Title: **A Simple Model of Filesystem Synchronization**

Tina Nolte 6.898 Final Project

Abstract:
There are a number of software tools that offer the ability for a user to synchronize data in the face of conflicting updates on the part of multiple users, but most often it is unclear what properties and policies these tools employ, as well as difficult to understand what the policies might guarantee in the synchronized filesystems. Alloy is used in an exploration of filesystem synchronization properties and algorithms. The Alloy language has a fully automatic simulation and checking tool which is used to interactively construct a model of filesystem synchronization as well as understand intricacies associated with synchronization policies.

1 Background and Motivation

There are a number of software tools that offer the ability for a user to synchronize data in the mobile computing world, but most often it is unclear what properties and policies these tools employ, as well as what the policies might guarantee in the synchronized filesystems. These are important considerations since no user wants to discover that after synchronizing their mobile computer's data with another user's mobile computer they have lost whole files that may have been the product of hours of work. Even more subtle, users don't want to look at the synchronized filesystem and see that the directory structure and directory contents are as expected and continue with the belief that files have the changes they made to them only to find that some files have been replaced with a file of the same name from another user's filesystem.

File synchronization can be split into two different tasks: *update detection* and *reconciliation*. Update detection is the identification of updates made to different replicas since the last synchronization. Reconciliation uses this information to produce new filesystems that take into account conflicting updates and reflect the updates made to the filesystems that are non-conflicting, where conflicting updates are those where the contents of modified filesystems are different.

2 Summary and Evaluation

The first thing to model when modelling filesystem synchronization is, of course, filesystems. I have three different kinds of atoms: Names, Paths, and Nodes. Names are what you expect; they name directories and files. Paths are sequences of names. Nodes are either Files or Filesystems. Of interest is the departure from the standard recursive filesystem description where filesystems are partial mappings from names to files or other filesystems. Instead, filesystems are partial functions mapping whole paths to contents, either a file or another filesystem.

Next is update detection. One description of update detection is as follows: Say filesystem S is an update of filesystem O. $dirty_S$ safely estimates updates from O if $p \notin dirty_S => O(p) = S(p)$. Also, it is convenient for reconciliation to have dirtiness be upclosed; if a path q is dirty then any pathprefix of q is also dirty.

One of the most exciting modelling challenges was reconciliation. Reconciliation, obviously, is recursive in nature. I did write a "true to a published algorithm" description of reconciliation, but it required some fudging, namely chaining a series of copies of the reconciliation function together to the depth that I needed for any given test. However, even this was difficult to do correctly since the algorithm did a lot of chained computations that would go something like, "when you hit a directory while reconciling, you order the children of the directory and reconcile each child one after the other, passing the results of each synchronization into the next." This was great to do since it feels very close to the algorithm that is employed. However, I had difficulties getting descriptions that did a great deal of passing to even compile. Hence, I needed a more declarative description which had to include statements such as ones constraining solutions to not change portions of the filesystems that should be left alone.

However, for a more elegant model that did not require recursion cheating I needed a different kind of description. I found one solution in introducing a new kind of atom: a synchronizer. This contained a relation that would map triplets of filesystems corresponding to an original and two updates as well as a path to a new pair of filesystems. This model would have to constrain the relation defined as part of a synchronizer to relate the results of reconciling a particular path in two filesystems to the results of reconciling its children in the same filesystems. What was key here was noting that there was a relationship between these things.

I stated and verified using the analyzer several properties about synchronization, including that there can only be one maximal synchronization of filesystems A and B off of an original filesystem O and that the published reconciliation algorithm employed by Unison calculated that unique maximal synchronization. Some others are:

• First is a property of the update detector: if a given portion of a filesystem is not found as dirty in either of two modified replicas then that portion must be the same in both replicas.

• Second is a property of synchronizations: given two replicas of an original filesystem and a particular rule for determining dirtiness, there can only be one pair of filesystems satisfying the requirements of synchronization.

• Third is a property of reconciliation: given two replicas of an original filesystem and a particular rule for determining dirtiness, reconciliation produces synchronized versions of the input filesystems.

What was perhaps more interesting was that the Unison synchronizer itself (based on formalized notions of filesystem synchronization) has an informal description in its user's manual that suffers from ambiguity, leading to an easily believed incorrect understanding of what Unison does. On another note, I discovered a very operational description buried deep in Microsoft's website for their Briefcase program that is installed on many Windows machines. Upon modelling what the description says I discover that it is the same as what Unison claims to do. However, there are many published examples of ”unusual” behaviour by Briefcase that actually demonstrate that while the spec might be right, the implementation doesn't have to abide by the spec.

3 Lessons

I've already discussed some difficulties relating to recursion. Making a general model without need for manual unfolding required use of a great deal of additional reasoning which made the entire verification task much more time consuming as well as difficult.

After I had models that did not have find counterexamples for any of my assertions, I decided to start examining the examples that the model generated. I saw examples of successful recursive reasoning and then I noticed something a little odd; there were no files in any of my models... just directories and subdirectories. I asked the analyzer to show me an example with a file and none were produced. There was a small and almost insignificant error in the description of filesystems that was even present in the published formalizations and which disallowed the existence of files. I then added a new assertion just for fun to make sure that all examples of filesystem synchronization were being tackled and that I wasn't overconstraining the model in any way. Of course, I was overconstraining the model. Moral of the story: be sure to have a lot of error checking for yourself... you aren't just verifying that an algorithm is correct, you are also verifying that your model is.

In a paper describing behaviour of Unison, maximality of reconciliations is described by saying that over all filesystems that are partial synchronizations of input filesystems, if a particular path is propogated as non-conflicting in some partial synchronization then that path is propogated in the maximal version. This gives uniqueness. However, when I modelled this I actually tried to write that exact statement: for all filesystems that are partial synchronizations the result of Unison's recon algorithm is maximal w.r.t. them. Unfortunately, this is not true since I had confused universes. When the statement is made in a paper, the universe of filesystems is all filesystems. However, in Alloy, it is restricted by the number of atoms I am considering. Hence, I would get counterexamples in a scope of 3 that showed me that the maximal reconciliation in that restricted universe was not generated by the Unison recon algorithm. However, this is because a maximal synchronized version of the filesystem in the real world would require more than 3 nodes in the universe.

Memory requirements were problematic. I could only check my models up to five atoms. During the course of my work, I received suggestions from Daniel and Manu that helped me cut down on the number of Boolean nodes that were needed by simple tricks like passing results into my uses of Alloy functions and avoiding nondeterminism.

Finally, I found the "how much is enough" question difficult to answer since the tendency is to do more or describe something just a little differently. As a result, I am currently standing at 19 separate models, each demonstrating some approach or combination of approaches that are different from the other models, which is difficult to manage.

References

[1] S. Balasubramaniam and B. C. Pierce. File synchronization. Technical Report 507, Computer Science Department, Indiana University, Apr. 1998.

[2] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Oct. 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[3] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering, Special Issue on Formal Methods in Software Practice*, 23(5), May 1997.

[4] D. Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Reference Manual; available through http://sdg.lcs.mit.edu/alloy/, 2001, 2002.

[5] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, Sept. 2001.

[6] D. Jackson and K. Sullivan. Com revisited: Tool assisted modelling and analysis of software structures. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, Nov. 2000.

[7] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Sept. 2000.

[8] B. C. Pierce and J. Vouillon. Unison: A file synchronizer and its specification. Technical report; available through http://www.cis.upenn.edu/ bcpierce, 2001.

[9] The PVS Specification and Verification System. http://pvs.csl.sri.com, April 2002.