An Alloy Analysis of CVS

Michael Harder   Alan Donovan

MIT Lab for Computer Science NE43-525
200 Technology Square
Cambridge, MA 02139 USA
{mharder,adonovan}@lcs.mit.edu

Abstract

CVS is the Concurrent Versions System, a free, distributed version–control system. We used Alloy to construct three models of CVS: a model to analyze file-level locking properties, a model to investigate the atomicity of checkins, and a model of CVS semantics from the user's point of view. We found undesirable properties in the locking and checkin models, and proposed and evaluated changes to fix the problems. The user-level model could serve as an aid for understanding CVS and provides a precise definition of various concepts. This paper assumes that readers are familiar with the usage and concepts both of CVS and of Alloy.

1. File-level locking

1.1 Background

CVS stores its internal data structures as ordinary files, and uses locks to protect these files. Any number of clients can be reading a file at a time, but if a client is writing then there must be no other readers or writers.

CVS also uses files to implement its locking protocol. Locks are created using the `mkdir()` system call, which is assumed to be atomic on all supported systems. Each lock protects a single directory in the repository, not including subdirectories which represent other directories under version control. To lock an entire tree, each directory must be locked separately.

There are three types of locks: master, read, and write. The master lock must be held before either read or write locks can be held. At most one client can hold the master lock for a directory. Read and write locks must be held before files can be read or written. Any number of clients can hold read locks for a directory, and at most one client can hold the write lock for a directory.

CVS uses the following algorithms for reading from and writing to directories [Ced].

To read files in a directory

1. Obtain the master lock.
2. Obtain a read lock.
3. Release the master lock.
4. Read files in the directory.
5. Release the read lock.

To write files in a directory

1. Obtain the master lock.
2. If there are any read locks, release the master lock and try again later.
3. Obtain the write lock.
4. Write files in the directory.
5. Release the write lock.
6. Release the master lock.

1.2 Alloy model

We created an Alloy model of the CVS locking protocol (Appendix A), and used it to check several properties of the protocol.

First, we checked that the protocol satisfies its main goals. The Alloy assertions `NoReadWrite` and `NoWriteWrite` assert that if a client is writing, there must be no other readers or writers. We checked these assertions for 3 clients and 10 states, and Alloy found no counterexamples. This makes us relatively confident that the protocol successfully protects the files.

Next, we checked that multiple clients could simultaneously read a directory (Alloy function `MultipleReads`). Alloy found an solution to this function, which means there is at least one instance where multiple clients can read simultaneously.

Finally, we investigated the fairness of the protocol to readers and writers. We defined fairness as follows:

> Assume each read and write is of finite length. Whenever a client desires to read or write, it should be able to do so within a finite amount of time.

We checked this property for both reads and writes (Alloy functions `StarveRead` and `StarveWrite`). Alloy found no solutions to the `StarveRead` function, but did find a solution to the `StarveWrite` function. This means a desire to write can be forced to wait indefinitely.

By examining the solution produced by Alloy, we determined the cause of the problem. Reads are allowed to overlap, and the write lock can never be held as long as there are read locks. An infinite chain of overlapping reads will prevent a write from ever taking place.

We made the locking protocol more fair by changing the second step in the write files protocol. In the original protocol, if any read locks are present after obtaining the master lock, the master lock is released and the client must try again later. In our new protocol, if any read locks are present after obtaining the master lock, the client simply waits until all the read locks are released. This will happen eventually, since no new read locks can be obtained. Once the read locks are gone, the client proceeds to obtain the write lock and write the files.

This change required a 1-line modification to our Alloy model. After modifying our model, we used Alloy to checked the `StarveRead` and `StarveWrite` function. Alloy found no solutions to either function, meaning neither reads nor writes can be starved indefinitely. Just to be safe, we re-checked the `NoReadWrite`, `NoWriteWrite`, and `MultipleReads` properties. These properties held as well, meaning our modification did not affect the other properties of the model.

2. Atomic check-ins

2.1 Background

This model addresses the following property, as stated in the CVS manual [Ced]:

> If someone commits some changes in one cvs command, then an update by someone else will either get all the changes, or none of them.

We refer to this property as the *atomic check-ins*. The manual states that CVS does *not* have this property. As far as we can tell, check-ins involving multiple files in one directory are atomic, but check-ins involving multiple directories are not atomic. This is because all locking is performed at the directory level.

For example, assume someone runs `cvs checkin a/one.c b/two.c`, and someone runs `cvs update` at the same time. The person running `update` might get the changes to `a/one.c`, but not to `b/two.c`.

This is more than a theoretical flaw in the system — it causes problems for real users. For example, the Mozilla daily builds have been known to fail if a developer is checking-in at the same time the build system is checking-out.

The CVS commands `update` and `checkout` are identical for our purposes, and we use the terms interchangably throughout this section.

2.2 Alloy model

We created an Alloy model of the CVS check-in and check-out protocols, to better understand this problem and propose solutions.

First, we had to reverse-engineer the locking scheme used by CVS when processing multiple directories. We ran CVS against a test repository, and observed the lock files created. The locks used are the same read and write locks from Section 1, but for the purposes of this section we consider a directory as being *locked* if a client holds either a read or write lock. The locking scheme is as follows:

To check-out directories A and B

1. Obtain the lock for A.
2. Read from A.
3. Release the lock for A.
4. Obtain the lock for B.
5. Read from B.
6. Release the lock for B.

To check-in directories A and B

1. Obtain the lock for A.
2. Obtain the lock for B.
3. Write to A.
4. Write to B.
5. Release the lock for A.
6. Release the lock for B.

The locking scheme is extended in the obvious way for more than two directories. The directories are processed in no particular order.

We created an Alloy model of the above locking scheme (Appendix B), and used it to check properties of the scheme. First, we asserted that check-ins are atomic (Alloy assertion `AtomicCheckin`). Alloy found the following counterexample:

1. Client 1 is checking-out A and B, Client 2 is checking-in A and B.
2. Client 1 obtains lock for A, reads from A, releases lock for A.
3. Client 2 obtains locks for A and B, writes to A and B, releases locks for A and B.
4. Client 1 obtains lock for B, reads from B, releases lock for B.

Client 1 got the changes for B, but not for A. To solve this problem, we made the locking scheme for check-in the same as check-out — obtain locks for all directories before reading any directory. We changed our model, and used Alloy to re-check the `AtomicCheckin` assertion. This time, Alloy found no counterexamples, giving us confidence our new scheme is correct.

As an orthogonal problem, we analyzed the potential for deadlock in this model. Multiple check-ins can deadlock in the original model, and multiple check-outs can deadlock as well in our revised model. We wrote an assertion that the model can never reach a state of deadlock (Alloy assertion `DeadLock`). Alloy found the following counterexample:

1. Client 1 is checking-in A and B, Client 2 is cheking in A and B.
2. Client 1 obtains lock A.
3. Client 2 obtains lock B.

CVS solves this problem as follows: if a needed lock is held by someone else, release all locks and try again later. We propose a more elegant solution: apply a total ordering to the locks, and obtain the locks in ascending order. We added this to our Alloy model, and re-checked the `DeadLock` assertion. Alloy found no counterexamples.

3. User-level model

3.1 Background

In this section, we will discuss another model of CVS; the level of abstraction of this model is the user level, by which we mean it describes the operation of CVS using concepts such as Event, Status, Client, File and Version, which are familiar to users of CVS, not just its maintainers.

Such a model is useful to gain an understanding of how the CVS application works, which would be essential for someone considering making extensions to it. Though it is largely a stable codebase now, we believe such a model would be extremely valuable during the design and implementation phases of CVS, or of any similar tools involving distributed state-machines.

3.2 Alloy model

This Alloy description models the state of the CVS repository and one or more clients, while users at each client issue a stream of CVS events. One event corresponds to one CVS command (e.g. `checkin`, `update`) operating on exactly one file. In practice, users typically issue operations on all the files in a directory; however since we are not concerned with atomicity or synchronisation, multiple-file operations can be serialised into a stream of similar operations.

Our model is incomplete: it examines only a single repository; it supports only a single branch; it does not include all possible CVS commands, and it has no notion of history. However, we believe it would be relatively easy to extend it to include branches and history, at which point the model would cover the majority of all CVS scenarios occurring in practice.

3.2.1 Signatures

The principal signatures in our model are as follows:

File represents the set of all files.[1]

Version is a totally-ordered set, representing the various versions of a file.

Client represents the set of all clients using the repository.

Event represents the type of a single CVS command. Each `Event` is associated with exactly one `File` and one `Client`. Users are familiar with these events from the various CVS commands: `UpdateEvent`, `ModifyEvent`, `AddEvent`, `RemoveEvent`, and `CheckinEvent`.

Status represents the status of a particular file from the point of view of a given client. Users of CVS are familiar with these statii from the various messages displayed by the CVS `update` command: `UpToDate`, `LocallyAdded`, `LocallyRemoved`, `LocallyModified`, `NeedsMerge`, `NeedsPatch`, `HadConflicts`, `NotCheckedIn`, and `NeedsCheckout`.

State represents an ordered set of points on the timeline as the simulation proceeds.

3.2.2 Relations

All the relations of this model belong to the signature `State` which is shown below.

[1]Note that `File`-equivalence models name-equality, so two clients can refer to the same `File` even before it has been checked-in, if the names are the same.

```
sig State
{
// representation:
    // global state:
    event_type:         Event,
    event_client:       Client,
    event_file:         File,
    repository:         File ->! Version,

    // per-client state:
    entries:            Client -> File ->! Version,
    localMods:          Client -> File,
    localConflicts:     Client -> File,
    localRemoved:       Client -> File,

// 'public interface':
    status_p:           Client -> File -> Status,
    event_p:            Client -> File -> Event
}
```

The first three relations define the pending event, which consists of an `Event` type, an issuing `Client`, and a `File` on which to operate. The next relation, `repository`, abstracts the state at the server, and is a partial function from a `File` to its latest checked-in `Version`. Together, these relations make up the global state.

The following relations model the per-client state: `entries` (which has the same type as `repository` once the client has been projected away) abstracts the `CVS/Entries` database at the client, and maps locals files to the server version they were last synchronised with. The following three relations are per-file flags (membership $\implies$ true) that, in the absence of file data, abstract the the contents of files. They model whether a file has been modified locally (`localMods`), whether it had conflicts during a merge (`localConflicts`), and whether it has been locally removed (`localRemoved`).

The remaining two relations, `status_p` and `entries_p`, represent the 'public' interface: they model the aspects of the system seen by a casual user who does not peek into the database, and they are each defined as a total function of the current `State`. Note that `status` is never used by the model, but is provided to clients of the model so that they need not be concerned with the relations forming the concrete implementation, upon which the state-transition logic is based. It is particularly useful for visualisation, and for terse formulation of test scenarios.

The `status` relation is totally defined by the `whatStatus` function, which is implemented with one predicate per `Status`. We have proved with an Alloy assertion that not only do these predicates cover all possible values of `State` (constrained only by its manifest facts), but that they are all mutually exclusive. In fact, it was only by using Alloy assertions to ensure the validity of the predicates that we discovered we had omitted the status `NeedsPatch` — once we added it, the assertion succeeded. The partitioning induced by these predicates is valuable, since their definitions serve as the invariants of the CVS system, and would be useful to anyone wishing to modify or extend it.

3.3 Example model

We will briefly describe a few steps of the model during simulation. Please read the following in conjunction with the visualisation diagrams in Appendix C.

3.3.1 Single client, simple modify/check-in

In the first example, we have a single client and a single file; initially, the client is working on version 1 of that file, and is about to check it in.

1. Note that the version at the client and repository match, that the file is in the client's `localMods` set, and that the status is `LocallyModified`. A `CheckinEvent` is pending.

2. Now the file is no longer modified, and the repository and client's CVS/Entries database show version 2. A `ModifyEvent` is pending, which changes the status back to `LocallyModified`.

3.3.2 *Multiple clients, conflicting update*

The second example shows two clients, working on a single file. Initially, both clients are working on version 3 of the file, and both have made local modifications.

1. Both clients have made modifications, so the file's status is `LocallyModified`. A `CheckinEvent` from Client 1 is pending.

2. Client 1 is now `UpToDate` with version 4, but Client 0, which is working on a stale file, needs a patch, hence `NeedsMerge`. It issues an `UpdateEvent`.

3. Having updated, Client 0 is now working on a modified version 4, but during the merge, the file `HadConflicts`.

3.4 Evaluation

We hope to use the `Event/Status` abstraction to build a description of the state-transition logic that matches the users' conceptual model instead of the more detailed representation model. In particular, we believe the interesting cases are when an `Event` at one client causes a change in the `Status` of files at another client. Also, we hope to use this model to prove various other consistency properties of CVS, such as monotonicity of version numbering, etc.

The model is unsound in a number of regards:

No branches. For simplicity, we have modelled the common-case usage of CVS in which the version tree degenerates to a linear list. Although adding support for branches would require a change in the chosen abstractions, it should not cause any fundamental problems; however, adding complexity increases the run-time of the analyser, making debugging slower.

File contents are approximated The merge operations that CVS performs are complex, so we make no attempt to model the actual file contents. It suffices to abstract the file contents by the three `local<X>` flags. However, this is only an approximation: we treat merge updates pessimistically (i.e. it always causes a conflict if the file was stale) and we treat modifications optimistically (in that we assume any conflict is always resolved).

`Add` **and** `Remove` **events are inexact.** In reality, these two events can be issued in either order. `Add` before `Remove` is a no-op. `Remove` before `Add`, which is not possible in the model, should cause the latest version to be checked-out (as if by `update`).

Attic not supported. In the model, when a locally-removed file is checked-in, the corresponding file in the repository is deleted. In reality, however, CVS simply moves the file to a special directory called the *Attic*, which permits files that were removed to be restored later.

Model requires further testing. The model as shown in Appendix C, while sufficient to generate the visualisations shown in Appendix D, contains a number of minor faults, purely due to time pressure. We hope to eliminate these in the future.

The abstraction chosen for this model was in many ways the obvious one; it was decided upon quite early-on, and changed little during development. The `event` relation was originally defined as `State -> Client -> File -> Event` but was later split into three parts to avoid the more complex syntax it required to express constraints (in particular, when 'extracting columns'). However, the original quaternary relation was left in (now a simple function of the others) as it proved useful during visualisation.

We believe the current model, extended with history and branches, will serve as a useful tool for maintainers of the CVS system.

4. Remarks on the Alloy system

Modelling CVS with Alloy was for the most part straightforward; Alloy's syntax is very expressive and allows terse specification of constraints. For example, to specify "produce a model in which some file at some client gets into status `HadConflicts` and then subsequently returns to status `UpToDate`", we simply write:

```
some s: State, f: File, c:Client | {
    HadConflicts in s.status[c][f]
    UpToDate in OrdNexts(s).status[c][f]
}
```

However, the modelling was extremely time-consuming, due in large part to a number of specific difficulties encountered when using the tool:

Analyser diagnostics are too terse. When the analyser fails to find a solution to the model, it gives no indication as to what is the probable cause of the failure. This puts a heavy burden on the user of selectively enabling and disabling parts of the model until they can identify the contradictory constraints.

Scope clauses must be manually kept consistent with `sig` declarations. When defining a number of `static disj sigs` (such as the sub-sigs of `Event` or `Status`), the user must ensure that the model's `run` command accurately reflects the cardinality of these sets, or else no solution will be found. Failure to keep these consistent was frequently the cause of honerous debugging sessions. (In addition the user must provide and maintain a coverage constraint of the form $E = E_1 + E_2 + ... + E_n$).

Visualisation is powerful , but complex to use. We found that, with sufficient experimentation, it was possible to get very clear visualisations of the model simulations; however, the interactions of the various GUI features are unclear and a process of trial-and-error was usually required to achieve the desired results.

Semantics of `fun` are not transparent. Failure to use the `det` modifier on function declarations can result in complex and apparently incorrect behaviour, with no warning as to the possible fault. We understand this is a known issue with the use of functions in expressions (as opposed to formulae).

5. Lessons Learned

This section lists what we learned from this project.

- Models are much easier to write in Alloy if they are written in a certain way. The current version of our file-level locking model (Appendix A) is written as a state machine, with functions to identify each state (`Idle`, `NewReadReq`, etc.), and a single function (`Trans`) describing the legal transitions between states. Earlier versions of the model were just an ad-hoc collection of rules that seemed to work. The state-machine model is much easier to work with, since the relationship between the prestate and poststate is clearly defined.

  We made 5 different versions of the file-level locking model before we decided on the state machine representation. In contrast, the atomic checkin model (Appendix B) was created using a state machine right away, and the model presented in this paper is the first version.

- Writing the properties to check can be more difficult than writing the model itself. The hardest part of the file-level locking model was writing the `StarveRead` and `StarveWrite` functions.

- Alloy is useful for "regression testing" changes to a model, to make sure a change didn't have any unexpected consequences. After we changed the file-level locking model so writes can't be starved, we rechecked the earlier assertions that the locking protocol correctly protects files.

References

[Ced] Per Cederqvist. *Version Management with CVS.*

[Jac] Daniel Jackson. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy.*

Appendix A: Alloy model of file-level locking

```
module project

open std/ord

// A CVS process executing on a machine.  Could have used
// "client" instead.
sig Process {}

sig State {
  // The processes that want to read or write, but haven’t yet
  // completed the read or write
  readRequest: set Process,
  writeRequest: set Process,

  // Used to insert new requests into the system
  newReadRequest: set Process,
  newWriteRequest: set Process,

  // The processes that are currently reading or writing
  read: set Process,
  write: set Process,

  // The process that holds the master lock
  masterLock: option Process,

  // The processes that hold read or write locks.  Multiple processes
  // could theoretically hold write locks, but the transition rules prevent
  // this from happening.
  readLock: set Process,
  writeLock: set Process
}

fact FullSets {
  Process = univ[Process]
  State = univ[State]
}


/*
 * STATE MACHINE
 */
fun Trans(s, s’: State) {
  all p: Process {
    // Unconstrained, or constrained by external rules
    Idle(s, p) => Idle(s’, p) ||
                  NewReadReq(s’, p) ||
                  NewWriteReq(s’, p)

    NewReadReq(s, p) => ReadReq(s’, p)

    // Constrained by MasterLockMustTry
    ReadReq(s, p) => ReadReq(s’, p) || ReadReqMaster(s’, p)

    ReadReqMaster(s, p) => ReadLockMaster(s’, p)

    ReadLockMaster(s, p) => ReadLockNoMaster(s’, p)

    ReadLockNoMaster(s, p) => Read(s’, p)

    // Unconstrained
    Read(s, p) => Read(s’, p) || ReadLockNoReq(s’, p)
    ReadLockNoReq(s, p) => Idle(s', p)


    NewWriteReq(s, p) => WriteReq(s', p)

    // Constrained by MasterLockMustTry
    WriteReq(s, p) => WriteReq(s', p) || WriteReqMaster(s', p)

    // Orignal model: writes can be starved by overlapping reads
    // WriteReqMaster(s, p) && some s.readLock => WriteReq(s', p)

    // Fixed model: writes *cannot* be starved by overlapping reads, since
    // the writer holds the master lock until the read locks are released
    WriteReqMaster(s, p) && some s.readLock => WriteReqMaster(s', p)

    WriteReqMaster(s, p) && no s.readLock => WriteLockMaster(s', p)

    WriteLockMaster(s, p) => Write(s', p)

    // Unconstrained
    Write(s, p) => Write(s', p) || WriteLockNoReq(s', p)

    WriteLockNoReq(s, p) => MasterLock(s', p)

    MasterLock(s, p) => Idle(s', p)
  }
}

fun Idle(s: State, p: Process) {
  p not in s.masterLock
  NoRead(s, p)
  NoWrite(s, p)
}

fun NewReadReq(s: State, p: Process) {
  p not in s.masterLock
  p not in s.readLock
  p not in s.read
  p not in s.readRequest
  p     in s.newReadRequest
  NoWrite(s, p)
}

fun ReadReq(s: State, p: Process) {
  p not in s.masterLock
  p not in s.readLock
  p not in s.read
  p     in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun ReadReqMaster(s: State, p: Process) {
  p     in s.masterLock
  p not in s.readLock
  p not in s.read
  p     in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun ReadLockMaster(s: State, p: Process) {
  p     in s.masterLock
  p     in s.readLock
  p not in s.read
  p     in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun ReadLockNoMaster(s: State, p: Process) {
  p not in s.masterLock
  p     in s.readLock
  p not in s.read
  p     in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun Read(s: State, p: Process) {
  p not in s.masterLock
  p     in s.readLock
  p     in s.read
  p     in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun ReadLockNoReq(s: State, p: Process) {
  p not in s.masterLock
  p     in s.readLock
  p not in s.read
  p not in s.readRequest
  p not in s.newReadRequest
  NoWrite(s, p)
}

fun NewWriteReq(s: State, p: Process) {
  p not in s.masterLock
  p not in s.writeLock
  p not in s.write
  p not in s.writeRequest
  p     in s.newWriteRequest
  NoRead(s, p)
}

fun WriteReq(s: State, p: Process) {
  p not in s.masterLock
  p not in s.writeLock
  p not in s.write
  p     in s.writeRequest
  p not in s.newWriteRequest
  NoRead(s, p)
}

fun WriteReqMaster(s: State, p: Process) {
  p     in s.masterLock
  p not in s.writeLock
  p not in s.write
  p     in s.writeRequest
  p not in s.newWriteRequest
  NoRead(s, p)
}

fun WriteLockMaster(s: State, p: Process) {
  p     in s.masterLock
  p     in s.writeLock
  p not in s.write
  p     in s.writeRequest
  p not in s.newWriteRequest
  NoRead(s, p)
}

fun Write(s: State, p: Process) {
  p     in s.masterLock
  p     in s.writeLock
  p     in s.write
  p     in s.writeRequest
  p not in s.newWriteRequest
  NoRead(s, p)
}

fun WriteLockNoReq(s: State, p: Process) {
  p     in s.masterLock
  p     in s.writeLock
  p not in s.write
  p not in s.writeRequest
  p not in s.newWriteRequest
  NoRead(s, p)
}

fun MasterLock(s: State, p: Process) {
  p     in s.masterLock
  NoWrite(s, p)
  NoRead(s, p)
}


fun NoWrite(s: State, p: Process) {
  p not in s.writeLock
  p not in s.write
  p not in s.writeRequest
  p not in s.newWriteRequest
}

fun NoRead(s: State, p: Process) {
  p not in s.readLock
  p not in s.read
  p not in s.readRequest
  p not in s.newReadRequest
}

// If a process has a read or write request, it must actively try to obtain
// the master lock
fun MasterLockMustTry(s, s': State) {
  all p: Process {
    p in s.readRequest && p not in s.readLock
    =>
    some s'.masterLock

    p in s.writeRequest && p not in s.masterLock
    =>
    some s'.masterLock
  }
}


/*
 * UTILITY FUNCTIONS
 */
fun FirstProcess(): Process { result = Ord[Process].first }
fun SecondProcess(): Process { result = OrdNext(FirstProcess()) }
fun ThirdProcess(): Process { result = OrdNext(SecondProcess()) }

fun FirstState(): State { result = Ord[State].first }
fun SecondState(): State { result = OrdNext(FirstState()) }
fun ThirdState(): State { result = OrdNext(SecondState()) }
fun FourthState(): State { result = OrdNext(ThirdState()) }
fun FifthState(): State { result = OrdNext(FourthState()) }
fun LastState(): State { result = Ord[State].last }

fun SameState(s, s': State) {
  s.newReadRequest = s'.newReadRequest
  s.newWriteRequest = s'.newWriteRequest
  s.readRequest = s'.readRequest
  s.writeRequest = s'.writeRequest
  s.read = s'.read
  s.write = s'.write
  s.masterLock = s'.masterLock
  s.readLock = s'.readLock
  s.writeLock = s'.writeLock
}

fun Loop(s, s': State) {
  s != s'
  SameState(s, s')
}

// True if the process had a chance to obtain the master lock between
// the two states
fun ChanceAtMasterLock(s, s': State, p: Process) {
  no s.masterLock || (s.masterLock != s'.masterLock)
}


/*
 * MODEL MECHANICS
 */
fun LegalTransition(s, s': State) {
  Trans(s, s')
  MasterLockMustTry(s, s')
}

fun AllLegalTransitions() {
  all s: (State-LastState()) {
    let s' = OrdNext(s) {
      LegalTransition(s, s')
    }
  }
}

fun Init(s: State) {
  no s.newReadRequest
  no s.newWriteRequest
  no s.readRequest
  no s.writeRequest
  no s.read
  no s.write
  no s.masterLock
  no s.readLock
  no s.writeLock
}
```

```
fun InitAllLegal() {
  Init(Ord[State].first)
  AllLegalTransitions()
}


/*
 * PROPERTIES TO CHECK
 */

// True if a read request can be starved
fun StarveRead() {
  InitAllLegal()
  some begin: State {
    Loop(begin, LastState())
    let loopStates = begin + OrdNexts(begin) {
      some p: Process {
        all s: loopStates {p in s.readRequest}
        no s: loopStates - LastState() | let s' = OrdNext(s) {
          ChanceAtMasterLock(s, s', p)
        }
      }
      no p: Process | all s: loopStates {
        p in (s.masterLock + s.readLock)
      }
    }
  }
}

// True if a write request can be starved
fun StarveWrite() {
  InitAllLegal()
  some begin: State {
    Loop(begin, LastState())
    let loopStates = begin + OrdNexts(begin) {
      some p: Process {
        all s: loopStates {p in s.writeRequest}
        no s: loopStates - LastState() | let s' = OrdNext(s) {
          ChanceAtMasterLock(s, s', p) && no s'.readLock
        }
      }
      no p: Process | all s: loopStates {
        p in (s.masterLock + s.readLock)
      }
    }
  }
}

// No two processes should be able to read and write simultaneously
assert NoReadWrite {
  InitAllLegal()
  =>
  (all s: State | (some s.write) => (no s.read))
}

// No two processes should be able to write simultaneously
assert NoWriteWrite {
  InitAllLegal()
  =>
  (all s: State | sole s.write)
}
```

```
// Multiple processes should be able to read simultaneously
fun MultipleReads() {
  InitAllLegal()
  some s: State | #s.read > 1
}


/*
 * TESTS
 */
fun test1() {
  InitAllLegal()
  all s: State - SecondState() | no s.newReadRequest
  SecondState().newReadRequest = FirstProcess()
  no State$newWriteRequest
}

fun test2() {
  InitAllLegal()
  all s: State - SecondState() | no s.newWriteRequest
  SecondState().newWriteRequest = FirstProcess()
  no State$newReadRequest
  all p: Process | Idle(LastState(), p)
}


run test1 for 2 Process, 10 State
run test2 for 2 Process, 10 State
check NoReadWrite for 3 Process, 10 State
check NoWriteWrite for 3 Process, 10 State
run MultipleReads for 2 Process, 10 State
run StarveWrite for 3 Process, 13 State
run StarveRead for 3 Process, 13 State

/*
 * Local Variables:
 * compile-command: "java -cp alloy.jar alloy.cli.AlloyCLI -E -n cvs-sm.als"
 * End:
 */
```

Appendix B: Alloy model of atomic check-ins

```
module project

open std/ord

// A CVS process executing on a machine.  Could have used
// "client" instead.
sig Proc {}

// Directories in the repository.  Don't need to model files in the
// directory, since locking is done at the directory level.
sig Dir {}

// Possible versions for the directories.
sig Ver {}

sig State {
  // Version of the directory on the server
  serverVer: Dir ->! Ver,

  // Version of the directory on each client
  clientVer: Proc -> Dir ->! Ver,

  // The processes that want to checkout or checkin directories,
  // but haven't yet completed the checkin or checkout
  coRequest: Proc -> Dir,
  ciRequest: Proc -> Dir,

  // Used to insert new requests into the system
  newCoRequest: Proc -> Dir,
  newCiRequest: Proc -> Dir,

  // The process that holds the lock for a directory
  lock: Proc ?-> Dir
}

fact FullSets {
  Proc = univ[Proc]
  Dir = univ[Dir]
  Ver = univ[Ver]
  State = univ[State]
}


/*
 * STATE MACHINE
 */
fun Trans(s, s': State) {
  all p: Proc {
    Idle(s, p)
    =>
    (Idle(s', p) || NewCoRequest(s', p) || NewCiRequest(s', p)) &&
    NoClientVersionChange(s, s', p)

    NewCoRequest(s, p)
    =>
    CoRequest(s', p) &&
    s'.coRequest[p] = s.newCoRequest[p] &&
    NoClientVersionChange(s, s', p)

    NewCiRequest(s, p)
    =>
    CiRequest(s', p) &&
    s'.ciRequest[p] = s.newCiRequest[p] &&
    NoClientVersionChange(s, s', p)

    CoRequest(s, p)
    =>
    ((CoRequest(s', p) &&
      s'.coRequest[p] = s.coRequest[p] &&
      NoClientVersionChange(s, s', p))
     ||
     (CoRequestLock(s', p) &&
      s'.coRequest[p] = s.coRequest[p] &&
      // Add only one lock in the transition
      one s'.lock[p] &&
      NoClientVersionChange(s, s', p)))

    CiRequest(s, p)
    =>
    ((CiRequest(s', p) &&
      s'.ciRequest[p] = s.ciRequest[p] &&
      NoClientVersionChange(s, s', p))
     ||
     (CiRequestLock(s', p) &&
      s'.ciRequest[p] = s.ciRequest[p] &&
      // Add only one lock in the transition
      one s'.lock[p] &&
      NoClientVersionChange(s, s', p)))


/*
    // Original behavior, as implemented by CVS
    CoRequestLock(s, p) && #s.coRequest[p] > 1
    =>
    (CoRequestLock(s', p) &&
     s'.coRequest[p] = s.coRequest[p] &&
     s'.lock[p] = s.lock[p] &&
     NoClientVersionChange(s, s', p))
    ||
    (CoRequest(s', p) &&
     s'.coRequest[p] = s.coRequest[p] - s.lock[p] &&
     CheckoutVersionChange(s, s', p))

    CoRequestLock(s, p) && #s.coRequest[p] = 1
    =>
    (CoRequestLock(s', p) &&
     s'.coRequest[p] = s.coRequest[p] &&
     s'.lock[p] = s.lock[p] &&
     NoClientVersionChange(s, s', p))
    ||
    (Idle(s', p) &&
     CheckoutVersionChange(s, s', p))
*/

    // Fix to make checkins atomic. Obtain all locks before checking out.
    // Same transitions as CiRequestLock
    CoRequestLock(s, p) && s.lock[p] != s.coRequest[p]
    =>
    (CoRequestLock(s', p) &&
     s'.coRequest[p] = s.coRequest[p] &&
     s.lock[p] in s'.lock[p] &&
     // Add at most one lock in a transition
     sole (s'.lock[p] - s.lock[p]) &&
     NoClientVersionChange(s, s', p))
    CoRequestLock(s, p) && s.lock[p] = s.coRequest[p]
    =>
    (CoRequestLock(s', p) &&
     s'.coRequest[p] = s.coRequest[p] &&
     s'.lock[p] = s.lock[p] &&
     NoClientVersionChange(s, s', p))
    ||
    (Idle(s', p) &&
     CheckoutVersionChange(s, s', p))

    CiRequestLock(s, p) && s.lock[p] != s.ciRequest[p]
    =>
    (CiRequestLock(s', p) &&
     s'.ciRequest[p] = s.ciRequest[p] &&
     s.lock[p] in s'.lock[p] &&
     // Add at most one lock in a transition
     sole (s'.lock[p] - s.lock[p]) &&
     NoClientVersionChange(s, s', p))

    CiRequestLock(s, p) && s.lock[p] = s.ciRequest[p]
    =>
    (CiRequestLock(s', p) &&
     s'.ciRequest[p] = s.ciRequest[p] &&
     s'.lock[p] = s.lock[p] &&
     NoClientVersionChange(s, s', p))
    ||
    // CheckinVersionChange handled by ServerVer function
    (Idle(s', p) && NoClientVersionChange(s, s', p))
  }
}

fun Idle(s: State, p: Proc) {
  no s.newCoRequest[p]
  no s.newCiRequest[p]
  no s.coRequest[p]
  no s.ciRequest[p]
  no s.lock[p]
}

fun NewCoRequest(s: State, p: Proc) {
  some s.newCoRequest[p]
  no s.newCiRequest[p]
  no s.coRequest[p]
  no s.ciRequest[p]
  no s.lock[p]
}

fun CoRequest(s: State, p: Proc) {
  no s.newCoRequest[p]
  no s.newCiRequest[p]
  some s.coRequest[p]
  no s.ciRequest[p]
  no s.lock[p]
}

fun CoRequestLock(s: State, p: Proc) {
  no s.newCoRequest[p]
  no s.newCiRequest[p]
  some s.coRequest[p]
  no s.ciRequest[p]
  some s.lock[p]
  s.lock[p] in s.coRequest[p]
}
```

```
fun NewCiRequest(s: State, p: Proc) {
  no s.newCoRequest[p]
  some s.newCiRequest[p]
  no s.coRequest[p]
  no s.ciRequest[p]
  no s.lock[p]
}

fun CiRequest(s: State, p: Proc) {
  no s.newCoRequest[p]
  no s.newCiRequest[p]
  no s.coRequest[p]
  some s.ciRequest[p]
  no s.lock[p]
}

fun CiRequestLock(s: State, p: Proc) {
  no s.newCoRequest[p]
  no s.newCiRequest[p]
  no s.coRequest[p]
  some s.ciRequest[p]
  some s.lock[p]
  s.lock[p] in s.ciRequest[p]
}

fun NoClientVersionChange(s, s': State, p: Proc) {
  s'.clientVer[p] = s.clientVer[p]
}

fun CheckoutVersionChange(s, s': State, p: Proc) {
  all d: Dir {
    d in s.lock[p] => s'.clientVer[p][d] = s.serverVer[d]
    d not in s.lock[p] => s'.clientVer[p][d] = s.clientVer[p][d]
  }
}

// Updates the server version of a file when some process checks in
// the file.  Must be constrained outside of Trans, because we must look
// at *all* processes in a state to determine the new server version, while
// trans looks at one process at a time.
fun ServerVer(s, s': State) {
  all d: Dir {
    // Because of locks, there can only be one updating proc for a directory
    (some p: Proc | CiRequestLock(s, p) && Idle(s', p) && d in s.lock[p])
    =>
    let updatingProc = ~(s.lock)[d] {
      s'.serverVer[d] = s.clientVer[updatingProc][d]
    }

    !(some p: Proc | CiRequestLock(s, p) && Idle(s', p) && d in s.lock[p])
    =>
    s'.serverVer[d] = s.serverVer[d]
  }
}

// Solves deadlock problem
// Imposes total ordering on the order in which locks can be obtained
fun LockOrdering(s, s': State) {
  all p: Proc {
    all d: (s'.lock[p] - s.lock[p]),
        d': (s.ciRequest[p] + s.coRequest[p]) - s.lock[p] {
      OrdLE(d, d')
    }
  }
}


/*
 * UTILITY FUNCTIONS
 */
fun FirstProc(): Proc { result = Ord[Proc].first }
fun SecondProc(): Proc { result = OrdNext(FirstProc()) }
fun ThirdProc(): Proc { result = OrdNext(SecondProc()) }

fun FirstDir(): Dir { result = Ord[Dir].first }
fun SecondDir(): Dir { result = OrdNext(FirstDir()) }
fun ThirdDir(): Dir { result = OrdNext(SecondDir()) }

fun FirstVer(): Ver { result = Ord[Ver].first }
fun SecondVer(): Ver { result = OrdNext(FirstVer()) }
fun ThirdVer(): Ver { result = OrdNext(SecondVer()) }

fun FirstState(): State { result = Ord[State].first }
fun SecondState(): State { result = OrdNext(FirstState()) }
fun ThirdState(): State { result = OrdNext(SecondState()) }
fun FourthState(): State { result = OrdNext(ThirdState()) }
fun FifthState(): State { result = OrdNext(FourthState()) }
fun SixthState(): State { result = OrdNext(FifthState()) }
fun SeventhState(): State { result = OrdNext(SixthState()) }
fun EighthState(): State { result = OrdNext(SeventhState()) }
fun NinthState(): State { result = OrdNext(EighthState()) }
fun TenthState(): State { result = OrdNext(NinthState()) }
fun LastState(): State { result = Ord[State].last }


/*
 * MODEL MECHANICS
 */
fun LegalTransition(s, s': State) {
  Trans(s, s')
  ServerVer(s, s')
  LockOrdering(s, s')
}

fun AllLegalTransitions() {
  all s: (State-LastState()) {
    let s' = OrdNext(s) {
      LegalTransition(s, s')
    }
  }
}

fun Init(s: State) {
  no s.newCoRequest
  no s.newCiRequest
  no s.coRequest
  no s.ciRequest
  no s.lock
}

fun InitAllLegal() {
  Init(Ord[State].first)
  AllLegalTransitions()
}
```

```
/*
 * PROPERTIES TO CHECK
 */

// If you do a checkout, you actually get the new versions of files
assert CheckoutWorks {
  {InitAllLegal()
   all p: Proc, d: Dir {
     FirstState().serverVer[d] = SecondVer()
     FirstState().clientVer[p][d] = FirstVer()
   }
   all s: State-SecondState() | no s.newCoRequest
   SecondState().newCoRequest[FirstProc()] = FirstDir() + SecondDir()
   all s: State | no s.newCiRequest
   all p: Proc | Idle(LastState(), p)}
  =>
  all d: Dir | LastState().clientVer[FirstProc()][d] = LastState().serverVer[d]
}

// If you do a checkin, you actually update the files on the server
assert CheckinWorks {
  {InitAllLegal()
   all d: Dir {
     FirstState().serverVer[d] = FirstVer()
     FirstState().clientVer[FirstProc()][d] = SecondVer()
   }
   all s: State-SecondState() | no s.newCiRequest
   SecondState().newCiRequest[FirstProc()] = FirstDir() + SecondDir()
   all s: State | no s.newCoRequest
   all p: Proc | Idle(LastState(), p)
  }
  =>
  all d: Dir | LastState().serverVer[d] = LastState().clientVer[FirstProc()][d]
}

// If someone commits some changes in one cvs checkin, then a checkout
// by someone else will either get all the changes, or none of them.
assert AtomicCheckin {
  {InitAllLegal()

   // For each client, all directories start at the same version
   all p: Proc, disj d, d': Dir {
     FirstState().clientVer[p][d] = FirstState().clientVer[p][d']
   }

   // For the server, all directories start at the same version
   all disj d, d': Dir {
     FirstState().serverVer[d] = FirstState().serverVer[d']
   }

   // All checkouts and checkins must be all directories, or no directories
   // Avoids trivial solution where client only checks out/in some directories
   all s: State, p: Proc {
     s.newCoRequest[p] = univ[Dir] || s.newCoRequest[p] = none[Dir]
     s.newCiRequest[p] = univ[Dir] || s.newCiRequest[p] = none[Dir]
   }

   all p: Proc | Idle(LastState(), p)
  }

  =>

  no p: Proc {
    some disj d, d': Dir {
      LastState().clientVer[p][d] != LastState().clientVer[p][d']
    }
  }
}

// If two processes each hold a lock that the other process needs
assert Deadlock {
  InitAllLegal()
  =>
  no disj p, p': Proc, s: State {
    some l: s.lock[p], l': s.lock[p'] {
      l in (s.coRequest[p'] + s.ciRequest[p'])
      l' in (s.coRequest[p] + s.ciRequest[p])
    }
  }
}


/*
 * TESTS
 */
fun test1() {
  InitAllLegal()
  all p: Proc, d: Dir {
    FirstState().serverVer[d] = SecondVer()
    FirstState().clientVer[p][d] = FirstVer()
  }
  all s: State-SecondState() | no s.newCoRequest
  SecondState().newCoRequest[FirstProc()] = FirstDir() + SecondDir()
  all s: State | no s.newCiRequest
  all p: Proc | Idle(LastState(), p)
}

fun test2() {
  InitAllLegal()
  all d: Dir {
    FirstState().serverVer[d] = FirstVer()
    FirstState().clientVer[FirstProc()][d] = SecondVer()
  }
  all s: State-SecondState() | no s.newCiRequest
  SecondState().newCiRequest[FirstProc()] = FirstDir() + SecondDir()
  all s: State | no s.newCoRequest
  all p: Proc | Idle(LastState(), p)
}

// Manually construct non-atomic checkin instance
fun test3() {
  InitAllLegal()
  all d: Dir {
    FirstState().serverVer[d] = FirstVer()
    FirstState().clientVer[SecondProc()][d] = SecondVer()
     all p: Proc - SecondProc() {
       FirstState().clientVer[p][d] = FirstVer()
     }
  }

  NewCoRequest(SecondState(), FirstProc())
  SecondState().newCoRequest[FirstProc()] = FirstDir() + SecondDir()
  Idle(SecondState(), SecondProc())

  CoRequest(ThirdState(), FirstProc())
  NewCiRequest(ThirdState(), SecondProc())
  ThirdState().newCiRequest[SecondProc()] = FirstDir() + SecondDir()

  CoRequestLock(FourthState(), FirstProc())
  FourthState().lock[FirstProc()] = FirstDir()
  CiRequest(FourthState(), SecondProc())

  CoRequest(FifthState(), FirstProc())
  CiRequestLock(FifthState(), SecondProc())

  CoRequest(SixthState(), FirstProc())
  CiRequestLock(SixthState(), SecondProc())

  CoRequest(SeventhState(), FirstProc())
  Idle(SeventhState(), SecondProc())

  CoRequestLock(EighthState(), FirstProc())
  Idle(EighthState(), SecondProc())

  Idle(NinthState(), FirstProc())
  Idle(NinthState(), SecondProc())
}

// When checking out files, you may only hold one lock at a time
assert CheckoutSoleLock {
  InitAllLegal()
  =>
  all p: Proc, s: State {
    some s.coRequest[p] => sole s.lock[p]
  }
}


check CheckoutWorks for 1 Proc, 2 Dir, 2 Ver, 10 State
check CheckinWorks for 1 Proc, 2 Dir, 2 Ver, 10 State
check AtomicCheckin for 3 Proc, 2 Dir, 2 Ver, 10 State
check Deadlock for 2 Proc, 2 Dir, 2 Ver, 5 State
run test1 for 1 Proc, 2 Dir, 2 Ver, 7 State
run test2 for 1 Proc, 2 Dir, 2 Ver, 10 State
run test3 for 2 Proc, 2 Dir, 2 Ver, 9 State
check CheckoutSoleLock for 2 Proc, 2 Dir, 2 Ver, 10 State

/*
 * Local Variables:
 * compile-command: "java -cp alloy.jar alloy.cli.AlloyCLI -E -n multidir.als"
 * End:
 */
```

Appendix C: User-level Alloy model of CVS

```
module CvsMethods

open std/ord

// --- Principals --- //

sig File {}
sig Client {}
sig Version {}

// only one event can happen per state (a somewhat simplistic model of
// concurrency!)
sig State
{
    // -------- REPRESENTATION -------- //

    // These relations abstract the concrete facts of the CVS system.

    // Global state:
    event_type:         Event,
    event_client:       Client,
    event_file:         File,

    repository:         File ->! Version,

    // Client state:
    entries:            Client -> File ->! Version, // models CVS/Entries
    localMods:          Client -> File,
    localConflicts:     Client -> File,
    localRemoved:       Client -> File,

    // -------- PUBLIC INTERFACE -------- //

    // These relations, derived purely from the above (not in terms of
    // their previous values) represent the user's view of CVS.

    status_p:           Client -> File -> Status,
    event_p:            Client -> File -> Event
}
{
    // This redundant relation prevents important "unconnected" nodes
    // disappearing during visualisation:
    event_p = event_client -> event_file -> event_type
    one event_p

    all c: Client, f: File |
    {
        // define status relation by whatStatus function
        status_p[c][f] = whatStatus(this,c,f)

        // no client "entry" version can be later than the corresponding
        // repository if both exist:
        // XXX Really, we'd like to prove this like (if P true initially,
        // P is always true)
        let svr_ver = currentVersion(this, f) |
        let cli_ver = entries[c][f] |
        some svr_ver && some cli_ver =>
            cli_ver in OrdPrevs(svr_ver) + svr_ver
    }

    // Can't be both removed and modified
    all c: Client | no localRemoved[c] & localMods[c]
    // A conflicting file is always modified.
    localConflicts in localMods

    // A file can only conflict if there exists a version at the server
    // [XXX and in the entries!] [XXX this rule seems to have no effect]
    all f: localConflicts[Client] | some currentVersion(this, f)
}

// --- Status --- //

sig Status {}
{
    this in UpToDate + LocallyAdded + LocallyModified + LocallyRemoved +
            NeedsMerge + NeedsPatch + HadConflicts + NotCheckedIn +
            NeedsCheckout
}

static disj sig UpToDate,
                LocallyAdded,           // "Locally Added"
                LocallyRemoved,         // "Locally Removed"
                LocallyModified,        // "Locally Modified"
                NeedsMerge,             // "Needs Merge"
                NeedsPatch,             // "Needs Patch"
                HadConflicts,           // "File had conflicts on merge"
                NotCheckedIn,           // "Unknown"
                NeedsCheckout           // "Needs Checkout"
                extends Status {}

fun isUpToDate(s: State, c: Client, f: File)
{
    s.entries[c][f] = currentVersion(s, f) // Versions match at client + repos
    one currentVersion(s, f)    // exists in repository

    f !in s.localMods[c]        // no mods
    f !in s.localConflicts[c]
    f !in s.localRemoved[c]
}

fun isLocallyModified(s: State, c: Client, f: File)
{
    s.entries[c][f] = currentVersion(s, f) // Versions match at client + repos
    one currentVersion(s, f)    // exists in repository

    f in s.localMods[c]         // locally modified
    f !in s.localConflicts[c]
    f !in s.localRemoved[c]
}

fun isNeedsPatch(s: State, c: Client, f: File)
{
    isFileStale(s, c, f)

    f !in s.localMods[c]        // unmodified
    f !in s.localRemoved[c]
}

fun isNeedsMerge(s: State, c: Client, f: File)
{
    isFileStale(s, c, f)

    f in s.localMods[c]         // modified (don't care about conflicts)
    f !in s.localRemoved[c]
```

```
}

fun isHadConflicts(s: State, c: Client, f: File)
{
    s.entries[c][f] = currentVersion(s, f) // Versions match at client + repos
    one currentVersion(s, f)    // exists in repository

    f in s.localMods[c]         // mods with conflicts
    f in s.localConflicts[c]
    f !in s.localRemoved[c]
}

fun isLocallyAdded(s: State, c: Client, f: File)
{
    f in s.entries[c].Version   // local entry exists
    no currentVersion(s, f)     // doesn't exist in repository

    f !in s.localMods[c]        // no mods
    f !in s.localConflicts[c]
    f !in s.localRemoved[c]
}

fun isLocallyRemoved(s: State, c: Client, f: File)
{
    f in s.entries[c].Version   // local entry exists (but marked "removed")
    one currentVersion(s, f)    // exists at server

    f in s.localRemoved[c]      // removed
}

fun isNotCheckedIn(s: State, c: Client, f: File)
{
    f !in s.entries[c].Version  // no local entry
    no currentVersion(s, f)     // not in repository

                                // local<..> flags don't matter.
}

fun isNeedsCheckout(s: State, c: Client, f: File)
{
    f !in s.entries[c].Version  // no local entry
    one currentVersion(s, f)    // exists in repository

                                // local<..> flags don't matter.
}

det fun whatStatus(s: State, c: Client, f: File) : Status
{
    // We can use Alloy to prove that these predicates are disjoint;
    // to do so, we must remove all other constraints on whatStatus
    // that might mask a counterexample, such as any applied to
    // State$status_p.

    isUpToDate(s,c,f)           <=> UpToDate            in result
    isLocallyAdded(s,c,f)       <=> LocallyAdded        in result
    isLocallyRemoved(s,c,f)     <=> LocallyRemoved      in result
    isLocallyModified(s,c,f)    <=> LocallyModified     in result
    isNeedsPatch(s,c,f)         <=> NeedsPatch          in result
    isNeedsMerge(s,c,f)         <=> NeedsMerge          in result
    isHadConflicts(s,c,f)       <=> HadConflicts        in result
    isNotCheckedIn(s,c,f)       <=> NotCheckedIn        in result
    isNeedsCheckout(s,c,f)      <=> NeedsCheckout       in result
}
```

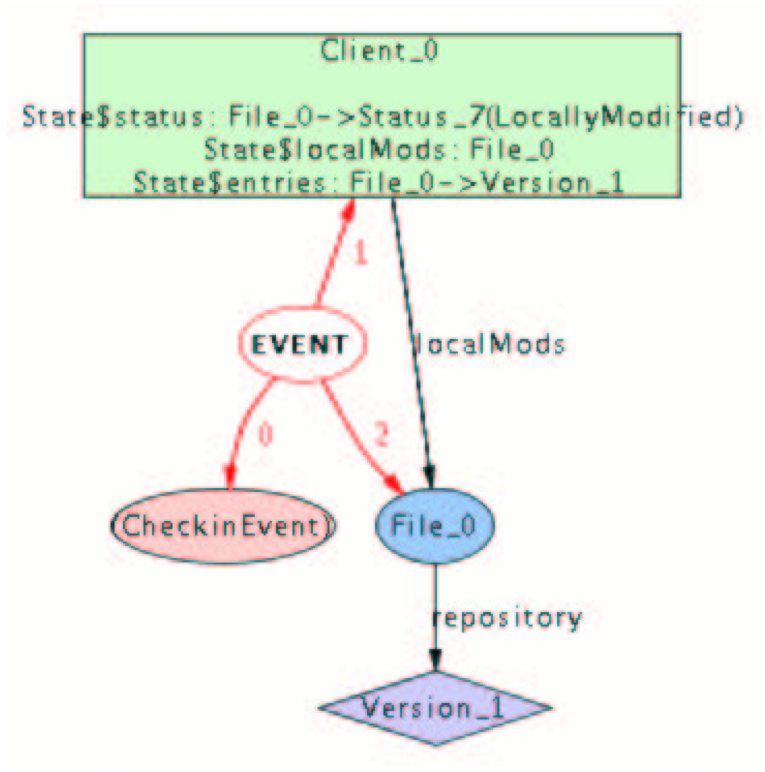
```
// This assertion is always true, FALSE, even without other
// constraints on whatStatus, proving that whatStatus is always
// uniquely defined.
assert statusPredicatesIntersect
{
    all s: State, c: Client, f: File | one whatStatus(s,c,f)
}
check statusPredicatesIntersect for 7 but 1 File, 2 Client, 9 Status, 6 Event


// --- Event --- //

sig Event {}
{
    this in UpdateEvent + ModifyEvent + AddEvent + RemoveEvent +
            CheckinEvent + NonEvent
}

static disj sig UpdateEvent,            // "cvs update foo.c"
                ModifyEvent,            // "emacs foo.c"
                AddEvent,               // "cvs add foo.c"
                RemoveEvent,            // "cvs remove foo.c"
                CheckinEvent,           // "cvs ci foo.c"
                NonEvent                // used for last state only
                extends Event {}

// --- Helper predicates --- //

fun repositoryUnchanged(s, s' : State)
{
    s'.repository = s.repository
}

fun false() { 0 = 1 }
fun true() {}

// In the event-client, all locals other than the event-file remain unchanged.
fun thisClientOtherLocalsUnchanged(s, s': State)
{
    let client = s.event_client |
    let file   = s.event_file |
    {
        // apart from the event_file, mods/conflicts/removed are unchanged
        // We don't specify any constraints on the event_file
        s'.localMods     [client] - file = s.localMods     [client] - file
        s'.localConflicts[client] - file = s.localConflicts[client] - file
        s'.localRemoved  [client] - file = s.localRemoved  [client] - file

        all f: File - file |
            s'.entries[client][f] = s.entries[client][f]
    }
}

// All clients not in this event don't change state (but they may change
// Status).
fun otherClientsLocalsUnchanged(s, s': State)
{
    all c: Client - s.event_client |
    {
        s'.entries[c]           = s.entries[c] // file->versions
        s'.localConflicts[c]    = s.localConflicts[c]
        s'.localMods[c]         = s.localMods[c]
        s'.localRemoved[c]      = s.localRemoved[c]
    }
}

// any change to the repository only affects the event file
fun reposOtherFilesUnchanged(s, s': State)
{
    // all other files unchanged in repository
    all f: File - s.event_file |
        currentVersion(s', f) = currentVersion(s, f)
}

fun currentVersion(s: State, f: File) : Version
{
    result = s.repository[f]
}

fun eventFileIsModified(s: State) {
    s.event_file in s.localMods[s.event_client]
}

// file exists at both ends but versions differ
fun isFileStale(s: State, c: Client, f: File) {
    s.entries[c][f] in OrdPrevs(currentVersion(s, f))
    some s.entries[c][f]
    some currentVersion(s, f)
}

fun clearConflicts(s, s': State) {
    s'.localConflicts[s.event_client] =
        s.localConflicts[s.event_client] - s.event_file
}

fun clearModified(s, s': State) {
    s'.localMods[s.event_client] =
        s.localMods[s.event_client] - s.event_file
}

fun clearRemoved(s, s': State) {
    s'.localRemoved[s.event_client] =
        s.localRemoved[s.event_client] - s.event_file
}

fun setConflicts(s, s': State) {
    s'.localConflicts[s.event_client] =
        s.localConflicts[s.event_client] + s.event_file
}

fun setModified(s, s': State) {
    s'.localMods[s.event_client] =
        s.localMods[s.event_client] + s.event_file
}

fun setRemoved(s, s': State) {
    s'.localRemoved[s.event_client] =
        s.localRemoved[s.event_client] + s.event_file
}

fun preserveConflicts(s, s': State) {
    s'.localConflicts[s.event_client] = s.localConflicts[s.event_client]
}

fun preserveModified(s, s': State) {
    s'.localMods[s.event_client] = s.localMods[s.event_client]
}

fun preserveRemoved(s, s': State) {
    s'.localRemoved[s.event_client] = s.localRemoved[s.event_client]
}

// --- Legality constraints on Event state-transitions --- //

fun legalEvent(s, s' : State)
{
    s.event_type != NonEvent // NonEvent is never used in state transitions

    let e = s.event_type |
    {
        AddEvent     in e => legalAddEvent(s, s')
        UpdateEvent  in e => legalUpdateEvent(s, s')
        ModifyEvent  in e => legalModifyEvent(s, s')
        RemoveEvent  in e => legalRemoveEvent(s, s')
        CheckinEvent in e => legalCheckinEvent(s, s')
    }

    // These are common invariants for all Events
    thisClientOtherLocalsUnchanged(s, s')
    otherClientsLocalsUnchanged(s, s')
    reposOtherFilesUnchanged(s, s')

    // Repository version numbers only go forward (no rollback)
    // between adjacent states, but the version numbering may start
    // again at zero after a removal and files may appear and
    // disappear over time.

    // XXX We would like this to be a testable assertion!

    all f: s.repository.Version | // all files in this state's repository
    let v  = currentVersion(s,  f) |
    let v' = currentVersion(s', f) |
        some v && some v' =>      // if file exists in later repository,
            v' in OrdNexts(v) + v // must have same or later version
}

fun legalUpdateEvent(s, s' : State)
{
    // -- preconditions --

    // -- postconditions --

    repositoryUnchanged(s, s')

    // Version brought up to date with repository (true for empty set too)
    s'.entries[s.event_client][s.event_file] =
        currentVersion(s, s.event_file)

    preserveRemoved(s, s')
    preserveModified(s, s')

    // (Pessimistically) flag a conflict if the the server version has
    // increased and the event-file has been locally modified
    eventFileIsModified(s) && isFileStale(s, s.event_client, s.event_file) =>
        setConflicts(s, s'), preserveConflicts(s, s')
}

fun legalModifyEvent(s, s' : State)
```

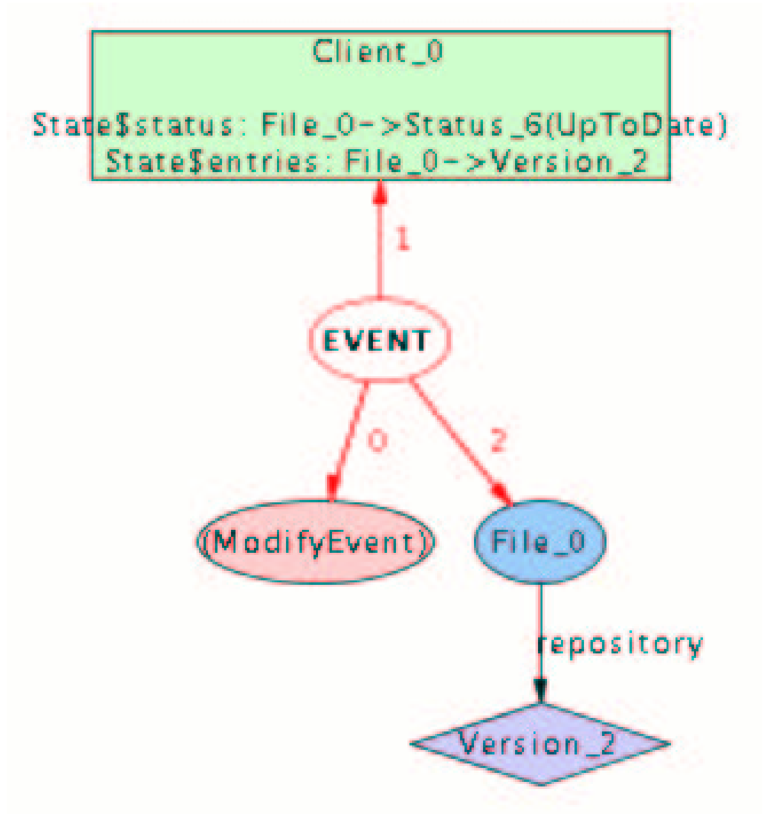
```
{
    // -- preconditions --

    // file must already be in repository:
    one currentVersion(s, s.event_file)

    // Not marked for removal (removed => no file on disk)
    // [XXX Incorrect: might be newly added. If this is the case we
    // want the ModifyEvent not to change its status.]
    s.event_file !in s.localRemoved[s.event_client]

    // -- postconditions --

    repositoryUnchanged(s, s')

    // local entries unchanged
    s'.entries[s.event_client] = s.entries[s.event_client]

    setModified(s, s')
    preserveRemoved(s, s')
    clearConflicts(s, s') // (optimistically) assume that user clears conflicts
}

// XXX Incorrect:
fun legalAddEvent(s, s': State)
{
    // -- preconditions --

    // CVS will not let you add a file (to the local CVS/Entries) if
    // the same file exists at the server. It issues the warning: "cvs
    // server: <file> added independently by second party".
    no currentVersion(s, s.event_file)

    // either it's a new file (no local or repos entries)
    let v = s.entries[s.event_client][s.event_file] |
        no v || (one v && s.event_file in s.localRemoved[s.event_client])

    // -- postconditions --

    repositoryUnchanged(s, s')

    s'.entries[s.event_client][s.event_file] = Ord[Version].first

    clearConflicts(s, s')
    clearRemoved(s, s')
    clearModified(s, s')
}

fun legalRemoveEvent(s, s': State)
{
    // -- preconditions --

    // can't remove if "removed" flag already set!
    s.event_file !in s.localRemoved [s.event_client]

    // -- postconditions --

    repositoryUnchanged(s, s')

    let cli = s.event_client |
    {
        setRemoved(s, s')
        clearModified(s, s')    // removed files are considered unmodified
        clearConflicts(s, s')

        s'.entries[cli]   = s.entries[cli]      // local entries unchanged
    }
}

fun legalCheckinEvent(s, s': State)
{
    let f = s.event_file |
    let modified = s.localMods[s.event_client] |
    let removed = s.localRemoved[s.event_client] |
    let added = s.entries[s.event_client].Version - s.repository.Version |
    {
        // -- preconditions --

        // must be in exactly one category:
        f in added || f in removed || f in modified

        // can only check in modified if repository/client versions agree:
        f in modified =>
            currentVersion(s, f) = s.entries[s.event_client][f]

        // can't check in a file with conflicts
        f !in s.localConflicts[s.event_client]

        // -- postconditions --

        f in removed =>
        {
            no currentVersion(s', f)            // remove from repository,
            no s'.entries[s.event_client][f]    // and from CVS/Entries
        }
        f in modified =>
        {
            // increment version
            currentVersion(s', f) = OrdNext(currentVersion(s, f))
            some currentVersion(s', f) // ensure we don't run out of Versions!
        }
        f in added =>
        {
            // added at first version [XXX problem: add, ci, remove, ci, add]
            currentVersion(s', f) = Ord[Version].first
        }

        // (works even for empty-set)
        s'.entries[s.event_client][f] = currentVersion(s', f)

        clearRemoved(s, s')
        clearModified(s, s')
    }
}

// --- Behavioural assertions --- //

// XXX Incomplete.  We want to write a user-level consistency
// constraint, using only the "public" interface relations, "status_p"
// and "event_p", to describe the changing status of files in terms of
// events received, without looking at the representation.

fun assertStatusChanges(s, s': State)
{
    let ev    = s.event_type |
    let stat  = s.status_p [s.event_client][s.event_file] |
    let stat' = s'.status_p[s.event_client][s.event_file] |
    {
        ev : UpdateEvent => {
            stat  = UpToDate + LocallyModified + HadConflicts
            stat' = UpToDate + LocallyModified
        }
        ev : ModifyEvent => {
            stat  in LocallyModified + HadConflicts + UpToDate
            stat' in LocallyModified + HadConflicts
        }
        ev : AddEvent => {
            stat  = NotCheckedIn
            stat' = LocallyAdded
        }
        ev : RemoveEvent => {
            stat in LocallyModified + UpToDate + HadConflicts
            stat' = LocallyRemoved
        }
        ev : CheckinEvent => {
            stat in LocallyAdded + LocallyModified + NotCheckedIn =>
                stat' = UpToDate
            stat in LocallyRemoved =>
                stat' = NotCheckedIn
        }
        // XXX
    }
}

// --- Main stuff --- //

fact FullSets
{
    // No trivial solutions please
    File    = univ[File]
    Client  = univ[Client]
    State   = univ[State]
}

// Constraints to filter out legal but uninteresting solutions:
fact Pretty
{
    // don't modify files that are already modified (effect is a no-op)
    all s: State |
        s.event_type = ModifyEvent <=>
            s.event_file !in s.localMods[s.event_client]

    // don't do updates to files that aren't stale
    all s: State |
        s.event_type = UpdateEvent <=>
            isFileStale(s, s.event_client, s.event_file)

    // NonEvent is used only in the last state. That's because Alloy
    // might otherwise put a different event type in the last slot, but it
    // can't check it's preconditions. NonEvent has no preconditions.
    all s: State | s.event_type = NonEvent <=> s = Ord[State].last
}

fun legal()
{
    all s: State - Ord[State].last |
        let s' = OrdNext(s) |
            legalEvent(s, s')
```

```
//  XXX We want to test this assertion, rather than state it as fact:
//  assertStatusChanges(s, s')
}

// ---- Test Scenarios ---- //

// Scenario helper functions:
det fun firstState()  : State { result = Ord[State].first }
det fun secondState() : State { result = OrdNext(firstState()) }
det fun thirdState()  : State { result = OrdNext(secondState()) }
det fun fourthState() : State { result = OrdNext(thirdState()) }

fun Scenario1()
{
    // Initial state: everything is up-to-date + clean
//      let firststate = Ord[State].first | {
//          no firststate.localConflicts
//          no firststate.localMods
//          no firststate.localRemoved
//          firststate.status_p[Client][File] = UpToDate
//      }

    // XXX NeedsCheckout, LocallyAdded are problematic

//    NeedsCheckout /* LocallyAdded*/ in State.status_p[Client][File]

    legal()
}
run Scenario1 for 5 but 2 File, 2 Client, 9 Status, 6 Event

fun Scenario2()
{
    some s: State, f: File, c: Client | {
        s.status_p[c][f] = UpToDate
        some s' in OrdNexts(s) | s'.status_p[c][f] = NeedsMerge
    }

    legal()
}
// run Scenario2 for 7 but 1 File, 2 Client, 9 Status, 6 Event

fun Scenario2a()
{
//    some s: State, f: File | NeedsMerge in s.status_p[Client][f]


//      }
//    some s: State, f: File, c:Client | {
//      HadConflicts in s.status_p[c][f]
//      some OrdNexts(s).status_p[c][f] - HadConflicts - NeedsMerge - LocallyRemoved
        // ie it eventually got out of it
//      }
}
// run Scenario2a for 7 but 1 File, 2 Client, 9 Status, 6 Event

fun Scenario3()
{
    some c1, c2 : Client {
        c1 != c2
        firstState().event_type = AddEvent // doesn't work yet
        firstState().event_client = c1

        secondState().event_type = ModifyEvent
        secondState().event_client = c1

        thirdState().event_type = CheckinEvent
        thirdState().event_client = c1

        fourthState().event_type = UpdateEvent
        fourthState().event_client = c2
    }

    legal()
}
//run Scenario3 for 7 but 1 File, 2 Client, 9 Status, 6 Event

/*
 * Local Variables:
 * mode: alloy
 * compile-command: "java -cp alloy.jar alloy.cli.AlloyCLI -E -m methods.als"
 * End:
 */
```

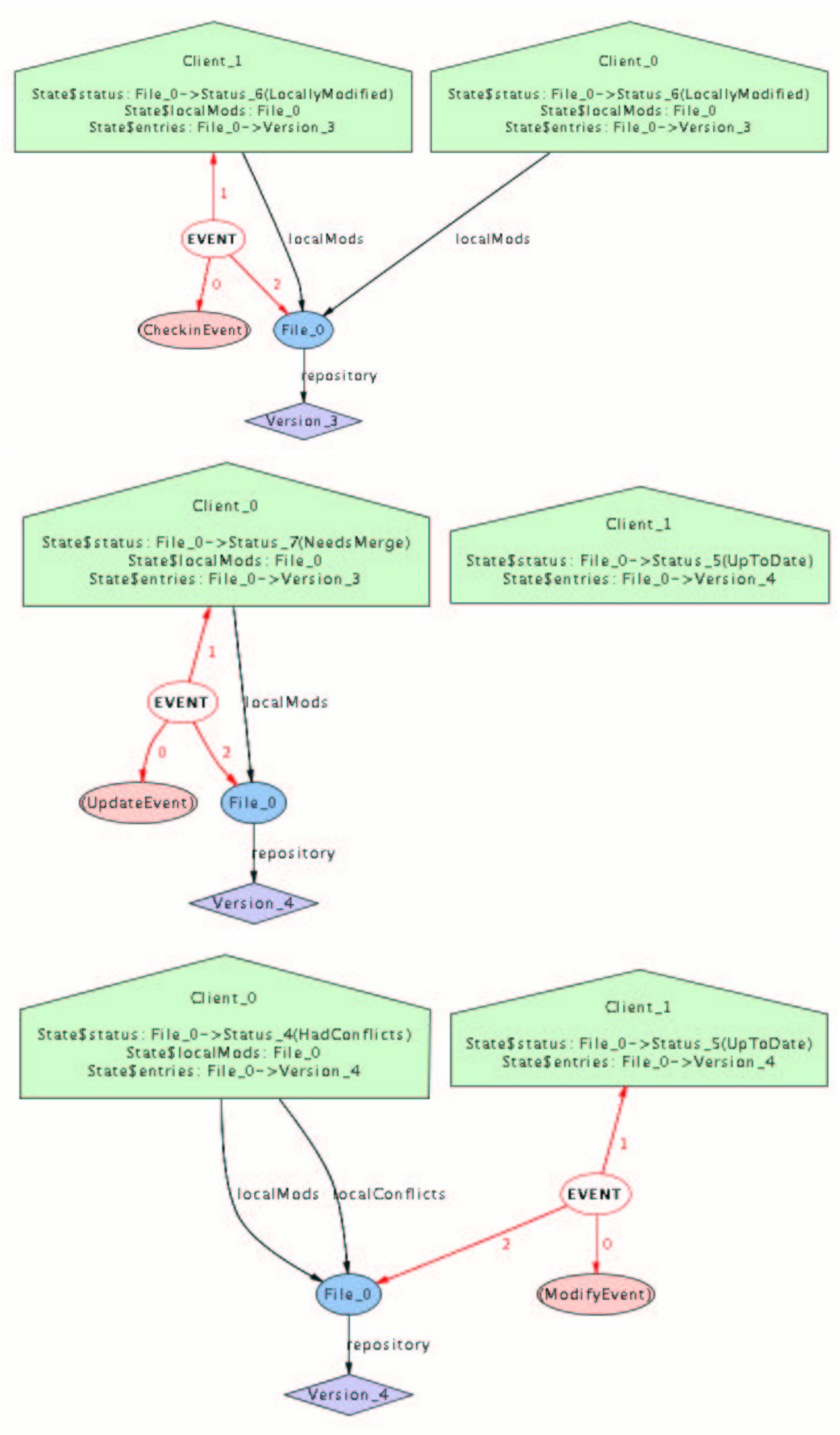
Appendix D: Visualisation of user-level model

D.1 Single clients, simple modify/check-in




Client_0
State$status: File_0->Status_6(UpToDate)
State$entries: File_0->Version_2
1
EVENT
0
2
ModifyEvent
File_0
repository
Version_2

D.2 Multiple clients, conflicting update


Client_1
State$status: File_0->Status_6(LocallyModified)
State$localMods: File_0
State$entries: File_0->Version_3
Client_0
State$status: File_0->Status_6(LocallyModified)
State$localMods: File_0
State$entries: File_0->Version_3
EVENT
1
0
2
localMods
localMods
CheckinEvent
File_0
repository
Version_3
Client_0
State$status: File_0->Status_7(NeedsMerge)
State$localMods: File_0
State$entries: File_0->Version_3
Client_1
State$status: File_0->Status_5(UpToDate)
State$entries: File_0->Version_4
EVENT
1
0
2
localMods
UpdateEvent
File_0
repository
Version_4
Client_0
State$status: File_0->Status_4(HadConflicts)
State$localMods: File_0
State$entries: File_0->Version_4
Client_1
State$status: File_0->Status_5(UpToDate)
State$entries: File_0->Version_4
localMods
localConflicts
EVENT
1
2
0
File_0
ModifyEvent
repository
Version_4