Technical Report No. 242

The Relation of Version Control to Concurrent Programming

(Extended Version)

Annette Bieniusa Peter Thiemann Stefan Wehr

Universität Freiburg, Germany

{bieniusa,thiemann,wehr}@informatik.uni-freiburg.de

Abstract

Version control helps coordinating a group of people working concurrently to achieve a shared objective.
Concurrency control helps coordinating a group of threads working concurrently to achieve a shared objective.
The seemingly superficial analogy between version control and concurrency control is deeper than expected. A comparison of three major flavors of version control systems (exemplified by RCS, Subversion, and Darcs) with three influential and representative approaches to concurrency control (monitors, STM, and message passing) exhibits a surprisingly close correspondences in terms of mechanism and workflow. The correspondence yields new perspectives on both, version control and concurrency control.

Keywords version control, monitors, message passing, software transactional memory

1. Introduction

Version control is a mature field of software engineering that traces its origins back to the legendary 1968 NATO conference [27]. It is one of the few fields that have attracted academic and industrial research alike and which has a multi-million dollar market [10]. Besides commercial systems, there are numerous open source systems that implement new approaches as well as new combinations of existing concepts.

Concurrent programming has an even longer and fickle history. There are numerous approaches some of which date back to the 1960s. Each of these approaches has advantages and drawbacks. Much recent work in this area concentrates on software transactional memory [12, 19, 21, 35, 42]. This approach to concurrency transfers ideas from database transactions [4] to the problem of synchronizing concurrent modification of shared datastructures. It is seen by some as an alternative API to concurrent datastructures that is particularly easy to master from a programmer perspective.

Grossman [17] suggests an inspiring analogy between transactional memory and garbage collection. His essay examines several key problems and their solutions in either area and puts them in a common perspective by presenting them with textual templates using different keyword substitutions. It then goes on to elaborate further points and develop some projections.

The present work suggests another analogy of transactional memory with a certain flavor of version control.[1] It further extends this analogy to the other main flavors of version control and compares them with monitors and message passing. As with Grossman's analogy there is some potential of transferring ideas from version control to concurrency paradigms and back. Analogies like these are valuable because they enable alternative views on old and new ideas. They clarify the design space, facilitate the classification of different approaches, and enable the anticipation of new approaches.

This paper is structured as follows. Section 2 presents a first cut at the correspondence by comparing the workflow of Subversion (a popular version control system) with the workflow of an implementation of software transactional memory. This section works entirely on an informal level and appeals to the reader's intuitive understanding of version control terms and software transactional terms. Subsequently, Section 3 gives an overview of basic terms and definitions on version control as used in this paper. Section 4 does the same for concurrent programming paradigms. Thus equipped, Section 5 extends the analogy to two other version control approaches and compares them with monitors and message passing. Section 6 collects further observations about the relationship between version control and concurrent programming in general. Section 7 concludes.

2. The Relation of Subversion to Software Transactional Memory

Toru Inemuri and Miki Kuruma work in the Tokyo office of MetaHacks, a software company that operates on a global scale. Toru is a project manager and Miki is a developer in Toru's project. Sometimes, Toru and Miki go out together to have dinner and to enjoy some Karaoke. Because they work

[1] We do not consider the exact relationship between database transactions and version control or software transactional memory in this paper.

Toru: Developing a Project

Toru is responsible for a *project*, which involves multiple *developers*. The main resource he is worried about is a *repository* of *artifacts* (source code, configuration files, documentation, reports, ...) created by the developers. This repository is shared among the developers that contribute to the project. Each of the developers needs to be able to manipulate artifacts. A developer may access or modify an artifact, he may create a new artifact, or delete an existing one. Clearly, this is a job for a *version control* system.

Toru considers using *Subversion* for the new project. He contemplates the workflow of a developer that wants to achieve a certain objective (implement a feature X, document feature Y, file a report).

1. No developer is allowed to operate directly on artifacts in the shared repository.
2. The developer performs a *check-out* operation to make the *head revision* of the shared artifacts available for reading and writing. Thereby, the check-out operation copies the artifacts into the developer's *workspace*.
3. The developer creates, reads, updates, and deletes artifacts as required to achieve the objective.
4. The developer might find out that his chosen approach does not work with the current repository. In this case, he performs a *revert* operation and restarts from the previous check out.
5. The developer might come to the conclusion that the objective cannot be achieved, in which case he *abandons* his modifications and proceeds with another task.
6. Once the developer has achieved his objective, he attempts to integrate his modifications into the shared repository by performing a *commit* operation.
7. On receiving the commit command, Subversion attempts to merge the modifications with the head revision of the artifacts. The merge operation is successful if no other developer has changed an artifact in a way that overlaps with changes made by the developer. In this case, the head revision of the artifacts is atomically updated to reflect the changes made by the developer and he can proceed to the next task.

   Otherwise Subversion signals a *conflict*. In this case, the developer reiterates all steps starting from the check-out operation.

Miki: Developing a Component

Miki is responsible for a *component*, which involves multiple *threads*. The main resource she is worried about is the *state* of the *variables* created by the threads. This state is shared among the threads that run inside the component. Each of the threads needs to be able to manipulate variables. A thread may access or modify a variable, it may create a new variable, or delete an existing one. Clearly, this is a job for a *concurrency control* system.

Miki considers using *STM* for the new component. She contemplates the workflow of a thread that wants to achieve a certain objective (fill a buffer with data, update a graph structure, write to a log file).

1. No thread is allowed to operate directly on variables in the shared program state.
2. The thread performs an *enter atomic block* operation to make the *current values* of the shared variables available for reading and writing. Intuitively, the enter atomic block operation creates a *local copy* of the variables.
3. The thread creates, reads, updates, and deletes variables as required to achieve the objective.
4. The thread might find out that its chosen approach does not work with the current program state. In this case, it performs a *retry* operation and restarts from the previous enter atomic block.
5. The thread might come to the conclusion that the objective cannot be achieved, in which case it *aborts* its modifications and proceeds with another task.
6. Once the thread has achieved its objective, it attempts to integrate its modifications into the shared program state by performing a *commit* operation.
7. On receiving the commit command, the STM implementation attempts to merge the modifications with the current values of the variables. The merge operation is successful if no other thread has changed a variable in a way that overlaps with changes made by the thread. In this case, the current values of the variables are atomically updated to reflect the changes made by the thread and it can proceed to the next task.

   Otherwise the STM implementation signals a *conflict*. In this case, the thread reiterates all steps starting from the enter atomic block operation.

Figure 1. Toru and Miki's views on program development.

Toru	Miki
project	component
developer	thread
version control	concurrency control
repository	program state
artifact	variable
head revision	current value
Subversion	STM implementation
workspace	local copy
check out	enter atomic block
commit	commit
abandon	abort
revert	retry
conflict	conflict

Figure 2. Corresponding terms in Toru's and Miki's tales.

on the same project, part of the dinner conversation revolves around work.

On one such occasion Toru has to select a version control (VC) system for a new project, whereas Miki tries to come to grips with software transactional memory (STM), a new paradigm for concurrent programming that she recently read about and wants to use in her next project. Toru is most familiar with the open-source VC systems CVS [7, 41] and Subversion [29, 38] and hence heavily influenced by their approach. Miki has read a number of papers on STM and is currently excited about a lightweight approach for Java [18]. Figure 1 contains condensed renditions of Toru's and Miki's contributions to that conversation and Figure 2 contains a table that lists the corresponding terms in their tales.

The conversation demonstrates that a concurrent thread using STM has a workflow which is very similar to the workflow of a developer who manages a project using a VC system similar to Subversion. There are a few dissimilarities, though.

- A developer working with a VC system has a private workspace with local copies of the artifacts created by a check-out operation. The private workspace enables the developer to perform experiments as well as to build and test the system without affecting anyone else.

  Some implementations of STM follow a similar strategy of keeping local copies of the variables changed in an atomic block. However, these copies do not outlive the atomic block in which they were created. Furthermore, there is no wholesale creation of local copies of all shared variables. Rather, these copies are created as needed.

  In a corresponding VC usage pattern, a developer would check out artifacts lazily on demand. However, this pattern does not appear to be useful to a developer because building and testing involves more artifacts than just the modified ones. Still, often the developer need not check out the entire repository.
- A developer populates the workspace with one initial check-out operation and then keeps the workspace up-to-date by performing *update* operations. Modulo local changes that the developer might want to preserve, the effect of an update operation is equivalent to clearing the workspace and then performing a fresh check out.
- If the VC system detects a conflict on an attempt to commit, then the developer has to take action to get an up-to-date view of the artifacts and to adapt its changes to the new situation. If STM fails to commit, then the STM implementation automatically makes the changes of other threads visible and reruns the body of the atomic block.
- A VC system records the history of each artifact in the repository, whereas STM keeps only the most current value of a shared variable.

3. Version Control

Version control (VC) systems manage multiple versions of the same artifact.[2] Software projects typically use a VC system to manage source code, configuration files, documentation, and so on. VC is not limited to software projects, but it is also used in such diverse areas as word processors, wiki applications, or content management systems.

Two features are the cornerstones of a VC system:

Accessibility The system allows access to all versions of an artifact at any time.

Accountability The system automatically logs who made a change to an artifact, when it was made, and what it was.

These features can be traced back already to one of the first VC systems, SCCS [32]. They are standard in all modern systems. VC systems can be further classified according to the following dimensions [10]:

Repository model How does the VC system store artifacts?

The system may store all artifacts in a *central repository*.

The system may rely on several *distributed repositories*. With this approach, each user owns one or more local repositories.

Concurrency model How does the VC system prevent conflicting changes to the same artifact?

The system may pessimistically use *locking* to prevent concurrent edit operations.

The system may optimistically allow concurrent edit operations and attempt to *merge* the resulting changes. If merging is not possible, the system signals a conflict that the user must resolve manually.

[2] Some people use the term "revision control" instead of "version control". We prefer the latter because an "revision" is a special kind of "version", as explained in the following.

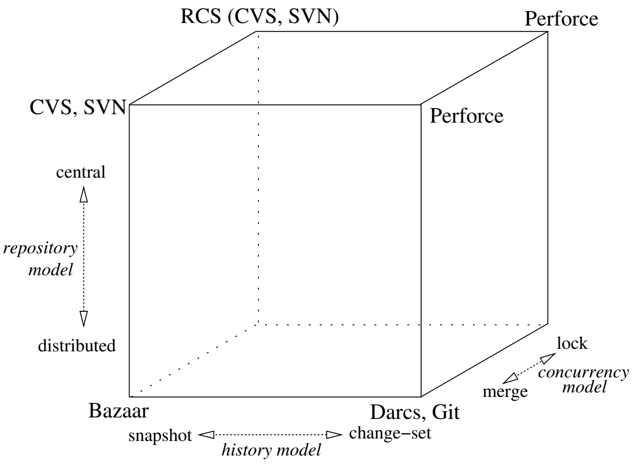

Figure 3. The VC cube.

History model How does the VC system record changes between different versions of an artifact?

The system may record different versions as a *snapshot* and use a *version graph* to represent several relations between them: the *revision-of* relation records sequential evolution of an artifact, the *variant-of* relation records parallel evolution of an artifact, and the *merge* relation records the combination of changes made in several variants of an artifact.

The system may record the differences between two versions of the same artifact as a *change set*. Users may then combine change sets in ways that go beyond the relations captured in a version graph.[3]

Figure 3 positions some existing VC systems with respect to these three dimensions. The list of VC systems is by no means exhaustive. The figure merely serves to indicate which combinations are implemented and used in practice. Apparently, there is no VC system supporting "distributed repositories" with "locking". In fact, it seems rather strange to insist on acquiring a global lock for editing artifacts in a local repository.

Unfortunately, there is no established common terminology in VC research and practice. Thus, the rest of this section introduces the terminology used in this paper for concepts common to many VC systems.

Repository A *repository* contains all information necessary to ensure accessibility and accountability for all artifacts stored in the system.

Workspace To modify artifacts stored in a repository, users copy specific versions of the artifacts into a local *workspace*. (This step is often implicit with distributed repositories.)

The workspace serves as a sandbox where artifacts can be created, modified, and deleted without changing the content of the repository. It is up to the user to synchronize between workspace and repository.

History The *history* is part of a repository. It records the meta information associated with changes to artifacts (who made the change, when it was, what it was, etc).

Branch A *branch* is a designated group of versions of some artifacts that evolve in parallel to some other group of versions based on the same artifacts.

Baseline A *baseline* is a designated group of versions from which new branches may be started.

Head revision The *head revision* denotes the most recent version of an artifact.

Conflict In most cases, a VC system uses a *merge algorithm* [5,13,23,31,43] to combine two different versions of an artifact. If the algorithm fails, the system signals a *conflict* and prompts the user for intervention.

Commit The *commit* operation propagates changes from the workspace to the repository. The changes may be specified with different *granularity*: per line, per group of lines, per file, per group of files. A commit is *atomic* if either all or none of the changes in the workspace are transferred to the repository.

Update The *update* operation propagates changes from the central repository to the workspace. For VC systems with distributed repositories, the notion of update is inapplicable, although the *pull* operation (explained next) is similar to an update.

Pull and push The *pull* operation propagates changes from a remote repository to the local one. The *push* operation propagates changes from the local repository to a remote one. The notion of pull and push is applicable only to VC systems with distributed repositories.

Add The *add* operation adds a file to the workspace.

Delete The *delete* operation removes a file from the workspace.

Revert The *revert* operation undoes all changes made inside the workspace, so that the workspace is up-to-date with a specific group of versions of the corresponding repository.

Pre/post event hooks VC systems often provide *hooks* that are invoked before or after a specific event. For example, after completion of a commit operation, a post-commit hook may be used to notify all users via email.

4. Concurrent Programming

Concurrent programming [3, 11] is the art of controlling (pseudo-) simultaneous execution of multiple interacting computations. The primary objective for using this programming technique is to increase the application throughput and

[3] VC systems with a snapshot-based history model often use delta mechanisms [24] to record only the differences between consecutive snapshots. Despite some similarities with change sets, the key difference is that change sets are accessible by the users of the VC system, whereas delta mechanisms are an implementation detail.

to use available hardware resources efficiently. Furthermore, there are problem instances for which concurrent programming is a natural paradigm (*e.g.*, client-server architectures and event-based architectures).

Programming in a concurrent style is difficult:

- The partitioning of a program into several *threads*,[4] each executing a part of the whole program's task, is in general challenging because of data and control flow dependencies between the threads.
- The coordination of multiple threads is a complicated task which requires communication among the threads. This communication overhead can impede the scalability of the system.

Synchronization [39] refers to the coordination of simultaneously running threads and the maintenance of a coherent view of data shared between threads. Synchronization requires communication. Communication between threads comes in two major flavors, via shared memory or via message passing. Access to shared memory can be coordinated in many different ways, among them locking and transactional memory. The following subsections examine the resulting three paradigms in more detail. Most modern programming languages support these paradigms, either by linguistic means or through libraries.[5]

4.1 Lock-Based Synchronization

Shared memory imposes only little overhead on data synchronization. The classical approach to synchronization in this setting grants code fragments (*critical sections*) only *mutually exclusive* access to the shared memory. This *locking* of resources guarantees that a thread obtains exclusive access for some time to complete its memory operations undisturbed. Unfortunately, excessive locking can reduce parallelism. Even worse, *deadlocks* can arise when threads that have already obtained some locks are blocked, mutually waiting for further locks to be released. Similarly, threads can end up in a *livelock* where their state constantly changes but no progress is made. Explicit synchronization via locking is commonly thought to be error-prone due to its delicate semantics.

A *monitor* [22] mediates all accesses to and modifications of some portion of shared memory. It guarantees that the procedures associated with the monitor obtain mutually exclusive access to the resources guarded by it. The standard implementation is via locks (generated by the compiler).

[4] This paper considers "thread" and "process" as synonyms and uses the word "thread" throughout. The usual reading is that threads run in a shared address space whereas processes may run in separate address spaces.

[5] There are also pure hardware and hybrid platforms that implement thread communication. The paper nevertheless concentrates on the software implementations.

4.2 Transaction-Based Synchronization

Software Transactional Memory (STM) [35] is seen by some as a more user-friendly approach to synchronization in a shared memory setting. It offers a high-level mechanism and shifts the implementation of mutual exclusion as well as some data management tasks to the runtime environment.

Central to STM is the notion of an *atomic block* which is used to encapsulate the accesses to the shared memory in a safe manner. A *transaction*[6] starts when entering an atomic block and ends when leaving it. The STM system guarantees that the computations inside a transaction either execute as a whole or not at all. Moreover, other concurrently running threads cannot observe intermediate states of the computation inside an atomic block. These intrinsic features are referred to as *atomicity* and *isolation*.

The implementation of STM has to cope with concurrently running transactions accessing and modifying the same memory locations. To ensure the absence of *conflicts*, a conflict detection mechanism checks the system's consistency and eventually arbitrates between the conflicting parties. In general, this arbitration may lead to the abortion of a transaction and a retry later in time. A transaction that finishes its computation without conflict *commits* when leaving the atomic block. Otherwise the transaction performs a *rollback*.

Several design choices configure the exact behavior of an STM implementation:

Atomicity *Strong atomicity* ensures that an atomic block executes in isolation with respect to all other computations. *Weak atomicity* guarantees isolation only with respect to other atomic blocks.

Conflict detection *Pessimistic conflict detection* checks the validity of read and written data progressively, so conflicts are detected early and transactions which are bound to fail are aborted quickly. *Optimistic conflict detection* postpones data validation until the end of an atomic block.

Granularity of conflicts Conflicts may occur at the *object level*, the *cache-line level*, or the *word level*. Whereas object conflict detection is a sensible choice in object-oriented programming languages, the other options are useful in less structured settings.

Data versioning *Eager versioning* performs in-place memory update during a transaction. It saves overwritten values in an undo-log structure for reconstruction on a potential rollback. With *lazy versioning* each atomic block maintains its own local write buffer whose values are later on committed to the shared memory.

Nesting A nested transaction [26] arises when an atomic block is enclosed in another atomic block. One approach

[6] Transactional memory evolved from ideas of transaction processing in database systems and borrows much of its terminology.

flattens this nesting into a single transaction. *Closed nesting* supports rollback of a nested transaction such that a successful commit of an outer transaction requires the successful completion of all nested transactions. To increase parallelism *open nesting* allows nested transactions to commit their result globally and irreversibly.

If the execution of an atomic block fails, all its effects must be undone and the former state restored. The runtime has to provide the means to perform this rollback, for example by logging of values. To sustain the isolation property, many implementations forbid irreversible operations, such as I/O, inside of atomic blocks. Often a type system restricts the mutation of shared data to atomic blocks.

4.3 Message-Based Synchronization

In a parallel architecture with *distributed memory*, threads do not share a common address space. Thus, it is more appropriate to manage data sharing via *message passing* (MP) than to grant remote memory access. In this setting pairs of corresponding send and receive operations transport data between threads. The message passing interface (MPI [36]) and its successor MPI2 [16] define standard APIs that many programming languages implement. Message passing is also the basis of the Erlang programming language [1, 2]. Communication operations can be classified with respect to the following categories:

Point-to-point vs. global A thread can send a message either to one other thread or to all other threads (*broadcast*). It is also possible to group threads for communication purposes.

Synchronous vs. asynchronous In synchronous mode, the sender blocks until the receiving thread has started its receive operation. In asynchronous mode, the send operation does not block. Instead, the run-time system buffers the message until the receiver requests it.

Accumulation vs. non-accumulation A special receive operation can accumulate messages from multiple threads with a specified reduction operation. The receiver sees only the final result. Alternatively, all messages are sent directly to the receiver.

The description of lock-, transaction-, and message-based synchronization suggests that these concepts are fundamentally distinct. Nevertheless, there are many hybrid forms. For example, STM systems can be implemented using some kind of locking when checking and writing data at the end of a transaction [9].

5. Playing the Analogy

The thorough description of concepts in version control and in concurrent programming in Section 3 and 4 suggest further analogies than just the correspondence between Subversion and software transactional memory considered in Section 2.

5.1 Prologue

Section 3 distinguishes three dimensions in VC systems, namely the repository model (central vs. distributed), the concurrency model (lock-based vs. merge-based), and the history model (snapshot vs. change set). The history dimension concerns the navigation in the space of accessible versions. In a concurrent system, the aim is to have one coherent view of the program state, so that only one version is interesting and worth preserving. Thus, it does not appear reasonable to consider the analogy along the history dimension.

A similar comment applies to accountability. In a VC system, accountability is of utmost importance because it enables tracking down the person responsible for a certain change. In concurrent programming, accountability is mostly not cared for, but it may have unexpected uses (see Section 6.1.2).

After squashing the history axis, it remains to consider all combinations of repository models and concurrency models. All combinations but one are worth exploring.

- Section 2 deals with Subversion, which is an instance of a merge-based VC system with a central repository, and compares it to software transactional memory.
- Section 5.2 relates lock-based VC with a central repository to concurrent programming with monitors.
- Section 5.3 explores the combination of merge-based VC with distributed repositories and relates it to message passing approaches in concurrency.
- The combination of lock-based VC with distributed repositories does not make sense (see Section 3).

5.2 The Relation of Lock-Based VC to Concurrent Programming with Monitors

There is a close correspondence between lock-based VC with a central repository and the monitor approach to synchronizing concurrent access to shared variables. Although the change to a lock-based VC affects only few parts of the workflow, it is clearer to restate both stories rather than just report the changes. Figure 4 contains the revised version of the stories in Figure 1 and Figure 5 contains the adapted version of the correspondence in Figure 2. Significant differences are highlighted in bold face. The following subsections contain further discussion.

5.2.1 Lock-Based VC

In a VC system with locking, no developer can change an artifact without previously acquiring its lock. Thus, the *commit* operation never fails. It just copies artifacts from the workspace to the repository.

Lock-based VC is considered too limiting because its granularity is too coarse. Typically, one artifact (*e.g.*, source file) contains many independently editable and clearly separated entities (*e.g.*, declarations) and only few of them are af-

Lock-based VC

Item 2 of Toru's story in Figure 1 gets extended by the addition of locking, item 5 gets extended by unlocking, and items 6 and 7 get replaced by item 6.

1. No *developer* is allowed to operate directly on *artifacts* in the shared *repository*.
2. The developer *checks out* to make the *head revision* of the shared artifacts available for reading and writing.

   The check-out operation copies the artifacts into the developer's *workspace*. **The developer *locks* all artifacts that he wants to modify.**
3. The developer creates, reads, updates, and deletes artifacts as required to achieve the objective.
4. The developer might find out that his chosen approach does not work with the current repository. In this case, he *reverts* and restarts from the previous check out.
5. The developer might come to the conclusion that the objective cannot be achieved, in which case he *abandons* his modifications, *unlocks* all artifacts, and proceeds with another task.
6. Once the developer has achieved his objective, he publicizes his modifications by **performing a *commit* operation** and unlocking the artifacts.

Concurrent Programming with Monitors

Items 4 and 5 of Miki's story need minor adjustment. Items 6 and 7 get replaced by item 6.

1. **Outside a monitor procedure**, no *thread* is allowed to operate directly on *variables* in the shared *program state*.
2. The thread *enters a monitor* to make the *current values* of the shared variables available for reading and writing.
3. The thread creates, reads, updates, and deletes variables as required to achieve the objective.
4. The thread might find out that its chosen approach does not work with the current program state. In this case, it *cleans up*, and restarts with item 3.
5. The thread might come to the conclusion that the objective cannot be achieved, in which case it cleans up its modifications, *exits* the monitor, and proceeds with another task.
6. Once the thread has achieved its objective, it publicizes its modifications by exiting the monitor.

Figure 4. Two tales of lock-based VC and concurrent programming with monitors

Toru	Miki
project	component
developer	thread
version control	concurrency control
repository	program state
artifact	variable
head revision	current value
RCS	**monitor implementation**
workspace	**—**
check out and lock	**enter monitor**
commit and unlock	**exit monitor**
revert	**cleanup**
abandon	**cleanup**
unlock	**exit monitor**

Figure 5. Adapted correspondence for lock-based VC and concurrent programming with monitors.

fected by a change. This separation makes it easy for merge-based approaches to do the right thing in many cases.

In lock-based VC, most developers do not obtain their locks all at once but gradually over time. This practice is prone to deadlock so that all lock-based VC systems support an operation to deliberately break a lock. Breaking a lock is not considered good style but it is sometimes unavoidable because the lock holder may be on a business trip, on vacation, or otherwise unavailable.

In VC, it is easy to back out of a failed objective because the developer works on local copies of the artifacts in his workspace. The repository still retains a previous version of the artifacts and simply falls back to that version.

The VC lock effectively serves as a read/write lock. The VC system imposes a write lock because it stops anyone but the lock holder from modifying an artifact. It imposes a (kind of) read lock because changes to the artifact only become visible after the lock holder has committed and released the lock. In the terminology of database transactions, this strategy avoids dirty reads [4].

5.2.2 Concurrent Programming with Monitors

In concurrent programming with monitors, the *commit* operation is implicit in exiting the monitor as the monitor guarantees that no other thread can change the variables that it guards while the thread is active in the monitor. However, the developer of the monitor has to anticipate that an objective may be abandoned and save data manually if that is required. Furthermore, the relationship between the monitor and the objects which the monitor is deemed to protect is often only informally specified. A concept like Java's synchronized methods [15] alleviates this problem but programmers can still inadvertently violate the locking protocol, for example, by using Java's `synchronized` blocks inappropriately.

The developer of the monitor must choose the granularity of the monitor procedures so that soundness is guaranteed and the program is reasonably efficient.

In item 4 in the right column of Figure 4, it is more fair to leave the monitor temporarily before trying another approach. This way, other threads waiting to enter the monitor get a chance to proceed. In contrast to a VC system, it is not possible to break a lock and enter a monitor occupied by another thread.

Contrary to the VC situation, there is no easy way to back out of a failed objective. The developer has to implement his own error detection and cleanup strategy. At least, the monitor ensures that there is no execution path leaving the critical section of the program without releasing the locks. With "bare-bones locking" the developer would have to take care of that matter, too.

Unlike the VC situation, the scope of the monitor is up to the programmer. In particular, if an isolation property that avoids dirty reads is desired, it has to be implemented by the programmer, for example, by also annotating a method that reads a shared variable as `synchronized` in Java.

5.3 The Relation of VC with Distributed Repositories to Concurrent Programming with Message Passing

While distributed repositories are popular in state-of-the-art VC systems, there is no directly matching concurrency paradigm. However, it is straightforward to synthesize a sensible distributed programming pattern based on message passing from the VC workflow. Figure 6 presents the synopsis of the two workflows and Figure 7 adapts the correspondences from Figures 2 and 5 to the new situation. The subsections contain further explanation and discussion, first on the VC side and then on the message passing side.

For concreteness, the VC part of this section borrows the terminology from Darcs [33], a merge-based VC with distributed repositories.

5.3.1 VC with Distributed Repositories

In a VC with a distributed repository, all developers have their own local repository and all commits are performed on the local repository. A developer may synchronize the local repository with a remote repository, either by pulling changes from the remote repository or by pushing local changes to it. Locally, the push operation has no effect.

The left column of Figure 6 contains the VC story. The choice of remote changes to integrate locally in step 3 is sometimes called *cherry picking*.

The Darcs documentation [34] does not always distinguish between the local repository and the workspace, although both exist and the workspace may contain changes not committed to the repository.[7] In particular, it is not clear from the description which of them is affected by the pull command (presumably both). Thus, the local check-out operation is implicit in the preceding pull operation.

Toru	Miki
project	component
developer	thread
version control	concurrency control
repository	program state
artifact	variable
head revision	current value
Darcs	**MP implementation**
workspace	**—**
pull	**custom pull**
push	**custom push**
local check out	**—**
local commit	**continue**
revert	**cleanup**
abandon	**cleanup**

Figure 7. Adapted correspondence for VC with distributed repositories and concurrent programming with message passing.

The discussion on message passing in Section 4.3 lists a classification of communication operations that need to be discussed in the context of VC.

Point-to-point vs. global Distributed VC systems employ point-to-point connections, only. No system employs broadcast communication to push changes, although it might make sense to define groups of remote repositories that should be updated at the same time.

Synchronous vs. asynchronous Both variants exist for VC systems, at least when embedded in an SCM system. For example, the standard API of Subversion is synchronous but its Eclipse embedding Subclipse [37] can update and check out asynchronously in the background.

Accumulation vs. non-accumulation There is no corresponding, useful concept in VC.

5.3.2 Concurrent Programming with Message Passing

The right column of Figure 6 synthesizes a suitable concurrency model from the VC workflow in the left column. The result concurrent programming paradigm employs message passing, but it is not straightforward to construct a good correspondence.

Both the pull and push operations are implemented by sending a message to the remote program state. Pull asks for changes in the remote program state. Push sends a set of

[7] Darcs calls the local commit operation *record*.

VC with Distributed Repositories	Concurrent Programming with Message Passing
1. Each developer is allowed to operate on *artifacts* in his *local repository*. He cannot directly operate on any *remote repository*.	1. Each thread is allowed to operate on *variables* in its *local program state*. It cannot directly operate on any *remote program state*.
2. The developer may choose to perform a *pull* operation to bring his repository up-to-date with the *head revision* of a remote repository.	2. The thread may choose to perform a *custom pull* operation* to bring its program state up-to-date with the *current values* in a remote program state.
3. As long as there is a conflict caused by the pull operation, the developer should address it by reconciling the changes of the local repository with those of the remote repository and *committing* locally.	3. As long as there is a conflict caused by the pull operation, the thread should address it by reconciling the changes of the local program state with those of the remote program state and *continuing*.
4. The developer (implicitly) performs a local *check-out* operation to make the head revision of the artifacts available for reading and writing.	4. The thread locally has the values of the variables available for reading and writing.
5. The developer creates, reads, updates, and deletes artifacts as required to achieve the objective.	5. The thread creates, reads, updates, and deletes variables as required to achieve the objective.
6. If he finds out that the chosen approach does not work with the current repository, then he *reverts* the local modifications and restarts from the last check-out operation.	6. If it finds out that the chosen approach does not work with the current program state, then it *reverts* the local modifications and restarts from the last pull operation.
7. He might come to the conclusion that the objective cannot be achieved, in which case he proceeds with another task.	7. It might come to the conclusion that the objective cannot be achieved, in which case it proceeds with another task.
8. Once he has achieved his objective, he commits locally.	8. Once it has achieved its objective, it continues.
9. He may attempt to integrate the current local repository into a remote one by performing a *push* operation.	9. It may attempt to integrate the current local program state into a remote one by performing a *custom push* operation.*
10. The developer proceeds with another task.	10. The thread proceeds with another task.

* Custom pull and push operations are implemented using send and receive operations of the underlying MP implementation.

Figure 6. Two tales of VC with distributed repositories and concurrent programming with message passing.

changes to be integrated in the remote program state. Both operations require suitable permissions.

There is no analogon to a workspace in message-passing concurrency, hence there is neither a notion of a local check out nor of a local commit.

The reconciliation of changes in step 2 requires some reflection facilities. In the VC world, the developer obtains both versions and can apply tools like `diff` to them. Analogous tools would be needed as an API at the thread level.

6. Corollaries

This section collects observations and differences that came up while building the correspondences in the previous sections. The observations are not sufficiently specific to a particular correspondence to merit inclusion in the section dedicated to it. However, they contain a sufficient element of surprise to deserve mention.

6.1 More on the Relation of VC to STM

The description of STM implementations in Section 4.2 mentions some design choices that still need to be put in relation to the VC context.

Atomicity A VC system always provides strong atomicity because all changes happen locally in the developer's workspace. Thus, they remain invisible to everyone unless the developer commits them.

Conflict detection All non–lock-based VC systems perform optimistic conflict detection. Strictly lock-based systems need no conflict detection at all.

Granularity of conflicts VC systems also check for conflicts on different levels. The standard level is one line for a textual artifact and the whole object for a binary artifact. The sophistication of the underlying diff and merge algoritm determines the level. Depending on them, other choices are possible, for example, syntactic merging [43] or semantic merging [23, 31].

Data versioning The workspace of a VC system dictates that all versioning happens lazily.

Nesting VC systems do not have a notion of nested transaction. However, a developer can easily emulate nesting by creating a branch and merging that back into the head revision once the branch development was successful.

6.1.1 Revisiting Consistency

Most VC systems support the notion of a *commit hook*, that is, a command that the VC system runs to perform some action before finalizing a commit. If this action signals an error, then the system aborts the commit. A popular use for commit hooks is to perform some consistency check on the artifacts. This check might evaluate some static property of a program, for example, whether the program is syntactically correct. Such a check guarantees that all artifacts later checked out of the repository fulfill the consistency check.

A similar notion exists for one STM implementation [20]. It supplies a facility for specifying constraints on variables which are checked at commit time. A failed check results in a failed commit.

6.1.2 Revisiting Accessibility and Accountability

The prologue of Section 5 makes the observation that accessibility and accountability, which are key features of VC, have no counterpart in concurrent programming. While this observation is true in the standard mode of operation of a deployed program, it may by worth to reconsider it in the light of debugging.

Concurrent programs are known for exhibiting intermittent problems (such as race conditions) that are very hard to reproduce and fix. While searching for such problems, it would be desirable to find out which thread is responsible for setting some variable to the wrong value. It would also be advantageous to be able to access all possible execution histories and test them against some consistency criterion. Indeed, it works so well that people are doing it all the time using model checking [6].

6.2 From STM to VC

One of the underlying ideas of this essay is that insights from work on VC may yield insights and inspiration for further work on concurrency abstractions. Interestingly, there is at least one instance where this transfer may work the other way round, in this case from STM implementation to VC.

In an STM implementation with lazy versioning and optimistic conflict detection, each thread builds a log structure consisting of all read and write accesses to shared memory during the execution of an atomic block. Conflict detection scans the read log and checks that the variables read still have the same values as before. STM signals a conflict if the logged value of a variable differs from the current value.

In contrast, a VC system usually detects a conflict by comparing the head revision with the version in the user's workspace. It signals a conflict if some other user has modified the head revision in a way that is incompatible with the changes in the workspace. That is, only a modification counts as a conflict, but a dependency caused by the user reading other artifacts does not.

Here is an example, where this strategy leads to a problem. Consider two software developers A and B who work concurrently.

- A reads the declaration of `x` in module `M` and writes a hunk of code in module `N` that uses `M.x`.
- B renames `x` to `y` consistently in module `M`.

Both developers can commit in arbitrary order (potentially with an intervening update) without the VC system signalling a conflict. However, the resulting program certainly does not compile.[8]

The problem here is that the VC is not aware of the dependency between `M.x` and `N`. A more advanced VC might be able to avoid the problem by registering developer A's "read accesses" to `M.x` (or better the dependency of the new code on `M.x`) in analogy to the log structure in the STM implementation.

7. Conclusion

What have we learned?

The blurry observation that started the investigation underlying this paper was that software transactional memory behaves similarly to a merge-based version control system with central repository. Close inspection of concepts, workflows, and systems in version control and concurrent programming lead us to collect many coincidences that helped pinpoint the correspondence exactly. It also lead us to broaden the correspondence to include monitors and message passing on the concurrent programming side, as well as lock-based approaches and multiple distributed repositories on the version control side.

It might be said that both version control and concurrent programming are tackling the same problem underneath. However, version control has facets without parallel on the concurrent programming side (*e.g.*, accessibility and accountability) and, vice versa, concurrent programming has facets not found in version control systems. For example, fully automatic conflict resolution is a conditio sine qua non, whereas human intervention is both acceptable and desired in version control. Thus, while the underlying problem of

[8] VC etiquette requires that the head revision of a program should compile, but this is rarely enforced.

managing concurrent work on shared resources is similar, there are many differences in terms of specific characteristics.

There is another indication that version control and concurrent programming are perceived very differently by the community. This indication is progress over time in the adoption of different methodologies.

- Early version control systems are lock-based with central repository (SCCS, RCS). The next stage of progress is merge-based systems with central repository (CVS, Subversion, etc). The most recent stage is merge-based systems with distributed repositories (Darcs, Bazaar, Git, etc).
- Early concurrent programming techniques are lock-based (mutexes, semaphores, monitors, etc). Next, there are message passing systems (MPI, Erlang, etc). The most recent stage is software transactional memory.

The surprising observation is that while both, version control and concurrent programming, start with lock-based approaches, they progress differently through the stages of the correspondence outlined in this essay. While message passing is a technique long established in concurrent programming, version control systems with distributed repositories are a comparably recent development. Vice versa, while merge-based version control systems with a central repository are used pervasively in software development (as exemplified by CVS and Subversion), software transactional memory, the corresponding concurrent programming technology, is still in a research stage.

Lock-based techniques for concurrent programming are error-prone and difficult to master. Of the two alternative approaches, message passing and software transactional memory, both are likely to stay because they yield a definite added value with respect to lock-based systems and they have different merits and application areas (distributed vs. concurrent programming).

A similar situation holds for version control systems. Clearly, the lock-based approach is obsolete, but there is no clear case for central vs. distributed repositories. In fact, this choice depends on the organizational structure of software development. Distributed repositories fit with bazaar-style open-source software development where developers are independent and may have private variants of a system [30]. Centralized repositories fit better with industrial (cathedral-style) software development where some authority bears responsibility for the resulting system.

Thus, there is some potential for cross-fertilization in the sense that high-level ideas and concepts as well as automatic techniques may be transferred between version control and concurrent programming. Full convergence seems impossible because of the fundamental differences between the areas.

References

[1] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, LLC, 2007.

[2] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, NY, 1993.

[3] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[4] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, Inc, 1997.

[5] Jim Buffenbarger. Syntactic software merging. In *Software Configuration Management*, volume 1005 of *Lecture Notes in Computer Science*, pages 153–172, 1995.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[7] CVS. Homepage. `http://www.nongnu.org/cvs/`, 2006.

[8] Darcs. Homepage. `http://darcs.net/`, 2008.

[9] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[10] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, 2005.

[11] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[13] W. M. Gentleman, A. MacKay, and D. A. Stewart. Commercial realtime software needs different configuation management. In *Proceedings of the 2nd International Workshop on Software configuration management*, pages 152–161, New York, NY, USA, 1989. ACM.

[14] Git. Homepage. `http://git.or.cz/`, 2008.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, June 2005.

[16] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1998.

[17] Dan Grossman. The transactional memory / garbage collection analogy. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming,*

Systems, Languages, and Applications, pages 695–706, Montreal, QC, CA, 2007. ACM Press, New York.

[18] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Anaheim, CA, USA, 2003. ACM Press, New York.

[19] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Sixteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, USA, June 2005. ACM Press.

[20] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06*, June 2006.

[21] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the Twentysecond Annual Symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, 2003. ACM Press, New York, NY, USA.

[22] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[23] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[24] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An empirical study of delta algorithms. In *Software Configuration Management*, volume 1167 of *Lecture Notes in Computer Science*. Springer, 1996.

[25] Mercurial. Homepage. `http://www.selenic.com/mercurial/wiki/`, 2008.

[26] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[27] Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the nato science committee. `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`, January 1969.

[28] Perforce. Homepage. `http://www.perforce.com/`, 2008.

[29] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2008.

[30] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[31] Thomas W. Reps, Susan Horwitz, and Jan Prins. Support for integrating program variants in an environment for programming in the large. In *Proceedings of the International Workshop on Software Version and Configuration Control*, volume 30 of *Berichte des German Chapter of the ACM*, pages 197–216. Teubner, 1988.

[32] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[33] David Roundy. Darcs: Distributed version management in Haskell. In Daan Leijen, editor, *Proceedings of the 2005 ACM SIGPLAN Haskell Workshop*, pages 1–4, Tallinn, Estland, September 2005.

[34] David Roundy. Darcs 1.1.0pre1 (unknown). `http://darcs.net/manual/`, 2007.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles of Distributed Computing*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press, New York, NY, USA.

[36] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[37] Subclipse. Homepage. `http://subclipse.tigris.org/`, 2006.

[38] SVN. Homepage. `http://subversion.tigris.org/`, 2006.

[39] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[40] Walter F. Tichy. RCS – a system for version control. *Software—Practice & Experience*, 15(7):637–654, 1985.

[41] Jennifer Vesperman. *Essential CVS*. O'Reilly Media, Inc., 2003.

[42] Adam Welc, Suresh Jagannathan, and Anthony Hosking. Transactional monitors for concurrent objects. In Martin Odersky, editor, *18th European Conference on Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 519–542, Oslo, Norway, June 2004. Springer-Verlag.

[43] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 68–79, New York, NY, USA, 1991. ACM.