Property Driven Development in Erlang, by Example

Samuel Rivas
R&D Department
LambdaStream S.L.
samuel.rivas@lambdastream.com

Miguel Ángel Francisco
R&D Department
LambdaStream S.L.
miguel.francisco@lambdastream.com

Víctor M. Gulías
MADS Group
University of A Coruña, Spain
gulias@udc.es

ABSTRACT
As an example of the combination of Test Driven Development and Property Based Testing, this paper presents the implementation of a template library from scratch using the functional language Erlang for development and the tool QuickCheck for testing.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Testing Tools

General Terms
Design, Reliability, Verification

Keywords
TDD, Erlang, QuickCheck, property based testing

1. INTRODUCTION
Testing has become an integral part of any development methodology. Initial methodologies delayed testing until the end of the project, but the modern tendency is to promote testing as soon as possible [6]. The main benefit from early testing is quick feedback about software quality. In the long term, high quality test suites ease fast software changes preserving good quality results, allowing developers to quickly adapt to changing requirements.

Traditionally, the workflow for writing automated test suites implied writing them after production code. With this approach, however, tests are likely to be skipped under time pressure. Test driven development [5, 10] (TDD) addresses this problem: developers do not write any production code without having a failing test case first, that results in greater test coverage and less untested code.

In the long term, automated testing suites greatly reduce the effort invested in testing evolutionary software at the cost of writing the testing code. Yet, to be profitable, programmers must limit the scope of tests to meet deadlines. This may have a negative impact on testing suites quality.

Property-based testing [8, 9] aims to increase test coverage without proportionally increasing the effort of writing test suites. QuickCheck, a tool initially implemented for Haskell [7] and later ported to Erlang [1] uses properties to guide the generation of random test cases, reporting any failure as counterexample. It also shrinks counterexamples to find simpler ones to ease fault diagnosis. Hence, developers concentrate on generating good data distribution to challenge the defined properties, rather than on concrete tests.

Previous QuickCheck studies [2, 3, 12, 13] are experiments retrofitting properties in already existing software. In this work, we use Property Driven Development (PDD) [4] to drive a development from scratch.

2. IMPLEMENTING THE CASE STUDY
The proposed case study is to write a template library in Erlang using the PDD process described in figure 1. We will focus on a function that inserts literals in a string with variable placeholders enclosed between @ characters. Thus, "hello @name@" is a template with a placeholder name; replacing name by luke yields "hello luke". A public repository1 has been published so that interested readers can follow the code changes in detail.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
prop_string_empty_list() ->
    ?FORALL(S, ql_gen:string(),
        S == lstd_template:string(S, [])).

The string generator (from the ql_gen library) generates a list, possibly empty, of printable characters. A trivial implementation to meet the property could be:

string(String, _Subs) -> String.

It is unlikely that properties are implemented absolutely correct the first time. In fact, this property is wrong and will fail later, after adding more requirements. However, starting simple ensures that behaviour and properties evolve as parallel as possible, so that any feature has tests baking it up and all tests have failed in the past, ensuring that they test behaviour introduced after writing them.

2.2 Implementing the Scanner

The next step is to define properties on substitution. That is achieved more easily as a combination of simpler properties. Let’s start thinking about syntax, for example, specifying a scanner to detect template tokens.

To write properties, the expected output for any generated input must be known in advance. A common technique is to generate both a representation of the input and its expected output, along with functions to derive the actual input and output from it. The following data types are defined to represent a template with variables and text sections:

template() = [{var, string()} | {text, string()}]
text() = string()

Thus, [{text,"hi "},{var,"name"}] is the representation of "hi @name@". A possible first generator:

template() ->
    eqc_gen:list(
        eqc_gen:oneof(
            [{var,ql_gen:string()}, {text,ql_gen:string()}]).
    )

text() = text()

From the template representation, it is possible to generate its string form—the one used by production code—and the expected list of tokens implementing functions like these:

fold_text([],_ = []) -> [].
fold_text([H|T],_ = [H|T]) -> [H|fold_text(T)]

This defines the terminals for template language string literals and at symbols. The scanner can now be specified as for all templates, tokens return the expected token list:

prop_tokens() ->
    ?FORALL(T,template(),
        to_tokens(T) = lstd_template:tokens(to_string(T))).

Generated tests fail; we can start writing the scanner:

tokens([],_ = []) -> []
tokens([H|T],_ = [at|tokens(T)])
tokens(S) =
    fold_text([S],_ = fold_text([S]))

It may seem that this straightforward implementation is correct and complies with the specification, but it has a number of issues. QuickCheck, reports the next counterexample:

> eqc:quickcheck(lstd_template_eqc:prop_tokens()).
..Failed! After 3 tests.
[[text,[]]]
false

It seems we have a corner case when using empty strings as text. Debugging in the shell we find the reason: This is an error in the specification, right now it is impossible to scan the empty string as a template deterministically:

> lstd_template_eqc:to_tokens([[text,""]]).
[[text,[]]]
> lstd_template_eqc:to_string([[text,""]]).
[]
> lstd_template:tokens("").
[]

We fix it using a non-empty string generator for text.

Another counterexample is found: [[var,[ ]]]. Here, the question is how to scan strings like "@@#". Should we consider it as empty text in the middle? We decide not to do that, but use the non-empty string generator for variables as well.

The code also fails with [[text, ""],[text, " "]]. The scanner cannot distinguish between two consecutive string sections and a single one concatenating their texts. Thus, a normalization of string sections is added to the generator.

Next, QuickCheck finds an error in a somewhat complex template ([[var,""],[var,"<0">],[var,"@@"]])

After shrinking to a smaller counterexample (C,ql_gen:non_empty_list(valid_char()).

The parser responsibility is to locate @ symbol pairs to split the input in text terminals or variable terminals. At first glance, the same symbolic templates can be used as expected parser output, it has all the information the variable substitution process needs.

prop_parse() ->
    ?FORALL(T,template(),
        T == lstd_template:parse(to_string(T))).

The following simple implementation passes all the tests QuickCheck generates to prove the property above.

parse(Tokens) -> parse(Tokens,text).
parse([at|T], Terminal) ->
    parse(T,switch_terminal(Terminal));
parse([[string,3]|T], Terminal) ->
    [[Terminal,3]|parse(T,Terminal)];
parse([],_ = []) -> []
switch_terminal(text) -> var;
switch_terminal(var) -> text.

At this point, we have a reliable and easy to test implementation that transforms a string into a template symbolic representation. We also have properties that serve as specification and documentation for the parser and scanner.
2.4 Implementing the Substitution Process

Our symbolic templates must be enriched to write properties to specify the substitution process. The symbolic variables must also contain the value for the substitution:

\[
\text{var()} \rightarrow \{\text{var, valid_string()}, \text{ql_gen: string()!}\}
\]

After adapting \text{prop_parse} to this change, we formulate the next property, that defines any possible cases for \text{string}:

\[
\text{prop_string()} \rightarrow \text{?FORALL(T, template(), ?LET(\{Subs, Str, Expect\}, \{\text{to_subs(T), to_string(T), to_result(T)}\}, \text{Expect}:=\text{lstd_template:string(Str, Substs)!})}).
\]

This property does not hold with "@ @". The initial naive \text{string} implementation is no longer complying. Hence, there is a failing test that requires a modification in the current implementation of \text{string} in order to pass.

Note that, even \text{prop_string} is the core property of the template library, specifying it before the parser and the scanner allows the developer to work in small and easy to control steps. Attempting to write the right \text{prop_string} at the initial stages would require too much effort before passing all generated tests; the developer would probably end up with a solution similar to the one proposed here, but without having automated tests to back up the scanner and parser development. This is the new string function:

\[
\text{string(Str, Subs) \rightarrow app_sub(parse(tokens(Str)), Subs).}
\]

\[
\text{app_sub(Prsd, Subs) \rightarrow lists:concat(}
\]

\[
\{\text{to_string(X, Subs) || X<Prsd}\}
\]

\[
\{\text{to_string(\{text, T\}, Subs) \rightarrow T};"\}
\]

\[
\{\text{to_string(\{var, V\}, Subs) \rightarrow lstd_lists:keysearch(V, Subs)}\}
\]

And now \text{prop_string} holds. However, running the whole test suite, \text{prop_string_empty_list} now fails. The problem is that @ is now special. Thus, \text{string} being capable of replacing variables by content, not all strings remain untouched. Now some inputs are actually invalid. At this stage we may want to specify the error cases; for the sake of simplicity, we leave that out of the scope of this paper. To fix \text{prop_string_empty_list}, we just change the generator to create only strings without @ symbols.

2.5 Adding Support for Textual @ Symbols

There is a basic feature that the tests are not covering. The defined properties specify how the template library works for \text{template strings without @ symbols}. All we know is that placing @ symbols in the wrong place \text{may} raise exceptions, but we are not even sure what kind of exceptions or whether it raises exceptions at all. The behaviour for strings with misplaced @ symbols is \text{undefined}.

Negative testing apart, this undefined behaviour raises a question: how do we specify templates with @ symbols in the text part? The specification may explicitly forbid @ symbols in variable names, but we must allow their introduction in text sections. Before even thinking about how to implement it, we know that we must update the properties to consider that case. After that, there will be failing cases to guide the implementation.

Right now, @ is not valid in text sections; specification must change to include an escaped @ and, hence, the symbolic representation of templates must also be extended properly. We add an escaped_at symbol to text generator and some hints to guide escaped @ distribution:

\[
\text{text()} \rightarrow \text{eqc_gen: frequency(}
\]

\[
\{[], \text{to_string(\{valid_string()\})}, \{1, \text{escaped_at}\}\}.
\]

<table>
<thead>
<tr>
<th>string</th>
<th>tokens</th>
<th>parsed</th>
<th>result</th>
<th>subs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;@@&quot;</td>
<td>[at, at]</td>
<td>{text,&quot;@&quot;}</td>
<td>&quot;@&quot;</td>
<td>[]</td>
</tr>
</tbody>
</table>

Table 1: Interpretation of escaped_at

According to the specification of escaped_at shown in table 1, @ in text is escaped as @@; the scanner is not aware of any special treatment to this behaviour and the parser will not merge text sequences for @ symbols, i.e., "a@@" parses to \{\text{[text,"a"],\{text,"@"\}}\} instead of \{\text{[text,"a@"}\}}. The tests drove us to a working, and easy to test, solution.

Running the tests again reveals that only the \text{prop_parse} property does not hold. Thus, it is the only one that must be modified to support escaped @ symbols.

Now that the required behaviour is defined by a property, and there is a counterexample, the parser can be fixed to pass the tests. At this level, the test-first strategy starts to pay off with quick feedback to small changes. Suppose the next case to the parse function is included:

\[
\text{parse(\{at, at|T\}, Term=text)} \rightarrow \{\text{[text,"@"|parse(T, Term)}\};
\]

After executing the tests, we can see \text{prop_parse} still does not hold for "@ @@ @". If we debug in the shell:

\[
\text{prop_parse(\{at, at|T\}, Term=text)} \rightarrow \{\text{[text,"@"|parse(T, Term)}\};
\]

The parser is mistakenly interpreting the @@ sequence as a text terminal. The problem is that @@ sequences must be interpreted as text only when the parser is not waiting to close an open @. This modification fixes the problem:

\[
\text{parse(\{at, at|T\}, Term=text)} \rightarrow \{\text{[text,"@"|parse(T, Term)}\};
\]

And now all properties are holding again and the template library supports escaped @ symbols.

2.6 Specification and Generator Fixes

There is still a hidden bug in the properties that is difficult to spot, since the cause that triggers it is very unlikely to appear with the generators used so far. However, a failure like the following may show up scarcely (even for the test run to find counterexamples in previous sections):

\[
\text{prop_string: ....Failed! After 48 tests.}
\]

\[
\text{|\{var,\"%\",\{"\}|, \{var,\"W\",y)8uFE\},}
\]

\[
\text{|\{text,\:"5OH\",\"\}|, \{var,\"sPtd4BH\",\"\",\"\}|,
\]

\[
\text{|\{var,\"k\",\{"\eg\",\"\}|, \{var,\"k\",\"e4\",\"\}|}
\]

\[
\text{Expected: ";\}|\{text,\:"\}|parse(T, Term)}\};
\]

The parser is mistakenly interpreting the @@ as a text terminal. The problem is that @@ sequences must be interpreted as text only when the parser is not waiting to close an open @. This modification fixes the problem:

\[
\text{prop_string: ....Passed! After 48 tests.}
\]

\[
\text{|\{var,\"%\",\{"\}|, \{var,\"W\",y)8uFE\},}
\]

\[
\text{|\{text,\:"5OH\",\"\}|, \{var,\"sPtd4BH\",\"\",\"\}|,
\]

\[
\text{|\{var,\"k\",\{"\eg\",\"\}|, \{var,\"k\",\"e4\",\"\}|}
\]

\[
\text{Expected: ";\}|\{text,\:"\}|parse(T, Term)}\};
\]

After QuickCheck shrinks the counterexample, the problem is clear: the symbolic representation is inconsistent, we cannot specify different substitution values for the same variable name. There are many ways to solve this, in the repository we adopted the simple solution of filtering repeated variable names and fixing them so that there is only one substitution for each variable in the substitution list.
However, that unveils a flaw in the specification. When a bug shows up after thousands of tests it is a symptom of poor test case distribution; i.e. the generators must be fixed.

Furthermore, to_substs returns the substitutions in the same order and repeated as many times as they appear in the symbolic template representation. For example, for a symbolic template

\[
(\text{var}, "V1", "dou"), (\text{var}, "V2", "sin"), (\text{var}, "V1", "dou")
\]

QuickCheck will check next expression:

\[
\text{string}(\text{"SV10SV20SV10"},\ 
(\{\text{"V1"},\text{"dou"}\}, (\{\text{"V2"},\text{"sin"}\}, (\{\text{"V1"},\text{"dou"}\})
\):="dousindou"
\]

That is consistent with the implementation (the second "V1" tuple is ignored) but is not the expected test. Also, it does not test what happens when substitutions are not in the same order as variables appear in the template. A simple solution is to create a generator to shuffle—and possibly randomly repeat—the entries in the substitution list. The git repository offers solutions to all those problems. See commits from 7c15254 to 8d99984.

It’s important to have a test failing before fixing errors. For the bug above, we artificially biased the generator distribution to enable QuickCheck to detect it more frequently. After fixing the fault, the right generator was restored. Another example of changes oriented to make test fail before fixing anything is the saboteur [11] explained in 8d99984.

3. CONCLUSION

We showed how to write a template library in Erlang using a Property Driven Development approach. This is the natural evolution of Test Driven Development when using a Property Based Testing approach. Before writing any production code, we specify the new requirement by first writing a property that the code must meet; then we use the QuickCheck tool to find a counterexample; and finally we write the required code to fix all the counterexamples that QuickCheck finds until we are confident properties hold.

We find that both TDD and property based testing work very well together, boosting much of the strengths of those techniques when applied alone.

Writing properties rather than test cases means programmers think less on the actual implementation and more on expected results. Even though this claim can be used for traditional testing, we believe that in property based testing it is even more sound. This improves one of the accepted benefits of TDD: tests as documentation, as properties are more informative and maintainable than concrete test cases.

PDD reduces the risk of choosing a poor set of test cases to drive development when using TDD alone, or a good set of test cases that turns into a bad one after a change in the code. It is more likely that the test generator covers new corner cases introduced in future code changes.

In our example, we show how the @ symbols become special in the middle of the implementation and how the test generator starts to find a number of issues in the implementation because of that. If we used a traditional testing approach, new test cases should have been written once this behaviour was added.

We also showed that PDD alone cannot avoid weak specifications caused by poor data generators. Developers must be careful to get good data distribution from their generators when using QuickCheck to verify properties. In our example, a fault in the specification slept through all the implementation process because the trigger was extremely unlikely to pull with the initial template generator. This issue has nothing to do with TDD, but we believe it is critical to put extra care in verifying the data distribution to succeed using QuickCheck for PDD.

Last, but not least, PDD encourages a better selection of the properties written to specify the code. As opposed to writing the properties after production code, using the TDD approach ensures the programmer verifies that the property being written tests something that is failing and should be changed. In other words, it is less likely to write trivial properties that always hold and give developers with a false sense of security about the code that they supposedly are testing.

4. REFERENCES