Object Versioning in the Presence of File-based Version Control Systems

Tobias Pape, Jakob Reschke, Patrick Rein, Fabio Niephaus, Marcel Taeumel, Robert Hirschfeld

▶ **To cite this version:**

Tobias Pape, Jakob Reschke, Patrick Rein, Fabio Niephaus, Marcel Taeumel, et al.. Object Versioning in the Presence of File-based Version Control Systems. International Workshop on Smalltalk Technologies, Aug 2019, Koln, France. hal-04053780v1

HAL Id: hal-04053780
https://hal.science/hal-04053780v1

Submitted on 31 Mar 2023 (v1), last revised 11 May 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tracking Objects

Object Versioning in the Presence of File-based Version Control Systems

Tobias Pape
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tobias.pape@hpi.uni-potsdam.de

Jakob Reschke
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jakob.reschke@student.hpi.uni-potsdam.de

Patrick Rein
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Fabio Niephaus
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Marcel Taeumel
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Version control reduces the risk of changing their software and allows them to manage different configurations and to collaborate with others. Most version control systems track files but some programming environments such as Smalltalk do not operate on files, but on objects instead.

Specialized version control systems, such as the ones available for Smalltalk systems, focus on a small subset of objects that can be versioned. Most of these systems concentrate on the tracking of methods, classes, and configurations of these.

To improve the situation for version control of arbitrary objects, we propose a framework for tracking, converting, and storing of objects. It allows version of objects to be stored in an exchangeable, existing backend version control system. In particular, we describe a generic way for capturing and restoring that supports most kinds of objects.

The proposed architecture is evaluated through a prototype implementation that can be used to track arbitrary objects in Squeak/Smalltalk.

The prototype improves the suboptimal standing of user objects with respect to version control and simplifies some version control tasks for classes and methods as well. Thereby, it supports collaboration not just on source code artifacts but also on living object graphs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IWST, August, 2019, Cologne, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Classes and objects*; *Integrated and visual development environments*.

KEYWORDS

Version control, Object-oriented programming, Exploratory programming, Serialization

ACM Reference Format:
Tobias Pape, Jakob Reschke, Patrick Rein, Fabio Niephaus, Marcel Taeumel, and Robert Hirschfeld. 2019. Tracking Objects: Object Versioning in the Presence of File-based Version Control Systems. In *Proceedings of IWST*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Version control is an established practice for source code since at least the 1970s [22]. Version control systems (VCSs) typically manage files, as most programming systems use files to store the source code of programs. However, since source code files seldom convey the current state of a running system, such as its objects, those entities are hard to track with conventional VCSs. Yet, using systems that built

upon a model of long-living state, such as image-based programming systems like Smalltalk and Self, database-oriented systems, or live- and exploratory programming systems, can lead to a split experience. The system's source code can be put in a VCS, enabling safe changes, easy sharing, and recovery of code from earlier versions of a program. The systems's state, however, can hardly be versioned because it does not easily fit into files. Developers who wish for the benefits of VCSs for state have to e.g. serialize state to source code or separate files, and manage the loading of it.

When objects must be converted before they can be put under version control, sometimes there are multiple options of how this can be done. For any given purpose, there are typically multiple file formats in use and multiple perspectives from which an object could be modeled.

Version control supports collaboration. When people make changes to their local copies of a system simultaneously, contemporary version control tools support users in managing different streams of development (branches). Internet platforms such as *GitHub*[1] have made collaborative, distributed development of software easier.

We want to provide support for collaboration on artifacts that are not text, or files, but arbitrary objects built in an exploratory programming such as Squeak/Smalltalk. We propose a version control solution for arbitrary objects that works on top of an established VCS to promote platform reuse.

Contributions. We present an architecture to put arbitrary objects under version control. It allows for customization of how versions of objects are represented and how they could be stored in files, so the most suitable representation can be chosen. The storage of versions is delegated to a backend version control system, so existing systems can be used for their maturity and familiarity to users, and their platforms can be reused for their utility and to save costs.

Further, we present an overview of an implementation of this architecture in Squeak/Smalltalk and discuss its features and limitations.

We do *not* propose a solution for synchronous collaborative editing of objects. That is, the proposed architecture assumes that the users of the exploratory programming environment need to synchronize their changes to objects explicitly.

Structure of this work. In the following section (section 2) we illustrate basic considerations when providing VCS for objects. We present our architecture for putting objects under version control based on established VCSs (section 3). We illustrate our implementation prototype in Squeak/Smalltalk (section 4) and discuss its merits and limitations (section 5). Finally, we discuss related approaches (section 6) and conclude the paper (section 7).

[1] http://github.com (last accessed June 10, 2019).

2 BACKGROUND

In this section we will first illustrate the challenges of object version control by looking at two existing approaches. We then explicate our goals for a VCS for objects in general. Further, we motivate our approach of basing the VCS on an established system in order to ease collaboration. Finally, we describe a set of options the system should provide to users according to previous work.

Due to the large variety of VCS today, a variety of terms and concepts exists. In order to clarify our interpretation of these terms we collected and explained them in Appendix A.

2.1 Challenges of object versioning

In Self, objects are transferred from one image to another via the *Self Transporter* [30]. There are no classes in Self. Objects implementations are reused as prototypes, like in JavaScript. The prototype objects are fully-capable objects themselves, which blurs the gap between meta-objects and "actual instances". For this reason, the Self Transporter can transport any object. It does so by traversing a graph of objects — guided by annotations to objects and slots to fill in information about user intentions — and writing files that contain Self source code expressions that rebuild the captured object graph when evaluated. The serialization format is therefore text-based and general enough to describe arbitrary objects, but it is specific to the Self language. Objects can be exchanged with file-based VCSs, like Smalltalk file outs, but the version control system is not integrated in the Self environment.

The Lively Kernel environment comes provides its users with an exploratory programming environment for JavaScript entirely inside the web browser [8]. Lively components can be shared via a *parts bin* that uses Subversion for version control and publishing. On top of that, version control features like difference detection and merging have been integrated into the Lively Kernel environment [1, 11]. It employs a serialization based on JavaScript Object Notation (JSON) with support for object graphs (with cycles) and instance-specific behavior (i.e. functions that do not belong to a class). This is an elaborate solution for version control of arbitrary objects, but the serialization format is still not adaptable.

All of the systems described above have in common that objects and their meta-objects (e.g. classes and prototypes) and the programming tools *live* in the same environment. More technically, the tools and manipulated objects share a single execution environment. This is different from programming environments for languages like C, Java, or Python, where the execution of the program under development is

usually short-lived in comparison to the programming environment: during a programming session, the source code is repeatedly compiled and run for testing or debugging. This entails a strong separation of software artifacts from the rest, and the ubiquitous use of the file system as a medium of data exchange between the two.

In contrast, loading equivalent software artifacts into a programming environment such as Squeak/Smalltalk can have immediate side effects. Checking out a class for a Java program will replace a single file of source code and any running programs will not be affected by that change until restarted. Checking out a class in Smalltalk does imply changing an existing class definition and possibly compiling some methods, immediately affecting all existing instances of that class. This is a fundamental difference between tracking *live* objects and tracking source code. Our VCS for objects has to account for that difference.

2.2 Goals for a new version control system for objects

Our goal is to put arbitrary objects under version control, not only meta-objects related to source code. To achieve this, source code and meta-objects should be strictly regarded as a special case of objects to be versioned. The diversity of domain objects and their possible repertoire of suitable data exchange formats should be accounted for by separating the serialization of objects from their captured snapshot representation. Moreover, certain version control operations, such as the handling of differences, should also be under the influence of domain-specific types. They may have special requirements for an operation (e.g. to produce differences that are at all useful for the consuming users) or the nature of a type might offer opportunities for improvements over a domain-unspecific, fixed set of procedures and tools. Another reason is that the knowledge and code about the representation of domain objects can stay close to the domain objects themselves. In contrast to version control via import/export mechanisms version control for objects should be controlled from inside the exploratory programming environment. This should support the construction of well-integrated tools.

On the other hand, the new system should not prompt for version control specific specialization too eagerly: users should focus on their project domain and supply specifics for version control only later in the process. An adequate solution that is already sufficient for many objects must be found. Specializations should be optional, rather than required.

One motivation for version control is collaboration which requires an agreement on tools and platforms for exchanging versions. In order to lower the entrance barrier, we want to base it on an existing wide-spread system, such as Git. Further, sharing versioned files (or objects) usually involves a central place to host and exchange the versioned data. While this is not technically necessary in distributed version control systems the presence of platforms such as *SqueakSource*[2], *GitHub*, or *Atlassian Bitbucket*[3] indicates that central repositories are very much desired. Reusing an existing VCS also means reusing existing platforms.

Reusing a file-based VCS benefits resources that are inherently stored as external files. These external files should also be version controlled along with the artifacts resident in the image. Support for external files becomes even more important when multiple programming languages are used in a software project, and the different parts are developed in different programming environments. Like resource files, the primarily file-based source code must be kept synchronized with the Smalltalk parts as part of the software configuration management.

2.3 Diverse representations of objects

There can be many ways to represent one kind of object. Smalltalk methods, for example, can be represented as byte codes or source text. Capturing the former may be brittle because the literals could change and it is specific to the byte code set employed by the virtual machine. But restoring a compiled method from such a snapshot could be much faster than compiling the source code again. Some common requirements for snapshot and serialization formats, among which a trade-off must be made, are performance, portability and interoperability, expressiveness and completeness (i.e. that no information is lost), and human-readability.

The most suitable form of representation might not even depend only on the type of object, but also on the use case of the representation. In the compiled- method example, for collaboration on GitHub, classes and methods are best represented in the form edited by the developers (i.e. as source code). However, if the use case is to distribute the software it can be more beneficial to share a binary representation for performance reasons.

2.4 Object graphs

In general, a single object does not have an inherent meaning on its own. Instead, what makes an object meaningful is a graph of objects that is reachable from it. For example, given a user interface that contains a box for text entry and a button to accept the entered text, the form alone would not be meaningful without the contained text box and button.

[2] http://squeaksource.com (last accessed June 10, 2019).

[3] https://bitbucket.org (last accessed June 10, 2019).

On the other hand, not all objects that are reachable from a given object might be relevant for the purpose of tracking this object for version control. For example, in Squeak's implementation of Morphic, each morph has a reference to its containing morph, the `owner`. If the user chooses to track one morph, its owner might not be relevant for versioning the morph; the owner can change whenever the user puts the morph into another space of the programming environment. Further, if the owner reference were always followed unconditionally, tracking any morph visible on the screen would mean to track the whole *world* of visible morphs. Consequently, the developers desire to manage clear boundaries in the object graph.

2.5 User intentions missing from object graphs

The generalized variant of the issue raised above with the example of morph owners has already been documented for the Self Transporter. According to [30], the following information is required when these objects should be transported:

- To which package a part of an object belongs (different parts could belong to different packages): As packages are meta-objects also, we should actually assume the reverse which is that a package is just a special composed object which we can use as a starting point for capturing a graph. Further, the reach of an object must be expressible, i.e. which references should be captured.
- Whether a reference from one object to another should be captured as is or whether the referent should be replaced by a different value in the transport representation: This applies for example to or strictly dynamic objects such as caches.
- Whether a referenced object is a global object that is assumed to be already present in the target system, thus making it preferable to capture only a symbolic reference to the object: This is relevant as some environments provide a fixed context which can be assumed and does not have to be transported.
- Whether the identity of an object matters when it is restored: This is especially relevant when multiple references to the same object must be transported. If the identity matters all these references have to point to the same instance again. This is not relevant for value objects for example geometric points in Squeak/Smalltalk.
- Whether an object should be recreated from an abstract expression rather than from a complete snapshot representation: This information is particular to the Self transporter. Since we do not want to dictate a particular serialization format or even prescribe the in-memory representation of snapshots for all kinds of objects, this point can be reformulated into "[It must be defined] whether a special type of snapshot should be used to represent an object".

All of these issues require that objects or whole object graphs must be complemented by additional information, which will be here referred to as *object metadata* (or only *metadata* for short).

3 AN ARCHITECTURE FOR OBJECT VERSIONING

In this section, we present our solution to track objects and store them in existing version control systems. We describe how object graphs can be captured and rematerialized, how object identity can be preserved in this process, how differences between two editions of object graphs can be handled, and how it can be supported to have different formats in which snapshots can be stored in an existing VCS. Finally, a generic way to capture any kind of object is presented, so that users do not have to provide own solutions for all types of objects they want to track.

3.1 Storing objects in versions

In order to put anything under version control from inside of an exploratory programming environment, we require a connector component to a version control system. It should be able to access the version history of a repository, create new versions, and possibly manage independent development streams, such as branches. How this connection can be established without focusing on a particular VCS has already been treated by previous work. However, some assumptions must be established on how the VCS deals with objects. This part of the architecture is based on a subset of an abstraction for VCS called Pur [9].

The basic concept is a *version* which describes a revision of a set of objects. Versions can have any number of parent versions. This relationship forms the version history in a repository. Each version contains a *snapshot* of object graphs and their associated metadata. The Pur architecture deliberately does not define what a snapshot consists of because it depends on the particular application. However, Pur defines an entity named *store* that can create and restore snapshots, updating the objects in the store. What "restore" means depends on the particular type of the store. The example implementation *Pur for Newspeak*[9] defines two stores: an *image store* to capture and load classes and methods with a snapshot object, and a *file store* to write and read snapshots to files.

In the following sections of this section, we will further define the snapshots of our version control solution for objects and what they must be able to do.

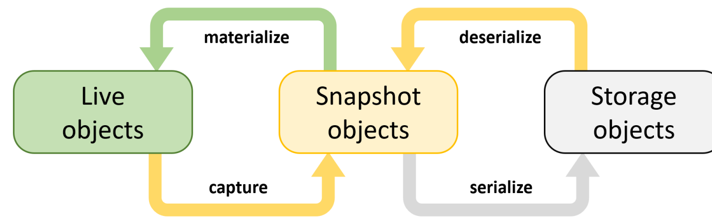

Figure 1: An overview of the three realms making up our architecture for object versioning.

3.2 Live objects, snapshot objects, storage objects

The described version control objects can be categorized into three realms. This makes communication about the different version control objects and about crossing the boundaries of the realms easier. The three realms are shown in Figure 1.

The objects that the users usually interact with are attributed to the *live realm*. Domain objects belong to this realm, as well as their meta-objects that define their types and behaviors, such as classes and methods in Smalltalk. A defining property for live objects is that they would still exist when version control is not even attempted.

The *snapshot realm* contains objects that stand in for editions of objects that originally came from the live realm. Objects in the snapshot realm reside in the memory of the running programming environment, just like live objects, but they solely exist for the purpose of version control. Objects for differences between editions, and other objects that explicitly deal with snapshots are also attributed to the snapshot realm.

The third realm includes all forms of objects that are intended to be stored outside of the programming environment. This realm is be called the *storage realm*. But even when snapshots from the version control were to be transferred directly between two programming environments, the representation "on the wire" between the two processes would belong to the storage realm.

Converting objects between the storage and the live realm always goes via the snapshot realm. Further, there are additional version control infrastructure objects. They deal with objects of one particular realm or operate at the boundary between realms, converting objects from one realm into another. Examples of these are stores, versions, and repositories. They will be attributed to the realm on which they operate or if they operate in multiple realms, they will be attributed to the live or the storage realm. For example, serializers that convert snapshots to storage data are attributed to the storage realm.

Finally, stores, snapshots and versions contain objects in different forms (as indicated by the different realms), but ultimately they always contain graphs of objects. Therefore, at a higher level of abstraction, stores, versions and their snapshots can share a common set of operations, such as computing differences between each other. To generalize such operations, we introduce the abstract type *object container* as the union of the three concepts.

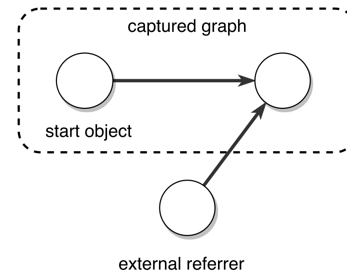

Figure 2: External reference into a captured object graph

3.3 Preserving object identity across system boundaries

In addition to the requirements for transferring objects between environments (see subsection 2.5), there are further requirements for tracking objects for version control: if an object already exists in a target environment, it should be possible to rediscover it there and different snapshots of the same object must have an identification.

Rediscovering objects in the target graph is needed to update objects in-place. An alternative would be to re-materialize the whole object graph with new objects. This would preserve object references inside the captured graph but references from outside of the captured graph would point to the then obsolete instances (see Figure 2). To identify objects in different snapshots, objects are assigned *names* if their identity needs to be preserved, which is the default. However some objects do not identification: value objects and objects that can be identified in the target environment based on their properties (e.g. a `PackageInfo` object in Squeak is fully identified by its name). Such names must be globally unique, even across the boundaries of the programming environment. They can take any suitable form but should be small,

resilient to collisions, and independent of their object's state (i.e. no hash values).

3.4 Capturing and materialization of object graphs

In this section, we describe how graphs of live objects can be captured to create graphs of snapshot objects, and how the inverse operation, materializing live objects from snapshot objects, can be performed (see Figure 1).

3.4.1 Composition of version snapshots and object graphs. The snapshot of a version or a store is a collection of object graphs. Each graph is associated with object metadata that saves decisions about the capturing or storage, which might also be needed to recover the graphs properly. In addition to that, a unique key is assigned to each graph to access it in a snapshot.

Each object graph stores the bidirectional mapping between object snapshots and the names of their objects. Additionally, each stores a start object, from which all other objects in the graph can be reached. New graphs are introduced to a store by telling it to track an additional start object under a given key. The other objects in the graph are then derived from the relationships among objects, guided or restricted by the metadata that is configured in the store.

To capture the snapshot of a store, the store must enumerate the live object graphs that are known to it, convert them to the snapshot realm and collect them into the overall snapshot, together with the metadata (see Figure 3).

3.4.2 Abstract algorithm to capture object graphs. To capture an object graph from a given start object, the graph has to be traversed and a snapshot has to be created for each object. All objects encountered that do not already have names from a previous capture operation must have new names assigned unless their identity does not matter. These names should be persisted in the store that tracks live objects, so future operations on the same graph of objects can look up the named live objects if they still exist. The names are also assigned to the respective snapshot objects, so the live object of a snapshot object can be looked up.

The graph traversal can be realized with an exhaustive search algorithm (e.g. breadth-first search for graphs). The algorithm should include multiple-path pruning, so each object is captured only once. The live objects being captured must be able to direct the graph traversal to related objects.

When a live object is encountered during the traversal, a message is sent to the live object to convert it into its preferred type of snapshot. This is the opportunity for a live object to decide that it should be replaced in the snapshot graph by another object (e.g. by a symbolic reference to itself).

Live objects that know that they are the root of a sufficiently independent subgraph could also decide to start another graph traversal that works differently (see Figure 4). They must also respect the multiple-path pruning of the main traversal. If the snapshots of the objects in this subgraph should be registered normally (with object names) in the snapshot graph that is built by the outer graph traversal, a way to pass the inner snapshots out to the main graph must be implemented. Alternatively, the aim of the separate traversal might be to encapsulate the results in a single snapshot object.

Live objects (and objects involved in the capturing traversal) must further be able to access the object metadata, so gaps of missing information can be filled with this metadata. For example, when one attribute of an object should be captured with a default value instead of the actual value.

There might be more than one way to capture one type of live object. For example, a `CompiledMethod` in Smalltalk could be captured through source code or as compiled byte code. To support choosing a different type of snapshot than the one the object prefers, it must be possible to specify this in the metadata associated with the object. Further, either the graph traversal or the live object itself has to respect the override.

3.4.3 Abstract algorithm to materialize object graphs. Converting snapshots to live objects can also be realized with a traversal of the snapshot object graph, starting from the snapshot of the start object. As during capturing, each object should only be rematerialized once.

When a snapshot is to be materialized, a message is sent to it to convert it back to its original live object. Depending on the kind of snapshot this may involve, for example, creating a new instance of the type of the captured object, or compiling source code. If the live object had itself replaced by another during capturing, the replacement object will be rematerialized instead. To get the original live object back, another message is sent to the materialized object, essentially telling it to "bring itself back to live". A symbolic reference to a globally accessible object would at this point resolve itself and return this global object. Live objects can also perform other post-materialization tasks, such as notifying observers of changes.

When snapshots are materialized, it is possible that a live object already exists (i.e. there is a live object with the same name). In this case, the snapshot object should be instructed to materialize itself *into* the existing live object if possible. This ensures that references to the live object do not become stale by materializing a new live object.

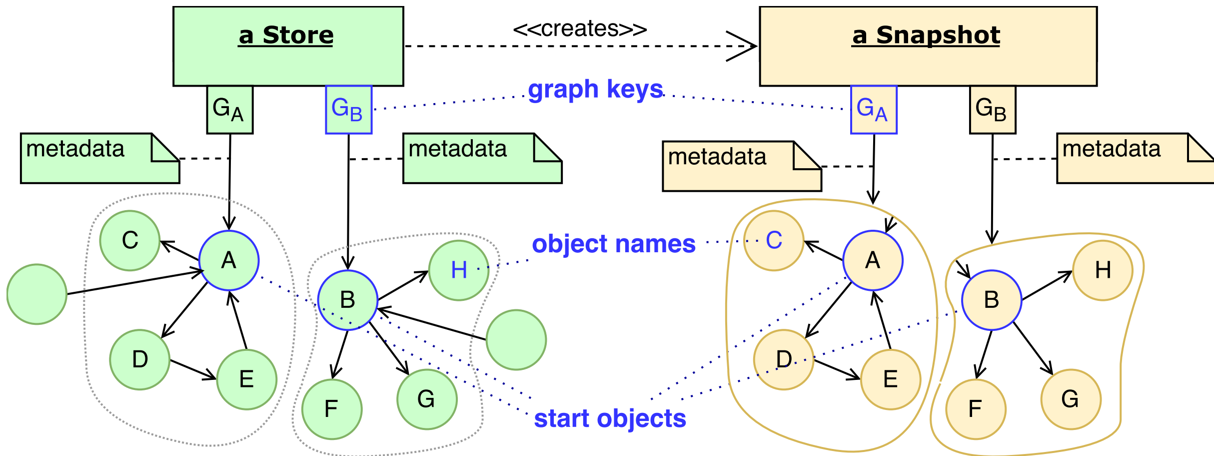

Figure 3: Example setting for object graphs, start objects, and object names. The store captures all objects that are reachable from its known start objects.

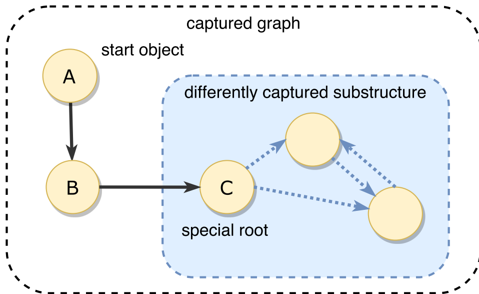

Figure 4: Different graph traversal strategy for a subgraph. Line styles indicate possibly very different relationships among the objects in the subgraph (e.g. pointers, naming conventions)

3.5 Differences between snapshots

After having described how objects can be converted between the live realm and the snapshot realm, in this section we motivate the need for differences between graphs, describe how they can be detected independent of a particular application domain, and how these differences can be applied to object graphs.

3.5.1 About the importance and granularity of differences. Differences are important, not only for the users to consume, but also for optimizing certain operations. For example, when a new version is to be saved, first computing the differences between the working copy and its parent version makes it possible to ignore all unchanged objects. The unchanged objects may not need to be serialized again and can be reused for caching of the newly created version. Assuming that only a small part of a system changes between versions, processing only the differences can mean that fewer objects must be processed overall.

Implementing differences requires additional implementation effort (in comparison to building a purely snapshot-based system). But the effort can be worth it for both the user experience and the performance of the version control system.

There are multiple levels at which differences can be computed.

(1) At the object container level: Which object graphs have changed?
(2) At the object graph level: Which objects have changed?
(3) At the object level: Which parts of an object have changed?

3.5.2 Abstract detection of differences. The specific structure of differences depends on the structure of the snapshots. However, the structure of collections of differences in an object graph and the structure of differences between object containers can be generalized. The general principle of detecting changes is described in this section.

An object graph is defined by its start object and contains a mapping between names and object snapshots. Thus, the differences between two object graphs can be expressed as the differences between the start objects plus the differences for any objects with the same name.

To collect the differences for the snapshots, we can perform a simultaneous graph traversal in the the graphs of snapshots that should be compared. One of the graphs is the *left*-side graph, containing snapshots "before" certain changes that should be detected, the second is the *right*-side graph, which captures the situation "after" the changes, and an optional third graph would be the *base* graph, which

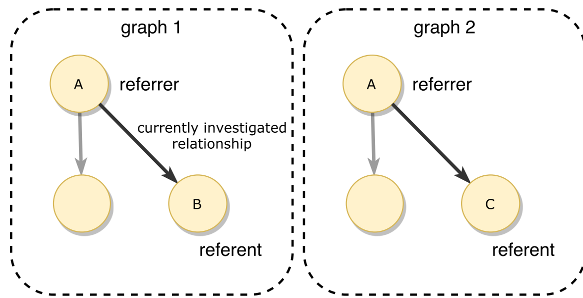

Figure 5: Two graphs being compared, currently inspecting a particular relationship from the two editions of the object named A. In graph 1, B is at the end of this relationship; in graph 2, it is a different object C. B from graph 1 and C from graph 2 are the referents in this relationship, while A from graph 1 and A from graph 2 are the referrers. The change in name from B to C means that the two captured editions of A relate to different objects.

contains snapshots from the base version of a merge, or more generally the base version of a three-way difference of graphs.

Assume two graphs that contain different snapshot contents for some names. Beginning from the snapshots of the start objects in each graph, two snapshots are compared in each step. The determination of the local differences (i.e. changes that apply to one object) is up to the implementation of the snapshots being compared because they know their structure best. The result of the comparison must be some kind of differences object. We assume that the comparison will somehow iterate over the relevant relationships of the snapshot object (the "referrer") and that the other ends of these relationships (the "referents") may need to be compared among the two graphs again (see Figure 5).

Depending on the names of the referents, three situations can arise:

First, if the referent snapshots have the same object name, then the referrer object is in relation with the same object in both graphs. This means that this relationship has not changed. The referents must then be compared in a subsequent traversal step to detect differences deeper in the graph.

Second, if the referents have different names, then the relationship has changed and, thus, a difference for the referrer exists. While users might be interested in the differences between these distinct referent objects, this comparison does not make sense in general, e.g. when two of the referents are of completely unrelated types.

Third, if one of the referents has no object name, meaning that the identity of the captured object is not tracked, further information is needed. If it is a snapshot of an immutable value object, then the difference applies to the referrer. If it is a snapshot of an object that should be mutated and that supports fine-granular differences in both graphs then there is no difference for the referrer.

If a three-way difference is computed and there are three different names for the snapshots in a set of referents (or two names and one nameless snapshot), then the relationship was changed both from the base to the left-side and from the base to the right-side, but to different objects. This is a conflict in the referrer and must be appropriately recorded.

When the referent from the right-side graph does not have a corresponding snapshot in the left-side graph, a new object has been introduced to the graph. This must be noted in the differences so it could be added to a graph to which these differences should be applied. If there is a difference for the referenced object, then it will be recorded under this name anyway. For the case when an existing object from the left-side graph can only be reached via the added object in the right-side graph, the relationships of the added object must be followed (see Figure 6). For each referent with a name that already exists in the left-side graph, the difference traversal must continue with these same-named snapshots from either side.

If a three-way difference is computed and a name can be found via this mechanism in the left-side and right-side graph, but not in the base graph, then the *same* object was added in both changes from the base. Any differences between the two sides are then automatically conflicts.

If the right-side graph does not contain an object from the left-side graph, that object has been removed from the graph. We can detect this by marking each name in the left-side graph which we have also seen in the right-side graph. All unmarked objects at the end of the traversal have been removed. Whether the removal should be noted in the differences depends on how the store to which these differences should be applied behaves. It does not need to be noted if simply removing references to the removed object is sufficient (e.g. when automatic garbage collection is used). It is also safe for uncaptured objects (outside of the graph) that refer to the object that has been "removed" from the captured graph. In case the store has to take care of the deletion itself (e.g. files on disk may not be deleted automatically), the removal of an object should be noted explicitly in the differences.

When one snapshot is told to compare itself to another, the snapshot implementation can decide to start an own graph traversal for differences.

3.5.3 Abstract application of differences. Applying differences relates to detecting differences like materialization relates to capturing. But since the differences between two snapshot graphs are made up of the collection of differences

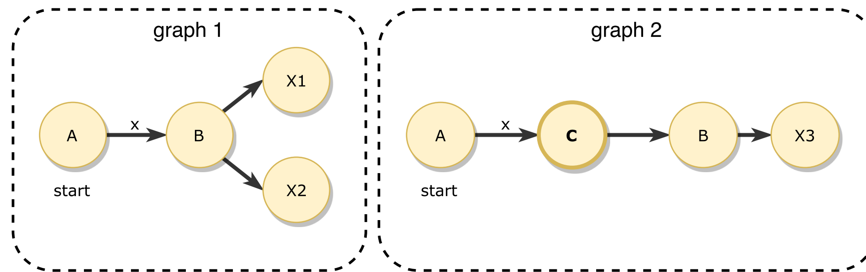

Figure 6: Two graphs being compared. In the right graph an object C has been added and it replaces B at the end of the reference x from A. B can only be reached from A via C in the right graph. The changes to B are "hidden" behind the added object. The difference detection graph traversal must therefore follow the relationships of C or it will not detect the changes in B.

to named objects (as defined above), no further object graph traversal is needed this time. Instead, the individual differences must simply be applied to their respective objects in the left-side graph.

Sometimes, objects appear in the right-side graph that do not exist in the left-side graph. In this case, these objects have been added and this addition must be reproduced when the differences are applied.

If a store on live objects implements the application of differences and new objects must be created, they must be materialized similar as described in subsubsection 3.4.3, but with a variation. If a materialized object refers to another named object, this referent object must be looked up in the target graph as usual. But if it already exists, it does not need to be materialized (which would be the case in the abstract materialization algorithm above). Instead, only the reference to this object must be materialized in the referrer, while the existing referent object is only subject to a change if there is an own difference for it.

3.6 Storing objects outside of the programming environment

Objects eventually have to leave the programming environment to the storage realm, in order to be shared with other programmers or authors.

When snapshots are exported from the programming environment, they must be converted into a representation that suits the target storage. Snapshots might have more than one form of representation. For example, a snapshot of a formatted text could be converted into Markdown or some specialized Extensible Markup Language (XML) format. For some objects, users may want to have control over the export format, but for others, they might not care.

We note that there can be a variety of storage strategies for each snapshot type. These strategies are represented through *serializers* and *deserializers*. Serializers convert graphs of snapshots into storage objects (e. g. files), and deserializers should do the inverse. Snapshot types should define a preferred serializer that is generally suitable for the objects they represent. For example, character string snapshot could refer to a serializer that outputs the text in Unicode-encoded text files.

The serializer used for a graph of objects must be recorded in the object metadata because users may have chosen a different serializer. Deserializers should be able to answer the question "Can you read the output of this serializer?", so a store can choose a suitable deserializer based on the information about the serializer.

Because the object metadata must be accessible before the correct deserializer is known, the format of the metadata must be determined by the store.

A store must therefore perform the following steps to serialize an object graph:

(1) Look up the serializer according to the metadata, if none is defined use the preferred serializer of the start object and add that information to the metadata.
(2) Instruct the serializer about the key of the object graph to be stored (the serializer may derive the storage location from the key).
(3) Invoke the serializer with the object graph.
(4) Write out the metadata to the storage medium.

The steps to deserialize storage objects to snapshots are:

(1) Read in the metadata from the storage medium.
(2) Based on the metadata about the serializer, look up a suitable deserializer.
(3) Instruct the deserializer about the location of the storage objects.
(4) Invoke the deserializer to obtain a graph of snapshots.

Finally, a store may need to find the storage locations of objects graphs in the first place. How it does that is basically implementation-defined, but a good strategy is to maintain a dictionary that connects graph keys with locations. The store would have to store this dictionary in a well-known location and format.

3.7 Generic snapshot format for objects

The previous sections described an abstract framework for version control of diverse types of objects. In this section, we describe how objects can be captured and compared when there are no specialized snapshots available for them. As a consequence, all objects should become trackable.

3.7.1 The structure of objects. Objects can be viewed as comprising a number of *slots*, as in Self or the Common Lisp Object System (CLOS) to denote "a component of an object that can store a value" [20]. There may be different types of slots, such as instance variables or unordered items of a collection. Slots can have an identifier, such as a variable name, symbol, or index, but they do not need to. However, it must be possible to look up a slot in an object. A slot can also refer to behavior, e. g. a method in case of Self or JavaScript.

The snapshot of an object is the collection of its captured slots, and its assigned name if its identity should be kept. This schema can be specialized to accommodate specia objects, such as primitive values. Slots could refer to other object snapshots directly or by name (if the referent has one). In programming languages with strong typing, we have to keep a reference to the type of the original live object — or its replacement. Thus, when the snapshot is rematerialized into a live object, the correct type can be instantiated. The default implementation of the capturing message should enumerate all slots in the live object and add slots of the appropriate type to the snapshot. Objects referenced by the captured slots also have to be traversed. When the snapshot slots are created, we have to consult the object metadata whether the slot should be captured at all or whether there is a default value.

To materialize an object snapshot, we create a new (uninitialized) instance of the type of the captured object (which is either the type of the original live object or the type of the replacement). Each slot must then be materialized into this object. What must be done to that object depends on the type of each slot. For example, a slot for an instance variable should assign its materialized value to the instance variable.

3.7.2 Differences in single objects. How objects can be mutated depends on the programming language. Commonly a slot value can be changed to a different object. In some languages, we can also add or remove slots. The differences between two object snapshots can therefore be described by the changes to the slots of the object (reassignments, additions, removals). If an object is replaced by another one everywhere in the system, there might also be the difference type "object replacement". But in most cases, it is sufficient to record that all slots referring to the "replaced" object have changed.

When we compare an object snapshot with another one (as described in subsubsection 3.5.2), we must match its slots with the slots of the other snapshot. If no matching slot can be found in the other snapshot, the slot was either added or removed. In both cases, this must be added to the collection of slot changes. If a matching slot is found, the two slots are compared. Should the slots be of a kind that references another snapshot, we determine the differences between the referents of the slots as described in subsubsection 3.5.2. Should it be determined that there is a local change to the referrer, which in this case means that a different object has been assigned to the slot, this reassignment is added to the collection of slot changes.

4 OBJECT VERSION CONTROL IN SQUEAK/SMALLTALK WITH GIT

A prototype implementation[4] of our approach has been realized in Squeak/Smalltalk and connects to Git as the backend VCS. The implementation is hosted on GitHub[5].

4.1 Object containers and object graphs

Capturing live objects is the responsibility of the `SquotImageStore`. Given an object and a key, it will traverse the object's graph for capturing, providing a bidirectional mapping between paths and start objects. For the prototype, the only backend VCS at the moment is Git, which is a file-based VCS, the keys of graphs are also the paths to the files or directories in which the graphs will be stored. Because graph keys need to be unique only within a store, the double role as paths is unproblematic. In addition, an image store also tracks metadata for each path, e.g., the desired serializers or deserializers, or arbitrary data.

For bookkeeping, the images store keeps an *object registry* that maps all known names to their live objects and a collection of `SquotObjectGraphs`, which in this case serve as local registries for object names. The *object registry* makes sure that object names are unique across the object graphs. UUIDs are used for names in this implementation.

Certain metadata are only relevant to the image store and not to be persisted. This *transient store info* is available to all objects that are being captured or materialized. Such information might include which instance variables of an object *not* to capture.

Capturing an image store's snapshot creates a `SquotSnapshot`. Snapshots of ordinary objects are called *shadows* in Squot. Each `SquotSnapshot` has a dictionary of `SquotArtifacts`. An artifact is the combination of an object graph, its key (or path), and the associated metadata. An

[4] Refer to Appendix B for supplemental information on the systems architecture.

[5] https://github.com/hpi-swa/Squot (last accessed June 10, 2019).

Listing 1: Default implementation `Object>>captureWithSquot:`

```
captureWithSquot: aCapturer
  | toCapture |
  self class isImmediateClass ifTrue: [^ aCapturer
        capturePrimitiveValue: self].
  self class isBits ifTrue: [^ aCapturer captureBits: self].
  toCapture := self squotReplacement: aCapturer.
  ^ aCapturer capture: toCapture as: toCapture squotShadowFactory
```

artifact is an element of an object container, which can be queried for `artifacts`. The result is a mapping from paths (as above) to `SquotArtifact`.

4.2 Generic object snapshots

Snapshots for generic objects, implemented in `SquotObjectShadow`, contain a collection of *slots*, the live object's class and the class of a potential replacement for the live object, if desired. Slot types are provided for the two kinds of Smalltalk object contents: instance variables (`SquotInstVarSlot`) and indexable variables (`SquotVariablePartSlot`). A slot is key-value–pair, whose key is the variables' name or index, respectively. The value is the shadow of the variables contents.

Certain kind of objects need special treatment. Objects with value semantics (i.e., `SmallInteger`, `Character`, and `SmallFloat64`) and certain well-known objects (`true`, `false`, and `nil`) are shadowed by `SquotPrimitiveValue`, a simple wrapper. This allows serialization by value or well-known name. `CompiledCode` is handled by a `SquotCompiledCodeShadow`, as its special layout (both indexable reference parts and indexable byte parts) does not fit the generic slot model. For efficiency, certain collections have their own shadows: indexable collections of well-known bit- or character-types (e.g. `ByteArray`, `WordArray`, `ByteString`, and `WideString`) are handled with a`SquotBitsObjectShadow` that wraps a copy of its contents rather than creating `SquotPrimitiveValue` wrapper for each slot.

4.3 Capturing

To start capturing live objects, a `SquotObjectCapturer` is handed to a start object (see Listing 1 for the default implementation). It can decide to be replaced (`squotReplacement:`), if necessary, and then hands the to-be-captured objects to the capturer. The capturer performs a *breadth-first* search to build a `SquotObjectGraph`. When new objects are encountered, the capturer (a) assigns a name to newly encountered objects, (b) registers them in the object graph and the snapshot's object registry and (c) recurses by sending `captureWithSquot:`. If necessary, the object can decide to be shadowed by something else than a `SquotObjectShadow` via `squotShadowFactory`.

The capturer then creates a new shadow and in a second step instructs it to initialize itself based on the live object. This separation allows cyclic object graphs. Generally, initialization comprises the enumeration of a live object's variables, adding them to the shadow, creating and populating slots, and queueing them for capturing.

4.4 Materialization

The `SquotShadowMaterializer` provides snapshot traversal for materialization in a recursive, depth-first manner. Given a start object shadow, the materialization starts in `rematerialize:` if the object shadow has not been materialized already. If a shadow has an object name, and a live object of that name exists, the shadow is materialized *in place* in the live object (cf. Listing 2). If this is not possible, a new live object with the shadows content will be created, which, when the object name is found, will *replace* the old one. During materialization, a shadow with slots will instruct the slot to materialize, recursively. The slots, in turn use the materializer to rematerialize their values and put them at the appropriate places in the live object. Similar to capture, the materialization is two-step, to allow for cyclic references to materialize correctly (cf. Listing 3).

4.5 Differences

The implementation supports differences at three levels: `SquotPatch` handles the object container level, `SquotObjectGraphDiff` the object graph level, and `SquotObjectDiff` an similar the object level (cf. subsubsection 3.5.1). The `SquotDiffBuilder` provides graph traversal for the differences computation, constructing a `SquotObjectGraphDiff`. In each step, a left-side shadow is compared with a right-side one, and optionally a base shadow. Their object names are compared as described in subsubsection 3.5.2. Cycles are dealt with by splitting the creation and initialization of difference objects (see Listing 4).

The multiple-path pruning in this traversal works by only checking that the left-side shadow has not been encountered yet. The rationale is that the changes to a single object do not depend on the path that led to it. The object must have changed in the same way wherever it is referenced. Anything else means that a former reference to this object was in fact changed to another object with a different name.

Analogous to capturing, the comparison of slot-based snapshots works by delegating the comparison to the contained slots, storing the result in a `SquotObjectDiff`. When the object names of the compared slot values are equal, no difference is recorded, but the referents are compared further. When the object names of the values of two compared slots are different or absent, a `SquotSlotReassignment` is recorded,

Listing 2: Materialization dispatch in `SquotShadowMaterializer`

```
1  findOrCreateAndRematerialize: aShadow
2    self sourceNameOf: aShadow ifPresent:
3      [:name | objectRegistry objectNamed: name ifPresent: [:existing |
4          existing squotShouldKeepIdentity ifTrue: [^ self materialize: aShadow in: existing]].
5    ^ aShadow squotMaterializeWith: self
6
7  materialize: aShadow in: anObject
8    | materializedObject |
9    convertedObjects at: aShadow undecorated put: anObject.
10   (anObject shouldHaveNameAssignedBySquot and: [sourceGraph notNil])
11     ifTrue: [objectRegistry nameOf: anObject ifAbsentAssign: [sourceGraph nameOf: aShadow]].
12   materializedObject := (aShadow materializeAs: anObject with: self) squotReactivateWith: self.
13   anObject becomeForward: materializedObject copyHash: false.
14   ^ anObject
```

Listing 3: `SquotObjectShadow` separates allocating new objects from filling slot values.

```
1  squotMaterializeWith: aMaterializer
2    | object |
3    object := self createMaterializedInstance.
4    ^ aMaterializer materialize: self in: object
```

Listing 4: The allocation of new difference objects for `SquotDiffBuilder` is separated from actually computing the differences.

```
1  diffFrom: leftShadow to: rightShadow ifNew: newBlock andInitialize:
       initBlock
2    | diff left |
3    left := leftShadow undecorated.
4    diff := objectDiffs
5      at: left
6      ifPresent: [:existing | ^ existing]
7      ifAbsentPut: newBlock.
8    initBlock cull: diff cull: leftShadow cull: rightShadow.
9    diff squotHasChanges ifTrue:
10     [graphDiff
11       addDiff: diff
12       forObjectNamed: (shadowNames at: left ifAbsent: []).
13     self noteChanged: left].
14   ^ diff
```

which remembers both values. Differences can be applied to shadows using a `SquotObjectPatcher`, which eventually sends `squotApplyTo:with:` to each difference object, which results in a modified object shadow: `SquotObjectDiffs` simply apply all their slot differences to the shadow, `SquotSlotReassignment` replace the slot's value with the remembered right-side shadow. Differences can be applied to a `SquotImageStore` to change live objects, re-using the materialization process with a patcher instead of a materializer.

4.6 Serialization and deserialization

For de/serialization of artifacts, Squot includes the `SquotFileSystemStore` that operates on a file directory. To aid deserialization, it stores object metadata and a table-of-contents as Smalltalk Object Notation (STON) [31] in dedicated files. Preferred serializers and deserializers can be set in an artifact's metadata. Since Squot provides general purpose shadow objects, it also provides general purpose serializers for them: A binary serializer based on Squeak's `SmartRefStream` and a serializer based on STON.

5 DISCUSSION

In this section, Squot and the architecture from section 3 are evaluated with regard to the goals described in section 2. Observations, issues and possible improvements are discussed.

5.1 Squot in practice

First of all, the architecture is in actual use through the Squot and Squit packages. The implementation as described in section 4 is capable of tracking, serializing, and restoring complex object graphs such as a workspace morph with variable bindings (see Listing 5). The example involves different kinds of objects and circular dependencies in the object graph, as well as metadata that influences the capturing. Further, the created file system structure can be managed through a file-based VCS. In the case of Squot, Git is used through the Squit Git connector component.

Further, we have created specialized snapshot and serialization formats for Squeak packages, hashed collections, and string objects. These specializations show that:

- the general (slot-based) approach to object snapshots is extensible to support types with special requirements,
- domain objects can opt out of the general approach and implement their own,
- existing classes for snapshots and differences (e.g. those of Monticello) can be reused and adapted for Squot, and
- the format of snapshots and the format of the generated files can be customized separately — to use a different file format, it is not always required to change the kind of snapshot used for an object.

Listing 5: A scenario in form of a test case which illustrates how Squot can manage the complex object graph of a graphical Squeak workspace.

```
1  workspace := Workspace new. systemWindow := workspace openLabel: 'Workspace'.
2  bindings := Dictionary new. bindings at: #This put: workspace.
3  workspace setBindings: bindings. workspace contents: 'This is a sample.'.
4  "store"
5  imageStore add: systemWindow at: 'sample workspace.morph'.
6  fileStore restoreToSnapshot: imageStore captureSnapshot.
7  "restore in different image store"
8  workspaceArtifact := fileStore captureSnapshot artifactAt: 'sample workspace.morph'.
9  restoredWindow := SquotWorkingCopy newOnObjectMemory
10 loadArtifact: workspaceArtifact; objectAt: workspaceArtifact path.
11 self assert: workspace contents equals: restoredWindow model contents.
12 self assert: restoredWindow model ~~ workspace description: 'new workspace from materialization'.
13 self assert: (restoredWindow model bindingOf: #This) value == restoredWindow model
```

5.2 Limitations

The following limitations span from limitations arising from the current implementation to conceptual limitations which require further research.

5.2.1 State migration. One problem associated with changing the structure of classes — such as adding or renaming instance variables — is migrating existing objects to the new schema. Any migrations that users apply to tracked objects in own environment will also be applied in other users' environments. These objects have been captured after receiving the treatment of the migration, after all. But objects that are not tracked with Squot cannot benefit from that. Therefore, it is still advisable to take other measures to migrate existing objects, such as migration code in pre- and post-load scripts of packages.

5.2.2 Merging objects. While the merging of packages has already been simplified in *Squot*, merging for arbitrary objects (with slot-based shadows) is not implemented yet. The reason is that further research must be conducted first on *how* objects can be merged correctly. In particular some basic assumptions from line-based merging via three-way differences do not hold in object graphs. This problem is out of scope for this work.

5.2.3 Sharing objects between graphs. One potential problem with the current implementation is that objects cannot be shared among multiple object graphs. If one object is referenced in multiple graphs, these graphs will overlap and create redundant snapshots (and storage data) for the objects reachable from the common object. At least the redundant snapshots will have equal names due to the shared object registry in an image store. Shared objects would be "rejoined" when they are materialized. But it could also lead to inconsistencies when changes to common objects are committed for only one of the involved graphs. So far users have to ensure that graphs do not overlap.

5.2.4 Re-integrating materialized objects. If multiple parts of the system are tracked in separate object graphs, it could be confusing where they are composed again in the target environment and and how this composition configuration should be tracked. More generaly speaking it is unclear *where* new live objects should be put when they are materialized from a snapshot. Materialized objects are added to an image store, which is probably not part of the software or document under development. The materialized objects might need to be assigned to some class variable or associated with content holders, such as graphical tools. Other approaches could be *places* and *generalized references* as found in Common Lisp [20]).

5.2.5 Partial materialization. Another concern is that of materializing only parts of a version. Currently, Squot assumes that an object graph should be deleted (i.e. untracked) if it is absent in a working copy's store, but present in the parent version. If a repository contains packages for different platforms, so not all of them can be loaded at the same time, support for unmaterialized object graphs without signifying their deletion becomes a requirement. It could be necessary to make the untracking of object graphs explicit in the state of a store to achieve this.

6 RELATED WORK

Version control has been investigate previously for both source code and object contents. Moreover, moving objects between systems as well as combining contents of different objects have their share in literature.

Version control approaches — code-centric. Pur [10] is a version control framework that abstracts from common VCS with a state-based history model and directed acyclic graphs as version history. Squot is a partial implementation of the Pur concepts. Image and file store, working copy and snapshot types make up the Pur frontend implementation in Squot.

Most Smalltalk systems (VA Smalltalk: *ENVY/Developer* [19], VisualWorks: *Store* [2], Dolphin *source tracking system* [25], Squeak and Pharo: *Monticello* [18], among others) provide specialized version control systems for their meta-objects, usually classes and methods. They do not support tracking arbitrary objects or only in a fixed serialization format. *Orwell* [29] is a Smalltalk VCS and configuration management system that supports versioning of methods, classes, applications and configurations, all of which are stored in a single object database. It does not support tracking arbitrary objects. *Iceberg* [6] is source code VCS tool for Pharo/Smalltalk to interact with Git and GitHub.

Version control approaches—object-centric. CoVer [5] adds version control to a collaborative hypermedia editing system with asynchronous editing. All versions of an object are combined into a *multi-state object* to track its identity. Since arbitrary objects can be mapped to hypertext [5], they can be managed using *CoVer*, collaboratively. Squot does not include any model for collaboration. *COOP/Orm* [14, 15] provides fine-grained version control and configuration management for sets of documents. Each document comprises a tree (as opposed to Squot's general graphs) that might contain classes, methods, or textual paragraphs, among other. Versions are created for documents and multiple versions of the same document can occur in a configuration. In Squot, only one version of an object can be checked out at the same time. *HistOOry* [21] is an object versioning system to record the state of selected fields of objects. It is provided as a language extension for Squeak and Pharo and, hence, does not integrate with external-storage VCSs. *HistOOry* can create views on the snapshots of an object that are polymorphic with the live object, which is not possible in Squot yet. *CoExist* [27, 26] provides continuous object versioning with new snapshots being created automatically whenever a program is changed. To achieve short response times a state-based approach for methods and classes is used. It does not integrate with external-storage VCS, either The *Lively Kernel* [28] includes object versioning facilities: any object is aware of its previous versions, an global queries allow to identify following versions of objects. *Lively Kernel* includes a *parts bin* of reusable graphical objects [12], which is versioned using Subversion. *Simulink* [4] is a programming environment that does not primarily involve source code, rather, hierarchical blocks are used to model simulation systems. The VCS integration bears similarities to Squot's architecture: different VCS backends [24, 23] and different kinds of objects that can be tracked and merged [16].

Tracking and transportation of objects. Vegdahl [32] describes the the challenges of moving objects between Smalltalk systems, including solutions to circular and symbolic references. The essence of this approach can be found in the standard serializer of Squeak as well as the more recent STON [31]: during serialization, objects are assigned unique identifiers which are used as references. Only the first occurrence of an object will serialize its contents. Certain objects, such as classes, are always serialized symbolically. *Self* uses the *Transporter* [30] to capture objects and transforms them into Self expressions that recreate the captured objects. In that course, annotations to the objects are use to preserve metadata such as categorization. The Transporter has no VCS capabilities. VisualWorks *Parcels* [17] are a fast binary deployment mechanism for objects and source. It supports circular dependencies among and partial loading of parcels. Similarly, *Fuel* [3] is a fast binary object serializer for Pharo and Squeak in the vein of parcels, but employs object clustering for efficiency. Fuel's object graph analysis is similar to capturing in Squot.

Object merging. Operation-based and *state-based* merging of objects emerged in the setting of asynchronous collaborative editing of graphical objects [7]. Operation-based merging has advantages in solving conflicts that would otherwise require manual resolution and is more efficient for large object sets. Nevertheless, Squot uses state-based merging because Squeak does not provide practical means tocapture all operations on arbitrary objects. Further, most contemporary VCSs assume a state-based approach, too. Despite that, for efficiency, Squot employs parts of the operation-based approaches by computing the differences between snapshots first and applying those instead of processing whole snapshots. For XML documents, a three-way merge approach has been documented [13]. The approach might be applicable to Squot, as XML documents are essentially "ordered trees with labeled nodes", which matches to Squot snapshots.

7 CONCLUSION AND OUTLOOK

We set out to devise a solution to the problem that, in exploratory programming environments that are built around objects, regular (non-code) objects are precluded from specialized version control solutions. Existing version control technology should be reused to ease collaboration. Ultimately, version control for objects should be as practical and accessible as it is for files today.

With the architecture presented, a first step towards that goal has been made. It provides a framework for modeling and comparing editions of objects. A generic solution to track arbitrary kinds of objects is provided, so the effort of versioning new objects is kept low. When a specialized way to handle editions of an object is needed, the architecture provides variation points, so that custom software can supply their own formats of snapshots and serializations. The prototype implementation *Squot* proves that the architecture

is functional and that the status-quo on object version control could be improved in Squeak/Smalltalk. By building on the ideas of Pur [9], the architecture should be portable to various backend version control systems.

The journey towards making version control for objects feel as natural as it has become for files has not come to an end yet. Besides the technical challenges illustrated in section 5, conceptual challenges remain. To provide a fully-functional version control experience for objects, merging facilities for arbitrary objects must still be implemented. In general, graphical tools would support working with snapshots and differences. To validate the generality of the proposed architecture, an implementation in another programming environment should be undertaken. While *Lively Kernel* already supports object versioning, it could be examined how our solution can extend their approach.

REFERENCES

[1] Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, and Jens Lincke. 2013. Explorative authoring of active web content in a mobile environment. In *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*. Number 72. Universitätsverlag Potsdam. ISBN: 978-3-86956-232-2. http://pub.ub.uni-potsdam.de/volltexte/2013/6405/.

[2] [n. d.] Cincom visualworks store repository. Retrieved 2017-09-02 from http://www.cincomsmalltalk.com/main/developer-community/store-repository/.

[3] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. 2014. Fuel: a fast general purpose object graph serializer. *Software: Practice and Experience*, 44, 4, 433–453.

[4] A. C. W. Grace. 1991. Simulab, an integrated environment for simulation and control. In *1991 American Control Conference*. (June 1991), 1015–1020. DOI: 10.23919/ACC.1991.4791532.

[5] Anja Haake and Jörg M. Haake. 1993. Take cover: exploiting version support in cooperative systems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (CHI '93). ACM, Amsterdam, The Netherlands, 406–413. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169320. http://doi.acm.org/10.1145/169059.169320.

[6] [n. d.] Iceberg. Retrieved 2017-09-21 from https://github.com/pharo-vcs/iceberg/.

[7] Claudia-Lavinia Ignat and Moira C Norrie. 2004. Operation-based versus state-based merging in asynchronous graphical collaborative editing. In *Proc. 6th International Workshop on Collaborative Editing Systems, Chicago*.

[8] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The lively kernel a self-supporting system on a web page. In *Self-Sustaining Systems*. Springer, 31–50.

[9] Matthias Kleine, Robert Hirschfeld, and Gilad Bracha. 2012. An abstraction for version control systems. In *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universtität Potsdam*. Number 54. Universtitätsverlag Potsdam. ISBN: 978-3-86956-158-5. http://pub.ub.uni-potsdam.de/volltexte/2012/5562/.

[10] Matthias Kleine, Robert Hirschfeld, and Gilad Bracha. 2012. An abstraction for version control systems. Technical report 54. Universität Potsdam, Potsdam, Germany, (March 21, 2012).

[11] Jens Lincke. 2014. *Evolving Tools in a Collaborative Self-supporting Development Environment*. PhD thesis. Hasso-Plattner-Institut, Universität Potsdam.

[12] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. 2012. The lively partsbin–a cloud-based repository for collaborative development of active web content. In *System Science (HICSS), 2012 45th Hawaii International Conference on*. IEEE, 693–701.

[13] Tancred Lindholm. 2004. A three-way merge for xml documents. In *Proceedings of the 2004 ACM symposium on Document engineering*. ACM, 1–10.

[14] Boris Magnusson and Ulf Asklund. 1996. Fine grained version control of configurations in coop/orm. *Software Configuration Management*, 31–48.

[15] Boris Magnusson, Sten Minör, and Ulf Asklund. 1993. Fine-grained revision control for collaborative software development. In *In Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering, Los Angeles CA*, 7–10.

[16] [n. d.] Merge simulink models from the comparison report - matlab & simulink. Retrieved 2017-09-09 from https://www.mathworks.com/help/rptgenext/ug/merge-simulink-models-from-the-comparison-report.html.

[17] Eliot Miranda, David Leibs, and Roel Wuyts. 2005. Parcels: a fast and feature-rich binary deployment technology. *Computer languages, systems & structures*, 31, 3, 165–181.

[18] [n. d.] Monticello. Retrieved 2017-09-26 from http://wiki.squeak.org/squeak/1287.

[19] Joseph Pelrine, Alan Knight, and Adrian Cho. 2001. *Mastering ENVY/Developer*. Cambridge University Press, New York, NY, USA. ISBN: 0-521-66650-3.

[20] Kent Pitman, editor. 2005. Common lisp hyperspec. http://www.lispworks.com/documentation/HyperSpec/Front/index.htm.

[21] Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. 2009. Executing code in the past: efficient in-memory object graph versioning. In *ACM SIGPLAN Notices* number 10. Volume 44. ACM, 391–408.

[22] Marc J Rochkind. 1975. The source code control system. *IEEE Transactions on Software Engineering*, 4, 364–370.

[23] [n. d.] Set up git source control - matlab & simulink. Retrieved 2017-09-09 from https://www.mathworks.com/help/simulink/ug/set-up-git-source-control.html.

[24] [n. d.] Set up svn source control - matlab & simulink. Retrieved 2017-09-09 from https://www.mathworks.com/help/simulink/ug/set-up-svn-source-control.html.

[25] [n. d.] Source tracking system. Retrieved 2017-09-02 from http://www.object-arts.com/downloads/docs/index.html?source_tracking_system.htm.

[26] Bastian Steinert. 2014. *Built-in recovery support for explorative programming*. PhD thesis. Universität Potsdam.

[27] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. 2012. Coexist: overcoming aversion to change. In *ACM SIGPLAN Notices* number 2. Volume 48. ACM, 107–118.

[28] Bastian Steinert, Lauritz Thamsen, Tim Felgentreff, and Robert Hirschfeld. 2014. Object versioning to support recovery needs: using proxies to preserve previous development states in lively. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (DLS '14). ACM, Portland, Oregon, USA, 113–124. ISBN: 978-1-4503-3211-8. DOI: 10.1145/2661088.2661093. http://doi.acm.org/10.1145/2661088.2661093.

[29] Dave Thomas and Kent Johnson. 1988. Orwell—a configuration management system for team programming. In *ACM SIGPLAN Notices* number 11. Volume 23. ACM, 135–141.

[30] David Ungar. 1995. Annotating objects for transport to other worlds. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA '95). ACM, Austin, Texas, USA, 73–87. ISBN: 0-89791-703-0. DOI: 10.1145/217838.217845. http://doi.acm.org/10.1145/217838.217845.

[31] Sven Van Caekenberghe, Stéphane Ducasse, and Johan Fabry. [n. d.] Ston: a smalltalk object notation. Retrieved 2017-08-17 from https://ci.inria.fr/pharo-contribution/job/EnterprisePharoBook/lastSuccessfulBuild/artifact/book-result/STON/STON.html.

[32] Steven R. Vegdahl. 1986. Moving structures between smalltalk images. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (OOPSLA '86). ACM, Portland, Oregon, USA, 466–471. ISBN: 0-89791-204-7. DOI: 10.1145/28697.28745.

A TERMS

For an overview of selected terms associated with the lifecycle of objects under version control, have a look at Figure 7.

The terms used in this work (with some deviations in section 4, which describes the prototype implementation of the proposed design), are as follows.

live object An object that would exist even without any support for support for version control in the programming environment.

snapshot object An object that represents a live object for the purpose of version control.

to capture an object Convert a live object to a snapshot object.

to materialize an object Convert a snapshot object to a live object. This may have side effects on other live objects, as noted earlier in this section.

tracked object A live object that is currently considered for version control and that can be captured at some point.

captured object Usually a live object that has been captured.

to serialize an object Convert an object to a series of bytes, for storing the object to a stream, which can end up in a file.

to deserialize an object Convert a series of bytes that was generated by serializing an object back to an object.

version (without a referent noun) An object that describes a set of object graphs at some point in time, with metadata such as the author who created this version. Such versions form the version history in a repository. The Git equivalent would be a commit.

version (of an object) An object as present in a *version* as defined above.

edition (of an object) An object in a state that needs not necessarily be present in any version. It could be a live object with changes not persisted in a version, or a modified snapshot object that was derived from applying only *some* of the differences between two versions to a version of an object. The set of versions of an object is always a subset of the set of editions of an object.

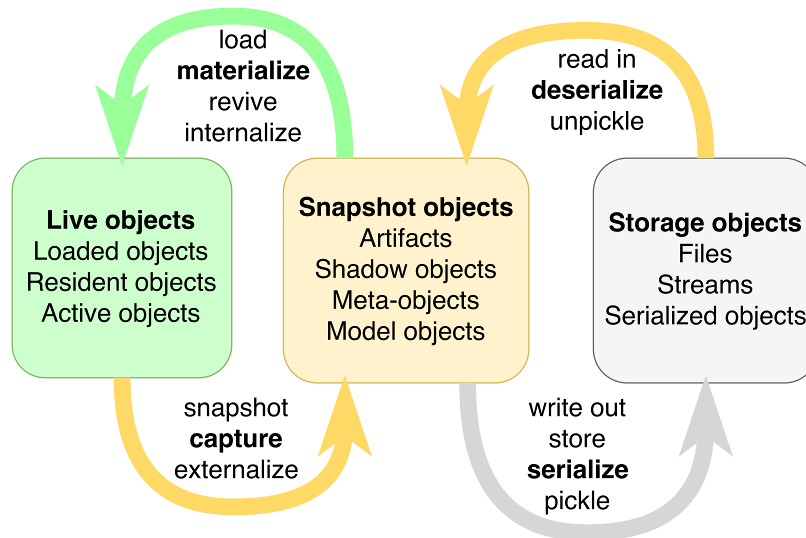

Figure 7: Common terms to describe operations of converting objects between different representations. Adopted variants in bold.

to apply differences to an object Transform an object from one edition into another edition of itself. Synonym: *to patch* an object.

merge The operation of combining three editions of each object in a set of objects into one edition, and the result of that operation.

The color code used in Figure 7 will also be used for other figures in this report when applicable: live objects are green, snapshot objects are yellow, and objects that result from serialization or are involved in that process are gray.

B SYSTEM ARCHITECTURE SUPPLEMENT

This section supplements the implementation description in section 4.

The relationships between the implementation entities of the Squot prototype as described in subsection 4.1 are depicted by the diagram in Figure 8.

The relationships between the capturing-related classes and the difference-related classes as described in subsection 4.5 are depicted by the diagram in Figure 8.

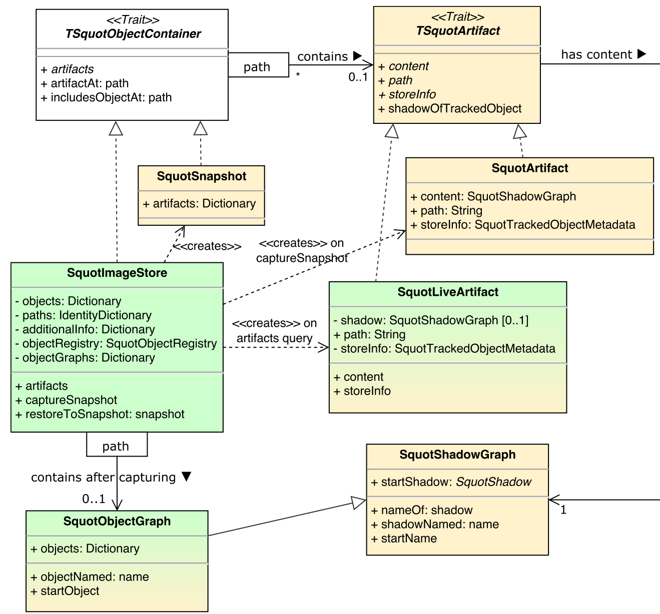

Figure 8: Relationships among object containers, SquotSnapshot, SquotImageStoree, artifacts, and graphs of shadow objects (snapshot objects). Key: UML

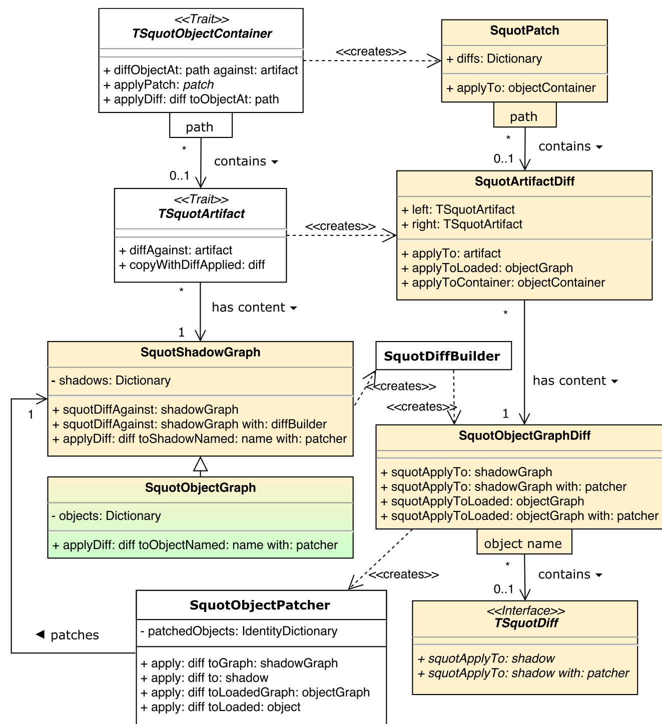

Figure 9: Relationships among object containers, artifacts, object graphs, and their respective difference classes. Key: UML