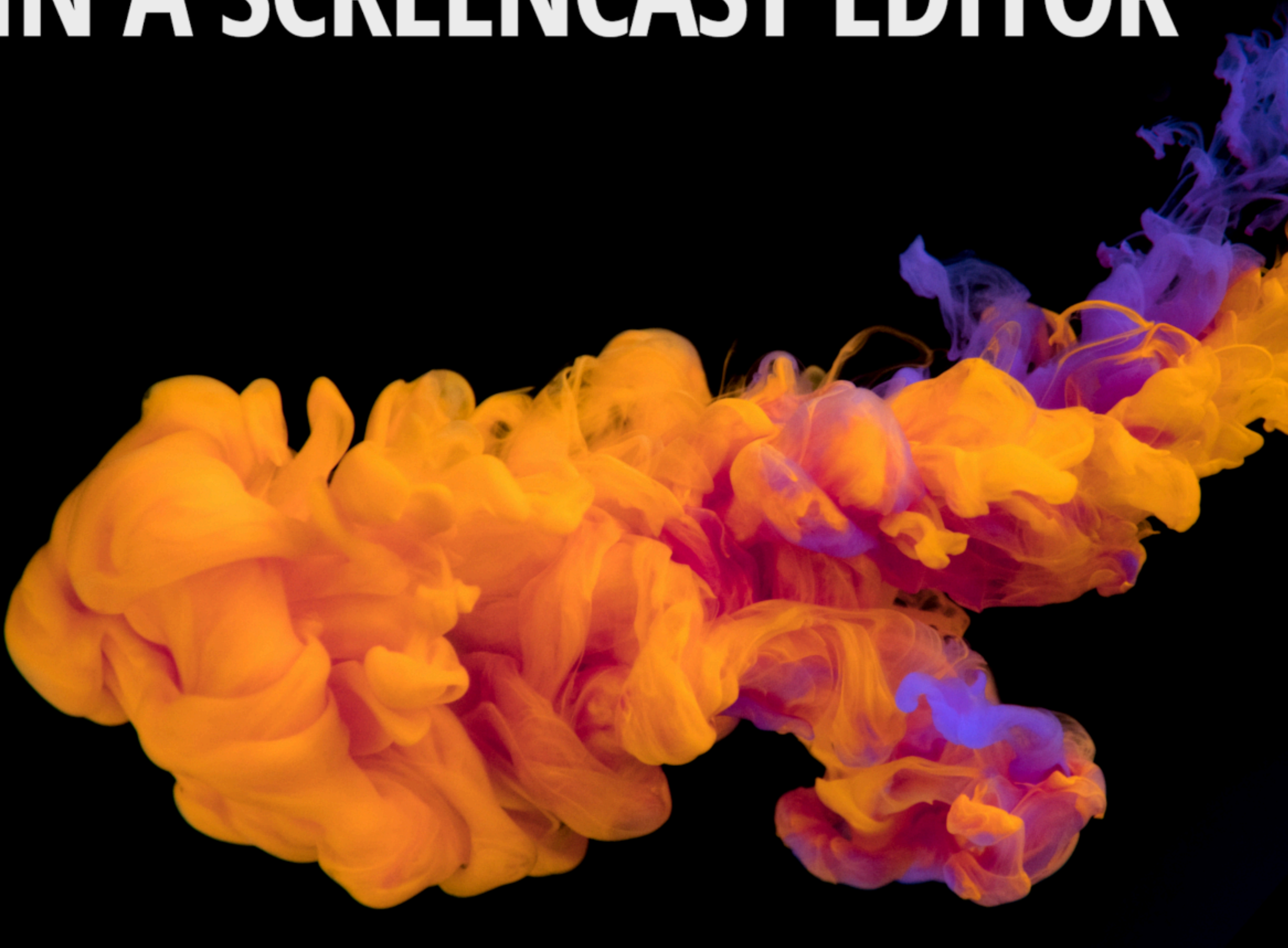
PROPERTY-BASED TESTING IN A SCREENCAST EDITOR



Case Studies from Komposition

Oskar Wickström

SAMPLE CHAPTER

August 2019

Preface

This is a short book on using property-based testing (PBT) within *Komposition*, a screencast editor. It's based on the articles published on wickstrom.tech[1], converted to book form.

PBT is a productive and delightful middle ground between the conventional testing techniques often used in industry and formal methods that are more commonly found in academia. While PBT doesn't offer the rigor of formal verification techniques, it's approachable and effective in real-world projects, making it a compelling choice.

I think that PBT should be leveraged more widely in industry. Just a few years ago, I didn't even know it existed. When I got into Haskell programming, I eventually learned about QuickCheck, a PBT framework. Most examples of using PBT are focused on teaching the essentials. When I wanted to leverage PBT in testing my screencast editor, the introductory examples didn't help. I got stuck, asking questions like "how do you test a GUI application using PBT?"

After a while of struggle, I learned how to leverage PBT in my project. This is why I wrote the articles on my website – to share my experiences, and hopefully to convince you and others that PBT can be useful in *your* projects. Not only on ideal pure functions, like list reversals or sorting algorithms, but in the code you write in your daily work.

Prerequisites

This is not a tutorial on PBT, but rather a collection of motivating examples. There are references in this book to other resources that explain the basics of PBT more thoroughly. If you are not familiar with PBT, be sure to check them out.

The code examples in this book are written in Haskell, using the Hedgehog testing framework. This is a consequence of Komposition using Haskell and Hedgehog. You are not expected to know much Haskell to follow along with this book. A basic understanding of functional programming should be enough.

How to Read This Book

The structure of this book is simple:

[1]https://wickstrom.tech

- First, it introduces the system under test (SUT): Komposition.
- Next, it goes into PBT and discusses challenges in testing properties of complex applications.
- The main part of the book is compromised of case studies. Each case study covers increasingly complex components and how they are tested. These include my personal reflections, what bugs the tests have found, and what still remains to be improved.
- Finally, the summary wraps up by motivating PBT, based on my experience.

You don't have to read this book in any particular order, but section **??** and **??** describe concepts that are useful to know about when reading the remainder of the book. Other than that, feel free to skip or jump around.

Notational & Typographical Conventions

This book uses a set of typographical conventions in body text and in code listings:

- *Italic* text is used for emphasis and for new terms
- `Constant Width` text is used for references to code elements, program listings, shell commands, and program output
- In Haskell program listings:
  - keywords (e.g. **`data`**) are bold
  - types, constructors, and modules (e.g. *`Text`*) are italic
  - circled numbers ((1), (2), ...) are annotations, not actual Haskell source code, used to mark sections of interest that are explained in prose

Abbreviations

PBT Property-based testing

SUT System under test

TDD Test-driven development

Credits

Thank you Chris Ford, Alejandro Serrano Mena, Tobias Pflug, Hillel Wayne, Ulrik Sandberg, Pontus Nagy, and Fredrik Björeman, for kindly providing your feedback on my drafts.

A huge thanks to my wife, Miranda, for your endless support, and for enduring lengthy outcries on programming, writing, and the Internet in general.

This book uses three fonts:

- Linux Libertine[2] (licensed under the GPL and Open Font License[3])
- Open Sans Condensed[4] (licensed under the Apache License Version 2.0[5])
- Iosevka[6] (licensed under the SIL OFL Version 1.1[7])

The cover photo is based on *Photograph of a Yellow and Purple Mixed Substance* by rawpixel.com from Pexels[8]. The fonts used are *Linux Libertine* and *Open Sans Condensed*, as in the book.

Emacs, XeLaTeX, Pandoc, pandoc-crossref, GNU make, Nix, GIMP, and GraphicsMagick were all used in writing this book.

Errata & Feedback

If you enjoy this book, please tell your friends, and share it in a social media channel. Spreading the word means more people learn about wonders of PBT.

If you find any typos, factual errors, or problems with the source code, please visit the Leanpub feedback page[9] and write a few words about it.

[2]http://libertine-fonts.org/
[3]http://libertine-fonts.org/libre/#more-89
[4]https://en.wikipedia.org/wiki/Open_Sans
[5]http://www.apache.org/licenses/LICENSE-2.0
[6]https://typeof.net/Iosevka/
[7]https://raw.githubusercontent.com/be5invis/Iosevka/master/LICENSE.md
[8]https://www.pexels.com/photo/photograph-of-a-yellow-and-purple-mixed-substance-1270954/
[9]https://leanpub.com/property-based-testing-in-a-screencast-editor/feedback

About the Author

After some years of musical education, Oskar Wickström began his journey into the world of software. He's now working remotely from the Swedish countryside, mostly focusing on functional programming, software architecture, and testing. He writes technical articles at wickstrom.tech[10], maintains various open source projects, and produces screencasts at Haskell at Work[11]. In his spare time, he enjoys long bike rides and working on his acoustic guitar compositions.

[10] https://wickstrom.tech
[11] https://haskell-at-work.com

Contents

1 Case Study: Video Scene Classification

In sec. **??**, we learned about timeline flattening. This case study covers the video classifier, how it was tested before, and the bugs that were found with property tests.

1.1 Classifying Scenes in Imported Video

Komposition can automatically classify *scenes* when importing video files. This is a central productivity feature in the application, effectively cutting recorded screencast material automatically, letting the user focus on arranging the scenes of their screencast. Scenes are segments that are considered *moving*, as opposed to *still* segments:

- A still segment is a sequence of at least S seconds of *near-equal* frames
- A moving segment is a sequence of *non-equal* frames, or a sequence of near-equal frames with a duration less than S

S is a preconfigured minimum still segment duration in Komposition. In the future it might be configurable from the user interface, but for now it's hard-coded.

Equality of two frames f_1 and f_2 is defined as a function $E(f_1, f_2)$, described informally as:

- comparing corresponding pixel color values of f_1 and f_2, with a small epsilon for tolerance of color variation, and
- deciding two frames equal when at least 99% of corresponding pixel pairs are considered equal.

In addition to the rules stated above, there are two edge cases:

1. The first segment is always a considered a moving segment (even if it's just a single frame)
2. The last segment may be a still segment with a duration less than S

The second edge case is not a desirable feature, but rather a shortcoming due to the classifier not doing any type of backtracking. This could be changed in the future.

1.2 Manually Testing the Classifier

The first version of the video classifier had no property tests. Instead, I wrote what I thought was a decent classifier algorithm, mostly messing around with various pixel buffer representations and parallel processing to achieve acceptable performance.

The only type of testing I had available, except for general use of the application, was a color-tinting utility. This was a separate program using the same classifier algorithm. It took as input a video file, and produced as output a video file where each frame was tinted green or red, for moving and still frames, respectively.

In fig. 1.1 you see the color-tinted output video based on a recent version of the classifier. It classifies still (fig. 1.1a) and moving (fig. 1.1b) segments rather accurately. Before I wrote property tests and fixed the bugs that I found, it did not look so pretty, flipping back and forth at seemingly random places.

At first, debugging the classifier with the color-tinting tool way seemed like a creative and powerful technique. But the feedback loop was long, having to record video, process it using the slow color-tinting program, and perform ocular inspection. In hindsight, I can conclude that PBT is far more effective for testing the classifier.

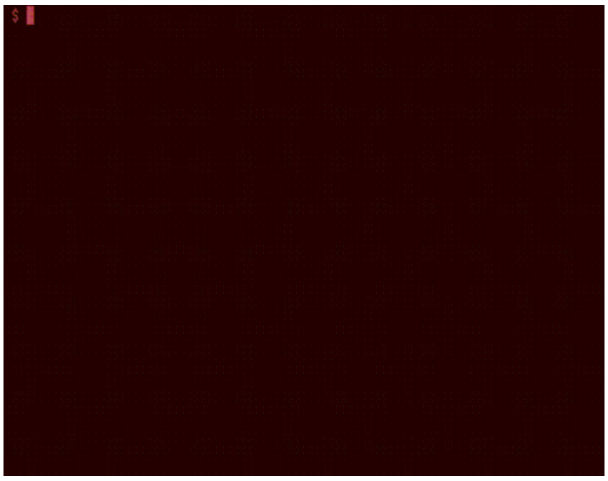
1.3 Video Classification Properties

Figuring out how to write property tests for video classification wasn't obvious to me. It's not uncommon in example-based testing that tests end up mirroring the structure of the SUT. The same can happen in PBT. With some complex systems it's very hard to describe the correctness as a relation between any valid input and the system's observed output. The video classifier is one such case. How do we decide if an output classification is correct for a specific input, without reimplementing the classification itself in the tests?

The other way around is easy, though! If we have a classification, we can convert that into video frames. Thus, the solution to the testing problem is to not generate the input, but instead generate the *expected output*. Hillel Wayne calls this technique "oracle generators" in *Finding Property Tests*[1].

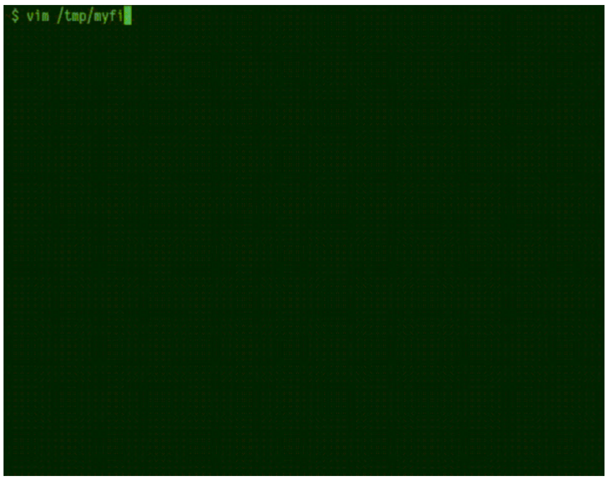
The classifier property tests generate high-level representations of the expected classification output, which are lists of values describing the type and duration of segments. fig. 1.2 illustrates how a generated sequence of classified segments might look.

[1] https://www.hillelwayne.com/post/contract-examples/



```
$
```

(a) Before any text is written, the frames remain unchanged, and the segment is classified as *still*



```
$ vim /tmp/myfi
```

(b) When the characters are entered, enough pixels change between frames, and the segment is classified as *moving*

Figure 1.1: Video classification shown with color tinting on a terminal session recording

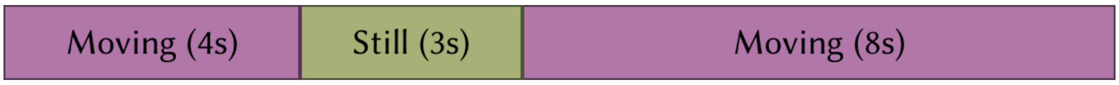


Figure 1.2: A generated sequence of expected classified segments

Next, the list of output segments is converted into a sequence of actual frames. Frames are two-dimensional arrays of RGB pixel values. The conversion is simple:

- Moving segments are converted to a sequence of alternating frames, flipping between all gray and all white pixels
- Still frames are converted to a sequence of frames containing all black pixels

The example sequence in fig. 1.2, when converted to pixel frames with a frame rate of 10 FPS, can be visualized like in fig. 1.3, where each thin rectangle represents a frame.

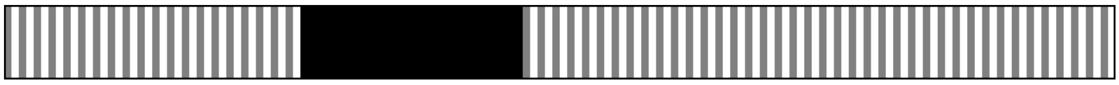


Figure 1.3: Pixel frames derived from a sequence of expected classified output segments

By generating high-level output and converting it to pixel frames, we have input to feed the classifier with, and we know what output it should produce. Writing effective property tests then comes down to writing generators that produce valid output, according to the specification of the classifier. In this chapter we'll study two such property tests.

1.4 Testing Still Segment Minimum Length

As stated in sec. **??**, classified still segments must have a duration greater than or equal to S, where S is the minimum still segment duration used as a parameter for the classifier. The first property, shown in lst. 1.1, asserts that this invariant holds for all classification output.

This chunk of test code is pretty busy, and it's using a few helper functions that I'm not going to bore you with. At a high level, this test:

- Generates a minimum still segment duration (❶) based on a minimum frame count (let's call it n) in the range $[2, 20]$ at a frame rate of 10. The classifier currently requires that $n \geq 2$, hence the lower bound. The upper bound of 20 frames is an arbitrary number that I've chosen.
- Generates valid output segments using the custom generator `genSegments` (❷), where
  - moving segments have a frame count in $[1, 2n]$, and
  - still segments have a frame count in $[n, 2n]$.

Listing 1.1 A property checking that still segments are at least of the minimum length

```
hprop_classifies_still_segments_of_min_length = property $ do
  minStillSegmentFrames <- forAll $ ❶
    Gen.int (Range.linear 2 (2 * frameRate))
  let minStillSegmentTime = frameCountDuration minStillSegmentFrames

  segments <- forAll $ ❷
    genSegments (Range.linear 1 10)
                (Range.linear 1
                              (minStillSegmentFrames * 2))
                (Range.linear minStillSegmentFrames
                              (minStillSegmentFrames * 2))
                resolution

  let pixelFrames = testSegmentsToPixelFrames segments ❸

  let counted =
        classifyMovement minStillSegmentTime (Pipes.each pixelFrames) ❹
          & Pipes.toList
          & countSegments

  countTestSegmentFrames segments === totalClassifiedFrames counted ❺

  case initMay counted of ❻
    Just rest ->
      traverse_ (assertStillLengthAtLeast minStillSegmentTime) rest ❼
    Nothing -> success
  where resolution = 10 :. 10
```

- Converts the generated output segments to actual pixel frames (❸). This is done using a helper function that returns a list of alternating gray and white frames, or all black frames, as described in sec. ??.
- Counts the number of consecutive frames within each segment (❹), producing a list like `[Moving 18, Still 5, Moving 12, Still 30]`.
- Performs a sanity check (❺) that the number of frames in the generated expected output is equal to the number of frames in the classified output. The classifier must not lose or duplicate frames.
- Drops the last classified segment (❻), which according to the specification can have a frame count less than n, and asserts that all other still segments have a frame count greater than or equal to n (❼).

Running 10000 tests, they all pass. Looks like it's working.

1.5 Sidetrack: Why generate the output?

Now, you might wonder why we generate output segments first, and then convert to pixel frames. Why not generate random pixel frames to begin with? The property test in lst. 1.1 only checks that the still segments are long enough, anyway.

The benefit of generating valid output becomes clearer in the next property test, where we use it as the expected output of the classifier. Converting the output to a sequence of pixel frames is easy, and we don't have to state any complex relation between the input and output in the property. When using oracle generators, the assertions can often be plain equality checks on generated and actual output.

But there's benefit in using the same oracle generator for the *minimum still segment length* property, even if it's more subtle. By generating valid output and converting to pixel frames, we can generate inputs that cover the edge cases of the system under test. Using property test statistics and coverage checks, we could inspect coverage, and even fail test runs where the generators don't hit enough of the cases we're interested in. John Hughes' talk *Building on developers' intuitions*[2] goes into depth on this topic. When these tests were written, Hedgehog did not support coverage checking. As of version 1.0[3], classification and coverage checks are supported.

Had we generated random sequences of pixel frames, it'd be likely the majority of generated examples would be moving segments. We could tweak the generator to get closer to either moving or still frames, within some distribution, but wouldn't that just

[2]https://www.youtube.com/watch?v=NcJOiQlzlXQ
[3]https://hackage.haskell.org/package/hedgehog-1.0

be a variation of generating valid scenes? It would be worse, in fact. We wouldn't then be reusing existing generators, and we wouldn't have a high-level representation to convert from and compare with in assertions.

1.6 Testing Moving Segment Time Spans

The second property, shown in lst. 1.2, asserts that the classified moving segments start and end at the same timestamps as the moving segments in the generated output. Compared to the previous property, the relation between generated output and actual classified output is stronger.

The arrangement is the same as in the previous property test (in lst. 1.1). This property test differs by:

- Converting the generated output segments into a list of time spans (❶). Each time span marks the start and end of an expected moving segment. Furthermore, it needs the full duration of the input (❷) when running the classifier.
- Classifying the movement of each frame (❸), i.e. deciding if it's part of a moving or still segment.
- Running the second classifier function called `classifyMovingScenes`, based on the full duration and the frames with classified movement data, resulting in a list of time spans (❹).
- Comparing the expected and actual classified list of time spans (❺).

While this property test looks somewhat complicated with its setup and various conversions, the core idea is simple. But is it effective?

1.6.1 Bugs! Bugs everywhere!

Preparing for a talk on PBT, I added the *moving segment time spans* property a week or so before the event. At this time, I had used Komposition to edit multiple screencasts. Surely, all significant bugs were caught already. Adding property tests should only confirm the level of quality the application already had. Right?

That was not the case. First, I discovered that my existing tests were fundamentally incorrect to begin with. They were not reflecting the specification I had in mind, the one described in sec. ??. Moreover, I found that the generators had errors. At first, I used Hedgehog to generate the pixels used for the classifier input. Moving frames were based on a majority of randomly colored pixels and a small percentage of equally colored pixels. Still frames were based on a random single color.

Listing 1.2 A property checking the start and end times of classified segments

```
hprop_classifies_same_scenes_as_input = property $ do
  minStillSegmentFrames <- forAll $ Gen.int (Range.linear 2 (2 * frameRate))
  let minStillSegmentTime = frameCountDuration minStillSegmentFrames

  segments <- forAll $ genSegments (Range.linear 1 10)
                                   (Range.linear 1
                                                 (minStillSegmentFrames * 2))
                                   (Range.linear minStillSegmentFrames
                                                 (minStillSegmentFrames * 2))
                                   resolution

  let pixelFrames = testSegmentsToPixelFrames segments

  let durations = map segmentWithDuration segments ❶
      expectedSegments = movingSceneTimeSpans durations
      fullDuration = foldMap unwrapSegment durations ❷

  let classifiedFrames = ❸
        Pipes.each pixelFrames
        & classifyMovement minStillSegmentTime
        & Pipes.toList

  let classified = ❹
        (Pipes.each classifiedFrames
         & classifyMovingScenes fullDuration)
        >-> Pipes.drain
        & Pipes.runEffect
        & runIdentity
  expectedSegments === classified ❺

  where resolution = 10 :. 10
```

The problem I had not anticipated was that the colors used in moving frames were not guaranteed to be distinct from the color used in still frames. In small-sized examples, where color values were picked close to 0, I got black frames at the beginning and end of moving segments, and black frames for still segments, resulting in different classified output than expected. Hedgehog shrinking the failing examples' colors towards 0, which is again black, highlighted this problem even more.

I made my generators much simpler, using the alternating white/gray frames approach described in sec. **??**, and went on to running my new shiny tests. fig. 1.4 shows the results I got. Where does 0s–0.6s come from? The classified time span should've been 0s–1s, as the generated output has a single moving scene of 10 frames (1 second at 10 FPS). I started digging, using the `annotate` function in Hedgehog to inspect the generated and intermediate values in failing examples.

```
┏━ test/Komposition/Import/Video/FFmpegTest.hs ━━
hprop_classifies_same_scenes_as_input = withTests 100 . property $ do
  -- Generate test segments
  segments <- forAll $ genSegments (Range.linear (frameRate * 1) (frameRate * 5)) resolution
  │ [ Scene 10 ]
  -- Convert test segments to timespanned ones, and actual pixel frames
  let segmentsWithTimespans = segments
                              & map segmentWithDuration
                              & segmentTimeSpans
      pixelFrames = testSegmentsToPixelFrames segments
      fullDuration = foldMap
                     (durationOf AdjustedDuration . unwrapSegment)
                     segmentsWithTimespans
  -- Run classifier on pixel frames
  classified <-
    (Pipes.each pixelFrames
     & classifyMovement 2.0
     & classifyMovingScenes fullDuration)
    >-> Pipes.drain
    & Pipes.runEffect
  -- Check classified timespan equivalence
  unwrapScenes segmentsWithTimespans === classified
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    │ Failed (- lhs =/= + rhs)
    │   [
    │ -   TimeSpan { spanStart = Duration 0 s , spanEnd = Duration 1 s }
    │ +   TimeSpan { spanStart = Duration 0 s , spanEnd = Duration 0.6 s }
    │   ]

  where resolution = 20 :. 20
```

Figure 1.4: A property test failing due to unexpected classified segments

I couldn't find anything incorrect in the generated data, so I shifted focus to the implementation code. The end timestamp 0.6s was consistently showing up in failing examples. Looking at the code, I found a curious hard-coded value 0.5 being bound and used locally in `classifyMovement`. The function is essentially a *fold* over a stream of frames, where the accumulator holds vectors of previously seen and not-yet-classified frames.

Stripping down and simplifying the old code to highlight one of the bugs, it looked something like the code in lst. 1.3. In the `InStillState` branch it uses the value `minEqualTimeForStill`, instead of always using the `minStillSegmentTime` argument (❶). This is likely a residue from a refactoring, where I meant to make the value a parameter instead of having it hard-coded in the definition.

Listing 1.3 A simplified version of the old frame classifier function

```haskell
classifyMovement minStillSegmentTime =
  case ... of
    InStillState{..} ->
      if someDiff > minEqualTimeForStill ❶
        then ...
        else ...
    InMovingState{..} ->
      if someOtherDiff >= minStillSegmentTime
        then ...
        else ...
  where
    minEqualTimeForStill = 0.5
```

Sparing you the gory implementation details, I'll outline two more problems that I found:

- In addition to using the hard-coded value, it incorrectly classified frames based on that value. Frames that should've been classified as "moving" ended up "still". That's why I didn't get 0s–1s in the output.
- Why didn't I see 0s–0.5s, given the hard-coded value 0.5? Because there was also an off-by-one bug, in which one frame was classified incorrectly together with the accumulated moving frames.

The `classifyMovement` function is 30 lines of Haskell code juggling some state, and I managed to mess it up in three separate ways at the same time. With these tests in place I quickly found the bugs and fixed them. I ran thousands of tests, all passing. Finally, I ran the application, imported a previously recorded video, and edited a short screencast. The classified moving segments were *notably* better than before.

1.7 Summary

A simple streaming fold can hide bugs that are hard to detect with manual testing. The consistent result of 0.6, together with the hard-coded value 0.5 and a frame rate of 10 FPS, pointed clearly towards an off-by-one bug. This is a great showcase of how powerful shrinking in PBT is, consistently presenting minimal examples that point towards specific problems.

Could these errors have been caught without PBT? I think so, but what effort would it require? Manual testing and introspection did not work for me. Code review might have revealed the incorrect definition of `minEqualTimeForStill`, but perhaps not the off-by-one and incorrect state handling bugs. There are of course many other QA techniques, and I won't evaluate all. But given the low effort that PBT requires in this setting, the amount of problems it finds, and the accuracy it provides when troubleshooting, I think it's a great choice.