

enables to understand the impact of individual changes with respect to test results.

CoExist also allows for running tests (and performing other computations) on previously created versions on-demand. This is useful when programmers want to avoid running the entire test suite for each newly created version, because this might take too long or might consume too many resources. However, running all tests late in the process can reveal problems that an early change has introduced. CoExist supports programmers in localize this change by running the failing tests on all previous versions (or a subset of them). This allows for finding the change that initially caused the test to fail with little effort.

This concept of performing computations on previous versions also presents advantages for other scenarios. For example, when programmers implement a unit test after implementing the corresponding feature, they can still check whether all of the changes have been necessary to make the test pass. Therefore, they can run the test on previous versions and inspect test results. Another scenario involves the desire to reproduce a recently observed problem that was absent in previous versions of the system but for which no tests exist. CoExist provides additional support for such scenarios where the source code of newly written tests does not exist in previous versions of interest. For such scenarios, the back-in-time analysis involves an additional step: CoExist first applies the changes that represent the newly written tests to the version where the tests need to be executed. Then, CoExist executes the tests, collects the results, and presents them. This feature is somewhat similar to “git-bisect”. CoExist advances “git-bisect” by integrating it into the working environment and by preserving access to all intermediate versions between two explicit commits.

2.5 Re-Assembling Changes

Programmers often perform mutual independent modifications during one task. Still, they might want to share these modifications separately. CoExist provides dedicated support for this need and allows for re-assembling changes into incremental improvements, to be shared with a regular VCS.

The support for re-assembling changes is based on the ability to select and re-apply individual changes to any version. CoExist associates with each version the change that caused its creation. Programmers can then apply such a change to a different version, thereby creating a new version. This approach is commonly referred to as *cherry picking*. As for all versions, programmers can explore this version in an additional environment, inspect test results, run the application and conduct acceptance tests.

The support for cherry picking enables programmers to extract incremental improvements that are spread over a set of many changes. Consider for example that a task has involved a refactoring that the programmer must manage and share as a separate improvement. Programmers can first have a look at the list of all versions to identify both the individual changes that constitute the refactoring and the version from which the main task started. Programmers can then apply the refactoring changes to this initial version which will implicitly create a new branch. If programmers consider this branch a meaningful increment, they can share it with other programmers by pushing it to the used VCS. The proposed support for cherry picking can also help programmers to correct problems in isolation of others changes and to clean the source code from remaining glitches, such as debugging instructions. Using the interactive mode of the “git-add” command can help programmers extracting small commits out of many changes. Our approach improves on this command by proposing the selection of changes to commit based on structured elements ordered by time of change instead of selecting lines ordered by their position in a file. Moreover,

our approach makes it easy to test a resulting source code prior to committing it.

In summary, CoExist relies on the idea of continuous versioning: any change made to the system triggers the creation of a new version storing the change as well as a complete snapshot of the current system. Because this approach creates many versions, we propose dedicated tools to help finding a version of interest quickly. Programmers can study the source code and behavior of these versions independently or in parallel for example to recover knowledge about a previous state of the system. In addition to that, CoExist reduces the effort needed for starting over from a previous state of development if necessary. Contrary to version control systems our approach copes with scenarios where snapshots have not been explicitly made by the programmer and where changes can affect multiple projects. CoExist also makes it easier to locate the cause of failures by running tests continuously and by allowing programmers to run any test, including new ones, one previous versions. Programmers can apply the changes associated to versions to already existing versions, which implicitly creates new branches and allows programmers to test and share in isolation.

3. Implementation

We have implemented CoExist in a Squeak/Smalltalk environment. First we briefly introduce the Squeak environment, and then show an overview of the offered tools. We finally give some details on how CoExist provides the developer immediate and full access to any version.

3.1 Extending the Squeak Environment

Squeak is an open-source implementation of Smalltalk. Like other Smalltalks, Squeak provides a development environment where programmers can open, move, and close windows, much like a desktop operating system (multi-window paradigm). Squeak is a “living” system: any system under development (and Squeak itself) is always running and can be modified without being stopped. This results in meta-objects representing the source code (such as classes and methods) being always available and changeable by the programmer. The Squeak environment automatically updates these meta-objects when the programmers modify their source code, for example when a method is changed. After updating the meta-objects Squeak fires an event that contains detailed information about the change, such as the name of the method that was changed and its previous implementation. Our implementation makes use of this event mechanism. When CoExist receives a change event, it creates a version and attaches the change information from the notification to the new version. Recording change information like this might require additional effort in IDEs that miss fine-grained change notifications.

CoExist provides two complementary interfaces to browse the versions resulting from a programmer’s activity (both shown in Figure 5):

The version browser is CoExist’s implementation of the mockup table presented in Figure 3. The main differences with the mockup are the presence of 2 columns for unit test results, a screenshot of the Squeak environment as it was when the selected version was created (top-right), and three panes giving more details about the version: the first displays the change associated to the version, the second lists all changes between the selected version and the current one, and the third gives details about the change selected in the second pane. For each version, the user can request a context menu, which provides various commands such as going back/forward to the associated version, juxtaposing the version with the current one, or applying

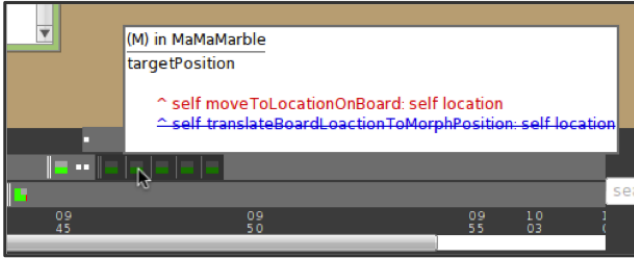


Figure 6. Hovering above version items shows the change.

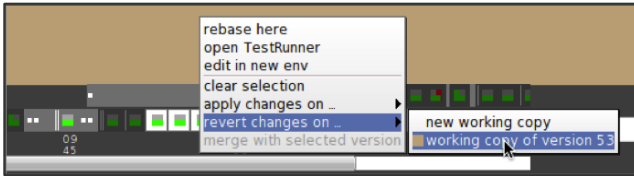


Figure 7. Context menu for one item, or a selection of items (with white background). The version browser provides a similar context menu.

the associated change to the current code base (or any version the programmer selects).

The *version bar* shown at the bottom of Figures 5 provides a condensed view of the recent versions. Contrary to the version browser, the version bar is always visible and ready to use by the programmer. However, it presents much less information compared to the browser, which makes it harder to find a version of interest. Each button on the bar represents a version and shows the result of the unit tests. Hovering over such a button displays a tooltip presenting the originating change (as in the second pane of the version browser). Clicking on a button reveals a menu with actions similar to the contextual menu of the version browser.

To provide integrated access to different program versions, we employed Squeak’s window metaphor. When the user requests to explore a previous version, CoExist opens a dedicated window inside the environment within which all tools will display code for that particular version. Such inner world windows support browsing and modifying source code, running programs, and debugging.

3.2 Snapshotting and Sharing in Squeak

During the implementation and evaluations of CoExist we understood that, for CoExist to be useful, it has to provide both *immediate* and *full access* to any version of interest. *Immediate access* implies that it should be easy to get the desired information, and that the information is provided fast. According to recommendations such as in [18, Ch. 10], CoExist needs to fulfill user requests that are common in not more than *two seconds*. If programmers have to wait too long to get access to a version they might refrain from applying the proposed approach. *Full access* to any version refers to the idea that programmers should be able to explore, modify, run, and debug any program version.

To provide immediate and full access, we use snapshotting and sharing techniques, similar to many VCS. A snapshot is a data structure that stores the state of a system at a particular point in time. As with many VCS, CoExist snapshots the state of the system under development to avoid applying changes before being able to present a version to the programmer. To limit memory

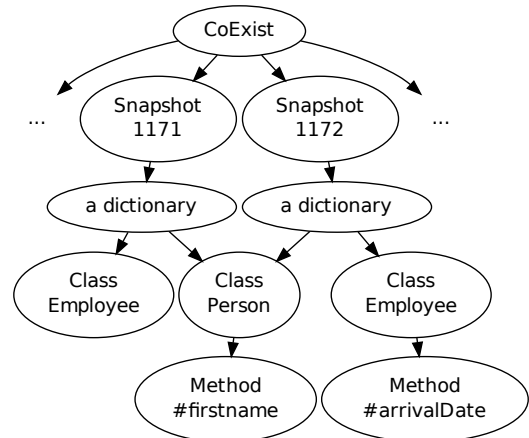


Figure 8. CoExist shares the meta-objects of the development environment between snapshots. The (extract of) snapshots 1171 and 1172 presented here are the same as those of the mockup tables of Figure 3.

consumption, CoExist takes care of sharing as much of the data structure as possible between snapshots.

CoExist improves responsiveness of the development environment by also snapshotting the meta-objects of the environment. Immediate access to a version’s source code is important but insufficient if the development environment takes a long time compiling all this source code and refreshing its internal state. To provide immediate access to any version, CoExist not only snapshots and shares the source code but also all the meta-objects (such as classes and compiled methods) of the Squeak environment. Figure 8 shows that each snapshot contains a dictionary which associates all class meta-objects to their names. Each class meta-object references all its compiled method meta-objects. In Figure 8, version 1172 introduces the `#arrivalDate` method in the class `Employee` (see Figure 3) which results in the duplication of the class `Employee` whereas all other classes (such as the class `Person`) of the system can be shared with version 1171. This means, we turned Squeak’s data structure for managing meta-objects⁴ into a kind of a purely functional data structure (persistent data structure) [13].

As Squeak notifies about a change only after updating corresponding meta-objects, the Squeak development tools edit copies of snapshots with no sharing. We call these copies of snapshots with no sharing *working copies*.

To maximize sharing of method meta-objects, CoExist adapts Squeak to defer the binding of class names to class meta-objects. In Squeak, class names in method source code are bound at compile time to their corresponding class meta-object. To maximize sharing among compiled methods, CoExist adapts the Squeak Virtual Machine to bind at run-time class names to their corresponding meta-objects in the current version. We show in the next section that this run-time binding does not prevent CoExist from providing good performances.

⁴Meta-objects refers to the set of all class objects, which are hold in the *SystemDictionary*, and all contained compiled method objects, which also know the respective source code.

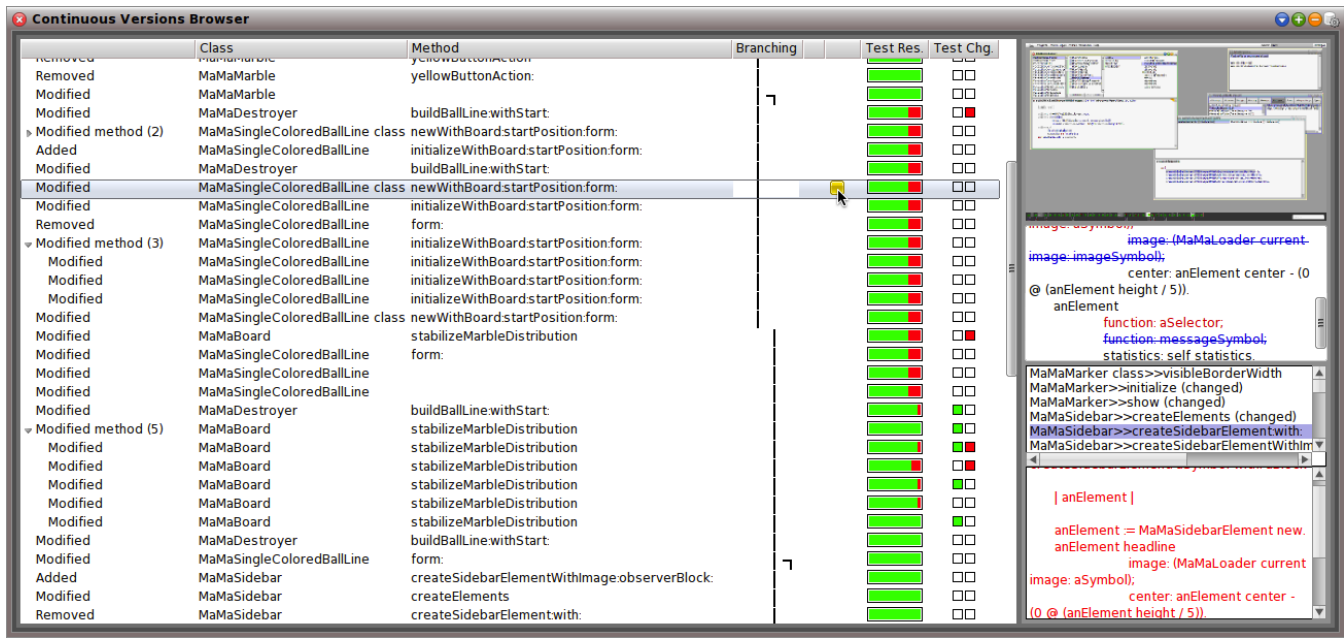


Figure 5. CoExist’s version browser and the version toolbar at the bottom (an enlarged part above).

3.3 Limitations

The current prototype is sufficient for many applications, but it still has some limitations:

- Some classes of the image need to be excluded from late class binding, and thus, changes to them cannot be versioned. This includes in particular all classes that are needed for implementing and running the late class binding mechanism and the core of the versioning mechanism.
- Versioning is limited to source code changes and does not include changes to any other kind of object state in the image (except source code). For example, application state is not under version control. Hence, switching between versions only changes the source code, and not the state of running applications. This means that switching between versions requires a restart of the application under development. A possible improvement could involve snapshotting the state of running applications alongside the source code and meta-objects. With this snapshotting in place, going to different version would immediately bring back the application in the state it was when the version was created. This improvement could be implemented using the same mechanism as the one used in worlds [21], a language construct to allow scoped-states for each object.
- The current implementation also lacks support for direct references to class objects, for example, when a class variable of class A holds a reference to class B (the meta-object). Assumed that a users makes some changes to class B, and then goes back to a previous version, the class variable of class A will still point to the most recent version of class B.

4. Evaluation of CoExist

In this section, we present the evaluation of CoExist concerning its usability and performance. Over the course of development we conducted three informal user studies, which helped us discover usability problems and opportunities. We improved our prototypes taking the results of those studies into consideration.

4.1 Informal User Studies

During our work on CoExist, we regularly asked members of our research group to use our system. We did informal user studies in three phases of our research project: (1) for an early prototype that only supported going back and forward one step at a time, (2) for a midterm prototype that improved speed and provided access to any existing version, and (3) for the final prototype as described in this article.

4.1.1 Early Prototype

Status: The early prototype provided only a simple undo/redo mechanism. Users could go back and forward by triggering commands. Visualizations were not yet available.

Purpose: We wanted to evaluate whether subjects find a project-wide undo/redo feature interesting and meaningful.

Procedure: We demonstrated our prototype to three graduate students. We asked them to perform arbitrary changes and to use the tools to undo/redo these changes.

Results: One student was skeptical about the benefit of the prototype whereas two others were enthusiastic and saw potential. All of them were dissatisfied with the responsiveness of the tool.

4.1.2 Midterm Prototype

Status: Whereas the early prototype only offered simple undo/redo commands, the midterm prototype provided a version bar to identify and go back to any version of interest. The midterm prototype also provided juxtaposing access to multiple versions. Finally, this prototype significantly improved responsiveness by snapshotting meta-objects in addition to source code.

Purpose: We wanted to evaluate if and how subjects make use of the tools. More specifically, we wanted to evaluate our assumption that programmers have a need for withdrawing changes when they perform tasks. We also wanted to better understand whether the provided version bar is appropriate for identifying a previous version.

Procedure: One graduate student, one PhD student, and one post-doc were asked to spend one to two hours improving the design of a 2D puzzle game named MarbleMania, implemented by undergraduate students in the context of a previous lecture. The game worked properly and was covered by 39 unit tests (all of them passing) but implementation showed room for many improvements. At the beginning of the experiment we introduced the game play and gave a conceptual overview of the implementation. Then we asked our subjects to study the source code and to freely refactor the pieces that they felt needed improvement. We encouraged the subjects to implement their ideas as they came to mind without evaluating them mentally. One evaluator sat next to each subject to take notes and to answer questions about the proposed prototypes. We also used screen recording for later analysis.

Results: All three subjects went back to a previous state and continued working from there (two subjects went back two times and one subject one time), which supports our assumption that programmers experience the need for withdrawing changes. In addition to that, we discovered that the version bar is inadequate for finding a version of interest, because it lacks an overview of version information. This motivated the creation of the version browser.

This study also highlighted the need for juxtaposing. Two subjects opened a previous version in a separate environment to study aspects of this version. One of them wanted to study a test execution in a previous version. To do this, he opened an additional environment, found the test, opened a debugger, and stepped to the place of interest. This led us to develop the replication feature that allows for re-creating the currently active view for a different version with just one click.

Furthermore, the subjects felt positive about the tools. Even some weeks after the studies, subjects sometimes dropped in and reported a situation where the proposed tools would have helped for their own project.

4.1.3 Current Prototype

Status: Compared to the midterm prototype, the current prototype includes the version browser (Figure 5), and provides features such as search, highlighting, and grouping.

Purpose: The purpose of this study was to evaluate whether the version browser appropriately allows for identifying a previous version of interest (because the version bar turned out to be insufficient in the preceding study).

Procedure: We asked two graduate students (different than the ones before) to perform a refactoring we specified on the MarbleMania game and then to find the version they started with after half an hour of work. To make finding the version harder, immediately before the experiment, we generated a couple of changes so that the target version is not the first item in the list and so that time-based information do not help the subjects.

Results: The first subject remembered the source code elements he changed first when he started the refactoring. Because the version browser shows the names of each modified element, the subject was able to find the target version within a few seconds. The second subject produced significantly more changes during the evaluation, which made the task of finding the target version more difficult. As a result, the subject took around 20 seconds to identify the target version. During that period of time, the subject first identified a range of candidate versions and then inspected the details of the associated changes.

4.1.4 Discussion

These informal user studies helped us understand the benefits and problems of both our approach and implementation. We understood

the need for responsiveness to make the tools attractive. We also discovered the need for the version browser. Indeed, the information presented by the version bar was insufficient to identify a previous version of interest.

4.2 Performance

User interface guidelines suggest to stay within two to four seconds for frequent user operations [18, Ch. 10]. Furthermore, the results of our informal user studies suggest that performance characteristics are an important factor for the adoption of the proposed tool support. In the following, we report on several performance evaluations.

Setup: As the system under evaluation, we use the Seaside⁵ web application framework, because it extends many parts of the Squeak environment, which challenges the current implementation strategy for sharing meta-objects.⁶ We artificially created 243 versions by loading 5 consecutive releases (from 3.0.0 to 3.0.4). The entire system including Seaside contains 3,312 classes and 68,950 methods. We used the 4.2 release of Squeak/Smalltalk. All measurements were performed on an Apple MacBook Pro 2.93 GHz Intel Core 2 Duo with 4 GB of RAM.

Performance of checking out: Checking out is the action of installing a snapshot into the IDE by copying all meta-objects (as explained in Section 3.2). Checking out requires 1.6 seconds on average, which is below the threshold of 2 seconds.

Performance of loading: Loading is the action of updating a working copy to a different version. Updating a working copy can be implemented either by checking out (1.6 seconds) or by applying a set of changes to the current working copy. We use the latter option as an optimization when the target version has few changes with respect to the current one. Using this option, CoExist requires 188 ms to withdraw 30 changes, approximately corresponding to 30-60 minutes of work (according to our experience from the studies). *These results suggest that our implementation of CoExist meets the desired response time.*

Memory consumption: CoExist consumes memory to maintain snapshots of source code and meta-objects. The 243 created Seaside versions roughly amount to a day of work (at a rate of 30 versions per hour). CoExist requires 68 MB of memory to maintain snapshots of these 243 versions of Seaside, indicating that the size of each version is less than 300 KB on average. *These results suggest that CoExist uses a reasonable amount of memory.*

Development slow-down: In a Squeak environment, applications are executed in the same process as the IDE and thus, IDE performance impacts applications. We measured the overhead that the presence of CoExist in the IDE introduces by timing 10 executions of the 617 Seaside unit-tests: on average, the execution takes 270 ms with CoExist installed and 217 ms without. Using CoExist thus makes executing programs around 1.24 times slower, mostly due to the binding of the class names that must be done at runtime with CoExist. *These results suggests that static class binding (compile/load-time) does not yield a significant performance improvement compared to class binding at run-time. These results also suggests that having CoExist always running does not significantly slowdown the execution of programs.*

⁵<http://www.seaside.st>

⁶For every change in a class meta-object, each subclass' meta-object needs to be copied to create the new version. The higher in the hierarchy a class is, the more subclasses need to be copied.

This first evaluation of CoExist is promising. The results show that subjects appreciated the tools and even missed them after the experiment. Furthermore, CoExist is fast enough for frequent user operations and only consumes a reasonable amount of memory. We plan to conduct empirical usability studies to identify to what degree our approach helps programmers perform better (get a better design and implement faster).

5. Related Work

We discuss the proposed concepts with respect to related approaches and highlight differences.

5.1 Versioning

The undo/redo feature of text editors is very convenient for changing recently entered text. However, undo/redo works on the level of characters, which makes going back to a less recent version of a file rather tedious. Mac OS X 10.7 provides the feature to regularly save files without explicit request. It provides visual feedback to support finding a previous version visual.⁷ This auto-save feature also allows for juxtaposing the current version with previous ones. Nevertheless, such undo/redo concepts handle files independently of each other, while programming typically involves the manipulation of multiple files. Thus, making undone changes requires the developer to manually apply the undo feature an unknown and different amount of time for each changed.

Version control systems (VCS), which are sometimes referred to or part of configuration management systems (CM), can manage multiple files of a project and allow for reverting all files to a snapshot where they were at a given point in time. Developers can employ a VCS to support withdrawing changes. Either developers snapshot manually from time to time, or they let tools automatically snapshot at regular intervals, for example, after each save operation. However, both approaches exhibit limitations. Using the former approach, developers are required to foresee the future, which remains hard. Developers have to continuously assess the likelihood that a future situation requires to discard changes and to go back to the current state. Unfortunately, this is hard to assess and there will be situations where a developer forgot to snapshot at the most appropriate time. Furthermore, this approach makes it hard to focus on the design task, because a control loop keeps reminding the developer to consider snapshotting now. To overcome these problems, another approach is to snapshot at regular intervals.⁸ However, if performed in an unstructured way it easily results in a huge amount of data that is hard to browse, because it lacks one of both meaningful meta-information or meaningful commit messages. In contrast, our approach creates versions that contain information such as the kind of change that happened that guide programmers in finding the snapshot they are looking for, even in the presence of many of them.

Orwell [20] is an early VCS / CM that provides both source and object sharing for Smalltalk systems. Compared to other VCS that employ files as the unit of versioning, Orwell manages versions of classes as well as editions of methods and classes among other. Orwell requires class ownership, but provides an owner with a complete development history. The programmer can also work on multiple versions (or releases) of one class. Nevertheless, according to our understanding, Orwell's versioning scheme is scoped to classes. This means, the user cannot easily withdraw changes that span multiple classes and application parts. The Envy system [14], which is based on Orwell, provides baselining as an additional

concept. This allows for creating named snapshots of all artifacts involved in a project. In contrast, CoExist continuously creates snapshots of the entire system, so the possibilities for exploring without hesitation are not restricted to scopes such as classes.

Changeboxes [3] is an approach to capture the history of a system and permit the existence of simultaneous versions in a single virtual machine and development environment. Because a Changebox encapsulates a particular change in the history of a system it is possible to use Changeboxes to go back to a particular state of this system. Nevertheless, Changeboxes have not been designed to allow going back and, as a result, doing so is tedious: going back requires finding a Changebox in a tree with no meta-information, creating a branch out of it, giving it a name and a color, closing all code browsers, and opening new ones. Our approach makes going back simpler by automating most of the process and presenting all versions of a system with visual meta-information. Moreover, the Changebox prototype implementation shows an important slowdown when used on a project with close to 2,000 methods: in this setup, the system becomes around 4.9 times slower on average. Contrary to the Changebox prototype, CoExist versions the entire workspace (around 69,000 methods). Still, our approach exhibits only insignificant run-time overhead (a slowdown of only 1.2 on average).

5.2 Change Recording for Software Evolution Analysis

The concept of preserving change information between explicit commits has been first introduced by Robbes and Lanza in [15]. The authors recognized that relying only on traditional, file-based VCS limits the possibilities of software evolution analysis. They have proposed recording fine-grained change information in IDEs (such as method modified, instance variable added), as well as semantic borders drawn by tool supported refactorings. Such dense change information, compared to plain diffs, additionally supports reverse engineering activities. It helps programmers to better understand the current state of a system, or why it has been designed or changed in a certain way. In particular, it supports reasoning about *development sessions* [16] of colleagues and to reconstruct what they have done and how. The implemented prototype called *SpyWare* enables programmers to browse the change history ordered by time, and to sort and re-arrange change information. It features support for interactive visualizations to track changes and to reason about statistics. In addition to browsing change history, *SpyWare* also allows for generating the source code of intermediate development states (by applying or reverting change operations).

The *SpyWare* approach has been continued, resulting, for example, in tools such as *Replay* [9] and *ChEOPS* [4]. Using *Replay*, programmers can replay change operations, which have been previously recorded in another development session, to better understand the evolution of the code base. A controlled experiment shows a statistically significant improvement in time required to complete software evolution analysis tasks. *ChEOPS* is an IDE prototype implemented to fulfill the needs of Change-Oriented Software Engineering, which is itself an extension of the *SpyWare* approach. *ChEOPS* extends *SpyWare* by managing changes in different levels of granularity or supporting the declaration of intents.

There is an overlap in the needs for software evolution analysis and explore-first programming: both require a detailed record of change information. But compared to our approach, the above mentioned line of work on software evolution analysis has a stronger focus on the recording of changes, their management, and the visual preparation. For example, the corresponding tools allow for tracking changes on a statement level, but also for combining fine-grained changes into composite ones, and for declaring the intent of change operations [4]. These and other aspects seem to be meaningful complements to our work. However, instead of recording

⁷ <http://www.apple.com/macosx/whats-new/auto-save.html>

⁸ <http://stackoverflow.com/questions/688546/continuous-version-control>

change operations, CoExist preserves the artifacts of intermediate development states including both source code and run-time artifacts, thereby avoiding the need for re-generation. Furthermore, in addition to preserving intermediate development states, which is similar to the above mentioned work, CoExist also provides features such as implicit branching, running tests in background, or browsing, running, and debugging previous versions in additional *inner* environments next to the current one. Such dedicated tool support is essential for explore-first programming.

Another approach that is close to our work is VPraxis, a language which models the history of source code through commands like “create class Person”, and “add field ‘name’”.⁹ Finding a version of interest can then be done via queries such as “which commit last changed this method?”. VPraxis can be attached to a development environment to monitor code changes while programmers are doing them. Except for the unit test results and the refactoring-level grouping, VPraxis could propose tables such as the one in Figure 5. Nevertheless, VPraxis lacks support dedicated to needs such as withdrawing changes to start over, juxtaposing program versions, or back-in-time impact analysis. Moreover, VPraxis is not integrated in a development environment in such a way that multiple versions can be browsed and edited independently.

5.3 Juxtaposing Versions

Orion is an interactive prototyping tool that allows to compare the impact of multiple changes on a software system [11]. Orion uses models as an abstraction to source code and model transformations as an abstraction to changes. Orion has been implemented to permit the manipulation of very large models including more than 600,000 entities, such as classes and methods. As a result of using abstractions instead of source code, Orion can not be used to execute a previous version of a software system or even study its source code. Moreover, even if Orion’s implementation shares some similarities with ours, its goal is inherently different: Orion has been designed for re-engineering changes (such as the removal of dependencies) whereas CoExist target all kinds of source code changes. As a result, CoExist is fully integrated in the development environment and versions are created automatically whereas, with Orion, changes have to be explicitly expressed as a model transformation.

Hartmann and others propose Juxtapose [8], a tool that facilitates the creation and comparison of alternatives through a dedicated code editor. Juxtapose also allows the execution of multiple alternatives in parallel. Nevertheless, Juxtapose is not associated to the history of a software system and thus cannot propose going back or re-assembling changes into incremental improvements.

5.4 Fine-grained Back-in-Time Impact Analysis

There are various approaches that use tests to support fault localization. For example, the continuous testing approach proposes to run tests automatically after each change [17]. Because this approach can only execute tests on the system currently in the development environment, the execution of a test suite is aborted after each change. Nevertheless, it immediately presents the results to the programmer so that he can fix problems as they appear. CoExist improves on that by recording the test results and link them to the corresponding changes, which allows for analyzing test results only when it is convenient. In addition, CoExist also supports running (long/acceptance) tests on previous versions.

The Git VCS provides the “git-bisect” command, which uses binary search on a sequence of commits to find the first commit that introduced a bug. CoExist advances this concept by integrating it into the working environment and, more important, by preserving

⁹<http://harmony.googlecode.com>

access to all intermediate versions between two explicit commits. Furthermore, CoExist can also run unit tests in the background during development, and it provides support for running newly defined tests on all previous versions.

A continuous integration server monitors commits to a VCS and automatically builds and executes tests on each one [6]. By providing history of test results for every commit, these servers allow for back-in-time impact analysis. Unfortunately, these tools can only provide fine-grained analysis if the commits are themselves fine-grained which requires time and discipline: When the commits contain many changes, the programmers have to study all these changes to locate faults.

5.5 Re-Assembling Changes

Best practices include doing small commits (sometimes qualified atomic or logical) that only affects one aspect of the system to facilitate activities such as going back, fault localization, and reviewing [1]. Using the interactive mode of the “git-add” command can help programmers extracting small commits out of many changes. This command allows programmers to define the boundaries of a commit by selecting a subset of the modified lines in each modified file. Our approach improves on this command by proposing change-level selection instead of line-level selection, by ordering the changes by timestamp, and by support to run tests for the defined increments.

6. Summary

We have pointed out limitations of following practices to deal with the risks of changing programs. We have called for *explore-first programming* and argued for the need to preserve immediate access to source code and run-time information of all development states.

We have proposed structured continuous versioning as foundational support. Our approach provides a dense and structured history, and allows for identifying a version of interest fast without requiring programmers to write commit messages. We have presented various tools, built on this foundation, to address the needs of *explore-first programming*. The presented tools make it easy to withdraw a set of selected changes and to recover knowledge by exploring and debugging a previous version next to the current one. Programmers can also defer quality assessment, because CoExist allows for running tests regularly for each commit as well as backward in time. We have implemented a prototype in Squeak/Smalltalk, guided by informal user studies. Our studies suggest that users identify a previous version of interest within a few seconds and that they appreciate the tools. We can conclude that *explore-first programming* is a valuable and feasible alternative to current approaches to deal with the risks of changing programs.

Acknowledgements

We thank Richard P. Gabriel, Theo D’Hondt and his group, Ralf Lämmel and his group for valuable discussions about the paper’s content; Julia Lawall for her detailed comments on a previous version; the various anonymous reviewers of this and previous submissions for pointing out the weaknesses in our presentation; and Felix Geller for joining in hacking the VM adaptation, and Tim Felgentreff for supporting the case study work. We gratefully acknowledge the financial support of the Hasso Plattner Design Thinking Research Program.

References

- [1] Apache2003SVNBestPractice. Subversion best practices, 2009. URL <http://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html>.

- [2] C. Beck, Kent and. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, 2004. ISBN 978-0321278654.
- [3] M. Denker, T. Gırba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and exploiting change with Changeboxes. In *ICDL'07: International Conference on Dynamic Languages*, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352681.
- [4] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 International Conference on Dynamic languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 3–24. ACM, 2007.
- [5] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [6] M. Fowler. Continuous integration, 2006. URL <http://www.martinfowler.com/articles/continuousIntegration.html>.
- [7] G. Goldschmidt. The dialectics of sketching. *Creativity Research Journal*, 4(2), 1991.
- [8] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Symposium on User interface software and technology*, 2008.
- [9] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu. Software evolution comprehension: Replay to the rescue. In *ICPC 2011: Proceedings of the 11th IEEE International Conference on Program Comprehension*, pages 161–170. IEEE Computer Society, 2011.
- [10] D. Kirsh. Thinking with external representations. *Ai & Society*, 25(4): 441–454, 2010.
- [11] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 2010.
- [12] Y.-K. Lim, E. Stolterman, and J. Tenenberg. The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 15(2), 2008.
- [13] C. Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.
- [14] J. Pelrine, A. Knight, and A. Cho. *Mastering ENVY/Developer*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-66650-3.
- [15] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166: 93–109, 2007.
- [16] R. Robbes and R. Lanza. Characterizing and understanding development sessions. In *ICPC 2007: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 155–166. IEEE Computer Society, 2007.
- [17] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE'03: International Symposium on Software Reliability Engineering*, 2003.
- [18] B. Shneiderman and C. Plaisant. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Addison Wesley, 5 edition, 2009. ISBN 978-0321601483.
- [19] M. Suwa and B. Tversky. External representations contribute to the dynamic construction of ideas. In *Diagrammatic Representation and Inference*, volume 2317. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43561-7.
- [20] D. Thomas and K. Johnson. Orwell — A configuration management system for team programming. In *OOPSLA'88: International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 1988. ISBN 0-89791-284-5.
- [21] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: Controlling the scope of side effects. In *ECOOP'11: Proceedings of the 25th European Conference on Object-Oriented Programming*, pages 179–203, Lancaster, UK, 2011. Springer. doi: 10.1007/978-3-642-22655-7_9.