The Relation of Version Control to Concurrent Programming

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 that work concurrently to achieve a shared objective. Concurrency control helps coordinating a group of threads that work 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 with three influential and representative approaches to concurrency control exhibits a surprisingly close correspondences in terms of mechanism and workflow. The correspondence yields new perspectives on both, version control and concurrency control.

1. Introduction

Version control is a mature field of software engineering that traces its origins back to the 1968 NATO conference [10]. It is one of the few fields that have attracted academic and industrial research alike and which has a multimillion dollar market. 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 history, dating back to the 1960s. Much recent work in this field concentrates on software transactional memory [13]. This approach transfers ideas from database transactions to the problem of synchronizing concurrent modification of shared datastructures. Its paradigm is claimed to be particularly easy to master from a programmer perspective.

Our work presents an analogy of transactional memory with a certain flavor of version control. It further extends it to the other main flavors of version control and compares them with monitors and message passing. As with Grossman's analogy [7] 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.

Outline § 2 and § 3 give an overview of basic terms in version control and concurrent programming. § 4 relates different flavors of version control systems with different approaches to concurrent programming. § 5 collects further observations about this relationship and § 6 concludes. An extended version of this article is available [2].

2. Version Control

Version control (VC) systems manage multiple versions of the same artifact. 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 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 consisted of.

These features can be traced back already to one of the first VC systems, SCCS [12], and are standard in all modern systems. VC systems can be further classified according to the following dimensions [5]:

Repository model Does the VC system store artifacts in a central or a distributed repository?

Concurrency model Does the VC system prevent conflicting changes via locking or via merging?

History model Does the VC system record changes in a version graph with snapshots or as changes sets?

3. Concurrent Programming

Concurrent programming [1, 6] is the art of controlling (pseudo-) simultaneous execution of multiple interacting threads. The primary objective is to increase the application throughput and 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 considered difficult. The partitioning of a program into several threads is in general challenging due to data and control flow dependencies

978-0-7695-3336-0/08 $25.00 © 2008 IEEE
DOI 10.1109/CSSE.2008.1079

creating data races. Furthermore, the coordination of multiple threads requires communication among them which can impede the scalability of the system. We will have a closer look at three approaches to concurrent programming:

Locking with Monitors This is the classical approach to synchronization via shared memory. A *monitor* [9], for example, grants critical sections only *mutually exclusive* access to shared resources. Unfortunately, excessive locking can reduce parallelism. Manual synchronization is also error-prone as deadlocks or livelocks are hard to avoid.

Software Transactional Memory (STM) STM [13] offers a high-level mechanism as it shifts responsibility to the runtime environment. Atomic blocks are used to encapsulate the accesses to the shared memory in a safe manner. At runtime, a transaction tries to commit the updates made within an atomic block to the shared memory. A conflict detection mechanism checks the system's consistency and eventually arbitrates between the conflicting parties. In general, this may lead to the abortion of a transaction and a retry later in time. As other concurrently running threads cannot observe intermediate states of the computation inside an atomic block, the paradigm implements *atomicity*, *isolation* and *consistency*. On failure of execution, all effects must be undone and the former state restored. The runtime has to provide the means to perform this rollback, *e.g.* by logging of values. Many implementations forbid irreversible operations, such as I/O, inside atomic blocks.

Message Passing In a distributed architecture, threads do not share a common physical address space. Thus, it is more appropriate to manage data sharing via message passing. Pairs of corresponding send and receive operations transport data between threads. MPI [14] defines a standard API including point-to-point and global communication, synchronous and asynchronous operation mode, accumulative and non-accumulative operations.

Most modern programming languages support these paradigms by linguistic means or through libraries. Though this brief description suggests fundamental differences, there exists many hybrid forms [4].

4. Playing the Analogy

In § 2, we distinguished three dimensions in VC systems: 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 worth preserving. Thus, it is not reasonable to consider the analogy along the history dimension.

In the following, we contrast the remaining two dimensions of VC systems with the three approaches to concurrent programming introduced in § 3. For reasons of space, we concentrate on the analogy between merged-based VC systems and STM (§ 4.1). The two other analogies are discussed very briefly (§ 4.2 and § 4.3); our extended version of this article provides more details [2].

4.1. Merge-Based VC vs. STM

Figure 1 tells the tale of Toru, a project manager and Miki, a software developer. It demonstrates that a concurrent thread using STM has a workflow which is very similar to that of a developer who manages a project using a VC system. 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. This enables him to perform experiments without affecting anyone else. Some implementations of STM follow a similar strategy by keeping local copies of the variables used in an atomic block. However, these copies are created on demand and do not outlive the atomic block. The corresponding VC usage pattern of checking out artifacts lazily on demand does not appear to be useful because building and testing involves more artifacts than just the modified ones. Still, often the developer does not need to check out the entire repository.
- A VC system always provides strong atomicity because all changes happen locally in the developer's workspace. They remain invisible to everyone unless the developer commits them.
- If the VC system detects a conflict on an attempt to commit, the developer has to take action to get an up-to-date view and to adapt his changes to the new situation. If STM fails to commit, the STM implementation is forced to rerun the body of the atomic block.
- A VC system records the history of each artifact in the repository. STM keeps only the most current value of a shared variable.
- VC systems do not have a notion of nested commits. 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.

4.2. Lock-Based VC vs. Monitors

In a VC system with locking, no developer can change an artifact without previously acquiring its lock. The corresponding concurrent programming model is locking, for example through monitors. The workflow in both situations is similar, but there are also differences:

- The scope of a monitor is up to the programmer whereas a lock-based VC system provides a predefined set of locks.
- To avoid deadlocks, all lock-based VC systems support an operation to deliberately break a lock. With monitors, this operation is typically not supported.

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.

- With a VC system, it is easy to back out of a failed objective: simply do not commit your changes. With monitors, a programmer has to implement his own error detection and cleanup strategy.

4.3. VC with Distributed Repositories vs. Message Passing

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 his local repository with some remote repository by either pulling changes from or pushing changes to the remote repository.

This workflow is similar to programming with the message passing paradigm. All processes have their own local state and all changes are performed on the local state. A process may synchronize its local state with the state of some other process by either receiving messages from or sending messages to the other process.

5. Corollaries

This section collects further observations and differences which are not sufficiently specific to a particular correspondence but contain a sufficient element of surprise to deserve mentioning.

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.

One STM implementation [8] supplies a similar facility for specifying constraints on variables which are checked at commit time. A failed check results in a failed commit.

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.

For example, it may be 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 some 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 [3].

Dependency 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.

6. Conclusion

Our primary observation was that software transactional memory behaves similarly to a merge-based version control system with central repository. Close inspection of concepts, workflows, and systems of version control and concurrent programming 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.

Truly, both version control and concurrent programming tackle the same problem underneath. However, version control has facets without an analogon on the concurrent programming side, such as accessibility and accountability. Vice versa, concurrent programming has unique aspects. For example, fully automatic conflict resolution is a conditio sine qua non, whereas human intervention is both acceptable and desired in version control.

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

- Early version control systems are lock-based with central repository (SCCS, RCS). Then, merge-based systems with central repository emerged (CVS, Subversion, etc). The most recent stage includes 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 STM.

While both, version control and concurrent programming, start with lock-based approaches, they surprisingly progress differently. 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, STM, its counterpart in concurrent programming, is still in a research stage.

The two more recent approaches, message passing and STM, are both likely to evolve because they yield a definite added value with respect to lock-based systems, and aim at different application areas (distributed vs. concurrent programming).

Clearly, the lock-based approach in VC is obsolete. Distributed repositories fit with bazaar-style open-source software development where independent developers may have private variants of a system [11]. Centralized repositories fit better with industrial (cathedral-style) software development with some authority bears responsibility for the resulting system.

As we have indicated, there is some potential for cross-fertilization as high-level ideas and even automatic techniques may be transferred between version control and concurrent programming.

References

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

[2] A. Bieniusa, P. Thiemann, and S. Wehr. The relation of version control to concurrent programming (extended version). `http://proglang.informatik.uni-freiburg.de/projects/transactions/index.html`, 2008.

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

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[5] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Software Eng. and Meth.*, 14(4):383–430, 2005.

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

[7] D. Grossman. The transactional memory / garbage collection analogy. In *Proc. 22nd ACM Conf. OOPSLA*, pages 695–706, Montreal, QC, CA, 2007. ACM Press, New York.

[8] T. Harris and S. Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06*, June 2006.

[9] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17(10):549–557, Oct. 1974.

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

[11] E. 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.

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

[13] N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th ACM Symp. PODC*, pages 204–213, Ottowa, Ontario, Canada, 1995. ACM Press, New York, NY, USA.

[14] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.