

Analysis, Modeling and Testing of Darcs Patch Theory Kernel

Formal Methods in Software Engineering
Department of Informatics
University of Minho

Iago Abal

`iago.abal@gmail.com`

June 5, 2011

Abstract

Darcs is a distributed version control system based on a unique algebra of patches called Patch Theory. Patch Theory was originally described informally by David Roundy and the formalisation of Darcs basis continue to be an unresolved topic nowadays, which keeps open the question of Darcs reliability since it may fail in some corner cases. In this work, I follow a different approach, instead of look for a formalisation which fits Darcs implementation, I propose to capture how Darcs works and verify if it satisfies the most well-known properties of Patch Theory. Next, I analyze Darcs in two levels. First, I present an analysis of Darcs maintainability identifying some important Darcs weaknesses. Finally, I measure the coverage of an important Darcs test suite showing that there is lot of room for improvement, and I show how to improve this coverage substantially.

1 Introduction

Darcs is a distributed version control system (dVCS) which is famous for its cherry-picking capabilities and because, unlike most VCS, it has a (more or less) mathematical background: it is based on a unique algebra of patches called Patch Theory. Patch Theory was invented by David Roundy in 2002, who also originally developed Darcs as a proof of concept of his theory of patches. Despite its name, Patch Theory was only informally described by David Roundy[2], and this description is very incomplete being insufficient to fully understand how Darcs works — specially with respect to conflicts handling. During the past years there was several attempts to formalize Patch Theory, some of these attempts are just descriptions a little more precise (but still informal) than the original, whilst others are based on well-established mathematical concepts[3]. However, after 10 years since first Darcs version was released, there is not still a complete formalization of Patch Theory that can be considered as the actual formalism behind Darcs. These facts prevents one to trust in Darcs reliability because it is based in a complex¹ and non-formalized theory which, moreover, is *supposed*² to work only in the case that Darcs implementation would satisfy some specific properties — and there is no real evidence of this.

This is not a new attempt to formalize Patch Theory. Instead, the first goal of this project is to capture how the actual Darcs Patch Theory implementation works by reverse engineering and verify if those properties, that are (supposedly) needed to get things working, hold. I will use Alloy³ for this purpose, which is a lightweight modeling language based on relational calculus and designed for fully automatic analysis. In summary, Darcs implementation behavior will be captured as an Alloy specification and then the Alloy Analyzer will be used to modelcheck the desired properties.

Once the verification is done the focus will be the analysis and improvement of Darcs implementation. First, I will analyze the maintainability of Darcs 2.5 following a method proposed by the Software Improvement Group (SIG)⁴, which is an experienced company in this field. Next, I will go delve into how Patch Theory implementation is being tested in Darcs, and how this testing could be improved. For the purpose of testing we will use QuickCheck[7], a tool for randomized testing in Haskell. Randomized testing is a more recent approach to software testing in which you write generic properties⁵ about your functions, that are then tested with multiple random test cases supplied by a set of test case generators.

This document is organized as follows. Section 2 introduces the key concepts of Patch Theory and describes how these concepts were modeled into Alloy. Section 3 gives a detailed explanation of the maintainability analysis of Darcs 2.5. Section 4 presents a coverage study of an important Darcs test suite and describes how this coverage was improved by (primarily) fixing or reimplementing some QuickCheck generators. Finally, Section 5 gives the main conclusions of this work.

¹Very few people fully understand some advanced parts of Patch Theory.

²There is no formal proof of that.

³<http://alloy.mit.edu>

⁴<http://www.sig.eu>

⁵Tests parametrized in the test data.

2 Modeling Darcs Patch Theory in Alloy

6

The concept of patch is the key concept of Patch Theory; as described by David Roundy in the Darcs manual⁷:

A patch describes a change to the tree. It could be either a primitive patch (such as a file add/remove, a directory rename, or a hunk replacement within a file), or a composite patch describing many such changes.

where here *tree* stands for repository tree; that is, the subtree of the filesystem which is under revision control. Primitive patches describe the most basic changes that could be made to a tree, and these are composed into sequences describing any other (arbitrarily complex) type of change. During this project we only deal with primitive patches and patch sequences are left as future work.

2.1 Repository tree

So, since patches are changes to a repository tree we will first model those trees and then we will come back to the question of patches. A tree contains files and directories identified by their path, where a path is defined as a pair of a parent path and a name relative to it; a name is just something abstract we do not care about. Note that there is no root path; if a path has no parent then it refers to a directory or file located in the root. A tree also contains a mapping of files (by path) to their content, where the content of a file is defined as a sequence of lines.

```
sig Name, Line {}

sig Path {
  parent : lone Path,
  name   : one Name
}

sig Tree {
  Dirs   : set Path,
  Files  : set Path,
  content : Path -> (seq Line)
}
```

We don't model the concept of a filesystem object because files and directories will be exclusively handled through their path. Paths are extremely useful because they can be manipulated and compared without the need of a tree acting as context, for instance, we can know if a file is contained in a directory just checking if the path of the directory is a prefix of the path of the file. This ability to compare and manipulate paths is crucial to model patches as we shall see later.

The use of paths to handle files motivates the modeling of trees in a non-standard way with respect to typical presentations of filesystem models in Alloy. Note that we don't have an hierarchy of objects defined by a *parent* relation; instead, our tree model just "store" paths and the hierarchy between files and directories is implicitly defined by those paths. We shall see that the application of patches requires the ability to access a file through its path, which is a trivial operation with our tree model but it is far from simple when working with a standard model of filesystem tree — since Alloy does not support recursive functions.

Another thing that deserves more explanation is the use of sequences to model the content of a file. A sequence *seq a* is just a relation $seq/Int \rightarrow a$ plus an axiom enforcing that the mapping contains no holes. The type *seq/Int* defines a subrange of integers whose elements are always greater than zero and strictly lesser than the biggest integer. Alloy sequences are the natural abstraction of Haskell lists, which makes them very convenient.

Of course, we need to enforce some invariants in order to keep trees sensible: *a*) a path is either a file or a directory, *b*) the parent of an item can only be a directory, and *c*) only files are mapped to content.

```
pred Inv[t : Tree] {
  no (t.Dirs & t.Files)
  all x : t.(Dirs+Files) | x.parent in t.Dirs
  t.content in t.Files -> (seq Line)
}
```

As in Darcs implementation patches won't be allowed to manipulate the tree directly. Instead, we offer an API inspired in the one offered by Darcs through the *ReadableDirectory* and *WritableDirectory* type classes defined in *Darcs.IO*.

```
pred isEmptyDir[t : Tree, d : Patch]
pred CreateDir[t : Tree, d : Path, t' : Tree]
pred RemoveDir[t : Tree, d : Path, t' : Tree]
pred CreateFile[t : Tree, f : Path, t' : Tree]
pred RemoveFile[t : Tree, f : Path, t' : Tree]
fun readFile[t : Tree, f : Path] : (seq Line)
pred WriteFile[t : Tree, f : Path, text : seq Line, t' : Tree]
pred Rename[t : Tree, src : Path, dest : Path1, t' : Tree]
```

⁶Joint work with João Melo.

⁷<http://darcs.net/manual>

2.2 Patch hierarchy

Now we can continue describing how we have modeled patches. Darcs considers a total of nine primitive patch types: *a) AddFile* patches to add a new empty file; *b) RmFile* patches to remove an empty file; *c) Hunk* patches to edit a portion of a file by removing a set of lines and adding another set of lines, starting in a specific line number *d) TokReplace* patches to replace all instances of a given token within a file with some other version; *e) AddDir* patches to add a new empty directory; *f) RmDir* patches to remove an empty directory; *g) Move* patches to rename a file or directory; *h) Binary* patches to replace the content of a binary file; and *i) ChangePref* patches to change repository preferences. Darcs implementation, shown below, also considers two abstract patch types *FilePatch* and *DirPatch* to group those patches that work on a single file or directory, respectively. These abstract patch types have no theoretical usefulness but they allow implementation shortcuts for some operations on patches as we will see later.

```
data Prim where
  Move :: !FileName -> !FileName -> Prim
  DP :: !FileName -> !DirPatchType -> Prim
  FP :: !FileName -> !FilePatchType -> Prim
  ChangePref :: !String -> !String -> !String -> Prim

data FilePatchType = RmFile | AddFile
                  | Hunk !Int [B.ByteString] [B.ByteString]
                  | TokReplace !String !String !String
                  | Binary B.ByteString B.ByteString
                  deriving (Eq,Ord)

data DirPatchType = RmDir | AddDir
                  deriving (Eq,Ord)
```

Modeling primitive patches in Alloy is almost straightforward and, since Alloy supports subtyping, the encoding of patches is even more natural than in Haskell. We omit *TokReplace*, *Binary*, and *ChangePref* patches mainly for simplicity; *TokReplace* patches would be particularly tricky to model. *ChangePref* is a special case because the implementation of these patches is known to be buggy, and they break most interesting properties about *commute*⁸.

```
abstract sig Patch {}

abstract sig DirPatch extends Patch {
  path : Path
}

sig Adddir, Rmdir extends DirPatch {}

abstract sig FilePatch extends Patch {
  path : Path
}

sig Addfile, Rmfile extends FilePatch {}

sig Hunk extends FilePatch {
  line : Int,
  old : seq Line,
  new : seq Line
}

sig Move extends Patch {
  source : Path,
  dest : Path
}
```

We canonize patches (by canonizing non-abstract patch types) to force that there not exist two patches describing the same change, which can be considered as our translation of the Darcs Eq⁹ instance for primitive patches. Although this canonization is not strictly needed it is very convenient: some operations on patches become injective and we can compare patches simply using the = operator.

```
fact CanonizeMove {
  no disj mv1, mv2 : Move | mv1.source = mv2.source and mv1.dest = mv2.dest
}
```

Without any additional restrictions patches may describe non sensible changes such as move a directory into one of its offspring (e.g. mv /home /home/foo), or others less obvious such as edit a file out of the filesystem limits (provided by seq/Int in this case). In Darcs the differ is assumed to create sensible patches only, but any operation which manipulates patches should preverse this invariant.

```
pred Inv[p : Patch] {
  p in Hunk => p.HunkInv
  p in Move => p.MoveInv
}

pred HunkInv[line : Int, old : seq Line, new : seq Line] {
  — Hunk does not try to add/remove content out of file limits.
  line in seq/Int
  old.lastIdx.add[line] in seq/Int
  new.lastIdx.add[line] in seq/Int
}
```

⁸If the bug were solved then different Darcs versions could “see“ different dependencies between patches in a Darcs2 repository, so it won’t be fixed until Darcs 3. Note how critical is this code for Darcs.

⁹The Haskell type class for types supporting equality.

```

}

pred MoveInv[mv : Move] {
  not isPrefix[mv.source, mv.dest]
  not isPrefix[mv.dest, mv.source]
}

```

2.3 Application of patches

A patch is applied to a repository tree by materializing the change described by the patch through the API offered to manipulate trees. Application of most types of patches is very straightforward and consists on a single call to a API function; application of hunks is a bit more tricky though. In Darcs, pattern matching is used to determine the type of a patch in order to apply it;

```

apply (FP f RmFile) = mRemoveFile f
apply (FP f AddFile) = mCreateFile f
apply (Move f f') = mRename f f'
...

```

in our model we use an encoding which recalls method dispatching in object-oriented languages to reproduce the same semantics. Note we had to add restrictions to force patch arguments to be of the desired type, since Alloy completely ignore types when evaluating the model which can lead to some unexpected behavior.

```

pred Apply[t : Tree, p : Patch, t' : Tree] {
  ApplyDirpatch[t,p,t'] or ApplyFilepatch[t,p,t'] or ApplyMove[t,p,t']
}

pred ApplyDirpatch[t : Tree, dp : DirPatch, t' : Tree] {
  dp in DirPatch
  ApplyAdddir[t,dp,t'] or ApplyRmdir[t,dp,t']
}

pred ApplyAdddir[t : Tree, add : Adddir, t' : Tree] {
  add in Adddir
  CreateDir[t,add.path,t']
}

```

There are some definitions based on patch application that are of interest for verification purposes, despite they are ignored in Darcs implementation. We say that a patch is *sensible* if it can be applied to some tree; analogously, a pair of patches is sensible if they can be applied in sequence to some tree, in which case we say both patches are *sequential*. Moreover, the change described by a patch can be modeled as an injective and simple relation from trees to trees in terms of the *apply* operation.

```

pred isSensible[p : Patch] {
  some t : Tree | p.isApplicableTo[t]
}

pred sequential[p, q : Patch] {
  some t1, t2, t3 : Tree | t1.Inv and Apply[t1,p,t2] and Apply[t2,q,t3]
}

fun Effect[p : Patch] : Tree -> Tree {
  { t1, t2 : Tree | t1.Inv and Apply[t1,p,t2] }
}

```

As we shall see, the most useful of these definitions is *sequential* because it is used as precondition for one of the most important operations defined on patches. Nevertheless, the other two are also of interest; for instance, we have tried to prove that any patch satisfying PatchInv is a sensible patch.

```

assert EveryPatchIsSensible {
  all p : Patch | p.Inv => p.isSensible
}

```

Unfortunately Alloy always find a counterexample for this assertion, due to (what we will refer as) the *unsaturated universe problem*. This problem usually appears when verifying assertions of the form $\langle \forall x. P \Rightarrow \langle \exists y. Q \rangle \rangle$, because it will try to find a universe in which some x satisfying P exists but there is no y satisfying Q , and this is likely to happen since Alloy works with finite universes. In this particular case, it is unlikely that each one of the universes generated to test this assertion will only contain patches for which the universe also contains some tree that makes them sensible.

2.4 Inversion of patches

Darcs manual defines the inverse of a patch p , denoted as p^{-1} , as the “simplest” patch for which the composition $p^{-1}p$ makes no changes to the tree. The notion of simplicity for patches is not defined anywhere, but Darcs provides an implementation that looks senseful.

```

invert (FP f RmFile) = FP f AddFile
invert (FP f AddFile) = FP f RmFile
invert (FP f (Hunk line old new)) = FP f $ Hunk line new old
invert (Move f f') = Move f' f
...

```

The *invert* operation was translated following the same pattern used for patch application.

```

pred Invert[p, p-inv : Patch]

```

The property we have called *invert rollback* is the most important one because it captures the theoretical meaning of *invert*. It states that we can undo the changes made by a patch p if we apply its inverse patch p^{-1} to the resulting tree. For example, if a patch removes an empty file from the repository, we can undo this effect by applying a new patch which creates a new empty file at the same path.

```
assert Invert_Rollback {
  all p, p_inv : Patch, t, t' : Tree |
    (p.Inv and t.Inv and Apply[t,p,t'] and p.Invert[p_inv]) => Apply[t',p_inv,t]
}
```

Darcs manual and formalizations such as [3] state that every patch must be invertible, which makes lot of sense because that ensures that we will be able to rollback to any previous repository state. However, our attempts to prove this with Alloy were frustrated by the *unsaturated universe problem* mentioned above.

```
assert EveryPatchIsInvertible {
  all p : Patch | p.Inv => some p_inv : Patch | p.Invert[p_inv]
}
```

2.5 Patch commutation

We say that a pair of sequential patches $p; q$ commutes when we can find another pair $q'; p'$ such that $p; q \equiv q'; p'$, which in most Patch Theory formalizations is written as $pq \leftrightarrow q'p'$. That is, the change described by $q'; p'$ is the same that the one described by $p; q$. The names for the patches were not picked arbitrarily, the idea is that p' describes—in essence—the same change than p , but in a different context; and the same applies for q' and q . Commutation is a very important concept of Patch Theory because it captures dependencies between patches: two patches are said to be independent if they commute. By one hand, the ability to identify dependencies between patches is what gives Darcs its famous cherry-picking capabilities. By other hand, any bug or change in *commute* implementation is critical because it may change dependencies between the patch set of a repository.

Consider the example shown in Figure 1, the change performed by the pair $p; q$ is “delete orange and replace lemon by banana”. This patch pair commutes to the pair $q'; p'$ which has the same effect than $p; q$ but primitive changes are applied in different order. Although q and q' edit `fruit.txt` in different positions they are also applied in different contexts, q is applied to a `fruits.txt` containing lemon in its first line whereas q' is applied to a `fruits.txt` file in which lemon is in the second line, so in essence both describe the same change in their respective contexts which is “replace lemon by banana”.

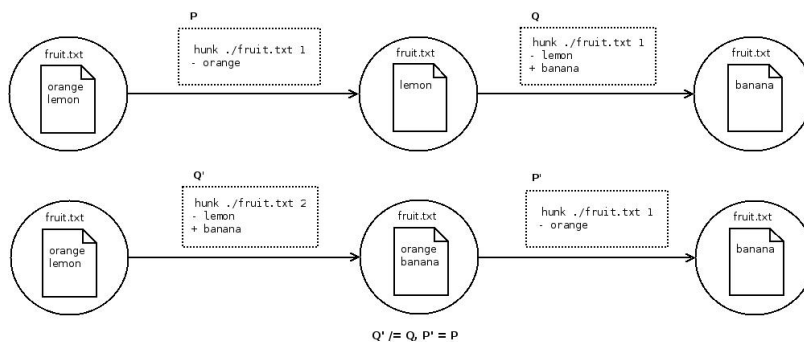


Figure 1: Example of patch commutation.

Our Alloy model tries to reproduce faithfully the Darcs commutation strategy for primitive patches defined in `Darcs.Patch.Prim.V1.Commute` (Darcs screened). The number of possible combinations for `commute` is quite high¹⁰, but there is only a small number of essential cases such that any other can be based on them. Darcs implementation uses some clever tricks in order to specify only those core cases. The first trick is to try to commute against abstract patch types before against concrete patch types so, for example, we can trivially commute two *FilePatches* modifying different files. Another more advanced trick relies on patch inversion and in the fact that the *commute* relation is symmetric; but this trick is not necessary a good idea in Alloy, recall that due to the finiteness of Alloy universes the inverse of a patch is not guaranteed to exist.

```
pred Commute[p, q, q', p' : Patch]
```

Commutation is only defined for sequential pairs of patches, and the result of commutation is always another sequential pair of patches. However, this cannot be verified in Alloy due to (again) the *unsaturated universe problem*.

```
assert Commute_Sequential {
  all p, q, q', p' : Hunk |
    (p.Inv and q.Inv and Commute[p,q,q',p']) => sequential[q',p']
}
```

The same problem arises in every property which contains a *commute* in the consequent of an implication, so we are force to verify weaker versions of some interesting properties by adding pair sequentiality as an hypothesis when necessary.

¹⁰Just consider all possible pairs of patch types.

```

assert Commute_Symmetric_Weak {
  all p, q, q', p' : Patch |
    Commute[p,q,q',p'] and sequential[q',p'] => Commute[q',p',p,q]
}

```

In the case of *commute* the property which captures its theoretical meaning is *commute effect preserving* — note how elegantly is it written when expressed in terms of patch effects. Unfortunately, since the definition of *Effect* also involve existential quantifiers, it cannot be verified due to the *unsaturated universe problem*.

```

assert Commute_EffectPreserving {
  all p, q, q', p' : Patch |
    (p.Inv and q.Inv and
     Commute[p,q,q',p']) => ((p.Effect) . (q.Effect)) = ((q'.Effect) . (p'.Effect))
}

```

Instead, we were only able to verify the much weaker version shown below. Nevertheless this version was very useful because it showed us that the manipulations of hunks line numbers made by *commute* may cause integer overflow in some situations as it is shown in Figure 2, which lead us to add checks preventing hunk commutation in case of overflow. This is in fact a real Darcs bug because Darcs uses the fixed-precision *Int* type to represent line numbers, however Darcs developers were not aware of it because this situation is unlikely to happen unless you are manipulating text files with billions of lines.

```

assert Commute_EffectPreserving_Left_Weak {
  all p, q, q', p' : Patch, t1, t2, t2', t3 : Tree |
    (p.Inv and q.Inv and t1.Inv and
     Apply[t1,p,t2] and Apply[t2,q,t3] and Apply[t1,q',t2'] and
     Commute[p,q,q',p']) => Apply[t2',p',t3]
}

```

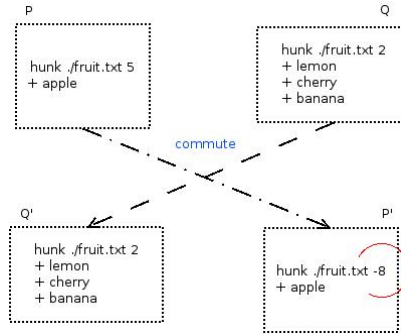


Figure 2: Overflow when commuting hunks (with 3-bit precision integers).

Since *commute* was not properly verified due to the limitations imposed by the finiteness of the universe, we have tried to apply a workaround consisting on the use of *generator axioms*. A *generator axiom* is a fact which is added to a model in order to force Alloy to generate a saturated universe. In our case we have forced Alloy to generate a universe which includes all possible trees containing at most one file with a maximum of three lines of content. Now, in these universes, we are able to verify stronger versions of *commute* properties such as *commute effect preserving*. Thanks to this we have found that *commute effect preserving* could be falsifiable due to the existence of filesystem limits.

```

assert Commute_EffectPreserving_Left {
  all p, q, q', p' : Hunk, t1, t2, t3 : Tree |
    (p.Inv and q.Inv and t1.Inv and
     Commute[p,q,q',p'] and Apply[t1,p,t2] and Apply[t2,q,t3])
    => some t2' : Tree | Apply[t1,q',t2'] and Apply[t2',p',t3]
}

```

A counterexample is shown in Figure 3: since q' adds content to *fruit.txt* it cannot be applied to those trees in which *fruit.txt* already has three lines because it would exceed the allowed file size limit, therefore we have found a tree for which $p; q$ is defined but $q'; p'$ is not.

3 Maintainability analysis of Darcs 2.5

The maintainability analysis was based on the method described in [5] but, since this article is focused on imperative and object oriented languages the following question arises: are the metrics and reference values given in [5] valid for a functional programming language such as Haskell? After an analysis of each of the proposed metrics I think they also make sense for measure quality of Haskell code. Nevertheless, the proposed methods to take those metrics, as well as the classification guidelines for them, could not be the most appropriate for a functional programming language. Hence, several tweaks were introduced to our reference method¹¹ in order to adapt it to the particular case of Haskell, as it will be described in the following. Many of the variations made

¹¹Many of them were suggested by Joost Visser (SIG) during an informal talk.

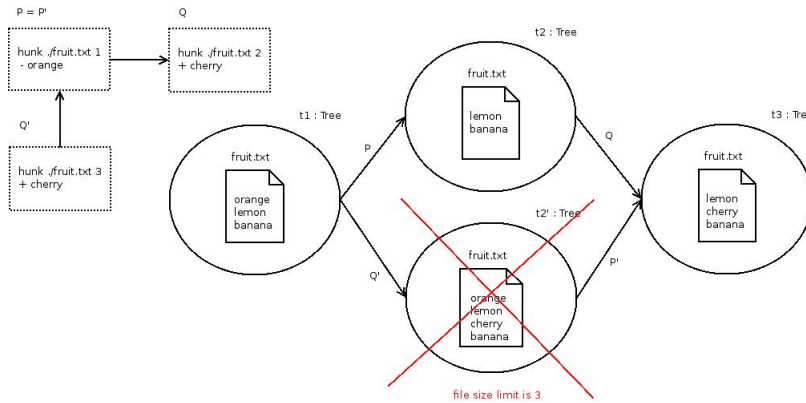


Figure 3: Counterexample of *commute effect preserving*.

to the reference method require special code support¹² which was developed thanks to the use of third-party libraries, most notably: *Haskell-Source with Extensions*¹³, *Uniplate*¹⁴ and *Scrap Your Boilerplate*¹⁵.

Once you introduce changes to the way metrics are taken and/or you change the classification guidelines for those metrics you no longer can be sure about the accuracy of the rates you obtain. Since I’m aware of this problem, but an statistical analysis over several Haskell projects is out of the scope of this work, I have decided to compare Darcs metrics with two other Haskell projects: XMonad¹⁶, a very small and (supposedly) high quality Haskell project; and GHC¹⁷, a big project which is the result of several years of development and it is (supposedly) hard to maintain.

3.1 Volume

Volume was measured as suggested by [5], that is, counting the number of KLOCs that are not comment or blank lines for any programming language involved, converting these KLOCs (through some given factors) to man years (MY), and summing volume in man years to finally obtain (through a given conversion table) a rate between one and five stars. KLOCs were easily counted thanks to CLOC¹⁸. Unfortunately, I have no access to a conversion factor between KLOC and MY neither for Haskell nor for other of the languages involved, so I had to obtain it indirectly through the conversion to KLOCs in an hypothetical third generation programming language provided by CLOC¹⁹. Once we know system KLOCs for this hypothetical language we can convert them to MY indirectly, using the same scale factors to obtain the corresponding KLOC for a language for which we know KLOC-MY conversion factor and then converting those KLOCs to man years. As an example, consider Darcs code which is formed by 27.11 Haskell KLOC, 1.47 C KLOC, 0.13 Perl KLOC and 0.11 C/C++ Header KLOC. In total, Darcs is 59.18 KLOC for the 3rd-gen. language considered by CLOC, and those KLOCs are equivalent to 39.19 C++ KLOC which correspond to 3.42 MY.

Despite this conversion process “makes sense”, I expect so many conversions will introduce lot of noise making the results very inaccurate, indeed the results are not the ones I have expected. As shown in Figure 4, Darcs has a five stars rate whilst GHC is considered four stars. But, in my opinion, GHC is a big²⁰ Haskell project and it should be rated as maximum with two stars; Darcs is a medium-size one so a more appropriate rate for it would be between three and four stars.

Certainly, to count KLOCs does not seem to be a good way to measure volume for a functional programming language because, in contrast with imperative/OO programming in which a LOC corresponds more or less with a single statement²¹, the use of nested function application and high-order functions make common to write complex expressions in one single (and even short) line. That is why I will use a different, which I judge more appropriate, code metric to measure volume in the following²², consisting in parse the source and count the number of nodes of the abstract syntax tree (AST). The reason because volume was not previously measured using this code metric is that it is not possible to establish a correspondence between AST nodes and MY based on my experience as Haskell programmer only, but an statistical analysis over a large sample of Haskell projects would be needed. It is interesting to note, that if you put both metrics together in a graphic as shown in Figure 5, it looks like both metrics were in some way correlated, and that a LOC is more or less equivalent to 5 AST nodes. However, to conclude this it would be necessary a much bigger sample of Haskell projects, and in the case that some correlation exists it may be caused by the use of similar style conventions.

¹²The code developed for this purpose is located in the `metrics/src/` directory; please, read the corresponding `metrics/README` file for further details about the code structure.

¹³<http://hackage.haskell.org/package/haskell-src-exts-1.10.1>

¹⁴<http://hackage.haskell.org/package/uniplate-1.6>

¹⁵<http://hackage.haskell.org/package/syb-0.3>

¹⁶<http://hackage.haskell.org/package/xmonad>

¹⁷www.haskell.org/ghc

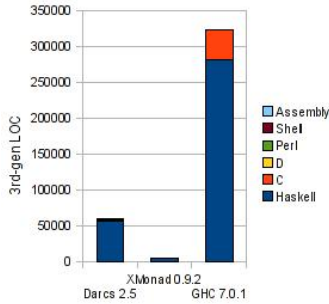
¹⁸<http://cloc.sourceforge.net/>

¹⁹http://cloc.sourceforge.net/#scale_factors explains how these scale factors were derived.

²⁰In fact it is perhaps the biggest Haskell project.

²¹In fact, write several statements in a single line is usually considered bad programming style.

²²There are some other code properties which are relative to the volume, such as *complexity per unit* or *unit size*.



	Haskell KLOC	3rd gen. KLOC	Man Years	Rate
Darcs 2.5	27.11	59.18	3.42	★★★★★
XMonad 0.9.2	1.9	4	0.23	★★★★★
GHC 7.0.1	133.55	324.61	18.76	★★★★★

Figure 4: KLOCs and Volume.

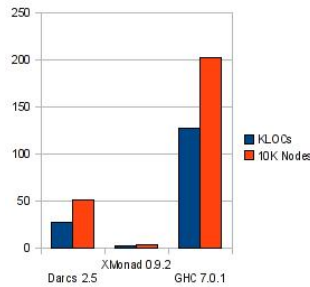
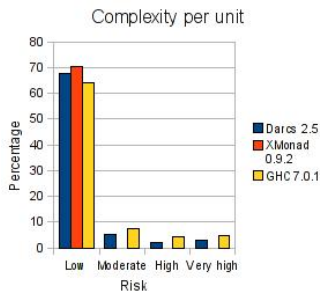


Figure 5: LOCs vs AST nodes.

3.2 Complexity per unit

Complexity per unit is calculated by determining the risk level of each code unit in terms of its cyclomatic complexity (CC), and then counting for each risk level what percentage of AST nodes falls within units categorized at that level. That is, I'm just following the same procedure described in [5] (including the same risk evaluation and rating tables) with the exception that I'm considering AST nodes instead of lines of code. The concept of code unit that I'm using for Haskell is basically the same than for imperative/OO languages: a unit is a top-level function or a type-class method instance.

Unfortunately I did not find any tool allowing to compute CC for Haskell programs which meets the requirements needed to be useful for this purpose. SourceGraph²³ was an interesting candidate but it considers modules to be the units of an application and so it does not provide metrics for specific functions or method instances; anyway, it is only able to generate a (not easy to parse) HTML report which makes it impractical for automated analysis. Finally I implemented a simple algorithm which computes a complexity measure using the abstract syntax tree, which coincides with cyclomatic complexity if no local definition²⁴ make use of alternative constructs²⁵. In case of multiple complex local definitions this measure is only an approximation to CC, and it considerably penalizes the complexity of those local definitions. Let me refer to this "almost cyclomatic complexity" measure as $\pm CC$ in the following.



	Moderate %	High %	Very High %	Rate
Darcs 2.5	5.04	1.8	2.85	★★
XMonad 0.9.2	0	0	0	★★★★★
GHC 7.0.1	7.55	4.04	4.87	★

Figure 6: Complexity per unit.

Figure 6 shows the complexity rates of our three considered applications. These rates reflect that XMonad is indeed an extremely simple application, whereas both Darcs and (specially) GHC have an excessive amount of complex code which negatively influences their maintainability. After a short analysis of the complex code units of both projects I was able to identify specific characteristics of these projects that highly influence their complexity rates. In the case of Darcs the abuse of local definitions is perhaps the most important cause of its complexity. Let us consider the function `treeDiff` in module `Darcs.Diff` as an example, this function involves 8 local definitions

²³<http://hackage.haskell.org/package/SourceGraph>

²⁴I am referring to Haskell `let` or `where` clauses.

²⁵Pattern matching, guards or `if-then-else` expressions.

where at least three of them could be top-level ones with no changes at all and have a \pm CC score above 2. Since these definitions are local to treeDiff they are also being considered to compute its complexity, as consequence, treeDiff is being classified as a very high risk unit when it could be a low risk one if those three definitions were made top-level. GHC seems to make a more reasonable use of local definitions, but being a compiler it deals with many large²⁶ data types which, when pattern matched, may result in definitions with a big complexity score. For instance, the Instr data type declared in module X86.Instr²⁷ has 70 data constructors. Although GHC is more complex than Darcs, it turns out that Darcs complexity could be substantially reduced just defining top-level functions instead of local ones whenever it makes sense, whilst in the case of GHC break the complexity of some units could be tricky, but perhaps generic programming may help.

3.3 Code duplication

Our reference method proposes to measure code duplication by calculating the percentage of all code that occurs more than once in equal code blocks of at least 6 lines. As I said early, in an imperative language a line of code is (more or less) equivalent to a statement, but it is a rather meaningless concept for a functional language. Due to this I have tried, in fist instance, to measure code duplication by searching duplicated blocks of words²⁸ instead of lines. Although this metric looked promised I end discarding it because I was not able to find a tool with native support for searching duplicates at word level. I have tried to do it by generating auxiliary files containing one word per line and then calling a standard clone detection tool, but this strategy makes the analysis of the reported duplicated blocks very cumbersome. Moreover, I were not able to find a good block (of words) size for this metric; if I search for relatively small blocks then I end by having lot of false positives due to functions sharing the same (or very similar) type signature, whilst if I use a more bigger block size then many interesting duplicated blocks escape to the analysis. Certainly the word-based clone detection could work well if type signatures were ignored and we had a specific tool for this purpose. Finally I decided to measure code duplication for Haskell as proposed by [5] but using blocks of 4 lines instead. The reason to use a smaller block size is the same given in subsection 3.1, that is, that a Haskell LOC is usually (semantically) richer than a single imperative statement, so it makes sense to consider smaller blocks. Figure 7 shows code duplication percentages and rates; this analysis was done —for both Haskell and C (GHC RTS)— with Simian²⁹.

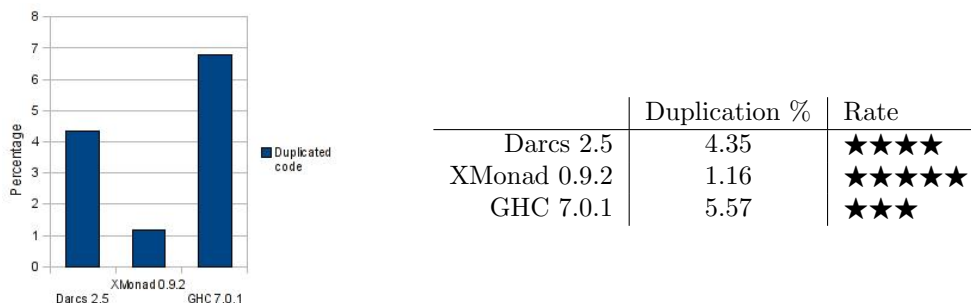


Figure 7: Duplicated code.

XMonad is loyal to its fame and has a very low percentage of duplicated code, though Darcs has also a good rate of four stars. The biggest source of duplication in Darcs is between modules IsoDate and Darcs.Patch.OldDate; they duplicated the IsoDate module in order to change the date format ensuring that they will continue supporting the old one³⁰. Some Darcs developers think that this duplication was a good way to handle this problem³¹ because it ensures that the old format won't be affected by new modifications, but this is no more than another symptom of low unit testing coverage. In the case of GHC I have found that most of its duplicated code could be also due to bad engineering decisions taken by their developers. For instance, inside the codeGen subfolder the *Cg** and *Stg** module families³² share about the 12% of their code (25% of duplicated code); both of them implement intermediate code generation into two different (but quite similar) abstract machines. In the case of GHC native code generation, I have observed a 18% of code duplication between the code for the different supported architectures. In the case of GHC RTS there is also a considerable amount of duplication between the code adding parallel support to the RTS and their non-parallel counterparts.

3.4 Unit size

Unit size is measured analogously to complexity per unit but classifying unit risk by the number of nodes of its AST. The risk classification guidelines are show in Table 1 which are completely based on my experience as Haskell programmer. Although the accuracy of these guidelines should not rely on my own experience only, an

²⁶In the sense of having lot of data constructors.

²⁷<http://www.haskell.org/ghc/docs/7.0-latest/html/libraries/ghc-7.0.1/X86-Instr.html>

²⁸Simply considering spaces and tabs as words separators.

²⁹<http://www.redhillconsulting.com.au/products/simian/>

³⁰See discussion <http://lists.osuosl.org/pipermail/darcs-devel/2011-January/012158.html>.

³¹Good practices suggest the way to handle this situation is to refactor common code for both date formats.

³²Most GHC code make no use of hierarchical modules so I am using the term “family” to refer to a set of modules that share the same prefix indicating their relationship.

statistical analysis to find out the most sensible risk bounds is out of the scope of this project. As an attempt to convince you that my risk bounds are more or less sensible I refer you to four Darcs units as representatives of each risk category: `writeAndReadPatch` in `Darcs.Repository` has 300 AST nodes being of low risk, `grabSSH` in `Ssh` has 561 AST nodes being of moderate risk, `fetchFileUsingCachePrivate` in `Darcs.Repository.Cache` has 1027 AST nodes being of high risk, and `applyToPop'` in `Darcs.Patch.Apply` has 1883 AST nodes being of very high risk.

AST nodes	Risk evaluation
0-300	small, without much risk
301-600	larger, moderate risk
601-1100	large, high risk
>1100	huge, very high risk

Table 1: Unit risk by size.

Unit size ratings are shown in Figure 8. Although I expected low rates for both Darcs and GHC due to the existing correlation between complexity per unit and unit size indicated in [5], I was surprised by the low rate assigned to XMonad. But the fact is that XMonad contains several large units with low complexity, such as windows in `XMonad.Operations`, which are also potentially hard to understand and test.

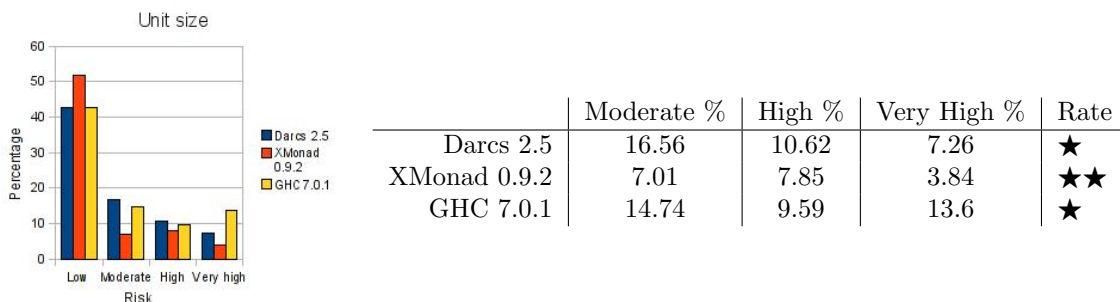


Figure 8: Unit size.

3.5 Module coupling

In addition to the code properties proposed by [5] I have also incorporated module coupling to this maintainability analysis. Module coupling tries to measure the degree to which each application module relies on each one of the other modules; this knowledge offers an estimation of how many code would be affected if we modify a given module. Low coupling is considered a sign of a well-structured computer system and a good design, supporting the general goal of maintainability[6]. Following the simplicity principles of our reference method I considered a very simple metric to measure module coupling.

Percentage of dependent code — For a given module I get the subset of the other modules that import some element from it and then I calculate the percentage of application volume that these modules represent. The risk level for the module is obtained by following the guidelines of Table 2. Let me illustrate this metric with an example: if a module *X* were imported by three other modules *A*, *B* and *C* representing the 15% of application code; then the module *X* is said to be of moderate risk according with Table 2 criteria.

Dependent code %	Risk evaluation
0-10	almost independent, without much risk
11-30	little coupled, moderate risk
31-60	coupled, high risk
>60	highly coupled, very high risk

Table 2: Coupling risk by percentage of code importing a module.

As well as for unit size, the risk classification guidelines for module coupling are completely based on my own experience and criteria. Figure 9 shows the module coupling rates for our three applications of interest. In this case the usual rating order was reversed, GHC has the lowest coupling rate whilst XMonad modules turn out to be extremely coupled. Certainly XMonad is highly coupled but, in my opinion, its module coupling rate should be weighted by its very small size. It is quite clear that coupling is in some way correlated with volume: the bigger an application is, the easier you can divide it into a set of almost independent logical blocks. In the case of GHC each compilation phase is an independent block, and any of these blocks is bigger than XMonad itself. In the case of Darcs, its most coupled modules are utility modules like `Printer` (for pretty-printing) or `ByteStringUtils`, and modules such as `Darcs.Patch` that only imports and re-exports elements from `Darcs.Patch.*` modules. All these observations suggest that the way I'm measuring module coupling is not as accurate as it should be, and hence

these rates should be taken with care. Despite this, it is not obvious to me how this property could be measured more accurately without making it too much complex.

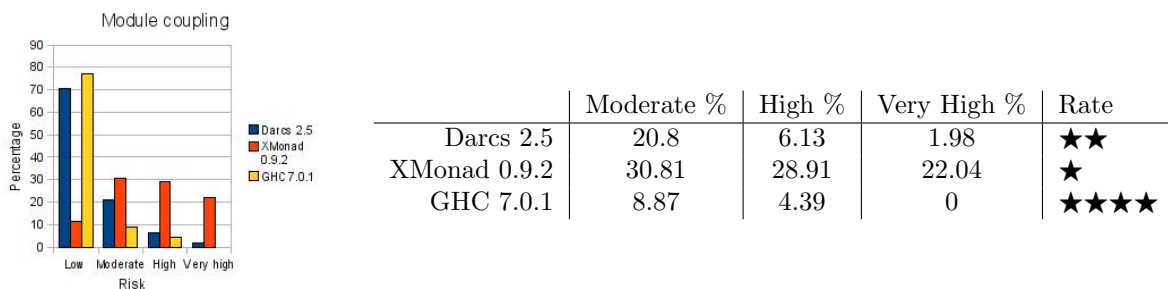


Figure 9: Module coupling.

3.6 Unit testing

Unit test coverage is a completely language independent metric, and so both the metric and the rate guidelines are perfectly valid for Haskell. To measure Darcs unit test coverage I have used the Haskell Program Coverage³³ (HPC) tool provided by GHC. Since Darcs unit tests cover the 29%³⁴ of Darcs expressions³⁵ it has, following [5] guidelines, a unit testing rate of ★★ (two stars). Note that unit testing was being measured exactly as indicated by our reference method, so there is no need for a comparison and that is why I did not measure unit testing neither for XMonad nor for GHC.

3.7 Maintainability rates

Table 3 shows Darcs final scores for sub-characteristics of maintainability and for maintainability itself. To allow some degree of comparison I have also included the final scores for XMonad and GHC, though many of these scores³⁶ may vary once included the unit testing rate which was omitted here, so take them with a grain of salt. Also note that, accidentally, Darcs would receive exactly the same scores if we were omitted module coupling rate.

	Analysability	Changeability	Stability	Testability	Maintainability
Darcs 2.5	★★★★	★★★★	★★	★★	★★
XMonad 0.9.2	★★★★*	★★★★	★*	★★★★*	★★★★*
GHC 7.0.1	★★★★*	★★★★	★★★★*	★*	★★★★*

Table 3: Maintainability score.

The maintainability analysis performed concludes that Darcs is not maintainable and identifies its excessive number of complex and large code units as one of the main causes. These complex and large units are hard to understand and test, and they possibly have an important influence on the low unit testing coverage of Darcs.

4 Improvement of test coverage for Darcs’s patch logic

Darcs test code is located under Darcs.Test namespace and it consists on several test suites which combine both QuickCheck³⁷ and HUnit³⁸ tests. HUnit is a typical unit testing framework inspired in JUnit³⁹; whilst QuickCheck is a tool for randomized testing, as I said previously. QuickCheck produces random test cases to instantiate your properties through a set of generators for the types involved, which are either provided by QuickCheck or build using QuickCheck combinators. While with HUnit, you are the test-case supplier and so the coverage (essentially) depends on how many tests you write; with QuickCheck, the coverage depends on the data distribution of the generators involved and how they work together.

³³http://www.haskell.org/ghc/docs/latest/html/users_guide/hpc.html

³⁴7844 expressions out of a total of 27020.

³⁵Line-based coverage data is rather senseless for a non-strict functional language such as Haskell.

³⁶The ones marked with an asterisk.

³⁷<http://www.cse.chalmers.se/~rjmh/QuickCheck/>

³⁸<http://hackage.haskell.org/package/HUnit-1.2.2.1>

³⁹<http://www.junit.org>

Although test coverage is usually measured as the percentage of lines (or expressions in the case of Haskell) executed during tests, this data is rather meaningless by itself. A function could be being executed during tests just as consequence of testing some property that, directly or indirectly, invokes it; but this does not mean that this function is being tested unitarily. The focus of this work is to improve the coverage, but in the sense of how well the test-case classes are being covered. Particularly, I will analyze and improve how QuickCheck generators are covering the interesting test cases of properties about operations on patches.

4.1 Existing test code

Darcs.Test.Patch module hierarchy contains code to test the Darcs's Patch Theory implementation which is located under the Darcs.Patch namespace. It consists of several modules containing different properties about patches, HUnit test cases and QuickCheck generators. There is no clear structure or organization, modules names are extremely ambiguous, there is a considerable amount of legacy test code waiting for clean-up, and the code contains few or no documentation at all. In short, it is a mess, and so I will just study a subset of this test code. Concretely, I have focused my work on the Darcs.Test.Patch.QuickCheck module which implements QuickCheck generators for testing Darcs2 patches.

Section 2 introduces primitive patches, which represent different types of basic changes that could be made over a repository. We can now offer a simplified description of the Darcs2 patch format as consisting of primitive patches plus a special patch type called *conflictor*. A *conflictor* is a patch that is seamlessly created by Darcs2 when a conflict occurs as the result of a merge, storing information about the conflicting patches. The user never handles *conflictor* patches directly, but they are there and all the operations we have seen for primitive patches, such as inversion or commutation, should now be extended to take them into account. Of course, all the properties about those operations must also continue to hold in the presence of *conflictors*, and test that is the goal of the QuickCheck module mentioned above.

The implementation of *commute* in the presence of *conflictors* is very tricky⁴⁰ and so it is one of main focus of testing. The testing of *commute* requires to have a generator of pairs of Darcs2 patches⁴¹, whose actual implementation works in the following way. First, it generates an arbitrary tree of hunk patches⁴² simulating changes over multiple branches of the same repository. Then this tree is flattened by merging its branches two by two obtaining a single sequence of patches as result. Finally, it picks a pair of patches from this sequence. The idea behind this algorithm is that a high percentage of patch pairs generated in this way will contain some *conflictor* produced during the flattening of the tree.

In contrast with common testing frameworks, with QuickCheck you don't write test cases but generators of test cases, so the proper coverage of the functions involved in a property has to be ensure by ensuring that the corresponding generators for the inputs of that property properly cover all the interesting cases. In the case we are working on, which is the testing of *commute* for Darcs2 patches, we are interesting in three properties for our pairs of patches: first, we want most of them to be commutable, since most properties consist on stating that some predicate holds for two commutable patches; second, they should cover (almost) all the possible cases of commute and; finally, and related with our second goal, it should exist a reasonable rate of conflictors within the generated pairs. These three coverage goals were measured using QuickCheck collect and classify combinators, and the result is shown in Figure 10.

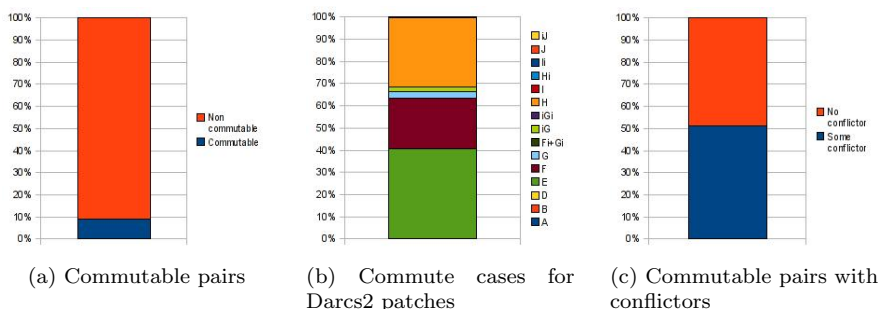


Figure 10: Coverage analysis for the generator of pairs of Darcs2 patches.

We can see that the current generator is not making a good job since, although the rate of conflictors within pairs is good enough, the rate of commutable pairs is under 10% and, most important, only about 5 out of 15 *commute* cases specific to Darcs2 patches are being properly covered.

4.2 Fix QuickCheck generator for pairs of Darcs2 patches

Unfortunately, I don't have the required skills to analyze why the generator is not properly covering all commute cases⁴³; but I have analyzed why it produces a very low rate of commutable pairs, I determined the underlying

⁴⁰You could take a look to the Commute instance for RealPatch data type defined in Darcs.Patch.Real module.

⁴¹Recall that *commute* takes a pair of patches as input.

⁴²For now, we will ignore how these hunks patches are generated.

⁴³Actually, very few people properly understand how commute works for Darcs2 patches.

cause, and I successfully fixed it. Concretely, I realized that the generator of trees of patches, which is called `arbitraryTree`, was generating an excessive rate of trees with size⁴⁴ smaller than one, as it is shown in Figure 11. When this situation occurs we cannot pick any pair from the flattened tree so a default pair of patches is used instead, but this default pair is neither useful for testing nor commutable. Although this situation should be rare the fact is that due to many factors it is happening very often, and that is why most generated pairs are not commutable.

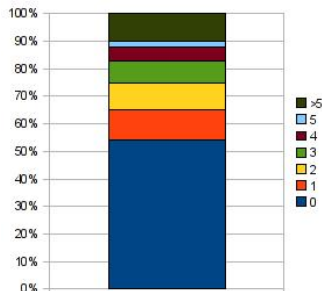


Figure 11: Size of trees generated by `arbitraryTree`.

After fix a minor bug in `arbitraryTree` code and make also some tweaks to the generator strategy, now the generation of those very small trees is indeed a rare case as shown in Figure 12. As expected this change had a great influence in the rate of commutable pairs, that was increased by a factor of 4. A 40% of commutable pairs is still a low rate and there is more room for improvement, however, to solve this requires changes to how primitive patches are being generated and that is precisely the subject of subsection 4.4.

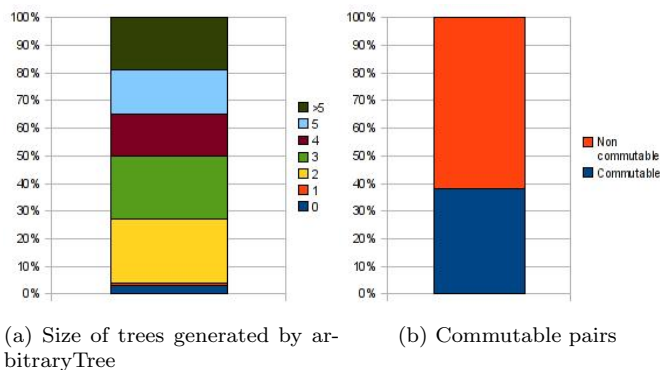


Figure 12: Coverage improvement after fixing patch trees generator.

4.3 Fix tests to discard useless test cases

As we mentioned above most properties about `commute`⁴⁵ are implications of the form $\forall x, y, y', x' : xy \leftrightarrow y'x' : P(x, y, y', x')$, that is, if a given pair of patches commute then there is some predicate that must hold involving both the given patch pair and the resulting commuted pair. These properties are written in `Darcs.Test.Patch.Properties` module as shown below⁴⁶.

```
<property> :: (Patch, Patch) -> Maybe Doc
<property> (x, y)
= case commute (x, y) of
    Nothing -> Nothing      -- Useless
    Just (y', x') ->
        ...
        case 'Some expression' of
            'Failed' -> Just 'Error Message'
            'Succeeded' -> Nothing
```

Darcs properties have range `Maybe Doc`, where `Nothing` is used to indicate that the test was succeeded whilst `Just` is used to indicate failure and carries the error message. A test may succeed due to two quite different reasons though. One possibility is that the input patch pair commutes and the predicate P ⁴⁷ holds, which is clearly a successful test case. The other possibility is that the input pair does not commute and hence the test succeeds because the implication is true given that the antecedent is false. The later is what we may call an useless

⁴⁴The number of nodes of the tree.

⁴⁵This also applies to other groups of properties.

⁴⁶Note the Darcs code showed here is very simplified, specially with respect to the type signatures which in the real implementation involve many typing tricks that allow to detect some common (Darcs specific) errors statically.

⁴⁷A generic predicate $P(x, y, y', x')$ as referred above.

test case, since we are not testing anything at all. Useless test cases are not a big concern in HUnit because the user is who specifies the cases to test; but in QuickCheck, where test cases are automatically generated, they could be a serious problem. If a big rate of test cases does not satisfy the antecedent of a conditional property, this property will not be properly tested despite being instantiated by tens of different test cases. In the case of Darcs this problem means that, prior to this work, for each block of 100 test cases generated to test properties about *commute* of Darcs2 patches, only about 8 of these cases were actually being useful for testing.

That is why QuickCheck offers the `==>` combinator to write conditional laws, which discard those test cases not satisfying the antecedent. However, `Darcs.Test.Patch.Properties` properties are used by both QuickCheck and HUnit test suites preventing the use of framework specific utilities. To solve this problem I have created a new data type `TestResult`⁴⁸ for expressing the result of a test which, in addition to the common succeeded and failed result types, it also let you to reject a test case indicating that it is useless for testing.

```
data TestResult = TestSucceeded
                | TestFailed Doc
                | TestRejected
```

This data type is made an instance of the `Testable` type class provided by QuickCheck, which allows us to instruct QuickCheck to discard rejected test cases.

```
instance Testable TestResult where
  property = ...
```

Other testing tools like HUnit only cares if a test was failed or not, being the responsibility of the user to write useful test cases.

```
isFailed, isOk :: TestResult -> Bool

isOk = not . isFailed
```

The generators included in `Darcs.Test.Patch.QuickCheck` are used to define multiple test cases as part of the `Darcs.Test.Patch.Unit2` test suite. Prior to this work the HPC coverage⁴⁹ of the `Unit2` test suite with respect to `Darcs.Patch.* modules`⁵⁰ was 25% and, with the improvements made until now, the coverage was increased only in 1%. Recall that this way to measure coverage is not accurate, so take this data with care. Although tests do not execute more code, this code is now being tested with more and richer test cases; there was a real coverage improvement, but it is not reflected in the kind of coverage information provided by HPC.

4.4 New generators for primitive patches

The actual `Darcs.Test.Patch.QuickCheck` implementation uses a smart strategy to generate primitive patches based on repository models, where a repository model is an in-memory representation of a repository. As shown below, the repository model handled by `Darcs.Test.Patch.QuickCheck` is quite simple consisting on a single file with some content.

```
data RepoModel
  = RepoModel {
    rmFileName :: !FileName,
    rmFileContents :: [B.ByteString]
  } deriving Eq
```

A generator receives a repository model as input, and returns an arbitrary patch applicable to this repository plus a new repository result of applying it to the input one.

```
arbitraryFP :: RepoModel -> Gen (Prim, RepoModel)
```

In order to produce a primitive patch the implementation starts with an initial repository with an empty file and then it generates a sequence of hunk patches modifying the content of the file. The last patch of the sequence is finally returned as the generated patch. To produce a pair or a triple of patches you just pick the last two or three patches of the patch sequence. Note that this strategy is using the first patches of the sequence to generate an arbitrary repository from the initial one, avoiding the implementation of a specific generator for repository models.

This method for primitive patch generation has a clear advantage, it is able to produce sequences of patches which are valid (sensible) by construction. In contrast with `Darcs.Test.Patch.Properties2`, where patches are first generated randomly and independently and then there is a filtering step, this valid-by-construction approach is clearly more sensible and efficient. However, the current implementation has many drawbacks: first, the repository model is so simple that it is only able to generate hunk patches; second, it produces a low rate of commutable pairs of hunks and; third, only 2 out of 4 commute cases for hunks are properly covered as shown in Figure 13.

In this work I wrote new code to generate primitive patches based on the same ideas presented above, but trying to solve all the drawbacks of the current implementation. The first step was to generalize the repository model in order to support most types of primitive patches. Fortunately a big part of this work is already done in the `Hashed Storage`⁵¹ library, which offers support to work with hashed file storage, and provides an abstraction of a filesystem tree making possible to manipulate the tree on memory. `Hash Storage` trees are parametrized by a monad as a way to parametrize how errors are raised; in particular the `IO monad` allows the tree to be read from and written to disk.

⁴⁸See `Darcs.Test.Util.TestResult` module

⁴⁹That is, the percentage of expressions evaluated during testing.

⁵⁰The `Patch Theory` implementation.

⁵¹<http://hackage.haskell.org/package/hashed-storage-0.5.5>

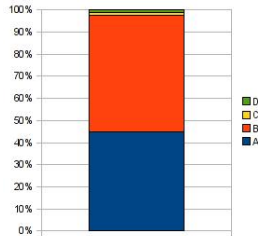


Figure 13: Coverage of commute cases for hunk patches.

The new repository model is defined in `Darcs.Test.Patch.RepoModel` as a wrapper over a `Hashed Storage` tree. In order to support application of patches to our new repository model we have to instantiate the `Tree` type with an instance of `MonadPlus`⁵², and the `Maybe` monad is the simplest one.

```
newtype RepoModel = RepoModel { repoTree :: Tree Maybe }
newtype RepoItem = RepoItem { treeItem :: TreeItem Maybe }

type Content = [B.ByteString]
type File = RepoItem
type Dir = RepoItem
```

It is offered a simplified API to work with repository models, allowing the user to forget low-level details exposed by `Hashed Storage` such as hashes.

```
makeFile :: Content -> File
fileContent :: File -> Content
find :: RepoModel -> AnchoredPath -> Maybe RepoItem
list :: RepoModel -> [(AnchoredPath, RepoItem)]
...
```

`Hashed Storage` provides us fast comparisons between repositories; when needed, hashes are computed on-demand. This ability to compare repositories is very important, since we are interested in write properties stating that two given sequences of operations result in the same repository.

```
instance Eq RepoModel where
  repo1 == repo2 = ...
```

One of the key reasons because I have based this new repository model on `Hashed Storage` trees is the fact that `Darcs` uses this library internally to handle its repositories, and therefore it is possible to apply a patch directly to a `Hashed Storage` tree. Thanks to this the application of a patch to a repository model will be performed by the same code that applies a patch in disk, which will give us more confidence about the correctness of this code.

```
applyPatch :: Patch -> RepoModel -> Maybe RepoModel
applyPatch patch (RepoModel tree) = RepoModel <$> applyToTree patch tree
```

This module also defines several generators for file and directory names, file contents, directories and repository models. The generator for names is designed to allow fast generation of unique names since it is a commonly used operation when generating patches like `AddFile`, `AddDir` or `Move`. File content generator is implemented in a way that the size of the generated files tends to increase as testing progresses. Finally, the generator of repositories receives the maximum allowed number of files and directories and it creates some arbitrary repository respecting those limits following a simple recursive algorithm. I prefer the use of generators for repository-related stuff instead of use the strategy followed by `Darcs.Test.Patch.QuickCheck` because I think that this approach is more efficient and, specially, because it provides more control on the type of repositories that are created.

Once the repository is general enough it becomes possible to generate almost any kind of patch. The idea to generate an arbitrary patch consist on generate a repository and then create a patch applicable to that repository. The problem is that given a repository only some types of patches are sensible, for example, a `RmFile` patch is only sensible if exists some empty file in the repository that the patch can remove. This makes the generator a bit tricky and so one need to think in some way to handle this situation. In this work I have done it in the following way. First, given a repository, the generator performs queries on the repository to collect data required to create the different types of patches. For example, it searches for a file in the repository to create a `Hunk`, and for an empty file to create a `RmFile` patch.

```
mbFile <- maybeOf repoFiles
mbEmptyFile <- maybeOf $ filter (isEmpty . snd) repoFiles
...
```

Once the generator has this data, it chooses what patch to create arbitrarily from a table of generators for each specific type of patch. Each entry of this table is enabled only if it is possible to create a patch of that type for the given repository.

```
patch <- frequency
  [ ( if isJust mbFile then 15 else 0
    , aHunkP $ fromJust mbFile )
    ...
    , ( if isJust mbEmptyFile then 12 else 0
    , return $ aRmFileP $ fst $ fromJust mbEmptyFile )
    , ... ]
```

⁵²A Haskell type class for monads that also support choice and failure.

This new patch generator covers all the patch types modeled in Alloy plus TokReplace patches. As we mentioned before, ChangePref patches does not worth to be modeled because its implementation is known to be buggy and they break several Patch Theory properties. Binary patches were not modeled because they introduce some complexities and it may not worth it since they are not involved in any interesting *commute* case. Pairs of primitive patches are generated pipelining two invocations to the single patch generator, so the second invocation works on the repository model generated by the first one.

This code was fine tuned until I obtained the coverage data shown in Figure 14. All the (considered) patch types are being properly covered, and the coverage of each type is more or less correlated with its importance. With respect to pairs of patches, the rate of commutable pairs is satisfactory, and the coverage of *commute* is in general good with the exception of Hunks patches. There are four possible cases for Hunks commutation and only two of them are being slightly covered, which is not acceptable given the importance of Hunk patches.

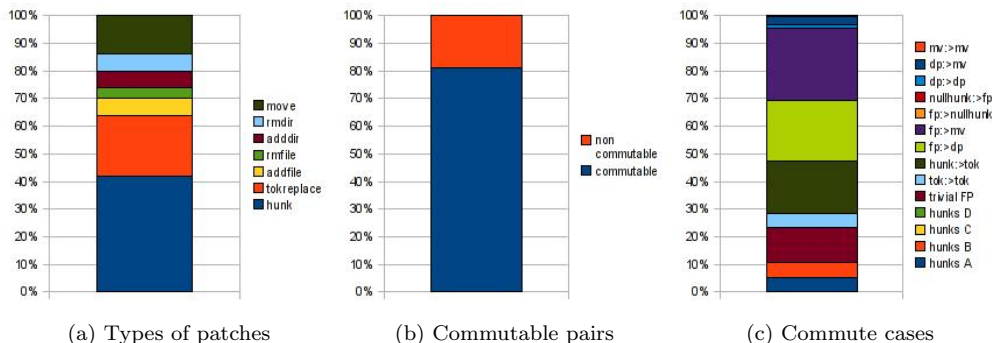


Figure 14: Coverage of the primitive patch generator.

To fix this situation I create an specific generator whose goal is to produce a big rate of commutable Hunk pairs, and I combined this generator with the generic one. As shown in Figure 15, the combination of both generators fix the problem and now hunks receives the attention they deserve, whilst the other *commute* cases continue to be well covered. The rate of commutable pairs was decreased slightly but it continues to be satisfactory.

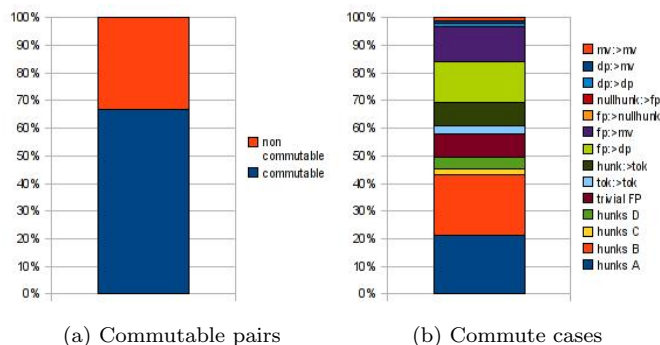


Figure 15: Coverage for improved primitive patch generator.

4.5 Results

I have added a total of four new properties to `Darcs.Test.Patch.Properties` — most of them are now possible to test thanks to the new repository model:

Invert Symmetry $\forall p, (p^{-1})^{-1} = p$

Invert Rollback $\forall a b p, b = p(a) \Rightarrow a = p^{-1}(b)$

Commute Effect Preserving $\forall a b p q q' p', pq \leftrightarrow q'p' \wedge b = q(p(a)) \Rightarrow b = p'(q'(a))$

Join Effect Preserving $\forall a b p q s, s = join(p, q) \wedge b = q(p(a)) \Rightarrow b = s(a)$

where a, b denote repositories; p, q, r, p', q' denote patches; and $b = p(a)$ denotes the application of patch p to the tree a resulting in the tree b . The operation *join* was not considered for the Alloy model, but it consist on (try to) combine two patches into a single patch without intermediary changes. For example, two hunk patches modifying adjacent lines can be coalesced into a bigger hunk patch.

The instantiation of these and other properties with the new generators resulted in 11 new test cases that were added to the `Darcs.Test.Patch.Unit2` test suite. As consequence of this work the Unit2 test suite now covers, as measured by HPC, the 29% of expressions in `Darcs.Patch.*`, which means that the coverage was increased in +4%. Recall this coverage data, measured by HPC, should be taken with a grain of salt. Despite the low increment in coverage shown by HPC, the fact is that once I have replaced `Darcs.Test.Patch.QuickCheck` generators by the

new ones, a new bug has been found⁵³. This new bug is (possibly) in the implementation of *commute* for Darcs2 patches and breaks the symmetry of commutation. The counterexample provided by QuickCheck shows that the problem appears when a conflictor between Move and RmDir patches is involved, so it is clear that it was detected thanks to the new patch generator.

5 Conclusions

During the verification part of this project we have modeled the Darcs patch logic kernel using Alloy, we have verified most interesting properties about *invert* and *commute* and, even more, we have found two rare corner cases in which Darcs would fail. Alloy was demonstrated to be a simple but powerful tool which allows cost-effective software verification. It is true that several complications arose due to Alloy limitations but we were able to verify many interesting things with very little effort; just compare our work with the work done in the Camp⁵⁴ project, which aims to verify a *simplified*⁵⁵ implementation of Darcs using Coq⁵⁶. Moreover, the verification job of Camp did not find the two rare bugs discovered by us because it is (wrongly) assuming that Darcs uses infinite precision integers⁵⁷ and that filesystems do not have limits.

Despite this experience was successful, we think that Alloy won't be the right tool to model and verify a full Patch Theory implementation. In this project we have only modeled the core concepts of Patch Theory and we were frustrated (in different degrees) by many Alloy limitations such as its lack of recursion support and the finiteness of universes. Our intuition is that those limitations would become more severe once we extend our models with sequences of patches. The *unsaturated universe problem* was very frustrating and, unfortunately, *generator axioms* are limited and cannot be applied in all situations. The use of SMT solvers such as Yices⁵⁸ may solve this limitation[8]; I wonder if would be possible to provide a theory for patches and repository trees in order to avoid the finitization of these types. Furthermore, Alloy could easily support a limited form of recursion like Yices does⁵⁹.

The maintainability analysis concludes that Darcs is hard to maintain, being its most important weaknesses an excessive number of complex and large functions and a low unit testing coverage. It is interesting to note that none of the Darcs developers have identified any of these aspects as being the cause of Darcs maintainability problems, which suggests that even experienced developers (at least Darcs ones) may have problems to identify maintainability risks in their own code. I am pleased to see that the maintainability rates agrees with my opinion, as well as with the opinion of some Darcs developers. However, these rates must be taken carefully since, as I mentioned above, in my opinion neither volume nor module coupling rates are accurate.

I would suggest Darcs people to make an effort in order to clean up Darcs testing code and add much more tests to increase unit testing coverage. During my analysis of Darcs and after many discussions with its developers I think that the lack of unit testing is leading to some bizarre situations. For example, some modules define their own private version of an utility function, sometimes already defined elsewhere, because the author is afraid of some other developer modifying how that function behaves if it were located in an utilities module. Certainly, good unit testing eliminates that kind of worries.

At the beginning I was surprised to see that commutation of Darcs2 patches, which is one of most critical parts of Darcs, was poorly tested. However, when you work implementing QuickCheck patch generators then you realize that it is not an easy task. For instance, write generators for pairs of patches is very tricky because a good coverage does not depend only on covering all possible combinations of patch types, but also depends on how many pairs are commutable or have some conflictor patch. It is hard to look to the code of a generator and deduce how good these properties are covered, and sometimes it is also difficult to understand what and why is causing a low coverage rate. All of this makes almost impossible to write a single, general purpose, patch generator.

Besides the particular problems mentioned above, I have found that some experience is required in order to write QuickCheck generators with a good data distribution. For instance, the first time I wrote a generator for repository models it produces too many empty repositories, and I had to add several fixes in order to reduce them significantly. My actual little experience with QuickCheck says me that you always must analyze the coverage of a generator to trust that it will cover test case space properly. Although QuickCheck potentially offers better coverage than HUnit, it turns out that with HUnit is much more simple to ensure that your tests are covering the interesting cases.

References

- [1] David Roundy. Darcs user manual. <http://darcs.net/manual>
- [2] David Roundy. Theory of patches. <http://darcs.net/manual/node9.html>
- [3] Judah Jacobson. A formalization of Darcs patch theory using inverse semigroups.

⁵³See <http://bugs.darcs.net/issue2047>.

⁵⁴<http://projects.haskell.org/camp/>

⁵⁵For instance, it only considers the two most trivial cases of hunk-hunk commutation, in contrast with Darcs which considers two additional cases.

⁵⁶<http://coq.inria.fr>

⁵⁷Darcs could use arbitrary precision integers, though certainly Haskell standard libraries "encourages" the use of fixed-precision integers.

⁵⁸<http://yices.csl.sri.com>

⁵⁹http://yices.csl.sri.com/language.shtml#language_set_nested_rec_limit

- [4] Jason Dagit. Type-Correct Changes — A Safe Approach to Version Control Implementation.
- [5] I. Heitlager, T. Kuipers and J. Visser. *A Practical Model for Measuring Maintainability*.
- [6] Wikipedia, the free encyclopedia. Coupling (computer programming).
- [7] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.
- [8] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing Alloy Constraints using an SMT Solver: A Case Study.