Computing and Informatics, Vol. 35, 2016, 890–913

MAKING PROPERTY-BASED TESTING EASIER TO READ FOR HUMANS

Laura M. Castro

Facultade de Informática, Universidade da Coruña
Campus de Elviña S/N, 15071, A Coruña, Spain
e-mail: lcastro@udc.es

Pablo Lamela, Simon Thompson

School of Computing, University of Kent
Canterbury, Kent, CT2 7NZ, UK
e-mail: {P.Lamela-Seijas, S.J.Thompson}@kent.ac.uk

Abstract. Software stakeholders who do not have a technical profile (i.e. users, clients) but do want to take part in the development and/or quality assurance process of software, have an unmet need for communication on what is being tested during the development life-cycle. The transformation of test properties and models into semi-natural language representations is one way of responding to such need. Our research has demonstrated that these transformations are challenging but feasible, and they have been implemented into a prototype tool called `readSpec`. The `readSpec` tool transforms universally-quantified test properties and stateful test models – the two kinds of test artifacts used in property-based testing – into plain text interpretations. The tool has been successfully evaluated on the PBT artifacts produced and used within the FP7 PROWESS project by industrial partners.

Keywords: Test artifacts, test models, stakeholders, semi-natural language, property-based testing, quickcheck

1 INTRODUCTION

Property-based testing (PBT) is a powerful approach to software testing [8], but it may be as challenging for developers who first use it [39] as it is for non-technical stakeholders to understand, in general, software-related artifacts [46].

Take, for instance, a QuickCheck [8] property such as:

```
?FORALL({I, L}, {int(), list(int())},
        not lists:member(I, lists:delete(I,L)))
```

Despite its declarative style and universally-quantified abstraction, it cannot be claimed to be readable for everyone, and it certainly is far less understandable than:

```
  If you have a number
 and you have a list of numbers
when you remove the number from the list
then the list does no longer contain the number.
```

The situation is much worse when the software under test is complex enough to need a stateful model to be tested. On the other hand, agile approaches to software development demonstrate that the involvement of all stakeholders in the software life-cycle is a key factor for project success [9]. Specifically, enabling clients or users to validate software requirements by direct inspection or even interaction with software quality assurance artifacts such as tests cases and test suites brings effective benefits [33].

Thus, it is in our own interest as computer scientists and software developers to explore, formulate and build techniques and tools to fill in the gap that exists not only between PBT and software developers, but also between the PBT artifacts and the software clients and users.

This paper presents `readSpec`, a tool which automatically generates versions of:

- QuickCheck[1] properties which are readable for non-technical people. As criteria for the requirement "*readable for non-technical people*", we have adopted the style used in the tool Cucumber [25], which uses a semi-natural language style for describing tests.
- QuickCheck stateful models which are readable for non-technical people. In this case, since there are no tools that use any artifacts similar to these, we have developed our own semi-natural language representation, which is produced through symbolic execution.

A prototype version of `readSpec` is publicly available as a GitHub repository (`https://github.com/prowessproject/readspec`), which includes a folder with

[1] Specifically, Erlang QuickCheck, the most advanced version of QuickCheck available.

examples and a user manual. These are the same examples used throughout this paper.

From the research point of view, the main contributions of our work are threefold:

- A way to close the communication gap that currently exists between stakeholders with regard to the use of PBT.
- A proposal to use the Cucumber language, Gherkin [50], as semi-natural language that can express the same abstractions typically represented by test properties in PBT.
- A text-based format to express the preconditions, postconditions and results involved in tests derived from stateful PBT models.

The rest of the paper is organised as follows: Sections 2 and 3 present in detail our proposals for textual representations of test properties and test models, respectively. Then, Section 4 describes the evaluation of the tool into which we implemented these proposals, in an industrial setting. We conclude by analysing related work in Section 5 and final remarks in Section 6.

2 GENERATING BEHAVIOUR-DRIVEN DEVELOPMENT SCENARIOS FROM QUICKCHECK PROPERTIES

2.1 Behaviour-Driven Development: Cucumber

We know that when users and/or clients are not involved in the software development process, this is more likely to fail [9, 5, 42]. Involving users and clients in the software development process has led to the development or encouraged the adoption of design artifacts that are approachable by non-technical people, such as UML [13] or STEPS [16].

We also know that quality assurance activities are essential to the development of *the right software* for the users. As such, quality assurance artifacts are key for software development, since their purpose is to check the user requirements in the software product that is being or has been built.

However, there have been few efforts to make quality assurance artifacts more approachable to users and/or clients. Test cases, test suites and/or test models remain a very technical area. One notable exception, though, is Cucumber [25], a behaviour-driven development tool whose popularity is on the increase (cf. Figure 1).

Behaviour-driven development (BDD) takes test-driven development (TDD) one step closer to the user or client. While TDD is a developer-oriented approach that demands that tests are written before the implementation, and actually used as a guide for development, BDD deals with the description of *features* in semi-natural language, in order to make them accessible to people outside the development or testing teams. On the one hand, functionalities are described in a similar way to specifications; on the other hand, they are described using a set of keywords and

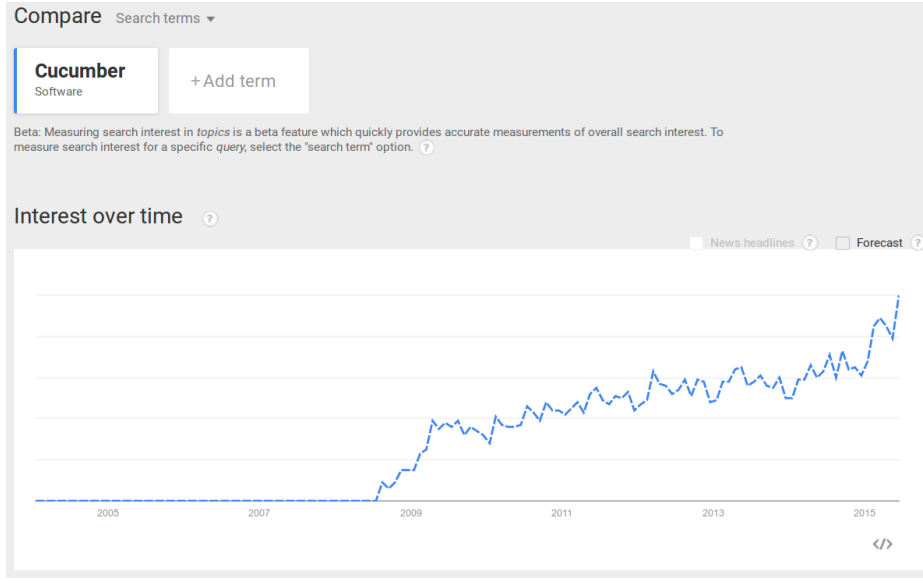

Figure 1. Google Trends: increased interest about "cucumber (software)"

a layout template that allows the automatic processing of those *scenarios* into actual test cases.

Cucumber is the most popular test tool for BDD, originally written in Ruby [25]. Since its first release in 2008, clone versions in a number of other languages have seen the light, also including two Erlang implementations [35, 40]. All of them execute plain-text functional descriptions as automated tests. Because Ruby is also the implementation language of the popular framework Ruby on Rails [23], a full-stack cross-platform web-app framework, the acceptance of Cucumber among web application developers has been wide.

With Cucumber, specifications are written as acceptance tests in *feature* files. Each *feature* can be described using a number of *scenarios*. Scenarios are transformed into executable test cases by the implementation of sequences of *steps*.

2.1.1 Cucumber Features

Feature files [50] are text-only files with the following header:

```
FEATURE: <name of the feature>
  In order to <general desc of the feature>
  I need to <goal of the feature>
```

The purpose of this general description of what the feature accomplishes is not operational, rather to provide context to the scenarios that follow.

2.1.2 Cucumber Scenarios

Each feature file can contain a number of scenarios which illustrate how the feature is expected to behave. The more diverse the scenarios are, the more complete the specification will be.

A Cucumber scenario [50] follows this template:

```
SCENARIO: <desc of scenario>
  GIVEN  <desc of input>
  (AND  <desc of additional input>)
   WHEN  <desc of action>
  (AND  <desc of additional action>)
   THEN  <desc of consequence of action>
  (AND  <desc of additional consequence of action>)
```

This template is the key to communication between developers, testers, and non-technical stakeholders. It represents a common ground because, thanks to being written in semi-natural language, it is understandable and assessable by users and clients. At the same time, it has a clear procedural style, which developers and testers can easily identify as a test scenario. Indeed, what tools like Cucumber do is to automatically translate these into executable test cases (cf. Figure 2).

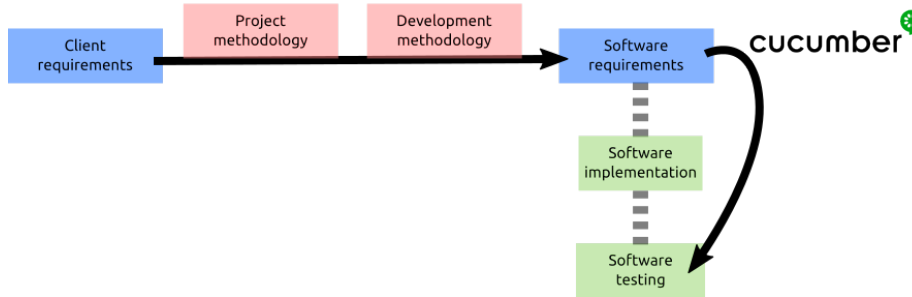

Figure 2. Role of Cucumber in the software development workflow

2.1.3 Cucumber Steps

The automatic translation of the scenarios included in a feature file into executable test cases is performed by defining each of the relevant steps (inputs, actions, and consequences) and implementing their translation into data, function calls, and result values in the target software. This translation is application-specific, and there is usually little room for code reuse [38, 19].

For the purpose of our research, we only focus on the automatic derivation of features and scenarios, and not on the implementation of steps. Our intention is

not to follow a BDD life cycle: we will not be writing cucumber-like specifications from scratch. Instead, we take QuickCheck properties and models as input sources. These test properties and test models are already executable and automatically transformed into test cases by QuickCheck. The gap we aim to fill in with this work is the one between the test properties and models, and the non-technical stakeholders (cf. Figure 3). That being said, the features and scenarios that our `readSpec` tool produces strictly follow the Cucumber conventions, so they can indeed be used with any Cucumber-like tool to derive test cases in their corresponding implementation languages (Ruby, Java, C++...).

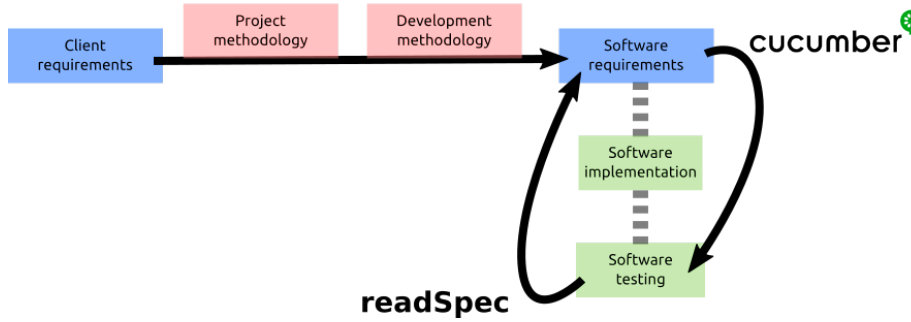

Figure 3. Role of `readSpec` in the software development workflow

2.2 Generation of Features and Scenarios from Properties

To illustrate how we have modelled and implemented the automatic generation of features and scenarios from QuickCheck properties, we will use a simple example. It is a basic QuickCheck module that contains only one test property, which follows the usual universally-quantifiable declarative statement format, and declares that any integer `I` should not be a `member` of the list that results from `delete`ing that integer from any given list of integers `L`.

```
prop_simple() ->
  ?FORALL({I, L}, {int(), list(int())},
     not lists:member(I, lists:delete(I,L))).
```

If we sample the generators `int()` and `list(int())` we can get and idea of the sort of input combinations that the property will be using when generating specific test cases:

```
> eqc_gen:sample({eqc_gen:int(), eqc_gen:list(eqc_gen:int())}).
{ 0, [-6, 10, -9]}
{ 4, []}
{-6, [5, 11]}
```

```
{−5, [7, 3, 11]}
{12, [−11, 15, 14]}
...
```

However, we would rather produce this output combined with the actual property body, in a more user-friendly way. Also, when testing QuickCheck properties, we would be running at least a hundred test cases, possibly many more, which will make us confident about the coverage of the subject under test (SUT). But when reporting a set of samples for human consumption, we would rather select a few that instead illustrate reasonable coverage. In the following subsections we explain how we have tackled this double challenge.

2.2.1 Selecting Sample Property Tests

In order to select a few test cases that we can transform into the scenarios that would describe a feature (i.e. property), we have chosen to use the QuickCheck `eqc_suite` library. This is an somewhat atypical subsection of QuickCheck features that allows us, among other things, to generate a reduced set of test cases that fulfill some criteria. One of those criteria is code coverage (`eqc_suite:coverage_based/2`).

For the criteria to be applied successfully, a coverage tool must be used in combination. We chose the coverage tool delivered with the rest of the Erlang/OTP standard libraries, `cover` [14].

Thus, the algorithm we follow is:

1. Cover-compile the module we want to test, to enable the collection of coverage measures.
2. Generate a reduced but representative set of test inputs using QuickCheck's `eqc_suite:coverage_based/2` and `eqc_suite:cases/1`.

This is better than just randomly sampling the generators, because it ensures a better distribution for our purposes (namely, that in addition to the random factor, the test data is generated trying to enforce the execution of all non-dead implementation code).

2.2.2 Property Features

As we already mentioned, each feature file has a header which only purpose is to give a general description of the feature described by the scenarios that follow.

Given that our input source for generating the feature file is an Erlang QuickCheck source file, we have assimilated this general description to the module header that developers should write when documenting their source code. The standard tool and format to document Erlang implementation modules is EDoc [15], a tag-based Javadoc-like format and tool that uses '`tags`' to identify pieces of commentary that can later on be extracted and transformed into HTML documentation.

The following is the EDoc header documentation for our simple example:

```
%%% @author Laura M. Castro <lcastro@udc.es>
%%% @copyright 2014
%%% @doc Simple QuickCheck properties
%%% @end
```

In this case, we are interested only in the '`doc`' tag, which is the one used to provide the general purpose of the module, namely the QuickCheck test module that contains the test properties.

The EDoc tool provides some utility functions to extract the EDoc comments from an Erlang module as an XML structure (`edoc_extract:source/2`), which we have used to our advantage. Obtaining not only the module description, but also the description of each property that has been written using the '`doc`' tag, is a matter of transversing the structure and locating the tags of interest.

```
%% @doc Deleting an integer from a list should result in a list
%%      that does not contain that integer.
%% @end
```

2.2.3 Property Scenarios

Once we have the input samples and have statically extracted from the module and properties documentation as much information as we can, it is the turn for generating the scenarios: one for each input sample.

There are two things we need to generate to comply with the Cucumber template:

1. A characterisation of the input data (GIVEN + ANDs part of the template), using the specific samples we have.
2. A description of the property (WHEN/THEN part of the template), in which we substitute variable names with their sample values.

The first task can be achieved by simple inspection of the generated sample data. For the second, we rely on the symbolic execution of the test property, as presented and discussed in Section 3 of this document. The symbolic execution capabilities are used to obtain a tree-like internal representation of the body function, which is then transversed (both in depth and width) to replace variable names with variable values.

The result of this process is a feature file which, for our simple example, includes scenarios such as:

```
FEATURE: simple
  Simple QuickCheck properties
SCENARIO: Deleting an integer from a list should result
          in a list that does not contain that integer.
GIVEN I have the integer 19
AND I have the list [7, -24, -18, 17, -8, -9, -8]
THEN IT IS FALSE THAT
```

```
the integer 19 is in
the result of calling lists:delete/2
              with 19 and [7, -24, -18, 17, -8, -9, -8].
```

Using `readSpec`, obtaining the complete feature file is a matter of just running `readspec:suite(simple_eqc, prop_simple)`, where `simple_eqc` is the name of the test module, and `prop_simple` the name of the property.

2.2.4 Features and Scenarios for Counterexamples

An additional functionality that we have incorporated into `readSpec` is the possibility of generating these scenarios not only from properties for which we sample their generators, but also from their counterexamples. We consider that obtaining a semi-natural language version of a failed test case (i.e., property counterexample) can be very useful for involving (and maintaining the involvement of) non-technical stakeholders in the decision of whether a given error is an implementation or a specification problem [34] (cf. Figure 4).

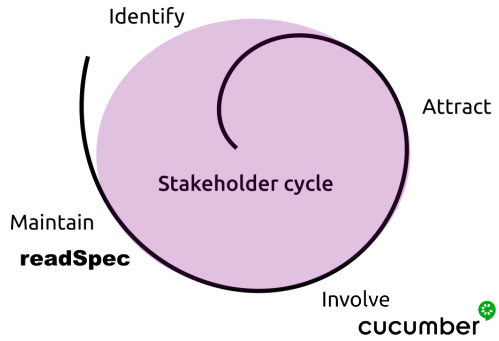

Figure 4. The optimal circle of stakeholder involvement in software development

Thus, when a counterexample is found during regular QuickCheck operation:

```
> eqc:quickcheck(simple_eqc:prop_simple()).
.....................................
...... Failed! After 51 tests.
{14, [-11, 14, -9, 12, 14, 14, -22, 18]}
Shrinking..(2 times)
{14, [14, 14]}
```

we can turn it into a cucumber-like scenario using `readspec:counterexample/3`, which generates a `prop_simple.counterexample.feature` file.

3 EXTRACTING HUMAN-READABLE DESCRIPTIONS FROM STATEFUL MODELS BY USING SYMBOLIC EXECUTION

Symbolic execution is a broadly researched topic for testing and test generation [31, 32, 51], for specification verification [43], and for invariant generation [11]. Symbolic execution has also been used sparingly for program comprehension as shown in [52], but we could not find it linked to natural language explanation generation.

Aside of symbolic execution, there is some work on automatic program documentation [17, 27], but it is mainly focused on providing information about structure and generic properties of functions, and not so much about behaviour description. On the other hand, existing work aimed at improving test readability usually focuses on getting tests out of use cases or either merging both worlds like it is the case of Cucumber [25].

In this section we illustrate the symbolic functionality of `readSpec` applied to automatically explaining QuickCheck stateful models using semi-natural language. The symbolic functionality of `readSpec` is aimed to help understand the QuickCheck `statem` models to people with a basic understanding of: the concept of state machine, the idea of testing, and programming artifacts like variables, collections and database tables; but that may not have any specific knowledge about the Erlang syntax nor the QuickCheck API.

As mentioned, Erlang QuickCheck is a PBT tool that supports the automatic generation (and execution) of test cases. The module `statem` is a QuickCheck library that makes it easier to design test models by describing them as abstract state machines. These stateful test models replace universally-quantified properties when the answers of the SUT depend on previous interactions with it.

A `statem` model must describe the abstract state machine by implementing a series of callbacks that provide information about:

- the data structure that holds the state,
- the commands or transitions that can be executed,
- the preconditions and postconditions that state must satisfy for each command to be appropriate,
- the effect that each command produces on the state,
- the arguments that each command takes as input.

QuickCheck `statem` models are analysed by `readSpec` with a dual approach. On the one hand, callbacks defined in QuickCheck models may be arbitrary Erlang functions, only restricted by the Erlang semantics. For this reason we use symbolic execution to analyse their possible behaviours, as shown in Section 3.1. On the other hand, the QuickCheck `statem` library enforces a well defined structure for the general model. This structure can be used to extract basic information about the model. Information like the structure of the record containing the state, the number of different commands defined, and the number of arguments required by each command, can be extracted easily by checking the basic structure of the callback

that are defined for each command (i.e., functions whose names end with: _pre, _post, _args...). In Section 3.2, we show how structural information is used to improve the results obtained with symbolic execution.

In addition, some programming patterns have special meaning to developers. Automatically generated explanation for these patterns may be perceived as unclear or hard to understand. With the aim of mitigating this problem, a mechanism is provided that allows the user to collect these patterns and provide clearer explanations for them. An example of usage of this mechanism is shown in Section 3.3.

Throughout the next sections we make use of a well-known example in the Erlang community, that has been repeatedly used to illustrate several aspects of testing: the process registry [7, 6, 30, 3, 45]. The Erlang's virtual machine has a process registry, meaning that after spawning a process, we can give it a readable name, and from then on refer to the process by its name instead of its process ID. The process registry provides the following API:

```
register(Name, Pid) -> ok
unregister(Name) -> ok
whereis(Name) -> Pid | undefined
```

were `register/2` associates a name with a process ID, `unregister/1` forgets that name, and `whereis/1` looks up a name to get a process ID. The QuickCheck model to test the process registry is about 150 SLOC of Erlang code; in the next sections, we show how different parts of that model get translated into text readable by non-technical people and discuss the complexities behind them.

3.1 Symbolic Execution

For each callback of each command, `readSpec` extracts a list of *possibilities*. A *possibility* represents a possible execution of the callback. Information provided about each *possibility* includes a semi-natural language explanation of the outcomes of the symbolic execution of the callback in terms of:

1. The arguments used to execute the command.
2. The fields of the record containing the *state* of the model.
3. The value returned by the command (the result).

If more information is required by the explanation, some of the following sections may also be included:

1. The definitions section, which provides new definitions of variables written in terms of other variables already defined, or function calls. These are usually due to function calls that the system could not analyse, i.e., those that contain calls to functions that are in different modules, or functionalities that are not implemented (like operations with binaries or list comprehensions).

2. The requirements section, which provides requirements that must be satisfied for the given result to be produced. These requirements may also be written in terms of other variables already defined.

If several *possibilities* are detected, they will be presented separated by "`OTHERWISE`" labels. These labels remind us that additional *possibilities*, as an implied extra requirement, must not satisfy the requirements of any of the previously described *possibilities*, even if they do not have any explicit requirements of their own.

Let us consider for example the function `register_pre/1`, that in the QuickCheck model for the process registry describes the preconditions needed to be able to `register` a process with a name. If we provide a single variable called `Arg1` as argument, we obtain the following output:

```
==========================================
        POSSIBILITIES ...
==========================================

*** DEFINITIONS ***
[*] We define the variable "Arg1" as in the arguments provided
[*] We define the variable "rec_elem_pids_of__:Arg1" as the
result provided by the function called "get_value" from the
module "?RECORD", when it is executed with the following 3
arguments:
    - the literal 'state'
    - the literal 'pids'
    - the variable "Arg1"

*** REQUIREMENTS ***
[*] The variable "Arg1" must contain a record of type "state".
[*] The variable "rec_elem_pids_of__:Arg1" must be equal to:
    - the empty list

*** RESULT ***
the literal 'false'

==========================================
        OTHERWISE ...
==========================================

*** RESULT ***
the literal 'true'
```

Symbolic execution will try to get the most concrete result out of the code of the callback `register_pre/1`. We can see that it has been detected that `Arg1` must contain a record of type `#state{}`. But because we have not bound the fields in this record to individual variables when setting the arguments, a new variable called `'rec_elem_pids_of__:Arg1'` has to be generated. Because records are transformed

into tuples at compilation time, we cannot rely on existing built-in-functions to unfold the record; consequently, we write the call in terms of a hypothetical function inside the fake module `?RECORD`. The value of this field cannot be determined either by looking at the code or by looking at the arguments provided, so the inequality cannot be deterministically resolved and two possibilities are considered: either it is not satisfied (the elements are equal and inequality returns the atom `false`), or else it is. If the result is assumed `false`, it can be implied that the field `pids` must contain an empty list. If this implication is not complied with, then we know the output of the function must be the atom `true` unless an exception is thrown, but our approach discards abnormal behaviours for simplicity.

Variable names like `'rec_elem_pids_of__:Arg1'` are not very intuitive, and so several mechanisms for renaming them are provided:

1. Variables can be manually renamed by calling `rename_vars_in_model/2` on the result of the symbolic execution. This replaces all the appearances of the variables provided in the extracted information.
2. If the expression is assigned to a variable in the code, the algorithm will try to use the name of that variable for the expression rather than the generated one, if that does not produce any conflict.
3. A more advanced mechanism called idiom definition is explained in Section 3.3; not only can it be used to mitigate the problem of variable names, but it may also be used to reduce the complexity of the whole explanation.

In addition, some variables may not be generated at all if we use the right structural information, as explained in Section 3.2.

3.2 Structural Information

Before symbolically executing a callback, we have the possibility to gather some extra information. In our example we can safely assume *a priori* that the state provided to the function `register_pre/1` will consist of a record of type `#state{}` if, for instance, the `statem` function `initial_state/1` returns a record of this type. Technically, the QuickCheck framework allows the type of the state to change, but this practice is uncommon, discouraged, and it would not provide any extra flexibility to a test model.

Thus, assuming the state will always consist of a record of the type provided by the function called `initial_state/1`, we can give this information to the symbolic analyser by providing it with the argument already structured as a record:

```
#state{pids=StatePids, regs=StateRegs, dead=StateDead}
```

Doing this gives us a slightly clearer output:

```
==========================
        POSSIBILITIES ...
==========================
```

```
*** DEFINITIONS ***
[*] We define the variable "StatePids" as in the arguments
provided

*** REQUIREMENTS ***
[*] The variable "StatePids" must be equal to:
    - the empty list

*** RESULT ***
the literal 'false'
```

```
OTHERWISE ...
```

```
*** RESULT ***
the literal 'true'
```

This technique can be used with all the arguments of the callback, and all QuickCheck stateful model callbacks. For example, in the case of `_next` and `_post` callbacks (that, respectively, implement the test model change of state and post-conditions), we can provide a list with the right number of elements since this information is given by the `_args` callback. And at the end of the process, when constructing the global semi-natural language explanation, we can replace the appearances of the arguments we provided with manual descriptions of their origin (i.e., we can replace occurrences of the variable `StatePids` with the phrase "the field called `pids`").

3.3 Natural Expression of Idioms

Whenever a call to a function that cannot be symbolically executed is detected, this call is automatically added to the list of definitions and given a generated variable name. These definitions make the explanation harder to read, and force the user to switch back and forward to and from the definitions section. Fortunately, lots of these calls follow programming patterns or "idioms" that can be collected and explained manually. For this reason `readSpec` provides a mechanism to define rules that will rewrite these patterns and transform them into natural explanations.

For example, let us consider the common practice of adding (appending) an element to the end of a list (`L ++ [E]`). The original output for code involving this practice will look like:

```
*** DEFINITIONS ***
[*] We define the variable "call_to-lists:append/2-1__" as the
result provided by the function called "append" from the module
"lists", when it is executed with the following 2 arguments:
```

```
    - the field called "pids"
    - a list with the following element:
          - the result of calling spawn/0
*** RESULT ***
a record of type 'state' with the fields:
    regs = the field called "regs"
    dead = the field called "dead"
    pids = the variable "call_to-lists:append/2-1__"
```

We can identify the definition of the variable `'call_to-lists:append/2-1__'` and see that the operator `++` has been translated into a call to `lists:append/2`, with two parameters that are written in abstract syntax. This can be transformed into a template by replacing the generic parts within `readSpec`'s internal code structure with 'template' tags conforming an idiom:

```
create_idiom(tvar(Ref, 'atom_identifier'),
   "the result of adding %ELEMENT% to the collection in %LIST%",
        [{"%ELEMENT%", var_name, tvar(Ref, 'element_varname')},
         {"%LIST%", var_name, tvar(Ref, 'collection_varname')}])
```

where `Ref` is used by the template system to distinguish the template terms from the normal ones, and `tvar` creates a tag that will match any Erlang term and give it the provided name. Internally, `readSpec` generates a dictionary of these template tags that is used during text generation.

When idiom templates such as the previous one are defined, textual representations like the following are produced by the pretty-printer:

```
*** RESULT ***
a record of type 'state' with the fields:
    regs = the field called "regs"
    dead = the field called "dead"
    pids = the result of adding the result of calling spawn/0
           to the collection in the field called "pids"
```

If we need to match several definitions, we may do so by using several place holders with common tags throughout them. When using several template tags with the same name, the system guarantees that all occurrences of those tags that have the same name are matched to the same Erlang structure each time, pretty much like normal Erlang pattern matching does but on a larger scale.

4 TOOL EVALUATION

To evaluate the tool beyond the examples used to conduct our research, we have used `readSpec` on properties and models produced and used within the PROWESS project [2] by industrial project partners.

With this pilot study, we aimed to answer two research questions:

RQ1 Can readSpec produce efficient text representations of QuickCheck properties and models?

RQ2 Are readSpec text representations an effective way to communicate what is being tested among stakeholders?

To answer the first question, we decided to measure the amount of time required for `readSpec` to generate the textual representations, and the number of lines of `readSpec` textual representations w.r.t. the number of lines of corresponding test properties/models. To answer the second question, we showed the stakeholders both the test properties and models that were being used, and the textual representations generated by the tool, and asked them to rate (0–10) whether they would use `readSpec`'s outputs instead of their current way of communication, and whether `readSpec`'s outputs were easy to understand, to follow, appropriate in verbosity, etc. (also on a scale of 0–10).

The results we gathered from this pilot can be summarised in the following points:

- The amount of time needed to produce the textual representations (in the order of milliseconds) does not seem to be a threat to applicability.
- The nature of the output (verbosity, number of lines) seems to be appropriate in terms of the number of cases which are needed to illustrate a property, which are in direct relationship with the data distribution of the corresponding data generators. Same happens with the number of lines to describe a stateful model, which is in direct relationship to the number and complexity of its preconditions and postconditions (c.f. Figure 5).
- The feedback from the interviews with the stakeholders showed that for simpler cases there was a high degree of influence of the persons skills and background as to whether a test property source code was considered more helpful and/or easier to understand that its corresponding textual representation, as produced by the `readSpec` tool. However, for more complex cases, two situations were consistent:

  1. there were some people that still would not make sense of either of the two alternatives,
  2. the rest of the interviewees consistently agreed that the text version was more helpful.

  This usefulness was in some cases even strong enough to provide people the insights needed to suggest more tests.

4.1 Improvements Derived from Evaluation

The industrial evaluation of our tool led us to identify several improvements to the tool, which constitute our main lines of future work.

Regarding the behaviour-driven development scenarios, QuickCheck test modules usually contain not one, but a number of test properties. This can lead to

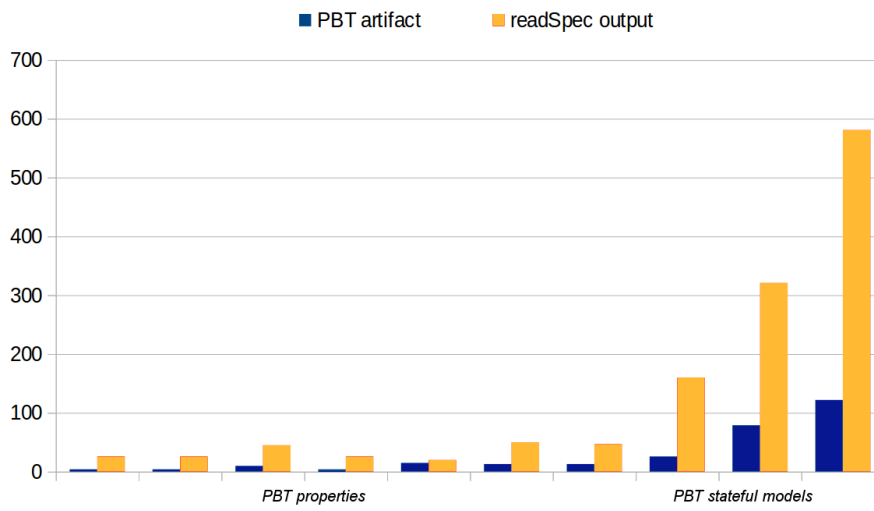

Figure 5. Lines of code of the examples shown to stakeholders during evaluation

very verbose feature files if all scenarios for those properties are condensed into one single feature file. On the other hand, it can lead to a big number of files if scenarios are written into separated files (i.e., one per property). It is yet unclear to us that one of the two possibilities will be unanimously preferred by developers and users, so the option to configure this behaviour at the tool level is a desired feature.

Besides, at the moment `readSpec` uses the coverage tool provided by the Erlang OTP libraries, `cover`. However, the PROWESS coverage tool Smother [47] provides an improved coverage measure that we would rather use. This will involve solving some integration issues that currently exist between Smother and the QuickCheck `eqc_suite` library.

For documentation purposes, and in order to guarantee the acceptance and use of `readSpec` within the Erlang community, we aim to develop some additional examples. For instance, to ensure seamless integration with Cucumber-like tools, we will implement the necessary steps in order for them to be executable using one of the Erlang implementations of Cucumber.

Regarding the descriptions of stateful models, a list of future improvements follows.

4.1.1 Contextualisation of Output

Depending on the kind of callback that is being explained, information could be presented differently. For example, the output

```
PRECONDITION
```

```
*** RESULT ***
the literal 'true'
```

tells us that there is no precondition for the current command. Or, effectively, that the precondition returns always the atom `true`. This could be expressed in a more intuitive way with a message like:

```
The command can always be executed independently
of the current state of the model.
```

In a similar way, we could further improve output such as

```
*** RESULT ***
a record of type 'state' with the fields:
    regs = the field called "regs"
    dead = the field called "dead"
    pids = the result of adding the result of calling spawn/0
           to the collection in the field called "pids"
```

since even after using idioms, this piece of output may be seen as a bit confusing, when it is merely meant to explain that the only field modified is the field `pids`. In future work, only the modified fields should be specified. In addition, it would also be interesting to describe the field modifications in a reflexive way, i.e., "the result of calling `spawn/0` is added to the collection in the field `pids`".

4.1.2 Removal of Graphical ASCII Markers

Lines and boxes drawn through the use of the characters "`=`" and "`*`" should not be necessary after the output has been contextualised and simplified. In cases of need, hierarchy could be expressed in a cleaner way by using indentation, numeration or bullets.

4.1.3 Parametrisation of Verbosity

While in most cases we want the output to be simple and non-verbose, in some scenarios condensed output may be harder to understand. In those cases, and for debugging purposes, it may be useful to be able to get less refined output on demand. Consequently, we would add this option as a configuration option that can be enabled or disabled at will.

4.1.4 Information about Argument Generation

It can be observed that, when using the symbolic functionality of the current version of `readSpec`, no information is extracted about the possible type of the arguments

used with the commands. This information could be extracted from QuickCheck's generators. This information could be useful to the user and should be provided in the semi-natural description of the model.

One last general concern about the `readSpec` tool is scalability. Symbolic execution is known to scale poorly. Every reachable bifurcation will in principle duplicate the number of possibilities that reach it. Because of this, even simple programs may produce an output too big for symbolic execution to be feasible. However, all the test models written by our industrial partners that we have analysed so far are processed by `readSpec` within seconds. Since most of these models correspond to the unit level of testing, this may indicate that at least the unit-level behaviour is, in general, simple enough to be handled by our tool without issues.

5 RELATED WORK

The research on how to improve the involvement of stakeholders in the software development process has been extensive in recent years, following the broad adoption of agile methodologies and practices [49, 12, 44, 29]. However, most efforts focus on how to make this involvement more effective where it is taken for granted: during requirements extraction [24, 26, 20]. Even if we know that sustained stakeholder involvement is key to the success of a software product [4, 22], when it comes to extending said involvement to other stages of the software development process, initiatives tend to focus on improvement of usability testing and user interface design [18, 48, 21], or final validation activities [37, 41].

Indeed, some of these activities explore the source of tools like Cucumber [36], but frequently to help developers connect human-readable test specifications with specific test strategies (i.e. model-based testing [10]) or to specific systems in which expressing such requirements is in itself challenging [28]. Naturally, the state of the art inspired the work presented here, as explained in Section 2.1, but to the best of our knowledge, this paper constitutes the first effort to close the circle and make the test artifacts of advanced testing strategies and techniques (i.e. properties, PBT stateful models), and also additional by-products (i.e. counterexamples), available *back* to the stakeholders, thus representing a key contribution for sustained involvement throughout software development.

In its novelty, our work opens additional lines of research that would deserve further investigation, such as the definition of alternative ways of describing stateful scenarios in a sensible way. We do not claim the proposal we have made in this regard to be optimal, and most certainly there is room for improvement around it.

6 CONCLUSION

The goal of `readSpec`, the tool we have presented in this paper, is to help non-technical stake-holders to understand the artifacts used in property-based testing. To this end, we have explored the transformation of test properties and test models

into semi-natural language representations. This transformation of PBT sources into their human-readable written descriptions is intended to allow developers (and/or testers) to communicate the requirements they are implementing and testing in the software to other stake-holders who may not have the technical knowledge to understand PBT artifacts.

This transformation presents a research challenge in that PBT artifacts are higher-level representations of requirements which are later on translated by tools such as QuickCheck into specific test cases, but are still expressed using a programming language. Also, test models are divided into a number of components (preconditions, postconditions, state transformations) which need to be integrated and presented in a coherent manner.

Our work closes the circle of communication with stakeholders, that could already use tools like Cucumber to express requirements that would later on generate test cases; with `readSpec` now, additional test properties and models that developers write, alongside counterexamples that test cases reveal, can be expressed in a human-friendly manner. Alongside this methodological contribution, two main technical contributions have been discussed:

1. Automatic transformation of test properties into Cucumber-like scenarios, which are semi-natural English text versions of sample test cases.
2. Automatic transformation of stateful test models into semi-natural English descriptions of the behaviour they represent.

Both of them have been implemented in a single tool, `readSpec`, available for download as a GitHub repository [1], together with the full examples that have been used through the paper, and a user manual. Last but not least, a number of improvements, based on the evaluation of the tool by an industrial partner, have also been presented, and are currently being incorporated into upcoming versions of the tool.

One of the aspects that came up as most interesting to practitioners during the evaluation of our tool was the stakeholder-readable presentation of counterexamples. Our current work is dealing with this aspect, working on the automatic generation of other kinds of representations for counterexamples, such as sequence diagrams, applicable to both stateless and stateful systems or components.

Acknowledgements

This research has been partially funded by the European Framework Program, FP7-ICT-2011-8 Ref. 317820.

REFERENCES

[1] readSpec: Making PBT Easier to Read for Humans. `https://github.com/prowessproject/readspec`, May, 2015.

[2] PROWESS Project (Property-Based Testing for Web Services). http://www.prowessproject.eu, October, 2012.

[3] Benac-Earle, C.—Fredlund, L.: Verification of Timed Erlang Programs Using McErlang. International Conference on Formal Techniques for Distributed Systems, 2012, pp. 251–267.

[4] Berki, E.—Georgiadou, E.—Holcombe, M.: Requirements Engineering and Process Modelling in Software Quality Management – Towards a Generic Process Metamodel. Software Quality Journal, Vol. 12, 2004, No. 3, pp. 265–283.

[5] Boehm, B. W.: A Spiral Model of Software Development and Enhancement. Computer, Vol. 21, 1988, No. 5, pp. 61–72.

[6] Christakis, M.—Sagonas, K.: Static Detection of Race Conditions in Erlang. 12th International Symposium on Practical Aspects of Declarative Languages, 2010, pp. 119–133.

[7] Claessen, K.—Palka, M.—Smallbone, N.—Hughes, J.—Svensson, H.—Arts, T.—Wiger, U.: Finding Race Conditions in Erlang with QuickCheck and PULSE. 14th ACM SIGPLAN International Conference on Functional Programming, 2009, pp. 149–160.

[8] Claessen, K.—Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. SIGPLAN Notices, Vol. 46, 2011, No. 4, pp. 53–64.

[9] Cockburn, A.—Highsmith, J.: Agile Software Development, the People Factor. Computer, Vol. 34, 2001, No. 11, pp. 131–133.

[10] Colombo, C.—Micallef, M.—Scerri, M.: Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing. Model Based Testing, Vol. 141, 2014, pp. 14–28.

[11] Csallner, C.—Tillmann, N.—Smaragdakis, Y.: DySy: Dynamic Symbolic Execution for Invariant Inference. 30th International Conference on Software Engineering, 2008, pp. 281–290.

[12] Dey, P. K.—Kinch, J.—Ogunlana, S. O.: Managing Risk in Software Development Projects: A Case Study. Industrial Management and Data Systems, Vol. 107, 2007, No. 2, pp. 284–303.

[13] Dobing, B.—Parsons, J.: Dimensions of UML Diagram Use: A Survey of Practitioners. Journal of Database Management, Vol. 19, 2008, No. 1, pp. 1–18.

[14] Erlang/OTP Library: Cover: A Coverage Analysis Tool for Erlang. http://www.erlang.org/doc/man/cover.html.

[15] Erlang/OTP Library: EDoc: The Erlang Program Documentation Generation. http://www.erlang.org/doc/man/edoc.html.

[16] Floyd, C.—Reisin, F.—Schmidt, G.: Steps to Software Development with Users. Lecture Notes in Computer Science, Vol. 387, 1989, pp. 48–64.

[17] Frank, C.: A Step Towards Automatic Documentation. MIT Artificial Intelligence Laboratory, 1980.

[18] Fruhling, A.—De Vreede, G.-J.: Collaborative Usability Testing to Facilitate Stakeholder Involvement. Value-Based Software Engineering, 2006, pp. 201–223.

[19] Gärtner, M.: ATDD by Example: A Practical Guide to Acceptance Test-Driven Development. Addison-Wesley Signature Series (Beck), Pearson Education, 2012.

[20] Gottesdiener, E.: Requirements by Collaboration: Getting It Right the First Time. IEEE Software, Vol. 20, 2003, No. 2, pp. 52–55.

[21] Grigera, J.—Rivero, J. M.—Robles Luna, E.—Giacosa, F.—Rossi, G.: From Requirements to Web Applications in an Agile Model-Driven Approach. Lecture Notes in Computer Science, Vol. 7387, 2012, pp. 200–214.

[22] Hansen, S.—Berente, N.—Lyytinen, K.: Requirements in the 21st Century: Current Practice and Emerging Trends. Lecture Notes in Business Information Processing, Vol. 14, 2009, pp. 44–87.

[23] Hansson, D.: Ruby on Rails: Web Development That Doesn't Hurt. `http://rubyonrails.org`, 2015.

[24] Harris, M. A.—Weistroffer, H. R.: A New Look at the Relationship Between User Involvement in Systems Development and System Success. Communications of the Association for Information Systems, Vol. 24, 2009, No. 1, pp. 739–756.

[25] Hellesøy, A.: Cucumber. `http://cukes.info/`, 2015.

[26] Henfridsson, O.—Lindgren, R.: User Involvement in Developing Mobile and Temporarily Interconnected Systems. Information Systems Journal, Vol. 20, 2010, No. 2, pp. 119–135.

[27] Hermenegildo, M.: A Documentation Generator for (C) LP Systems. Computational Logic (CL 2000), 2000, pp. 1345–1361.

[28] Hesenius, M.—Griebe, T.—Gruhn, V.: Towards a Behavior-Oriented Specification and Testing Language for Multimodal Applications. ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2014, pp. 117–122.

[29] Hester, P. T.—Bradley, J. M.—Adams, K. M.: Stakeholders in Systems Problems. International Journal of System of Systems Engineering, Vol. 3, 2012, No. 3-4, pp. 225–232.

[30] Hughes, J.: QuickCheck Testing for Fun and Profit. International Symposium on Practical Aspects of Declarative Languages, 2007, pp. 1–32.

[31] Khurshid, S.—Păsăreanu, C. S.—Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. Tools and Algorithms for the Construction and Analysis of Systems, 2003, pp. 553–568.

[32] King, J. C.: Symbolic Execution and Program Testing. Communications of the ACM, Vol. 19, 1976, No. 7, pp. 385–394.

[33] Kujala, S.: User Involvement: A Review of the Benefits and Challenges. Helsinki University of Technology, Software Business and Engineering Institute. Teknillinen Korkeakoulu, 2002.

[34] McManus, J.: Managing Stakeholders in Software Development Projects. Computer Weekly Professional Series, Elsevier, 2005.

[35] Membase: Cucumberl: Pure Erlang Implementation of Cucumber Parser and Driver. `https://github.com/membase/cucumberl`, 2015.

[36] Micallef, M.—Colombo, C.: Lessons Learnt from Using DSLs for Automated Software Testing. IEEE International Conference on Software Testing, Verification and Validation, 2015.

[37] Mäntylä, M. V.—Itkonen, J.—Iivonen, J.: Who Tested my Software? Testing as an Organizationally Cross-Cutting Activity. Software Quality Journal, Vol. 20, 2012, No. 1, pp. 145–172.

[38] Nelson-Smith, S.: Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure as Code. O'Reilly Media, 2013.

[39] Nilsson, A.—Castro, L. M.—Rivas, S.—Arts, T.: Assessing the Effects of Introducing a New Software Development Process: A Methodological Description. International Journal on Software Tools for Technology Transfer, 2013, pp. 1–16.

[40] OpenShine: Kucumberl: Pure Erlang, Open-Source Implementation of Cucumber. `https://github.com/openshine/kucumberl`, 2015.

[41] Poston, R.—Sajja, K.—Calvert, A.: Managing User Acceptance Testing of Business Applications. Lecture Notes in Computer Science, Vol. 8527, 2002, pp. 92–102.

[42] Drew Procaccino, J.—Verner, J. M.—Overmyer, S. P.—Darter, M. E.: Case Study: Factors for Early Prediction of Software Development Success. Information and Software Technology, Vol. 44, 2002, No. 1, pp. 53–62.

[43] Păsăreanu, C. S.—Visser, W.: Verification of Java Programs Using Symbolic Execution and Invariant Generation. In Model Checking Software, 2004, pp. 164–181.

[44] Roberts, T. L. Jr.—Leigh, W.—Purvis, R. L.: Perceptions on Stakeholder Involvement in the Implementation of System Development Methodologies. Journal of Computer Information Systems, Vol. 40, 2000, No. 3, pp. 78–83.

[45] Sagonas, K.: Using Static Analysis to Detect Type Errors and Concurrency Defects in Erlang Programs. International Symposium on Functional and Logic Programming, 2010, pp. 13–18.

[46] Soeken, M.—Wille, R.—Drechsler, R.: Assisted Behavior Driven Development Using Natural Language Proceessing. Objects, Models, Components, Patterns, Lecture Notes in Computer Science, Vol. 7304, 2012, pp. 269–287.

[47] Taylor, R.: Smother: MD/DC Analysis for Erlang. `http://ramsay-t.github.io/Smother`, 2015.

[48] Vines, J.—Clarke, R.—Wright, P.—McCarthy, J.—Olivier, P.: Configuring Participation: On How We Involve People in Design. International SIGCHI Conference on Human Factors in Computing Systems, 2013, pp. 429–438.

[49] Voinov, A.—Bousquet, F.: Modelling with Stakeholders. Environmental Modelling and Software, Vol. 25, 2010, No. 11, pp. 1268–1281.

[50] Wynne, M.—Hellesoy, A.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Programmers. Pragmatic Bookshelf, 2012.

[51] Xie, T.—Marinov, D.—Schulte, W.—Notkin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. Tools and Algorithms for the Construction and Analysis of Systems, 2005, pp. 365–381.

[52] Zhang, X.: Analysis-Based Techniques for Program Comprehension. `https://www.researchgate.net/publication/242283806_Analysis-based_techniques_for_Program_Comprehension`, 2005.

Laura M. Castro is a post-doctoral researcher and assistant teacher at the University of A Coruña (Spain). Her research focuses on distributed systems, functional programming, design patterns, and more recently, software testing. She received her Ph.D. degree Cum laude in 2010, with a dissertation entitled "On the development life cycle of distributed functional applications: a case study". She has spent several months in institutions as the University of Houston (USA), the University of Gothenburg (Sweden) and Chalmers University (Sweden). She has been teaching at the university since 2005, where she currently lectures on software architecture and software validation and verification. As a postdoc, she has been involved in several European research projects (FP7).

Pablo Lamela is a researcher at the School of Computing of the University of Kent where he is a member of the Programming Languages and Systems Group. In 2011 he completed his master thesis in computer science at the Chalmers Tekniska Högskola (Gothemburg), and he received the degree of Engineer in computer science at the Universidade da Coruña. He has participated in the EU ICT FP7 project PROWESS. He is author and co-author of several scientific papers. His research interests include web services, functional languages, source code manipulation and comprehension, grammar inference, and software models.

Simon Thompson is Professor of Logic and Computation in the School of Computing at the University of Kent, UK. His research interests include computational logic, functional programming, testing and diagrammatic reasoning. His recent research has concentrated on all aspects of refactoring for functional programming, including the tools HaRe and Wrangler for Haskell and Erlang. He is also the author of standard texts on Haskell, Erlang, Miranda and constructive type theory. He is a Fellow of the British Computer Society and has degrees in mathematics from Cambridge (MA) and Oxford (DPhil).