Mysteries of Dropbox

Property-Based Testing of a Distributed Synchronization Service

John Hughes*†, Benjamin C. Pierce‡, Thomas Arts*, Ulf Norell*†,

* Quviq AB, Göteborg, Sweden
† Dept of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden
‡ Dept of Computer and Information Science, University of Pennsylvania, PA, USA

***Abstract*—File synchronization services such as Dropbox are used by hundreds of millions of people to replicate vital data. Yet rigorous models of their behavior are lacking. We present the first formal—and testable—model of the core behavior of a modern file synchronizer, and we use it to discover surprising behavior in two widely deployed synchronizers. Our model is based on a technique for testing nondeterministic systems that avoids requiring that the system's internal choices be made visible to the testing framework.**

I. Introduction

File synchronization services—distributed systems that maintain consistency among multiple copies of a file or directory structure—are now ubiquitous. Dropbox claim 400 million users,[1] while Google Drive and Microsoft OneDrive are reported to have over 240 million users each.[2] In addition to these large-scale commercial offerings and many smaller ones, there are a plethora of open source synchronizers, enabling users to create their own 'cloud storage.' With so many people trusting their data to synchronization services, their correctness should be a high priority indeed.

Surprisingly, then, it seems that only one file synchronizer has been formally specified to date: Unison [1]–[3]. However, even Unison's specification is not directly useful for *testing*; moreover, Unison works rather differently from contemporary synchronization services: it synchronizes two peers at a time, rather than many clients with one server; synchronization is invoked explicitly by the user, rather than taking place automatically in the background; and conflict resolution involves user interaction, rather than being performed automatically.

Synchronizers are challenging not only to specify but also to test. They are large-scale, nondeterministic distributed systems, with timing-dependent behavior. They must cope with conflicts (when the same file is modified concurrently on two or more clients). They work in the background, and their state is unobservable. Moreover, they are *slow*. They share these difficulties with a large class of critical systems; thus, techniques for addressing these problems are valuable, not only for testing file synchronizers, but in a broader context.

[1] `techcrunch.com`, "Dropbox now has more than 400 million registered users", June 24^{th}, 2015.

[2] `fortune.com`, "Who's winning the consumer cloud storage wars?" November 6^{th}, 2014.

Our goal in this paper is to present a testable formal specification for the core behavior of a file synchronizer. We do so via a *model* developed using Quviq QuickCheck [4]. Despite the apparent simplicity of the problem, we encountered interesting challenges regarding both specification and testing. We used the model to test Dropbox, Google Drive, and ownCloud (an open source alternative), exposing unexpected behavior in two out of three.

In Section II, we introduce our testing framework. Section III gives a high-level overview of the concepts used in our model, in particular the *operations* performed by test cases, the *observations* made when a test case is run on the system under test (SUT), and the *explanations* that we construct to determine whether the test has passed or failed. Section IV presents the formal specification itself, beginning with a naive version and refining it in light of failed tests that reveal subtleties in the synchronizer's handling of corner cases. Section V describes further failed tests that, rather than pinpointing inadequacies in the specification, seem to us to exemplify unintended behaviors of both Dropbox and ownCloud, including situations where each system can lose data. In Section VI, we discuss the pragmatics of our testing framework, in particular our methods for triggering timing-dependent behaviors, for observing when the system has reached quiescence, and for shrinking tests to minimal failing cases in the presence of nondeterminism. In Section VII, we sketch an initially promising attempt to formulate the specification in terms of Lamport's "happens before" relation and explain why it ultimately proved unsuccessful. Section VIII surveys related work, and Sections IX and X present future directions and concluding remarks.

Our main contributions are as follows: (1) We construct the first formal model of the core behavior of modern file synchronizers. (2) To define the model, we develop a technique for testing nondeterministic systems that does not require that the system's internal choices be visible to the testing framework. Our technique is based on an explicit representation of hidden system state plus "conjectured events" that mutate it. (3) We validate our final model against two commercial synchronizers and one open-source one, showing that their behavior agrees with the model in most situations. (4) We demonstrate the effectiveness of our model and testing framework by using it to reveal a number of surprising behaviors, likely not intended by the developers of these systems.

II. Testing a Synchronization Service

We begin by making a rather drastic simplifying assumption: we consider only *one file*, with operations to *read*, *write*, and *delete* it. We consider only operations that read and write the *entire* file; we do not specify or test how the synchronizer interacts with open files as they are modified. Restricting our attention to one file may seem extreme, but even this simple setting forces us to confront essential issues of synchronization—in particular subtleties arising in the presence of conflicts—and it is enough to expose surprising behaviors in real systems. A plan for extending the specification to multiple files and directories is sketched in Section IX.

We developed our model using Quviq QuickCheck [4], a descendant of Haskell QuickCheck [5]. QuickCheck tests *properties*—universally quantified boolean formulæ—by generating random values for the quantified variables and checking that the formula evaluates to `true`. When a test fails, QuickCheck "shrinks" it, searching for a similar but smaller test that also fails and ultimately reporting a failing test that is 'minimal' in some sense. (This process is akin to delta-debugging [6], although the details are slightly different.) QuickCheck provides a domain-specific language for defining test data generators and shrinking searches; using this DSL, users can exert fine control over the random distribution of test cases. A part of this DSL is a notation for specifying state machine models, which generate test cases consisting of sequences of calls to an API under test [7], [8]; this is how we generated test cases for synchronizers. Quviq QuickCheck is embedded in the Erlang programming language—that is, specifications are just Erlang programs that call libraries supplied by QuickCheck—and the generated tests invoke the SUT directly via Erlang function calls. As a result, there is no distinction between 'abstract' and 'concrete' test cases, as there often is in model-based testing, and there is no need for a 'system adapter'; instead, generated test cases are directly executable.

We ran our tests on laptops running a host operating system (Windows 8.1 or Mac OS) together with several virtual machines running Ubuntu Linux. The file synchronizer under test was installed on each virtual machine and under the host operating system, so we could read and write files on any of the virtual machines, or on the host, and expect the synchronizer to propagate changes to all of the others. We ran distributed Erlang on the virtual machines and the host, using the host to coordinate each test by making remote procedure calls on the virtual machines. All machines were also connected to the internet, allowing us to test synchronization servers running either remotely (in the case of Dropbox and Google Drive), or on another VM (in the case of ownCloud). The use of multiple VMs on a single physical machine was purely a matter of convenience: Erlang's support for transparent distribution would make it straightforward to run the same setup on multiple nodes. We used 3 VMs for most tests.

We wanted a model that would apply (perhaps with minor adjustments) to *many* file synchronizers, not just the specific ones we have tested so far. In particular, we did not want to assume any direct access to remote servers outside our control. Therefore, we treat the synchronizer as a 'black box', which we communicate with only through the file system; our tests just read and write files on the virtual machines and check the results for validity. This allows us to write the specification without dependencies on synchronizer-specific APIs.[3]

III. Overview of the Specification

Our strategy for test case generation is very simple: we generate test cases consisting of random sequences of calls to a small set of basic filesystem operations. The basic operations we consider are

- $\textsc{Read}_N$, which reads the (one) file on node N (one of the VMs), and
- $\textsc{Write}_N\ V$, which writes the value V (a string) to the file on node N.

We will introduce a few additional operations below.

We use QuickCheck's state machine library to generate tests, with a trivial model state—our tests are simply random sequences of READs and WRITEs, with random arguments. One might expect to track the file contents in the model state, and check that READ returns the modeled contents—but this could only work if synchronization were instantaneous, which of course it is not. Instead of trying to express our specification in this *synchronous* style, we collect *observed events* as each operation is actually executed, and we use a separate state machine to validate the resulting sequence of events.

The observed events corresponding to the READ and WRITE operations are as follows:

- when $\textsc{Read}_N$ returns the value V, we observe the event $\textsc{Read}_N \to V$, and
- when $\textsc{Write}_N\ V$ overwrites the value V_{old}, we observe the event $\textsc{Write}_N\ V \to V_{old}$.

Notice that, when we write the file, we observe the previous contents as well as the new one (taking the 'previous contents' of a newly created file to be the special value $\perp$). The value that we overwrite matters, because of the way synchronizers handle *conflicts*. (This observation is not actually atomic, but it is very unlikely that the dropbox dæmon will overwrite the file between our read and write, and we have not observed it to happen.)

A conflict occurs when two or more clients write the file concurrently—that is, without having seen each other's updates. For example, if client 1 writes '`a`' to the file, and client 2 then writes '`b`' before '`a`' has been delivered to client 2 by the synchronizer, a conflict is created. One of the two values *wins*, and eventually all clients will see this value in the file if there are no more writes, but conflicting values are also saved in special files in the same directory, with names derived from

[3]For *testing* the specification, we found it convenient to use limited communication with the local dæmon, where available; for example, the Dropbox client on Ubuntu provides a handy Python script for querying whether the local dæmon thinks it is up to date, which we used to speed up testing a bit. This adds $<$10 lines of code per synchronizer to the test harness.

the name of the original file, such as '`paper.tex (John's conflicted copy)`'; these files are eventually replicated to other clients just like any other file. To avoid depending on implementation-chosen names, our model specifies just the set of *values* expected to appear in conflict files, and our test harness assumes that *any* files that appear in the same directory as the main one are conflict files.

One subtlety in the specification of conflicts is that, because conflicts can only be detected using global information, there may be a delay in the creation of conflict files. Consider the following sequence of observations.

Client 1	Client 2
WRITE$_1$ '`a`' $\to \perp$	
	WRITE$_2$ '`b`' $\to \perp$
	READ$_2$ $\to$ '`a`'

The two writes are in conflict, and the write of '`a`' has 'won', so the value '`b`' should appear in a conflict file. However, client 2 cannot determine this locally (since a different client could have overwritten '`b`' with '`a`' in the meantime). So we must wait for pending communications with the server to finish before checking for the existence of conflict files.

We therefore add another operation to our test cases, STABILIZE, which waits for synchronization to complete on all client nodes. At this point, the *same* value V should be in the file on all clients, and all clients should have the same set of conflict files. Once the system is stable, we observe the event STABILIZE $\to (V, C)$, where C is the set of values in the conflict files (i.e., both the value and the conflict files should be *eventually consistent*; we will see later that our model can always reach such a state). Note that this is a *system wide* event, not an event observed on just one client. To check that the correct conflict file is created in the example above, we would add a STABILIZE operation to the end of the test, which should result in the following observed events:

Client 1	Client 2
WRITE$_1$ '`a`' $\to \perp$	
	WRITE$_2$ '`b`' $\to \perp$
	READ$_2$ $\to$ '`a`'
STABILIZE $\to$ ('`a`', {'`b`'})	

If '`b`' were missing from the conflict set in the last observed event, this would represent lost data and the test would fail.

But how can we implement STABILIZE? That is, how can we tell that synchronization is complete, given that we treat the synchronizer as a black box? Our solution is a little *ad hoc*. Certainly, the value in the file and the conflict files must be the same on each client. Also, under Ubuntu, Dropbox provides a handy Python script to check the local Dropbox dæmon's status; the dæmons on each client must be reporting 'up to date'. We wait for these necessary conditions to become true—but they are not sufficient, since the server may still be holding data that will be sent to the clients. So if a STABILIZE $\to (V, C)$ operation ever leads to an observation that would cause test failure, then we wait a bit longer and retry the operation. We cannot wait *too* long, because *failed* tests may well result in a stable state never being reached! In practice we wait up to 30 seconds, long enough to allow synchronization to finish if it is going to but short enough to enable us to 'shrink' test cases (which involves running many, many failing tests—see Section VI) in a reasonable amount of time. After 30 seconds we record a "failed stabilization" observation of the form STABILIZE $\to \{(V_1, C_1), \ldots, (V_n, C_n)\}$, where each (V_i, C_i) records the the file contents and conflict set on one of the nodes. Such an observation is regarded as invalid by the specification, so any test that generates it is considered to fail.

Since stabilization is slow—and most interesting after *several* read and write operations—we include it only $1/10$ as often as READs and WRITEs. We also add a STABILIZE at the end of every test case, which improves the probability of detecting that something went wrong and also reduces the risk of one test influencing the outcome of the next one.

How can we decide whether a sequence of observed events is *valid*? We use a separate state machine, which accepts or rejects a sequence of observed events. However, the observations we make do not tell the whole story: in the background, the synchronizer is also performing actions. This makes our tests nondeterministic—we cannot tell, from the events we have observed, what state the whole system is in. We address this by adding *conjectured events* to the observed ones, representing actions taken by the synchronization service (messages between local dæmons and a central server, interactions between dæmons and local filesystems, etc.). In general, we can add conjectured events to a given sequence of observed events in many different ways, each resulting in a different combined sequence of observed and conjectured events, which we call an *explanation*. If any of the explanations is accepted by our state machine, we consider the test to have passed. If there is no way to insert conjectured events so the resulting explanation is accepted, then we consider that the test has failed.

Our conjectured events are uploads to, and downloads from, the server. We write these events as UP$_N$ and DOWN$_N$, where N is the node taking part in the up- or down-load. Thus, when we model the state of the whole system, we will need to include the server's state in the model. (Of course, in reality 'the Dropbox server' may itself be a replicated service involving many hosts. Our model implicitly assumes strong consistency among these hosts; weaker consistency in Dropbox's implementation might in principle cause our tests to fail, but we have not observed this.)

To make all of the above more concrete, here is an example of a simple test case:

Client 1	Client 2
WRITE$_1$ '`a`'	
	WRITE$_2$ '`b`'
READ$_1$	
	WRITE$_2$ '`c`'
STABILIZE	

Here is an observation that might arise from this test:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
	WRITE$_2$ 'b' $\to$ 'a'
READ$_1$ $\to$ 'b'	
	WRITE$_2$ 'c' $\to$ 'b'
STABILIZE $\to$ ('c', $\emptyset$)	

And here is a valid explanation of this observation:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
UP$_1$	
	DOWN$_2$
	WRITE$_2$ 'b' $\to$ 'a'
	UP$_2$
DOWN$_1$	
	WRITE$_2$ 'c' $\to$ 'b'
	UP$_2$
READ$_1$ $\to$ 'b'	
DOWN$_1$	
STABILIZE $\to$ ('c', $\emptyset$)	

Of course, the same test may give rise to different observations (and the same observation may be explained by many possible explanations). For example, here is another observation that might arise from running the test above:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
	WRITE$_2$ 'b' $\to \perp$
READ$_1$ $\to$ 'a'	
	WRITE$_2$ 'c' $\to$ 'b'
STABILIZE $\to$ ('a', {'c'})	

A valid explanation of this observation is:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
	WRITE$_2$ 'b' $\to \perp$
READ$_1$ $\to$ 'a'	
	WRITE$_2$ 'c' $\to$ 'b'
UP$_1$	
	UP$_2$
STABILIZE $\to$ ('a', {'c'})	

Conversely, suppose the synchronizer under test were misbehaving. Then the same test case might lead to an observation with no valid explanations. For example:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
	WRITE$_2$ 'b' $\to$ 'a'
READ$_1$ $\to$ 'b'	
	WRITE$_2$ 'c' $\to$ 'b'
STABILIZE $\to$ ('c', {'a'})	

Here, the final STABILIZE observation shows that a conflict file has been created for the file value 'a'. But 'a' was the first value written to the file, and it must have been the first value uploaded to the server (because the second WRITE saw 'a' as the previous value), so it cannot have been in conflict with any of the other WRITEs.

IV. FORMALIZING THE SPECIFICATION

Formally, we use a deterministic state machine to accept or reject explanations. We define the state of the machine, the *system state*, as follows:

- a global *stable value* $ServerVal$ (i.e., the value currently held on the server)
- a global *conflict set* $Conflicts$ (a set of values)
- for each node N,
  - a *local value* $LocalVal_N$,
  - a *local freshness*, $Fresh?_N \in \{\text{FRESH}, \text{STALE}\}$, and
  - a *local cleanliness*, $Clean?_N \in \{\text{CLEAN}, \text{DIRTY}\}$.

Values (i.e., file contents) are just strings. (We define the result of a READ or WRITE before the file has ever been written to be the special value $\perp$.)

In the initial state S_{init}, the stable value and all the local values are $\perp$, and all nodes are FRESH and CLEAN.

There are three kinds of *observed events* (READ, WRITE, and STABILIZE) and two kinds of *conjectured events* (UP and DOWN). A sequence of observed events is called an *observation*. A sequence of both kinds of events is called an *explanation*. An explanation E *explains* an observation O if deleting the conjectured events from E leaves just O.

For every event (of either kind), we will define a *transition* consisting of a *precondition* (which tells us whether the event can happen in a given system state) and an *effect* (which defines the change to the system state after the event has happened). Taken together, these preconditions and effects define a partial function $Next$ mapping a system state plus an event to a new state (or failing, if the event's precondition is not satisfied by the state).

An explanation E is *valid* with respect to some starting state S if either (1) E is the empty sequence of events, or else (2) E is a non-empty sequence $e : E'$ (where : is 'cons'), such that $Next(S, e)$ yields a new state S' and E' is valid with respect to S'.

An observation O is *valid* if it is explained by some explanation that is valid with respect to the initial state. There are only finitely many valid explanations for each observation, as we explain below, so validity of observations is decidable.

A *test* is a sequence of operations. It *fails* if the observation that arises by running it has no valid explanation; otherwise it *succeeds*.

It remains only to define the transitions themselves. The read and write transitions are straightforward:

READ$_N$ $\to V$
Precondition: $LocalVal_N = V$
Effect: none

WRITE$_N$ $V_{new} \to V_{old}$
Precondition: $LocalVal_N = V_{old}$
Effect: $LocalVal_N \leftarrow V_{new}$
$Clean?_N \leftarrow$ DIRTY

A write event *does* have a precondition, because it observes the value that is overwritten. That is, a read or write event on client node N is valid with respect to some model state if the value that the event observes in the filesystem agrees with the model's current value for that node. Observing a READ has no effect on the model state, while observing a WRITE changes the model's local value for node N to the one that was written by the WRITE operation and marks node N as DIRTY.

The STABILIZE $\to (V, C)$ event has no effects (it is like a READ in this respect), but it has a very strict precondition:

STABILIZE $\to (V, C)$
Precondition: $ServerVal = V$
$Conflicts = C$
for all N, $Fresh?_N =$ FRESH
$Clean?_N =$ CLEAN
Effect: none

The intuition for this precondition is that this observed event *is not considered valid* unless the system model has also reached a stable state. The precondition can be satisfied by adding conjectured upload and download actions to the explanation until all nodes in the system state are FRESH and CLEAN.

The transition for failed stabilization events has an even stricter precondition: such an event is *never* allowed!

STABILIZE $\to \{(V_1, C_1), \ldots, (V_n, C_n)\}$
Precondition: *False*
Effect: none

This ensures that any observation including a failed stabilization will be identified as a failing test case.

These are all the observed events. But we also need transitions for the conjectured events UP and DOWN. Downloading a value from the server to a client node stores the server's value as the local value for that node in the system state; it also changes the node from STALE to FRESH. However, it can only be performed if the client node is currently CLEAN. (Otherwise, it must be preceded by an UP event, which will reconcile the local value with the server's.)

DOWN$_N$
Precondition: $Fresh?_N =$ STALE
$Clean?_N =$ CLEAN
Effect: $LocalVal_N \leftarrow ServerVal$
$Fresh?_N \leftarrow$ FRESH

The most interesting case is the UP transition. Here is a simple first attempt (we will refine it below):

UP$_N$
Precondition: $Clean?_N =$ DIRTY
Effect: $Clean?_N \leftarrow$ CLEAN
if $Fresh?_N =$ FRESH then
$Fresh?_{N'} \leftarrow$ STALE for all $N' \neq N$
$ServerVal \leftarrow LocalVal_N$
else $Conflicts \leftarrow Conflicts \cup \{LocalVal_N\}$

UP$_N$ is only allowed if node N is DIRTY (written since the last DOWN). Its effect is either to update the server's value from node N's if N is currently FRESH—i.e., if it is not in conflict with a WRITE on another node that has already reached the server—or otherwise to mark the local value as a conflict. In either case, node N is marked as CLEAN.

To decide whether a test succeeded, we construct a valid explanation for the observation we made; that is, we insert a sequence of UP and DOWN events between each pair of observed events that makes the explanation valid. How many conjectured events might we need to insert? First of all, note that each UP event makes a DIRTY node CLEAN (and neither UP nor DOWN can make a CLEAN node DIRTY). So, if there are a total of N nodes, then at most N UP events can appear between consecutive observed events. Secondly, note that each DOWN event makes a STALE node FRESH. So there can be at most N DOWN events in a row. Since an UP event makes $N-1$ nodes STALE, each UP can be followed by up to $N-1$ DOWN events before another UP or an observed event must occur. Thus we need to insert at most $N + N \cdot (N-1)$ conjectured events between each pair of consecutive ordered events; there are only finitely many possible explanations for each observation, so it is decidable whether a test has passed.

In our implementation, we do not explore all possible explanations. We construct the *set of possible states* before and after each observed event in an observation. Given the set of possible states before such an event, we select those satisfying the event's precondition and apply the event's action to them, resulting in the set of possible states after the event. If the set of possible states ever becomes empty, then the test fails.

Now, given the set $\mathcal{S}$ of possible states *after* an observed event, we construct the set of states before the following one by taking the image of $\mathcal{S}$ under the transitive closure of UP and DOWN. In any such state, (a) every node will have a $LocalVal$ drawn from the set V of all *LocalVal*s plus the $ServerVal$ in the given state, (b) the $ServerVal$ will also be an element of V, (c) the $Conflicts$ will be the union of the $Conflicts$ in the given state, and a subset of V, and (d) each node will be FRESH or STALE, CLEAN or DIRTY. Since the size of V is at most $N+1$, it follows that there can be at most $(N+1)^{N+1} \cdot 2^{N+1} \cdot 4^N$ states reachable from $\mathcal{S}$.

Should this set become too large to deal with during testing, a pragmatic solution would be to abandon that test case and generate another. We conjecture that most bugs can be found by a relatively deterministic test, so we would not expect this solution to make many interesting bugs impossible to find.

However, in our experiments, N was at most 3, giving a bound of at most 262144 states reachable from each state—

a large but not unmanageable number. In practice, we have almost never seen more than 1,000 different possible states during a test. Synchronizers are so slow that there is *plenty* of time to compute 1,000 model states after each observed event!

Refining the model: repeated values

Perhaps not surprisingly, with the first-draft model as presented above, testing against Dropbox fails immediately. QuickCheck reports the following minimal counterexample:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
	WRITE$_2$ 'a' $\to \perp$
STABILIZE $\to$ ('a', $\emptyset$)	

The test fails because the two writes conflict—neither saw the value written by the other. Our model says that *both* nodes must upload their values before the STABILIZE and that, on the second UP required to enable it, the value should be added to the set of conflicts. Yet the set of conflicts is observed to be empty at the end. The *Next* function as defined above admits no valid explanations of this observation.

Evidently, Dropbox considers that there is no conflict if the *same* value is written independently by two clients. This is a sensible design decision, but it needs to be reflected in our specification. To make the implementation and specification agree, we need to refine the specification to add special cases in the UP event when the local and global values are identical:

UP$_N$
Precondition: $Clean?_N =$ DIRTY
Effect: $Clean?_N \leftarrow$ CLEAN
  if $Fresh?_N =$ FRESH then
    if $LocalVal_N \neq ServerVal$ then
      $Fresh?_{N'} \leftarrow$ STALE for all $N' \neq N$
      $ServerVal \leftarrow LocalVal_N$
  else
    if $LocalVal_N \neq ServerVal$ then
      $Conflicts \leftarrow Conflicts \cup \{LocalVal_N\}$

With this modification, the test case passes. Here is a (newly valid) explanation for the observation we made:

Client 1	Client 2
WRITE$_1$ 'a' $\to \perp$	
UP$_1$	
	WRITE$_2$ 'a' $\to \perp$
	UP$_2$
STABILIZE $\to$ ('a', $\emptyset$)	

Adding deletion to the model

Since we already model reading from a missing file by the special value $\perp$, *deletion* can be modelled simply as writing $\perp$ back to the file. Of course, when executing tests we actually implement this by performing a real file deletion, but the *event* that we observe is just WRITE$_N$ $\perp \to V_{old}$, where V_{old} is the contents of the file just before deletion. Since the model already encompasses WRITE events, we might have expected that no further changes would be required. But QuickCheck soon finds this failing test case:

Client 1	Client 2	Client 3
WRITE$_1$ 'a' $\to \perp$		
	READ$_2$ $\to$ 'a'	
WRITE$_1$ $\perp \to$ 'a'		
	READ$_2$ $\to \perp$	
		WRITE$_3$ 'b' $\to$ 'a'
READ$_1$ $\to$ 'b'		

Why does this test fail? Because: at step 4, the server value must be $\perp$, since client 2 saw $\perp$ after client 1 wrote it (and client 2 previously read the value 'a', so client 2 is not simply reading the initial $\perp$); at step 5, because the value client 3 overwrites is not the server value, a conflict is created; the value 'b' should thus *only* appear in conflict files on other nodes, never as the value in the file itself—yet it does just that in the last step. Thus, there can be no explanation for this observation.

(This test is, of course, quite sensitive to timing. For example, the second operation (READ$_2$ $\to$ 'a') reads the value written by the first WRITE$_1$ 'a' $\to \perp$. This is only possible if enough time passes after the first WRITE to allow the synchronizer to act. The actual test case includes SLEEP operations recording the need for these pauses, but they are not shown in the observations we present. We will return to the question of timing in more detail in Section VI.)

We have discovered another inconsistency with our model, but not yet a bug: in fact, the behavior we are seeing reflects another sensible design decision—that when a delete and a write conflict, the write should take precedence (and the deletion should be silently forgotten). We must amend our model to reflect this too:

UP$_N$
Precondition: $Clean?_N =$ DIRTY
Effect: $Clean?_N \leftarrow$ CLEAN
  if $Fresh?_N =$ FRESH then
    if $LocalVal_N \neq ServerVal$ then
      $Fresh?_{N'} \leftarrow$ STALE for all $N' \neq N$
      $ServerVal \leftarrow LocalVal_N$
  else
    if $LocalVal_N \notin \{ServerVal, \perp\}$ then
      $Conflicts \leftarrow Conflicts \cup \{LocalVal_N\}$

The change is in the second-to-last line, which now states that neither uploading the same value as the stable value, *nor a deletion*, ever generates a conflict. With this change, we believe our model reflects the intended behavior of the synchronizers we have tested.

V. SURPRISES

What about unintended behaviors?

Dropbox Surprises

Up to this point, we were essentially debugging our model using Dropbox as a reference implementation. However, con-

tinued testing revealed further inconsistencies between our model and Dropbox.

The first surprise was that *Dropbox can (briefly) delete a newly created file*:

Client 1	Client 2
WRITE$_1$ 'a' $\rightarrow \perp$ WRITE$_1$ $\perp \rightarrow$ 'a' WRITE$_1$ 'c' $\rightarrow \perp$ READ$_1$ $\rightarrow \perp$	WRITE$_2$ 'b' $\rightarrow$ 'a'

In this case, WRITE$_2$ 'b' and WRITE$_1$ 'c' are in conflict; the final READ$_1$ should see one of these values, with the other eventually appearing in a conflict file—but at the time we try to read the file, it is not there at all! In this case stabilization would restore a correct file contents, but the test fails because our model does not allow the file to be missing, even briefly. (Of course, observing this transient behavior requires executing the operations at just the right times; the test case found by QuickCheck includes SLEEP operations that make this more likely.)

The second surprise was that *Dropbox can re-create deleted files*, even when only *one* client is modifying the file![4]

Client 1
WRITE$_1$ 'b' $\rightarrow \perp$
WRITE$_1$ $\perp \rightarrow$ 'b'
READ$_1$ $\rightarrow$ 'b'

In this case Dropbox does not later 'correct the mistake' by deleting the file again: it remains there permanently. (Again, of course, Dropbox does not *always* restore files after they have been deleted: to provoke this behavior, the test case must include a SLEEP operation of just the right length.)

A similar test shows that *deleted files can reappear even if the creation and deletion take place on different nodes*:

Client 1	Client 2
WRITE$_1$ $\perp \rightarrow$ 'b' READ$_1$ $\rightarrow \perp$	WRITE$_2$ 'b' $\rightarrow \perp$
STABILIZE $\rightarrow$ ('b', $\emptyset$)	

The READ$_1$ $\rightarrow \perp$ verifies that the file was properly deleted; but, after waiting for the system to stabilize, it reappears.

The most alarming behavior we observed shows that *Dropbox can lose data completely*:

Client 1	Client 2
WRITE$_1$ 'a' $\rightarrow$ 'b' READ$_1$ $\rightarrow$ 'a'	WRITE$_2$ 'b' $\rightarrow \perp$
STABILIZE $\rightarrow$ { ('a', $\emptyset$), ('b', $\emptyset$) }	

[4]There were other machines logged in to the same Dropbox account, so Dropbox *was* synchronizing the file at the same time to these other machines. However, they were not involved in the test, and were not modifying the files in question; they were simply passive observers.

Here the file is created on one client, synchronized to the other, and overwritten there—but Dropbox does *not* copy the new value to the other client, and so the system never becomes stable. Further investigation shows that Client 1 behaves as though it is still FRESH, rather than STALE, so Client 2 never sees the value that Client 1 wrote, and if Client 2 writes another value to the file then it is just copied onto Client 1—no conflict is detected, and the value 'a' is lost forever. Again, the behavior is timing dependent, occuring when the second WRITE happens very soon after the value 'b' arrives on Client 1.

Dropbox offered the following response to these surprises: *Our engineers thoroughly investigated the file and data issues and were able to reproduce them programatically. Fortunately, we don't believe these issues have occurred outside of the lab due to the precise conditions necessary for any data loss to occur–namely, the local edit occurring within the same second as the remote change and the saved file having the identical file size as the original. While that fact gives us comfort, Dropbox takes any potential data issue, no matter how remote in possibility, extremely seriously and we are developing fixes for the issue. We are grateful to the researchers for their efforts in testing the Dropbox service using property-based testing and raising awareness of property-based testing within Dropbox.*

ownCloud Surprises

While most of our effort has been spent testing Dropbox, we have also used our model (unchanged) to test ownCloud and Google Drive. So far, Google Drive has behaved as expected, but we elicited some surprising behavior from ownCloud.

The first surprise was that *ownCloud can delete newly created directories*, instead of propagating them to other nodes. More surprising yet, we actually discovered this when our *test setup* failed! To mitigate the risk of one test case interfering with the next, we run each test in a new directory. We create these directories in batches, to reduce the time spent waiting for them to propagate to all the nodes. At the start of a test run, we delete left-over test directories, then recreate the ones we need. So, our preparation for a testing run looks like this: (1) On the host, delete all the left-over directories from previous tests. (2) On each virtual machine, wait for the left-over directories to disappear. (3) On the host, create several hundred 'fresh' directories for the first few hundred test cases to use. (4) On each virtual machine, wait for all these new directories to appear. To our surprise, when using ownCloud as the synchronizer, step (4) often failed to terminate. On checking progress, we found that not only had the test directories not appeared on the virtual machines, but *they had been deleted from the host*! We surmise that ownCloud may arrange deletion followed by recreation of the same directory in the wrong order, if they occur sufficiently close together in time.

After we worked around this issue, QuickCheck found one further discrepancy between the specification and ownCloud's

actual behavior: *ownCloud can lose changes*. The following observation illustrates what can happen:

Client 1	Client 2
WRITE$_1$ 'a' $\rightarrow \perp$	WRITE$_2$ 'b' $\rightarrow$ 'a'
WRITE$_1$ 'c' $\rightarrow$ 'a'	
STABILIZE $\rightarrow$ ('b', $\emptyset$)	

At the end, all clients have stabilized on the value 'b', while 'c' has been completely forgotten even though it was written independently from 'b' (both writes saw the previous value 'a') and, according to the specification, it should at least appear in the final conflict set. This time, we could confirm the reason by reading the ownCloud source code. The ownCloud client uses a simple test for when a file has been changed and needs to be uploaded to the server: it checks whether either the file's modification time or its length are different from their last seen values. But modification times are recorded in the filesystem with a 1-second granularity. So if the file is written twice in quick succession (i.e., during the same second) and the new contents is the same length as the old one, no change will be detected. Thus, 'b' is recorded as the file's stable value: no matter how long we wait, the value ('c') will never reach the server or Client 2. If the next write to the file occurs on Client 2, 'c' will be silently overwritten.

VI. Pragmatics of Testing

Many of the behaviors we encountered were timing dependent (we have not included timings here, since their values will vary with factors like connection speed). Our main technique for provoking timing-dependent behaviors was to include SLEEP operations in our tests, which cause the whole testing framework to pause for a specific period (up to one second, randomly chosen during test generation).

We found that timing-dependent tests often failed with fairly low probability, which we could increase manually—once we'd identified a test case that sometimes failed—by 'triggering' some of the WRITE operations on changes in the file. That is, we busy-wait on some node until a specified value appears in the file and then immediately execute an operation. Triggering an operation makes unexpected behavior more likely, since it may create a race condition between the test code running on node N, and the synchronization dæmon on that node. For example, it allowed us to observe the last two Dropbox surprises quite repeatably. It would be interesting to go a step further and automatically *generate* triggered operations as part of test cases; this would require a slightly richer generation-time state so that we can predict what value(s) might appear in the file.

We found the *slowness* of file synchronizers to be quite a problem; also the unpredictability of synchronization time. It is not easy to tell when synchronization is complete—in particular, the icons that synchronizers display to show their status are often wrong: the local dæmon itself is confused about what state things are in! Yet we must know, if we are to detect synchronization *failures* reliably. It does not work just to allow a fixed time for synchronization to complete, because the more file operations a test case performs, the slower synchronization becomes. It appears that synchronizers 'back off' when files are changing rapidly, waiting for a more opportune moment to do their job. This was the original reason for including stabilization operations in our test cases, combining observations from all of the virtual machines to try to detect when the synchronizer has nothing left to do.

The examples we presented above are *minimized* test cases, found by QuickCheck's shrinking search. Shrinking tries to reduce the size of test cases by dropping calls from the sequence, but also reducing the duration of SLEEP operations—shrunk test cases should wait no longer than necessary to provoke a failure. We also noticed that counterexamples leading to wrong file contents were found in two forms: ending in a READ operation, or ending in a deletion (a WRITE of $\perp$), which also observes the contents. We configured shrinking so that deletions shrink to READ operations (provided the test still fails, of course), so that the latter form would shrink to the former.

Because of the non-determinism inherent in the system, failing test cases may not fail every time they are run. This is problematic both when searching for a failing test case, and when shrinking one. Our solution in both cases is to run each test several times, and consider the test to have failed if *any* of the runs fails—thus reducing the probability of a 'false negative' result. While running random tests, we repeated each test three times, but while shrinking test cases, we repeated each one twenty times. (We work harder to avoid false negatives during shrinking, because they lead QuickCheck to report non-minimal failing tests, which can waste a great deal of human debugging time. In consequence, shrinking a failed test to a minimal one can take 10–20 minutes to finish, at 10–15 seconds per test. Twenty repetitions was usually enough to minimize failing tests.)

VII. An Alternative Specification Attempt

There is an appealing analogy between file synchronization services and the memory subsystems of modern multiprocessors. Network hosts with copies of a replicated file correspond to individual processor cores, the local filesystems correspond to per-core caches, and the central synchronization server corresponds to the main memory. This suggests that one might try to leverage ideas from the literature on specifications of memory-system behavior (see [9], [10] for surveys) to specify the desired behavior of a synchronizer. In particular, perhaps a specification could be based on Lamport's notion of *happened before* relations [11], which express the causal ordering of events in a distributed system. Indeed, an early version of our QuickCheck specification was written in this style, rather than the model-based style that we have described in this paper.

This early specification used the same notions of *tests* and *observed events* as our current one. But instead of trying to match the observed behavior against the transitions of a concrete model of the system (including the server), it directly specified which observations were legal by attempting

to construct a partial ordering $\prec$ on the observed events such that (1) if an event e happened before e' on the same client, then $e \prec e'$, and (2) if e is the WRITE event that writes the value observed by e', then $e \prec e'$. If such a relation exists, then it can be taken as an explanation for the observation. For example, the validity of the observation

Client 1	Client 2
e_1 : WRITE_1 'a' $\rightarrow \perp$	
e_2 : WRITE_1 'b' $\rightarrow$ 'a'	
	e_3 : WRITE_2 'c' $\rightarrow$ 'b'

is justified by this relation $e_1 \prec e_2 \prec e_3$. On the other hand, if no $\prec$ relation exists that is consistent with the observations, then a bug (or at least a discrepancy between the system and the spec) has been detected. For example, the observation

Client 1	Client 2
WRITE_1 'a' $\rightarrow \perp$	
WRITE_1 'b' $\rightarrow$ 'a'	
	$\text{READ}_2 \rightarrow$ 'b'
	$\text{READ}_2 \rightarrow$ 'a'

cannot be partially ordered in a way that respects the two conditions above. To deal with conflicts, we specified that, at stabilization points, all of the maximal values in the partial order (that is, the values written by every WRITE event e such that for no WRITE event e' do we have $e \prec e'$) must appear either as the local value or in the conflict set on all nodes.

Unfortunately, although this style of specification at first appeared quite natural and elegant, we found it difficult to extend to encompass all the behaviors we cared about. In particular, the fact that the same value can be written many times during the same test run (e.g., the value $\perp$ is written every time a file is deleted), renders the second condition above—"if e is *the* WRITE event that writes the value observed by e'..."—impossible to test with certainty.

We tried dealing with this indeterminacy by constructing $\prec$ relations for *all* possible ways of matching WRITES with later observations (and accepting a test case if we succeeded for any one of them), but the result was tricky to implement and slow because the set of possibilities quickly became large. Fortunately, this failed attempt gave us the idea of working with limited knowledge about what the system is doing by explicitly maintaining sets of possibilities, leading to the current model-based specification.

VIII. Related work

As far as we know, this is the first work to address testing a distributed synchronization service. But the problems of *specifying* the behavior of synchronizers and of *testing* the behavior of nondterministic and distributed systems have both received considerable attention.

A series of formal specifications of the Unison file synchronizer [12] by Pierce and Balasubramaiam [1], Pierce and Vouillon [2] and Ramsey and Csirmaz [3] were the starting point for the present work. Those specifications go further than ours in that they deal with multiple files and directories—in particular, they spend considerable effort on the subtleties of conflicts in this setting. However, they address a different distribution scenario, in which the execution of the synchronizer is a visible user-initiated action rather than a continuous background activity and in which there is no centralized "global value." They have not been used for testing.

We have implicitly assumed that the *local* filesystem is behaving correctly (so that discrepancies between our model and actual observations are attributable to Dropbox). Ridge et al. [13] show how this can be tested, using a specification with significant similarities to ours.

A distinct body of specification work deals with specifying the behavior of *operational transform* services—middleware layers that maintain consistency of replicated data structures (databases, documents, spreadsheets, etc.) under concurrent updates. Operational transform algorithms are widely used—for example, they underlie behind the collaboration features in Apache Wave and Google Docs—and their theory is well developed [14]–[16, etc.]. However, although it has been used for debugging replication algorithms using symbolic model-checking [17], the theory has not, to our knowledge, been applied to testing of actual distributed implementations. Indeed, since these specifications are based on notions of causal ordering, our experiences reported in Section VII suggest that it might be difficult to do so, at least in a black-box style.

Fraser and Wotowa [18] present a model-based testing method for non-deterministic systems, using a model-checker to generate test cases (sequences of transitions in the model) which fulfill selected structural coverage criteria. But when test cases are *run*, the implementation may choose to make different transitions, because of non-determinism. If the implementation diverges from the model at a deterministic point, then the test fails, but if divergence occurs at a non-deterministic choice point, then the test is considered *inconclusive*. Fraser and Wotowa show how to take inconclusive tests and generate new branches, again using the model checker, that fulfill the selected coverage goal, starting from the state that the implementation chose. The test is repeated, and if the implementation follows either the original or the newly generated path, then the coverage goal is reached. If the implementation diverges again, then the test is still inconclusive, and another branch can be added in the same way. Tests generated in this way are tree-structured, and hopefully eventually the implementation will follow one of the paths in the tree, and the coverage goal will be reached.

Arcaini *et al.* [19] generate tests with a model checker in a similar way, but instead of introducing branches, they reuse the model as a *run-time monitor*, to check that even if the system follows a different path from the test case, then its input-output behavior still conforms to the model. They do assume that the *inputs* in the generated test can be supplied to the SUT even though it is following an unexpected path, and they assume that outputs from the SUT always provide enough information to uniquely identify the corresponding model state ('strong conformance'). They evaluate their approach using a Tic-Tac-Toe game, in which the model requires moves to be

valid, but does not specify *which* moves the computer player should make. The Java implementation must be annotated in order to link it to the model. No errors were discovered in the Tic-Tac-Toe implementation (but neither were any expected).

In comparison, we use *two* state machines, a trivial one for generating tests, and a more interesting one as a run-time monitor. Our monitoring state machine is *deterministic*, but includes unobservable transitions—eliding those transitions makes it non-deterministic. We *allow* multiple possible model states during monitoring ('weak conformance' in the sense of [19]), and we treat the SUT as a black box—there is no need for instrumentation of the *implementation* to connect it to our model. Our examples are more complex real applications, and we found a number of unexpected behaviors.

Ulrich and König propose architectures for testing distributed systems [20], but assume that all internal actions of the SUT are observable by the tester, and that software probes are inserted into the SUT to allow the tester to control the timing of communications between nodes, and ensure that test runs are deterministic. Neither assumption holds in our setting.

Boy *et al.* report on an approach to testing servers by running random sequences of API calls from a number of clients, and checking that specified invariants hold over the resulting traces [21]. The invariants are specified as patterns that are matched against the traces, and assertions that must hold if a pattern matches. Boy *et al.* found a subtle timing bug in a lock server using this method. The approach does not address 'conjectured events', however, and does not include shrinking—the lock-server bug was minimized by hand.

A variety of work on testing nondeterministic systems can be phrased in terms of the theory of *input-output conformance (ioco) testing* [22], [23]. Indeed, there are some suggestive similarities between aspects of this theory and the structures we used in specification and testing of synchronizers—e.g., the inclusion in its labeled transition systems of *quiescence* transitions, which are reminescent of our stabilization events. It would be interesting to try to reframe our development in terms of ioco concepts.

QuickCheck was originally developed in and for Haskell [5], and it has become the most widely used testing tool in that community. The version we used was developed by Quviq and supports Quviq's core business: specification-based testing tools and services. Quviq QuickCheck [24] extends the original version with libraries tailored for testing industrial software, such as the state machine library used here. It has been used to test many large systems, including telecoms products [4], a messaging gateway [25], refactoring tools [26], [27], and quadcopters [28]. Probably the largest application so far was to test AUTOSAR basic software (C code which runs in cars), in which a million lines of C was tested against 3,000 pages of the AUTOSAR standard, using 20,000 lines of QuickCheck code [29]. Most of these systems are deterministic, but QuickCheck has also been used to test for race conditions in concurrent programs [30], finding two long-standing race conditions in the database distributed with Erlang [31]. However, the approach to handling non-determinism in those papers is quite different to the one used here.

IX. Future Work

We have considered just the case of a single file. Naturally, there are interesting questions to ask about a synchronizer's behavior in the presence of multiple files and directory structures, such as "what happens when a directory is deleted on one client, while a file is written into that directory on another?"

Extending the testing framework to multiple files and directories will require slightly richer model states at test case generation time, including the paths that have been created so far, so that operations can stay within this set with high enough probability to provoke bugs.

One challenge that can be expected when we make this extension is that the set of possible system states given a sequence of observed events is likely to grow much more quickly (it will be exponential in the number of files), and we will probably need to find clever representations for this set. Possibilities include representing the set as a cartesian product of smaller sets—we can *overapproximate* the set of possible states without introducing false positives, so ideas from abstract interpretation [32] should be applicable here.

Another rich source of incorrect behaviors in distributed systems is network partitions. To provoke such behaviors, it might be useful to extend our test cases with operations to disconnect and reconnect hosts from the network.

X. Conclusions

We have described an executable formal specification of the core behavior of file synchronization services. Since it's written in a black-box style, avoiding synchronizer-specific APIs and communicating only via the filesystem, we were able to apply it to three popular synchronizers—two commercial, one open source—and found surprising behaviors in two of them. This shows the effectiveness of the method.

Given that three different synchronizers appear to share the same core specification, we expect that our model should be applicable (perhaps with small changes to the testing framework) to many others—e.g., Microsoft OneDrive, Box.net, SpiderOak, Sugarsync, Seafile, Pulse, Wuala, Teamdrive, Cloudme, Cx, etc. Given the surprising behaviors already found, this should be a valuable exercise.

Acknowledgments

We thank John Lai and other Dropbox engineers for their feedback. This work is partially funded by the EU FP7 project *PROWESS* (#317820), the Swedish Strategic Research Foundation (RAWFP), and the National Science Foundation (CCF-1421243).

References

[1] 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.

[2] B. C. Pierce and J. Vouillon, "What's in Unison? A formal specification and reference implementation of a file synchronizer," Dept. of Computer and Information Science, University of Pennsylvania, Tech. Rep. MS-CIS-03-36, 2004.

[3] N. Ramsey and E. Csirmaz, "An algebraic approach to file synchronization," in *Proceedings of the 8th European Software Engineering Conference*. ACM Press, 2001, pp. 175–185.

[4] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ser. ERLANG '06. New York, NY, USA: ACM, 2006, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/1159789.1159792

[5] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[7] J. Hughes, "Quickcheck testing for fun and profit," in *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, ser. PADL'07, 2007, pp. 1–32.

[8] U. Norell, H. Svensson, and T. Arts, "Testing blocking operations with quickcheck's component library," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 87–92. [Online]. Available: http://doi.acm.org/10.1145/2505305.2505310

[9] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[10] L. Higham, J. Kawash, and N. Verwaal, "Defining and comparing memory consistency models," in *In Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, 1997, pp. 349–356.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] B. C. Pierce, T. Jim, and J. Vouillon, "UNISON: A portable, cross-platform file synchronizer," 1999–present, http://www.cis.upenn.edu/~bcpierce/unison.

[13] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "Sibylfs: formal specification and oracle-based testing for posix and real-world file systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 38–53.

[14] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Acm Sigmod Record*, vol. 18, no. 2. ACM, 1989, pp. 399–407.

[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*. ACM, 1995, vol. 29, no. 5.

[16] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[17] H. Boucheneb, A. Imine, and M. Najem, "Symbolic model-checking of optimistic replication algorithms," in *Integrated Formal Methods*. Springer, 2010, pp. 89–104. [Online]. Available: https://hal.inria.fr/inria-00524535/document

[18] G. Fraser and F. Wotawa, "Test-case generation and coverage analysis for nondeterministic systems using model-checkers," in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*. IEEE, 2007, pp. 45–45.

[19] P. Arcaini, A. Gargantini, and E. Riccobene, "Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 178–187. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.309.7963&rep=rep1&type=pdf

[20] A. Ulrich and H. König, "Architectures for testing distributed systems," in *Testing of Communicating Systems*. Springer, 1999, pp. 93–108.

[21] N. Boy, J. Casper, C. Pacheco, and A. Williams, "Automated testing of distributed systems," May 2004, final project report for MIT 6.824: Distributed Computer Systems.

[22] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software—Concepts and Tools*, no. TR-CTIT-96-26, 1996. [Online]. Available: http://doc.utwente.nl/65463/1/Tre96-CTIT96-26.pdf

[23] ——, "Model based testing with labelled transition systems," in *Formal methods and testing*. Springer, 2008, pp. 1–38. [Online]. Available: http://liacs.leidenuniv.nl/~bonsanguemm/Toos/P9_TestingTransSyst.pdf

[24] J. Hughes, "Software testing with quickcheck," in *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, ser. CEFP'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 183–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939128.1939134

[25] J. Boberg, "Early fault detection with model-based testing," in *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ser. ERLANG '08. New York, NY, USA: ACM, 2008, pp. 9–20. [Online]. Available: http://doi.acm.org/10.1145/1411273.1411276

[26] D. Drienyovszky, D. Horpácsi, and S. Thompson, "Quickchecking refactoring tools," in *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '10. New York, NY, USA: ACM, 2010, pp. 75–80. [Online]. Available: http://doi.acm.org/10.1145/1863509.1863521

[27] H. Li and S. Thompson, "Implementation and application of functional languages," O. Chitil, Z. Horváth, and V. Zsók, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Testing Erlang Refactorings with QuickCheck, pp. 19–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85373-2_2

[28] B. Vedder, J. Vinter, and M. Jonsson, "Using simulation, fault injection and property-based testing to evaluate collision avoidance of a quad-copter system," in *Dependable Systems and Networks Workshops (DSN-W), 2015 IEEE International Conference on*, June 2015, pp. 104–111.

[29] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing autosar software with quickcheck," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–4.

[30] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger, "Finding race conditions in erlang with quickcheck and pulse," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '09. New York, NY, USA: ACM, 2009, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/1596550.1596574

[31] J. M. Hughes and H. Bolinder, "Testing a database for race conditions with quickcheck: None," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '11. New York, NY, USA: ACM, 2011, pp. 72–77. [Online]. Available: http://doi.acm.org/10.1145/2034654.2034667

[32] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.