used to effect changes to software systems, the focus is on cross-cutting concerns, rather than on software changes in general. The availability of control flow based pointcuts enables different executions to execute different code, but it is normally not used to express versioning.

Gemstone [42] provides the concept of class versions. Classes are automatically versioned, but existing instances keep the class (shape and behavior) of the original definition. Instances can be migrated at any time. Gemstone provides (database) transaction semantics, thus state can be rolled back should the migration fail.

In Java, new class definitions can be loaded using a class loader [32]. Class loaders define namespaces, a class type is defined by the name of the class and it's class loader. Thus the type system will prohibit references between namespaces

defined by two different loaders. Class loaders can be used to load new versions of code and allow for these versions to coexist at runtime, but they do not provide a first-class model of change.

Dynamic software updating has seen a lot of research over the last years [19,27,41,30]. The focus here is updating a system at runtime, both code and data. All of these systems are not concerned with providing a first-class model of change.

In Erlang [5,4] two different versions of the same software artifact can be active at the same time. When code is loaded in the running system, it retains both the old and new version. Calling conventions define which code is called. This allows for a module to continue to execute old code until it is restarted. There are at most two versions active at any time. If a third version is loaded, all processes executing the oldest code are killed. Erlang focuses on on providing a robust model for dynamic code loading. It does not try to model change.

CLOS [12] provides a protocol for changing class definitions at runtime: whenever a class definition is changed, all existing instances are prepared to be updated to the new version when they are accessed subsequently. The process of updating can be customized by the programmer. In both systems, the focus is not on encapsulating change, nor on providing multiple execution contexts within the same running application.

DOORS and its Smalltalk prototype [35] enable dynamic object evolution. Depending on a condition, objects can be altered or extended at runtime. The usefulness of this feature is shown for modeling domain objects, it does not provide a general model of change.

8 Concluding remarks

Changeboxes offer a simple and uniform mechanism for encapsulating change specifications. They provide a consistent execution scope for running applications, which means that different versions of the same software elements can be simultaneously active within one software system. Changeboxes are immutable, so they can be safely combined and merged to form new Changeboxes without affecting existing ones.

Changeboxes have many potential applications for managing software evolution. Our prototype implementation illustrates how bug fixes, new features and refactorings can be safely integrated into a running system without impacting active sessions. Although the prototype is intended only as a proof-of-concept for demonstration purposes, its performance is more than adequate to illustrate the potential benefits of the approach.

Future directions include:

- developing a cleaner and more efficient implementation approach based on first-class execution scopes,
- providing support for expressing higher-level, composable change specifications,
- developing a broader spectrum of merging operations for changes,

- support for migrating the representation of running objects,
- mechanisms to support context-aware applications by automatically enabling Changeboxes based on properties of the current context,
- experimentation with new metaphors for developers to navigate the space of changes.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). We would also like to express our thanks to Alexandre Bergel, Pascal Costanza, Stéphane Ducasse, Orla Greevy and Roel Wuyts for their help in reviewing various drafts of this paper.

References

1. Franz Achermann and Oscar Nierstrasz. Explicit namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programning Languages, Proceedings of JMLC 2000 (Joint Modular Languages Conference)*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.
2. Franz Achermann and Oscar Nierstrasz. Applications = components + scripts — a tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.
3. Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proceedings ECOOP 2005*, volume 3586 of *LNCS*, pages 144–168, Glasgow, UK, July 2005. Springer Verlag.
4. Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, 2003.
5. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
6. Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Berne, November 2005.
7. Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with Squeak. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.
8. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.
9. Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.
10. Lucy M. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 181–193, October 1990.
11. Daniel G. Bobrow and Ira P. Goldstein. Representing design alternatives. In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, July 1980.

12. D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon. Common lisp object system specification, x3j13. Technical Report 88-003, (ANSI COMMON LISP), 1988.
13. Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly & Associates, Inc., 2004.
14. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, New York, NY, USA, October 2005. ACM Press.
15. Daniel Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, September 2005.
16. Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(2):83–107, April 2006.
17. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
18. Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, September 2004.
19. Dominic Duggan. Type-based hot swapping of running modules. In *Intl. Conf. on Functional Programming*, pages 62–73, 2001.
20. Erik Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.
21. Martin S. Feather. Detecting interference when merging specification evolutions. In *Proc. Fifth International Workshop on Software Specification and Design*, pages 169–176, 1989.
22. Ayse Göker and Hans I. Myrhaug. User context and personalisation. In *ECCBR Workshop on Case Based Reasoning and Personalisation*, Aberdeen, UK, 2002. invited paper.
23. Ira P. Goldstein and Daniel G. Bobrow. Descriptions for a programming environment. In *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*, August 1980.
24. Ira P. Goldstein and Daniel G. Bobrow. Extending object-oriented programming in Smalltalk. In *Proceedings of the Lisp Conference*, pages 75–81, August 1980.
25. Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox PARC, December 1980.
26. David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 321–335, Berlin, Germany, 2006. Springer-Verlag.
27. Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005.
28. Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.
29. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet

Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
30. J. Kramer, J. Magee, and A. Finkelstein. A constructive approach to the design of distributed systems. In *Proc 10th Intl Conf on Distributed Computing Systems*, pages 580–587. IEEE, June 1990.
31. Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
32. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998.
33. Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.
34. Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.
35. Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97*, pages 190–219. Springer-Verlag, June 1997.
36. Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.
37. Jonathan P. Munson and Prasun Dewan. A flexible object merging framework. *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 231–242, 1994.
38. Tien Nguyen, Ethan Munson, and John Boyland. An infrastructure for development of object-oriented, multi-level configuration management services. In *Internationl Conference on Software Engineering (ICSE 2005)*, pages 215–224. ACM Press, 2005.
39. Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind and Uwe Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.
40. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, and Adrian Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.
41. Manuel Oriol. *An Approach to the Dynamic Evolution of Software Systems*. Ph.D. thesis, Centre Universitaire d'Informatique, University of Geneva, April 2004.
42. Allen Otis, Paul Butterworth, and Jacob Stein. The GemStone object database management systems. *Communications of the ACM*, 34(10):64–77, October 1991.
43. Lukas Renggli. Magritte – meta-described web application development. Master's thesis, University of Bern, June 2006.
44. Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.
45. Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 268–285. ACM Press, 1996.
46. Walter F. Tichy. RCS — a system for version control. *Software Practice and Experience*, 15(7):637–654, July 1985.
47. Pascal Zumkehr. Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern, February 2007.